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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
30 June 2016 (30.06.2016)

WIPO | PCT

(10) International Publication Number
WO 2016/105761 A1

(51) International Patent Classification:
G06F 9/38 (2006.01) *G06T 1/20* (2006.01)
G06F 15/80 (2006.01)

(21) International Application Number:
PCT/US2015/062066

(22) International Filing Date:
23 November 2015 (23.11.2015)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/581,858 23 December 2014 (23.12.2014) US

(71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, Santa Clara, California 95054 (US).

(72) Inventors: **CHEN, Wei-Yu**; 1000 Kiely Boulevard, Apt 108, Santa Clara, California 95051 (US). **LUEH, Guei-Yuan**; 1239 Quail Creek Circle, San Jose, California 95120 (US). **MAIYURAN, Subramaniam**; 1915 Black Slate Court, Gold River, California 95670 (US).

(74) Agent: **WEBSTER, Thomas**; Nicholson De Vos Webster & Elliott, LLP, 217 High Street, Palo Alto, California 94301 (US).

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

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

Published:

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

(54) Title: METHOD AND APPARATUS FOR EFFICIENT EXECUTION OF NESTED BRANCHES ON A GRAPHICS PROCESSOR UNIT

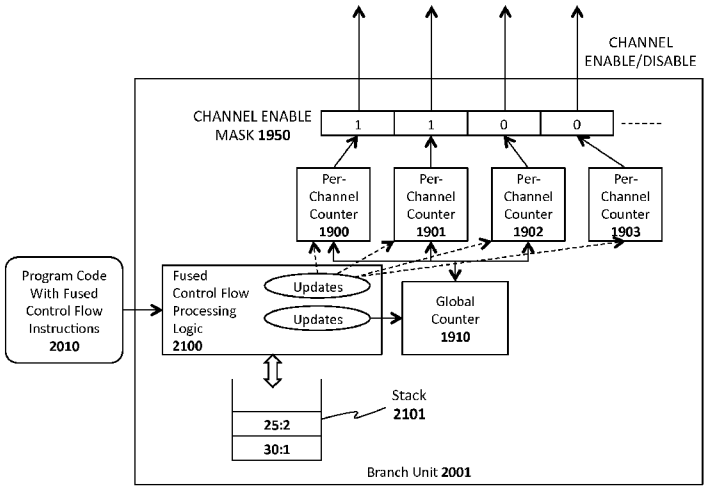


FIG. 21

(57) Abstract: An apparatus and method for executing nested control flow instructions on a graphics processing unit (GPU). For example, one embodiment of a processor comprises: an execution unit having a plurality of channels to execute control flow instructions including fused control flow instructions comprising two or more consecutive control flow instructions fused into a single fused control flow instruction; and a branch unit to process the control flow instructions and to maintain a global counter indicating a nesting level of the control flow instructions, wherein to process a fused control flow instruction, the branch unit is to store a value N in a stack indicating a number of control flow instructions fused into the fused control flow instruction, the branch unit to subsequently read the value N from the stack upon execution of the fused control flow instruction and decrement the global counter by a value of N responsive to execution of the fused control flow instruction.

WO 2016/105761 A1

METHOD AND APPARATUS FOR EFFICIENT EXECUTION OF NESTED BRANCHES ON A GRAPHICS PROCESSOR UNIT

BACKGROUND

Field of the Invention

[0001] This invention relates generally to the field of computer processors. More particularly, the invention relates to an apparatus and method for efficient execution of nested branches on a graphics processor unit.

Description of the Related Art

[0002] Managing control flow in single instruction multiple data (SIMD) programs is a complex problem. Traditionally, graphics processing units (GPUs) use scalar code and program routines to control instruction pointer (IP) addresses for each SIMD channel. This is inefficient both in terms of performance and power usage.

[0003] Control flow is managed on some architectures by maintaining a unique IP address for each channel. For example, when a control flow instruction is encountered, the IP of each channel is updated with a particular IP based on the predicate mask of the instruction. For each and every instruction, the execution IP is compared to the channel's IP to determine if that channel is enabled for a particular instruction at the current IP.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0005] **FIG. 1** is a block diagram of an embodiment of a computer system with a processor having one or more processor cores and graphics processors;

[0006] **FIG. 2** is a block diagram of one embodiment of a processor having one or more processor cores, an integrated memory controller, and an integrated graphics processor;

[0007] **FIG. 3** is a block diagram of one embodiment of a graphics processor which may be a discreet graphics processing unit, or may be graphics processor integrated with a plurality of processing cores;

[0008] **FIG. 4** is a block diagram of an embodiment of a graphics-processing engine for a graphics processor;

[0009] **FIG. 5** is a block diagram of another embodiment of a graphics processor;

[0010] **FIG. 6** is a block diagram of thread execution logic including an array of processing elements;

[0011] **FIG. 7** illustrates a graphics processor execution unit instruction format according to an embodiment;

[0012] **FIG. 8** is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline, a media pipeline, a display engine, thread execution logic, and a render output pipeline;

[0013] **FIG. 9A** is a block diagram illustrating a graphics processor command format according to an embodiment;

[0014] **FIG. 9B** is a block diagram illustrating a graphics processor command sequence according to an embodiment;

[0015] **FIG. 10** illustrates exemplary graphics software architecture for a data processing system according to an embodiment;

[0016] **FIG. 11** illustrates one embodiment of a system architecture for performing SIMD branch operations;

[0017] **FIG. 12** illustrates a branch unit in accordance with one embodiment of the invention;

[0018] **FIG. 13** illustrates a method for processing diverging control flow instructions in accordance with one embodiment of the invention;

[0019] **FIG. 14** illustrates a method for processing converging control flow instructions in accordance with one embodiment of the invention;

[0020] **FIG. 15** illustrates one embodiment of a branch unit with unstructured control flow logic;

[0021] **FIG. 16** illustrates additional details of one embodiment of the branch unit;

[0022] **FIG. 17** illustrates a method for processing unstructured diverging control flow instructions in accordance with one embodiment of the invention;

[0023] **FIG. 18** illustrates a method for processing unstructured converging control flow instructions in accordance with one embodiment of the invention;

[0024] **FIG 19** illustrates the operation of one embodiment which does not rely on fused control flow instructions;

[0025] **FIG. 20** illustrates one embodiment of the invention in which fusing logic fuses control flow instructions;

[0026] **FIG. 21** illustrates one embodiment of a branch unit including fused control flow instruction processing logic for processing fused control flow instructions;

[0027] **FIG. 22** illustrates the operation of one embodiment of the invention in which a particular type of control flow instructions (ENDIF) are fused; and

[0028] **FIG. 23** illustrates a method in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

[0029] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

EXEMPLARY GRAPHICS PROCESSOR ARCHITECTURES AND DATA TYPES

[0030] **System Overview**

[0031] **Figure 1** is a block diagram of a data processing system 100, according to an embodiment. Data processing system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In on embodiment, the data processing system 100 is a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

[0032] An embodiment of data processing system 100 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments, data processing system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system 100 can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system 100 is a television or set top box device having one or more processors 102 and a graphical interface generated by one or more graphics processors 108.

[0033] In some embodiments, the one or more processors 102 each include one or more processor cores 107 to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores 107 is configured to process a specific instruction set 109. In some embodiments, instruction set 109 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores 107 may each process a different instruction set 109, which may include instructions to facilitate the emulation of other instruction sets. Processor core 107 may also include other processing devices, such a Digital Signal Processor (DSP).

[0034] In some embodiments, the processor 102 includes cache memory 104. Depending on the architecture, the processor 102 can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor 102. In some embodiments, the processor 102 also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores 107 using known cache coherency techniques. A register file 106 is additionally included in processor 102 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 102.

[0035] In some embodiments, processor 102 is coupled to a processor bus 110 to transmit data signals between processor 102 and other components in system 100. System 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an input output (I/O) controller hub 130. Memory controller hub 116 facilitates communication between a memory device and other components of system 100, while I/O Controller Hub (ICH) 130 provides connections to I/O devices via a local I/O bus.

[0036] Memory device 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or some other memory device having suitable performance to serve as process memory. Memory 120 can store data 122 and instructions 121 for use when processor 102 executes a process. Memory controller hub 116 also couples with an optional external graphics processor 112, which may communicate with the one or more graphics processors 108 in processors 102 to perform graphics and media operations.

[0037] In some embodiments, ICH 130 enables peripherals to connect to memory 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple to ICH 130. In some embodiments, a high-performance network controller (not shown) couples to processor bus 110.

[0038] **Figure 2** is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of **Figure 2** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that

described elsewhere herein, but are not limited to such. Processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of cores 202A-N includes one or more internal cache units 204A-N. In some embodiments each core also has access to one or more shared cached units 206.

[0039] The internal cache units 204A-N and shared cache units 206 represent a cache memory hierarchy within the processor 200. The cache memory hierarchy may include at least one level of instruction and data cache within each core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units 206 and 204A-N.

[0040] In some embodiments, processor 200 may also include a set of one or more bus controller units 216 and a system agent 210. The one or more bus controller units manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent 210 provides management functionality for the various processor components. In some embodiments, system agent 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

[0041] In some embodiments, one or more of the cores 202A-N include support for simultaneous multi-threading. In such embodiment, the system agent 210 includes components for coordinating and operating cores 202A-N during multi-threaded processing. System agent 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of cores 202A-N and graphics processor 208.

[0042] In some embodiments, processor 200 additionally includes graphics processor 208 to execute graphics processing operations. In some embodiments, the graphics processor 208 couples with the set of shared cache units 206, and the system agent unit 210, including the one or more integrated memory controllers 214. In some embodiments, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. In some embodiments, display controller 211 may be separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent 210.

[0043] In some embodiments, a ring based interconnect unit 212 is used to couple the internal components of the processor 200. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some embodiments, graphics processor 208 couples with the ring interconnect 212 via an I/O link 213.

[0044] The exemplary I/O link 213 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 218, such as an eDRAM module. In some embodiments, each of the cores 202-N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

[0045] In some embodiments, cores 202A-N are homogenous cores executing the same instruction set architecture. In another embodiment, cores 202A-N are heterogeneous in terms of instruction set architecture (ISA), where one or more of cores 202A-N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set.

[0046] In some embodiments, processor 200 is a part of, or implemented on, one or more substrates using any of a number of process technologies, for example, Complementary metal-oxide-semiconductor (CMOS), Bipolar Junction/Complementary metal-oxide-semiconductor (BiCMOS) or N-type metal-oxide-semiconductor logic (NMOS). Additionally, processor 200 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

[0047] **Figure 3** is a block diagram of a graphics processor 300, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor 300 includes a memory interface 314 to access memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

[0048] In some embodiments, graphics processor 300 also includes a display controller 302 to drive display output data to a display device 320. Display controller 302 includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In some embodiments, graphics processor 300 includes a video codec engine 306 to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

[0049] In some embodiments, graphics processor 300 includes a block image transfer (BLIT) engine 304 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations

are performed using one or more components of the graphics-processing engine (GPE) 310. In some embodiments, graphics-processing engine 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0050] In some embodiments, GPE 310 includes a 3D pipeline 312 for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline 312 includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system 315. While 3D pipeline 312 can be used to perform media operations, an embodiment of GPE 310 also includes a media pipeline 316 that is specifically used to perform media operations, such as video post-processing and image enhancement.

[0051] In some embodiments, media pipeline 316 includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine 306. In some embodiments, media pipeline 316 additionally includes a thread spawning unit to spawn threads for execution on 3D/Media sub-system 315. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media sub-system 315.

[0052] In some embodiments, 3D/Media subsystem 315 includes logic for executing threads spawned by 3D pipeline 312 and media pipeline 316. In one embodiment, the pipelines send thread execution requests to 3D/Media subsystem 315, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In some embodiments, 3D/Media subsystem 315 includes one or more internal caches for thread instructions and data. In some embodiments, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

[0053] **3D/Media Processing**

[0054] **Figure 4** is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the GPE 410 is a version of the GPE 310 shown in **Figure 3**. Elements of **Figure 4** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0055] In some embodiments, GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. In some

embodiments, command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 403 receives commands from the memory and sends the commands to 3D pipeline 412 and/or media pipeline 416. The 3D and media pipelines process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to an execution unit array 414. In some embodiments, execution unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of GPE 410.

[0056] In some embodiments, a sampling engine 430 couples with memory (e.g., cache memory or system memory) and execution unit array 414. In some embodiments, sampling engine 430 provides a memory access mechanism for execution unit array 414 that allows execution array 414 to read graphics and media data from memory. In some embodiments, sampling engine 430 includes logic to perform specialized image sampling operations for media.

[0057] In some embodiments, the specialized media sampling logic in sampling engine 430 includes a de-noise/de-interlace module 432, a motion estimation module 434, and an image scaling and filtering module 436. In some embodiments, de-noise/de-interlace module 432 includes logic to perform one or more of a de-noise or a de-interlace algorithm on decoded video data. The de-interlace logic combines alternating fields of interlaced video content into a single fame of video. The de-noise logic reduces or removes data noise from video and image data. In some embodiments, the de-noise logic and de-interlace logic are motion adaptive and use spatial or temporal filtering based on the amount of motion detected in the video data. In some embodiments, the de-noise/de-interlace module 432 includes dedicated motion detection logic (e.g., within the motion estimation engine 434).

[0058] In some embodiments, motion estimation engine 434 provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In some embodiments, a graphics processor media codec uses video motion estimation engine 434 to perform operations on video at the macro-block level that may otherwise be too computationally intensive to perform with a general-purpose processor. In some embodiments, motion estimation engine 434 is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

[0059] In some embodiments, image scaling and filtering module 436 performs image-processing operations to enhance the visual quality of generated images and video. In some

embodiments, scaling and filtering module 436 processes image and video data during the sampling operation before providing the data to execution unit array 414.

[0060] In some embodiments, the GPE 410 includes a data port 444, which provides an additional mechanism for graphics subsystems to access memory. In some embodiments, data port 444 facilitates memory access for operations including render target writes, constant buffer reads, scratch memory space reads/writes, and media surface accesses. In some embodiments, data port 444 includes cache memory space to cache accesses to memory. The cache memory can be a single data cache or separated into multiple caches for the multiple subsystems that access memory via the data port (e.g., a render buffer cache, a constant buffer cache, etc.). In some embodiments, threads executing on an execution unit in execution unit array 414 communicate with the data port by exchanging messages via a data distribution interconnect that couples each of the sub-systems of GPE 410.

[0061] **Execution Units**

[0062] **Figure 5** is a block diagram of another embodiment of a graphics processor 500. Elements of **Figure 5** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0063] In some embodiments, graphics processor 500 includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 580A-N. In some embodiments, ring interconnect 502 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In some embodiments, the graphics processor is one of many processors integrated within a multi-core processing system.

[0064] In some embodiments, graphics processor 500 receives batches of commands via ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. In some embodiments, graphics processor 500 includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 580A-N. For 3D geometry processing commands, command streamer 503 supplies commands to geometry pipeline 536. For at least some media processing commands, command streamer 503 supplies the commands to a video front end 534, which couples with a media engine 537. In some embodiments, media engine 537 includes a Video Quality Engine (VQE) 530 for video and image post-processing and a multi-format encode/decode (MFX) 533 engine to provide hardware-accelerated media data encode and decode. In some embodiments, geometry pipeline 536 and media engine 537 each generate execution threads for the thread execution resources provided by at least one graphics core 580A.

[0065] In some embodiments, graphics processor 500 includes scalable thread execution resources featuring modular cores 580A-N (sometimes referred to as core slices), each having multiple sub-cores 550A-N, 560A-N (sometimes referred to as core sub-slices). In some embodiments, graphics processor 500 can have any number of graphics cores 580A through 580N. In some embodiments, graphics processor 500 includes a graphics core 580A having at least a first sub-core 550A and a second core sub-core 560A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In some embodiments, graphics processor 500 includes multiple graphics cores 580A-N, each including a set of first sub-cores 550A-N and a set of second sub-cores 560A-N. Each sub-core in the set of first sub-cores 550A-N includes at least a first set of execution units 552A-N and media/texture samplers 554A-N. Each sub-core in the set of second sub-cores 560A-N includes at least a second set of execution units 562A-N and samplers 564A-N. In some embodiments, each sub-core 550A-N, 560A-N shares a set of shared resources 570A-N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

[0066] **Figure 6** illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of **Figure 6** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0067] In some embodiments, thread execution logic 600 includes a pixel shader 602, a thread dispatcher 604, instruction cache 606, a scalable execution unit array including a plurality of execution units 608A-N, a sampler 610, a data cache 612, and a data port 614. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic 600 includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache 606, data port 614, sampler 610, and execution unit array 608A-N. In some embodiments, each execution unit (e.g. 608A) is an individual vector processor capable of executing multiple simultaneous threads and processing multiple data elements in parallel for each thread. In some embodiments, execution unit array 608A-N includes any number individual execution units.

[0068] In some embodiments, execution unit array 608A-N is primarily used to execute "shader" programs. In some embodiments, the execution units in array 608A-N execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g.,

vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders).

[0069] Each execution unit in execution unit array 608A-N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units 608A-N support integer and floating-point data types.

[0070] The execution unit instruction set includes single instruction multiple data (SIMD) instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

[0071] One or more internal instruction caches (e.g., 606) are included in the thread execution logic 600 to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 612) are included to cache thread data during thread execution. In some embodiments, sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler 610 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

[0072] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. In some embodiments, thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-N. For example, the geometry pipeline (e.g., 536 of **Figure 5**) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (**Figure 6**). In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

[0073] Once a group of geometric objects has been processed and rasterized into pixel data, pixel shader 602 is invoked to further compute output information and cause results to be written

to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, pixel shader 602 calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel shader 602 then executes an API-supplied pixel shader program. To execute the pixel shader program, pixel shader 602 dispatches threads to an execution unit (e.g., 608A) via thread dispatcher 604. In some embodiments, pixel shader 602 uses texture sampling logic in sampler 610 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

[0074] In some embodiments, the data port 614 provides a memory access mechanism for the thread execution logic 600 output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data port 614 includes or couples to one or more cache memories (e.g., data cache 612) to cache data for memory access via the data port.

[0075] **Figure 7** is a block diagram illustrating a graphics processor instruction formats 700 according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format 700 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

[0076] In some embodiments, the graphics processor execution units natively support instructions in a 128-bit format 710. A 64-bit compacted instruction format 730 is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit format 710 provides access to all instruction options, while some options and operations are restricted in the 64-bit format 730. The native instructions available in the 64-bit format 730 vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field 713. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit format 710.

[0077] For each format, instruction opcode 712 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data

channels of the operands. In some embodiments, instruction control field 712 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions 710 an exec-size field 716 limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

[0078] Some execution unit instructions have up to three operands including two source operands, src0 722, src1 722, and one destination 718. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 724), where the instruction opcode 712 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

[0079] In some embodiments, the 128-bit instruction format 710 includes access/address mode information 726 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction 710. When indirect register addressing mode is used, the register address of one or more operands may be computed based on field 726, which specifies an address register value mode and/or an address immediate field in the instruction. In one embodiment the access/address mode information 726 may determine the operand mode to define a data access alignment for the instruction. For example, when in a first mode, the instruction 710 may use byte-aligned addressing for source and destination operands and, when in a second mode, the instruction 710 may use 16-byte-aligned access mode and 1-byte aligned addressing for all source and destination operands. access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands.

[0080] In some embodiments instructions are grouped based on opcode bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb (e.g., 0x0x) and logic instructions are in the form of 0001xxxxb (e.g., 0x01). A flow control instruction group 744 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 746 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 748 includes component-wise arithmetic instructions (e.g., add,

multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 748 performs the arithmetic operations in parallel across data channels. The vector math group 750 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

[0081] **Graphics Pipeline**

[0082] **Fig 8** is a block diagram of another embodiment of a graphics processor 800. Elements of **Figure 8** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0083] In some embodiments, graphics processor 800 includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In some embodiments, graphics processor 800 is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 800 via a ring interconnect 802. In some embodiments, ring interconnect 802 couples graphics processor 800 to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect 802 are interpreted by a command streamer 803, which supplies instructions to individual components of graphics pipeline 820 or media pipeline 830.

[0084] In some embodiments, command streamer 803 directs the operation of a vertex fetcher 805 that reads vertex data from memory and executes vertex-processing commands provided by command streamer 803. In some embodiments, vertex fetcher 805 provides vertex data to a vertex shader 807, which performs coordinate space transformation and lighting operations to each vertex. In some embodiments, vertex fetcher 805 and vertex shader 807 execute vertex-processing instructions by dispatching execution threads to execution units 852A, 852B via a thread dispatcher 831.

[0085] In some embodiments, execution units 852A, 852B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units 852A, 852B have an attached L1 cache 851 that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

[0086] In some embodiments, graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain

shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of hull shader 811 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components 811, 813, 817 can be bypassed.

[0087] In some embodiments, complete geometric objects can be processed by a geometry shader 819 via one or more threads dispatched to execution units 852A, 852B, or can proceed directly to the clipper 829. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader 819 receives input from the vertex shader 807. In some embodiments, geometry shader 819 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

[0088] Before rasterization, a clipper 829 processes vertex data. The clipper 829 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 850. In some embodiments, an application can bypass the rasterizer 873 and access un-rasterized vertex data via a stream out unit 823.

[0089] The graphics processor 800 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 852A, 852B and associated cache(s) 851, texture and media sampler 854, and texture/sampler cache 858 interconnect via a data port 856 to perform memory access and communicate with render output pipeline components of the processor. In some embodiments, sampler 854, caches 851, 858 and execution units 852A, 852B each have separate memory access paths.

[0090] In some embodiments, render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. Associated render and depth buffer caches 878, 879 are also available in some embodiments. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 841, or substituted at display time by the display controller 843 using overlay display planes. In some embodiments, a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

[0091] In some embodiments, graphics processor media pipeline 830 includes a media engine 837 and a video front end 834. In some embodiments, video front end 834 receives pipeline commands from the command streamer 803. In some embodiments, media pipeline 830 includes a separate command streamer. In some embodiments, video front-end 834 processes media commands before sending the command to the media engine 837. In some embodiments, media engine 337 includes thread spawning functionality to spawn threads for dispatch to thread execution logic 850 via thread dispatcher 831.

[0092] In some embodiments, graphics processor 800 includes a display engine 840. In some embodiments, display engine 840 is external to processor 800 and couples with the graphics processor via the ring interconnect 802, or some other interconnect bus or fabric. In some embodiments, display engine 840 includes a 2D engine 841 and a display controller 843. In some embodiments, display engine 840 contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller 843 couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

[0093] In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) from the Khronos Group, the Direct3D library from the Microsoft Corporation, or support may be provided to both OpenGL and D3D. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

[0094] **Graphics Pipeline Programming**

[0095] **Fig 9A** is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Fig 9B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in **Figure 9A** illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 900 of **Figure 9A** includes data fields to identify a target client 902 of the command, a command operation code (opcode) 904,

and the relevant data 906 for the command. A sub-opcode 905 and a command size 908 are also included in some commands.

[0096] In some embodiments, client 902 specifies the client unit of the graphics device that processes the command data. In some embodiments, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In some embodiments, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode 904 and, if present, sub-opcode 905 to determine the operation to perform. The client unit performs the command using information in data field 906. For some commands an explicit command size 908 is expected to specify the size of the command. In some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

[0097] The flow diagram in **Figure 9B** shows an exemplary command sequence 910. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

[0098] In some embodiments, sample command sequence 910 may begin with a pipeline flush command 912 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline 922 and the media pipeline 924 do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command 912 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

[0099] In some embodiments, a pipeline select command 913 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some

embodiments, a pipeline select command 913 is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command is 912 is required immediately before a pipeline switch via the pipeline select command 913.

[00100] In some embodiments, a pipeline control command 914 configures a graphics pipeline for operation and is used to program the 3D pipeline 922 and the media pipeline 924. In some embodiments, pipeline control command 914 configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command 914 is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

[00101] In some embodiments, return buffer state commands 916 are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. In some embodiments, the graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. In some embodiments, the return buffer state 916 includes selecting the size and number of return buffers to use for a set of pipeline operations.

[00102] The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination 920, the command sequence is tailored to the 3D pipeline 922 beginning with the 3D pipeline state 930, or the media pipeline 924 beginning at the media pipeline state 940.

[00103] The commands for the 3D pipeline state 930 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based the particular 3D API in use. In some embodiments, 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[00104] In some embodiments, 3D primitive 932 command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive 932 command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive 932 command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive 932 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline 922 dispatches shader execution threads to graphics processor execution units.

[00105] In some embodiments, 3D pipeline 922 is triggered via an execute 934 command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

[00106] In some embodiments, sample command sequence 910 follows the media pipeline 924 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 924 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

[00107] In some embodiments, media pipeline 924 is configured in a similar manner as the 3D pipeline 922. A set of media pipeline state commands 940 are dispatched or placed into in a command queue before the media object commands 942. In some embodiments, media pipeline state commands 940 include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands 940 also support the use one or more pointers to "indirect" state elements that contain a batch of state settings.

[00108] In some embodiments, media object commands 942 supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command 942. Once the pipeline state is configured and media object commands 942 are queued, the media pipeline 924 is triggered via an execute command 944 or an equivalent execute event (e.g., register write). Output from media pipeline 924 may then be post processed by operations provided by the 3D pipeline 922 or the media pipeline 924. In

some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

[00109] **Graphics Software Architecture**

[00110] **Figure 10** illustrates exemplary graphics software architecture 1000 for a data processing system according to some embodiments. In some embodiments, software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. In some embodiments, processor 1030 includes a graphics processor 1032 and one or more general-purpose processor core(s) 1034. The graphics application 1010 and operating system 1020 each execute in the system memory 1050 of the data processing system.

[00111] In some embodiments, 3D graphics application 1010 contains one or more shader programs including shader instructions 1012. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions 1014 in a machine language suitable for execution by the general-purpose processor core 1034. The application also includes graphics objects 1016 defined by vertex data.

[00112] In some embodiments, operating system 1020 is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the operating system 1020 uses a front-end shader compiler 1024 to compile any shader instructions 1012 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

In some embodiments, user mode graphics driver 1026 contains a back-end shader compiler 1027 to convert the shader instructions 1012 into a hardware specific representation. When the OpenGL API is in use, shader instructions 1012 in the GLSL high-level language are passed to a user mode graphics driver 1026 for compilation. In some embodiments, user mode graphics driver 1026 uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. In some embodiments, kernel mode graphics driver 1029 communicates with graphics processor 1032 to dispatch commands and instructions.

APPARATUS AND METHOD FOR SIMD STRUCTURED BRANCHING

[00113] One embodiment of the invention performs SIMD control flow in hardware, thereby increasing performance and reducing power. In particular, a SIMD processor functional unit is described that can be used in SIMD threads where channels diverge and re-converge using

structured flow. One example of channel divergence is an IF/ELSE/ENDIF sequence where, if the "IF" condition is met, the instruction stream diverges to a first instruction sequence, but if the "IF" condition is not met, the instruction stream diverges to a second instruction sequence (specified by the "ELSE" condition). Channel convergence occurs in response to the ENDIF statement (e.g., when all divergent instructions required by the "IF" and/or "ELSE" statements have been completed).

[00114] In one embodiment of the invention, channel divergence and convergence are detected by a new branch unit (e.g., via new instructions) which responsively enables and disables each channel. The branch unit may be shared across all threads in an execution unit (EU) and may compute the result of all structured control flow instructions for all channels in the processor.

[00115] One embodiment of the branch unit computes a channel mask, one bit per channel, to enable or disable channels during structured control flow processing. This removes the necessity to compare channel IPs for every instruction. In one embodiment, the channel mask is used as-is for all non-control flow instructions, which are processed by another unit.

[00116] In one embodiment, the channel enable mask is generated from a counter per channel which indicates if the channel is disabled, and if so, by what control flow instruction (i.e., IF/ELSE/ENDIF, CALL/RETURN, etc). This saves gates when compared with existing per-channel IP solution and saves compute cycles per instruction. The embodiments of the invention may be implemented with any structured control flow instructions supported by a standard (e.g., c/c++). This includes, by way of example and not limitation: IF/ELSE/ENDIF, DO/BREAK/CONTINUE/WHILE, CALL/RETURN, HALT, BRANCH, and may work with any other structured control flow instructions.

[00117] Recursion is also supported in one embodiment using a counter that differentiates occurrences of control flow instructions. In particular, in one embodiment, a "Control Instruction Count" is maintained for each branching type (IF/ELSE/ENDIF, DO/BREAK/CONTINUE/WHILE, CALL/RETURN, etc), which provides a count of the current level of nesting of that branching type. For example, a value of 2 for an IF branching type means two nested IF statements. In one embodiment, the Control Instruction Count is zero if the current IP is outside of any branching conditional blocks.

[00118] One embodiment also maintains a "Per Channel Count" to determine if a channel is enabled. If the Per Channel Count is 0, the channel is enabled. In one embodiment, when a channel is disabled, its corresponding count contains the value of the Control Instruction Count at which the channel was disabled along with an encoded ID representing the branching type. (e.g., IF, 2 for an IF branching type with 2 nested IF statements).

[00119] In one embodiment, an n-channel entry stack is used to store the address (IP) of converging instructions (e.g., for an IF instruction, the IP of the ENDIF instruction is stored). A Top of Stack pointer is maintained for each branching type to identify the IP for the converging instruction within the stack.

[00120] In addition, in one embodiment, each instruction contains a predicate mask. Each bit of the predicate mask represents one channel, and is used to determine if the channel takes the conditional or not. (e.g., for IF, a channel can either enter the IF block, or jump to the ELSE block). In one embodiment, only enabled channels are evaluated for execution.

[00121] **Figure 11** illustrates an architecture for an execution unit 1100 in which a branch unit 1101 is implemented in accordance with one embodiment of the invention. A plurality of channels 0-2 execute instructions stored in an instruction memory 1105 which may be an instruction buffer, instruction cache or any other memory suitable for storing instructions prior to execution. While 3 channels are illustrated in **Figure 11**, the underlying principles of the invention may be implemented with any number of channels. An instruction pointer 1110-1112 associated with each channel 0-2, respectively, identifies the next instruction to be executed over its respective channel. Instruction dispatch logic 1120-1122 then dispatches the instructions for execution on a plurality of functional units 1130, which may include, for example, vector functional units, scalar functional units, and logical functional units (and/or any other type of functional unit designed to execute the instructions).

[00122] The execution unit 1100 shown in **Figure 11** may be implemented within a graphics processing unit architecture such as described above (see, e.g., execution units 608 shown in **Figure 6**). However, the underlying principles of the invention are not limited to a graphics processing implementation.

[00123] As mentioned above, channel divergence and convergence are detected by the branch unit 1101 which responsively enables and disables each channel. The branch unit 1101 may be shared across all threads in the EU 1100 and may compute the result of all structured control flow instructions for all channels in the processor. In one embodiment, each thread is executed on a different channel 0-2.

[00124] **Figure 12** illustrates additional details associated with one embodiment of the branch unit 1101. As illustrated, this embodiment of the branch unit 1101 computes a channel enable mask 1250, one bit per channel, to enable or disable channels during structured control flow processing (thereby removing the necessity to compare channel IPs for every instruction). In one embodiment, the channel mask 1250 is used as-is for all non-control flow instructions, which are processed by another unit.

[00125] In one embodiment, the channel enable mask 1250 is generated from a set of per-channel counters 1200-1202 which indicate whether the channel is disabled, and if so, by what control flow instruction (i.e., IF/ELSE/ENDIF, CALL/RETURN, etc). A set of control instruction counters 1211 differentiates occurrences of control flow instructions with a "control instruction count" maintained for each branching type (IF/ELSE/ENDIF, DO/BREAK/ CONTINUE/WHILE, CALL/RETURN, etc). In one embodiment, the control instruction count indicates the current level of nesting of that branching type. In the example shown in **Figure 12**, a value of 2 is shown for an IF branching type, meaning two nested IF statements. Values of 1 are shown for CALL/RETURN and DO/BREAK/CONTINUE/WHILE instructions indicating 1 level of nesting. In one embodiment, the control instruction count is zero if the current IP is outside of any branching conditional blocks.

[00126] In one embodiment, if the per channel count is 0 in a per-channel counter 1200-1202, then the corresponding channel is enabled, as indicated by a 0 in the channel mask 1250. In one embodiment, when a channel is disabled, its corresponding count in its per-channel counter 1200-1202 comprises the control instruction count at which the channel was disabled along with the encoded ID representing the branching type (provided by the control instruction counters 1211).

[00127] **Figure 12** also illustrates an n-channel entry stack 1260 used to store the address (IP) of converging instructions (e.g., for an IF instruction, the IP of the ENDIF instruction is stored). Top of stack pointers 1261 are maintained for each branching type to identify the IP 1250-1254 for the converging instruction within the stack 1260. In one embodiment, the IP of the converging instruction is added to the stack when executing the corresponding diverging instruction.

[00128] As mentioned, in one embodiment, each instruction contains a predicate mask where each bit represents one channel, and is used to determine if the channel takes the conditional or not (e.g., for IF, a channel can either enter the IF block, or jump to the ELSE block). In one embodiment, only enabled channels are evaluated for execution.

[00129] A method for executing diverging instructions is set forth in **Figure 13**. The method may be implemented within the context of the system shown in **Figures 11-12** but is not limited to any particular system architecture.

[00130] Starting at 1301, for each channel *n*, a determination is made at 1302 as to whether the per channel count is 0. As mentioned, in one embodiment, a channel count of 0 indicates that the corresponding channel is enabled, determined at 1303. If the per channel count is non-zero, the corresponding channel is not enabled, determined at 1304. The above process is performed for each channel *n*.

[00131] At 1305 a tag for the converging IP is set equal to the current IP + the converging offset (i.e., the distance from the current IP to the converging IP). At 1306, a determination is made as to whether all enabled channels have the same predicate value associated therewith (e.g., whether the predicate mask of the instructions are the same). If not, then at 1307, the control instruction count is incremented for the current control flow instruction type (e.g., the IF count is incremented within the corresponding control instruction counter 1211).

[00132] At 1308, for each channel *n*, a determination is made as to whether the channel is enabled at 1309. If so, then at 1310 a determination is made as to whether the channel is enabled for the current instruction based on the predicate enable field. If not, then the per channel count is set equal to the control instruction count for the control flow instruction at 1311 (thereby disabling the channel). The above process is repeated for each channel *n*. At 1312, the next instruction pointer is incremented (e.g., the next instruction pointer is set equal to the current instruction pointer + 1).

[00133] Turning back to decision block 1306, if all enabled channels have the same predicate value associated therewith (i.e., the predicate mask of the instructions are the same), then at 1313, a determination is made as to whether all active channels take the current condition (e.g., whether all channels can process the control flow instruction). If so, then at 1315 the next instruction pointer is set equal to the current instruction pointer added to the jump offset (i.e., the IP location of the diverging instruction). If not, then at 1314 the next instruction pointer is set equal to the current instruction pointer added to the converging offset (i.e., the location of the converging instruction).

[00134] As a result of the foregoing method, if the predicate is uniform for all channels, then the per channel count is not updated, and only the Execution IP is updated. If the predicate mask is not uniform for all channels, this is called divergence and, in this case, some channels take the condition and some do not. In this instance, the control instruction count is incremented, and the per channel count of each channel which does not take the condition is updated with the control instruction count of the current branching type. The Tag is pushed onto the stack (to store the IP of the converging instruction), and the top of stack pointer is updated.

[00135] **Figure 14** illustrates one embodiment of a method for processing converging control flow instructions. The method may be implemented within the context of the processor architecture shown in **Figures 11-12** but is not limited to any particular architecture.

[00136] At 1401, a tag is set equal to the current instruction pointer. At 1402, if the tag is equal to the instruction pointer at the top of the stack (e.g., stack 1260), then the process moves to 1403. For each channel *n*, if the per channel count is equal to the control instruction count, determined at 1404, then the per channel count is set equal to 0 at 1405 (enabling the channel).

[00137] Once the process completes for each channel, at 1406 a determination is made as to whether any channel count is equal to the control instruction count. If not, then at 1407, the control instruction count is decremented and the instruction pointer is popped from the stack. At 1408 a determination is made as to whether any channel is enabled. If so, then at 1409 the instruction pointer is incremented (i.e., the next instruction pointer is set to the current instruction pointer + 1). If not, then at 1410, the next instruction pointer is set equal to the current instruction pointer added to the jump offset (i.e., to arrive at the IP for the divergent control flow instruction).

[00138] As a result of the foregoing method, the current IP is compared to the Top of Stack IP. If those values match, then each per channel count is compared to the branching type's control instruction count, for channels with an enabled predicate mask bit. If these values match, the per channel count is set to 0 (indicating an active channel). If all channels are enabled corresponding to a particular control instruction count, then the count is decremented and the stack is popped. The IP is incremented and the program continues as normal.

APPARATUS AND METHOD FOR UNSTRUCTURED CONTROL FLOW FOR A SIMD EXECUTION ENGINE

[00139] As mentioned above, unstructured control flow is not currently supported in hardware for SIMD programs. Modern compilers produce unstructured SIMD program code which is traditionally converted back to structured program code by the compiler, and some optimizations are lost due to converting back to structured flow.

[00140] To address this limitation, one embodiment of the invention supports unstructured control flow instructions for SIMD programs in hardware, thereby maintaining the compiler optimizations which improves performance and reduces power. In particular, a branch unit may be equipped with logic to implement both the structured branching embodiments (described above) and unstructured branching embodiments (described below). It should be noted, however, that the structured branching techniques described above are not required for implementing the unstructured branching techniques described below (i.e., the unstructured branching techniques may be implemented in a branch unit with or without the structured branching techniques).

[00141] As illustrated in **Figure 15**, in one embodiment, the branch unit 1501 includes unstructured control flow logic 1500 to support new unstructured control flow instructions. The unstructured control flow logic 1500 may be used in SIMD threads where channels diverge and re-converge using unstructured control flow. Channel divergence and convergence is may be

detected from new instructions and the branch unit 1501 manages enables for each channel as described below.

[00142] As in the structured control flow embodiments, a plurality of channels 0-2 execute instructions stored in an instruction memory 1105 which may be an instruction buffer, instruction cache or any other memory suitable for storing instructions prior to execution. While three channels are illustrated in **Figure 15**, the underlying principles of the invention may be implemented with any number of channels. An instruction pointer 1110-1112 associated with each channel 0-2, respectively, identifies the next instruction to be executed over its respective channel. Instruction dispatch logic 1120-1122 then dispatches the instructions for execution on a plurality of functional units 1130, which may include, for example, vector functional units, scalar functional units, and logical functional units (and/or any other type of functional unit designed to execute the instructions). The execution unit 1100 shown in **Figure 15** may be implemented within a graphics processing unit architecture such as described above (see, e.g., execution units 608 shown in **Figure 6**). However, the underlying principles of the invention are not limited to a graphics processing implementation.

[00143] As mentioned above, channel divergence and convergence are detected by the branch unit 1101 which responsively enables and disables each channel. The branch unit 1101 may be shared across all threads in the EU 1100 and may compute the result of all unstructured control flow instructions for all channels in the processor. In one embodiment, each thread is executed on a different channel 0-2.

[00144] **Figure 16** provides additional details for implementing unstructured control flow including a channel mask 1250 to enable and disable channels. The channel mask 1250 may be computed for both structured and unstructured control flow instructions by the branch unit 1501 and may be used as-is for all non-control flow instructions, which are processed by another unit. In one embodiment, two new instructions are added for unstructured control flow, GOTO and JOIN, and a new counter 1605 is added to process these new instructions. In one embodiment, GOTO causes a jump to a specified location in the instruction stream and JOIN returns to a specified location in the instruction stream (e.g., both specified via an address/instruction pointer).

[00145] In the embodiments described above related to SIMD structured branching, the branch unit 1501 has defined a Control Instruction Count per branching type (e.g., IF/THEN/ENDIF, BREAK/WHILE/CONTINUE, CALL/RETURN, etc), which is a count of the current level of nesting of that branching type. In one embodiment, this mechanism is modified to handle unstructured instructions. Instead of using a count as the unique identifier, this embodiment uses the address of the IP tag in the local memory. While the unstructured

instructions used a stack structure 1260 shown in **Figure 12** (top of stack being the last address pushed onto the stack), the new unstructured instructions use a sorted linked-list structure 1660 to store IP tags 1650-1654 in the same memory space as shown in **Figure 16**. One difference with the stack structure 1260 is that in the sorted linked-list structure 1660 the unstructured instruction IP tags 1650-1654 are sorted by hardware (while in the stack structure the structured IPs are sorted by definition).

[00146] The existing Per Channel Count maintained in per channel counters 1200-1202 is used to determine if a channel is enabled. If the Per Channel Count is 0, the channel is enabled. In one embodiment, when a channel is disabled by an unstructured instruction its corresponding count is updated to contain the address of the IP tag 1650-1654 at which the channel was disabled along with an encoded ID representing the branching type (e.g., "GOTO2" to identify the GOTO instruction and IP tag at entry # 2 in the linked list memory 1660 structure 1660).

[00147] As in the prior embodiments, each instruction may include a predicate mask. Each bit in the predicate mask represents one channel, and is used by the unstructured control flow logic 1500 of the branch unit 1501 to determine if the channel takes the conditional or not. Only enabled channels are evaluated for execution.

[00148] One embodiment of a method for executing unstructured diverging instructions is set forth in **Figure 17**. The method may be implemented within the context of the system shown in **Figures 15-16** but is not limited to any particular system architecture.

[00149] Starting at 1701, for each channel *n*, a determination is made at 1702 as to whether the per channel count is 0. As mentioned, in one embodiment, a channel count of 0 indicates that the corresponding channel is enabled, determined at 1703. If the per channel count is non-zero, the corresponding channel is not enabled, determined at 1704. The above process is performed for each channel *n*.

[00150] At 1705 a tag for the converging IP is set equal to the current IP + the converging offset (i.e., the distance from the current IP to the IP of the converging instruction). At 1706, a determination is made as to whether all enabled channels have the same predicate value associated therewith (e.g., whether the predicate mask of the instructions are the same). If not, then at 1707, the sorted linked list memory structure (e.g., 1660 in **Figure 16**) is searched to determine whether the IP tag is found in the memory. If not, then at 1708, the tag is inserted into the memory and the tag address is saved (e.g., into a temporary storage such as counter 1606 accessible by the unstructured control flow logic 1500). If the IP tag is found in the memory, then at 1709 the tag address is saved.

[00151] Starting at 1710, for each channel *n*, a determination is made as to whether the channel is enabled at 1711. If so, then at 1712 a determination is made as to whether the channel

is enabled for the current instruction based on the predicate enable field of the instruction. If not, then the per channel count is set equal to the tag address (identifying the IP tag within the memory 1660). In one embodiment, the tag address is stored in the corresponding Per Channel Counter 1200-1202 for the current channel n (thereby disabling the channel). The above process is repeated for each channel *n*. At 1714, the next instruction pointer is set by adding the jump offset to the current instruction pointer (i.e., to arrive at the IP address of the diverging instruction).

[00152] Turning back to decision block 1706, if all enabled channels have the same predicate value associated therewith (i.e., the predicate mask of the instructions are the same), then at 1715, a determination is made as to whether all active channels take the current condition (e.g., whether all channels can process the control flow instruction). If so, then at 1717 the next instruction pointer is set equal to the current instruction pointer added to the jump offset (i.e., the IP location of the diverging instruction). If not, then at 1716 the instruction pointer is incremented, i.e., the next instruction pointer is set equal to the current instruction pointer + 1 (i.e., the diverging instruction is not taken).

[00153] As a result of the foregoing method, if the predicate is uniform for all channels, then the per channel count is not updated, and only the Execution IP is updated. If the predicate mask is not uniform for all channels, this is called divergence and, in this case, some channels take the condition and some do not. In this instance, the control instruction count is incremented, and the per channel count of each channel which does not take the condition is updated with the memory address of the converging IP. The IP Tag 1650-1654 is inserted into memory 1660.

[00154] **Figure 18** illustrates one embodiment of a method for processing unstructured converging instructions. The method may be implemented within the context of the processor architecture shown in **Figures 15-16** but is not limited to any particular architecture.

[00155] At 1801, the tag is set equal to the current instruction pointer. At 1802, the memory containing IP tags (e.g., sorted linked-list memory 1660) is searched using the tag. If a match is found, then the tag address is saved at 1803 (e.g., into a temporary storage such as a counter 1606 accessible by the unstructured control flow logic). At 1804, for each channel *n*, if the predicate enable bit is set (determined at 1805) and the per channel count is equal to the tag address (determined at 1806), then the per channel count is set equal to 0 at 1807 (thereby enabling the channel).

[00156] Once the process completes for each channel, at 1808 a determination is made as to whether any channel count is equal to the tag address. If not, then at 1809, the IP tag is deleted from memory. At 1810 a determination is made as to whether any channel is enabled. If so, then at 1811 the instruction pointer is incremented (i.e., the next instruction pointer is set to

the current instruction pointer + 1). If not, then at 1812 the next instruction pointer is set equal to the current instruction pointer added to the jump offset (i.e., to arrive at the IP for the next unstructured instruction).

[00157] As a result of the foregoing method, the current IP is searched for in memory. If it is found, then each Per Channel Count is compared to the memory address for channels with an enabled predicate mask bit. If these values match, the Per Channel Count is set to 0 (indicating an active channel). If all channels are enabled corresponding to a particular Control Instruction Count, then the Tag is removed from memory.

EFFICIENT SUPPORT FOR NESTED BRANCHES ON A GPU ARCHITECTURE

[00158] Some single instruction multiple data (SIMD) architectures includes special IF, ELSE, and ENDIF control flow instructions to support SIMD branching. One such example is Intel's integrated graphics (GEN) instruction set architecture (ISA). In an IF/ELSE/ENDIF block, only channels that are enabled via predicate control execute the THEN block, while the disabled channels execute the ELSE block. Control flow re-converges at the ENDIF instruction, when the execution mask, which controls the active channels, is restored to be the same as before the IF instruction. Hardware may implement this by maintaining a global counter of the active divergent branches together with a set of per-channel counters that mark the first branch that disables this channel. For example, **Figure 12** (described above) illustrates a set of per-channel counters 1200-1202 and a global counter 1211 (referred to as a control instruction counter). The global counter 1211 is incremented at each divergent if instruction and decremented at its matching ENDIF. In one embodiment, each per-channel counter 1200-1202 is reset to zero if it is equal to the global counter to indicate re-convergence. The execution mask 1250 is formed by comparing each per-channel counter against the global counter.

[00159] This requires the number of IF and ENDIF instructions to be identical in a kernel, and thus means that for nested IF/ENDIF blocks that converge at the same program point, there are multiple redundant ENDIF instructions at the nested IF/ENDIF exit. The embodiments of the invention described below include a mechanism that allows a single ENDIF instruction to be generated in these situations, thereby both improving performance and reducing power.

[00160] One embodiment of the invention detects the presence of multiple consecutive control flow instructions such as ENDIFs in a kernel and fuses them into a single control flow instruction (e.g., a single ENDIF). A hardware stack may be added in one embodiment to keep track of the number of divergent IFs that map to an ENDIF. When hardware sees a (possibly

fused) ENDIF, it may correctly restore the execution mask to that before the outermost IF for this ENDIF.

[00161] The embodiments of the invention described herein improve upon the known solutions by fusing back-to-back control flow instructions (e.g., ENDIFs) together for arbitrary level nests, thus avoiding the overhead of executing consecutive control flow instructions such as ENDIFs. This results in a 4% performance improvement for a 16 level deep IF-ENDIF nest in an OpenCL benchmark. These embodiments require modest hardware and software changes and can potentially benefit any GPU applications with divergent control flow.

[00162] While some of the embodiments of the invention described below are specifically directed to IF/ENDIF control flow instructions, the underlying principles of the invention may be implemented with other forms of control flow instructions.

[00163] **Figure 19** illustrates exemplary program code containing three sets of IF/ENDIF instructions. The parenthesized bits before the IF instruction are the predicate values, and each IF matches with its corresponding ENDIF via a unique label (L1, L2, L3 in the illustrated example). For the purpose of this demonstration the number of SIMD channels is set to 4, but the underlying principles of the invention can be applied to IF/ELSE/ENDIF blocks with arbitrary SIMD widths and nest levels.

[00164] In the illustrated embodiment, the global counter 1910 records the number of active divergent branches at the current program point. Thus, the global counter starts at 0, increments to 1 in response to IF L1, increments to 2 in response to IF L2 and increments to 3 in response to IF L3, as illustrated. It then decrements from 3 to 2 in response to ENDIF L3, decrements to 1 in response to ENDIF L2, and finally decrements back to 0 in response to the final ENDIF L1.

[00165] As illustrated, each channel also has its own counter to track the divergent branch that first disables the channel. Because the illustrated example uses four SIMD channels, four per-channel counters 1900-1904 are illustrated in **Figure 19**. The counters are incremented to indicate the current nesting level associated with each IF/ELSE instruction which may be determined from the global counter (i.e., in response to an IF/ELSE, the relevant per-channel counter may be set equal to the global counter as illustrated). In response to and ENDIF, each per-channel counter storing the value of the corresponding IF/ELSE may be reset to 0. For example, in Figure 19, per-channel counter 1900 is set equal to the global counter value of 1 in response to IF L1; per-channel counter 1901 is set equal to the global counter value of 2 in response to IF L2; and per-channel counter 1902 is set equal to the global counter value of 3 in response to IF L3. In response to the ELSE instruction (i.e., if the IF condition is not true), per-channel counter 1903 is set equal to the global counter value of 3. Then, in response to the first

ENDIF (L3), the global counter value is 2 so any per-channel counter value greater than 3 (e.g., counter 1903 or 1902) is reset to 0. On the next ENDIF (L2), any per-channel counter value greater than the global counter value of 1 is reset to 0 (i.e., counter 1901). For the final ENDIF (L1), the global counter value is 1, so all per-channel counters are reset to 0. As mentioned above, the channel enable mask 1950 is set based on the per-channel counter values. Any per-channel counter with a value greater than 0 sets its corresponding channel enable mask bit to 0 (disabled). Otherwise, the corresponding channel enable mask bit is set to 1 (enabled).

[00166] In one embodiment, the counters are updated according to the following rules:

global_counter = 0, counter[I] = 0 for all channels at program start

On divergent IF:

```
global_counter++;
If channel I is disabled by IF
        counter[I] = global_counter
```

On divergent ELSE:

```
If counter[I] == 0
        counter[I] = global_counter
Else if counter[I] == global_counter
        counter[I] = 0
```

On divergent ENDIF:

```
global_counter--;
If counter[I] > global_counter
        counter[I] = 0
```

[00167] Note that the counters are not updated at a uniform branch, since either the THEN block or the ELSE block will be skipped and the execution mask does not need to be restored at the ENDIF. The execution mask is formed by

[00168] Emask[I] = counter[I} == 0

[00169] One embodiment of the invention further improves upon the above solution by combining any back-to-back control flow instructions (e.g., ENDIFs). In particular, as illustrated in **Figure 20**, one embodiment includes control flow fusing logic 2005 which detects multiple consecutive control flow instructions such as ENDIFs (L2 and L3 in the above example) within program code 2001, and generates program code 2010 in which the consecutive control flow instructions are fused into a single control flow instruction. In one embodiment, the labels of the divergent control flow instructions (e.g., IFs) corresponding to the converging control flow instructions (e.g., ENDIFs) are also updated to all point to the fused instruction as described below (see, e.g., **Figure 22** and associated text).

[00170] The other components shown in **Figure 20** may operate substantially as described above (aside from the branch unit 2001 which may include additional logic for processing the fused control flow instructions as described herein). In particular, **Figure 20** illustrates an architecture for an execution unit 2000 in which a branch unit 2001 is implemented in accordance with one embodiment of the invention. Four channels 0-3 execute instructions stored in an instruction memory 2005 which may be an instruction buffer, instruction cache or any other memory suitable for storing instructions prior to execution. While four channels are illustrated in **Figure 20**, the underlying principles of the invention may be implemented with any number of channels. An instruction pointer 2010-2013 associated with each channel 0-3, respectively, identifies the next instruction to be executed over its respective channel. Instruction dispatch logic 2020-2023 then dispatches the instructions for execution on a plurality of functional units 2030, which may include, for example, vector functional units, scalar functional units, and logical functional units (and/or any other type of functional unit designed to execute the instructions).

[00171] The execution unit 2000 shown in **Figure 20** may be implemented within a graphics processing unit architecture such as described above (see, e.g., execution units 608 shown in **Figure 6**). However, the underlying principles of the invention are not limited to a graphics processing implementation.

[00172] As mentioned above, channel divergence and convergence are detected by the branch unit 2001 which responsively enables and disables each channel as described herein. The branch unit 2001 may be shared across all threads in the EU 2000 and may compute the result of all control flow instructions for all channels in the processor. In one embodiment, each thread is executed on a different channel 0-3.

[00173] **Figure 21** illustrates additional details of one embodiment of the branch unit 2101 including the set of per-channel counters 1900-1903 maintaining a count representing the current level of IF/ELSE/ENDIF nesting and responsively updating the channel enable mask 1950. As mentioned, each bit in the channel enable mask indicates whether its corresponding channel is enabled (1) or disabled (0). If the value in a per-channel counter is 0, then the corresponding channel enable mask bit is set to 1 (to indicate the channel is enabled). If the value in the per-channel counter is greater than 0, then the corresponding channel enable mask bit is set to 0 (to indicate that the channel is disabled). As discussed above, the per-channel counters 1900-1903 may be modified based on comparisons against the current value in the global counter 1910. In one embodiment, fused control flow processing logic 2100 processes the program code with fused control flow instructions 2010 (e.g., ENDIFs) to update the per-channel counters 1900-1903 and global counter 1910 as described herein. In hardware, a stack 2101 is added to record

the control flow instructions of active divergent branches, and each control flow instruction on the stack additionally has a counter to store the number of diverging control flow instructions (e.g., IFs) that match the converging control flow instructions (e.g., ENDIFs).

[00174] **Figure 22** provides an example of how fused ENDIFs may be processed in accordance with one embodiment of the invention. In this example, the ENDIF L3 and ENDIF L2 from **Figure 19** have been fused into a single ENDIF L2. As in **Figure 19**, the first IF L1 increments the global counter 1910 to 1 and sets the per-channel counter 1900 to 1 for the associated channel. The enable mask 1950 bit for this channel is then set to 0 (indicating that the channel is disabled). The next IF L2 increments the global counter 1910 to 2 and sets a value of 2 in the corresponding per-channel counter 1901, thereby causing the enable mask 1950 bit for this channel to be set to 0. The third IF statement also has the L2 label since the L2 and L3 ENDIFs were fused. It sets the global counter 1910 and the corresponding per-channel counter 1902 to 3, thereby disabling the channel. Alternatively, the ELSE instruction may set per-channel counter 1903 and global counter to 3, as illustrated.

[00175] In contrast to the embodiment shown in **Figure 19**, in **Figure 22**, a single ENDIF L2 decrements the global counter by 2 (from 3 to 1) and, in response, all values in the per-channel counters greater than 1 are reset to 0 (all but per-channel counter 1900 which remains at a value of 1 in the example). Thus, a single ENDIF instruction provides the same end result as two separate ENDIF instructions in the prior embodiments. In one embodiment, the stack 2101 is consulted to determine the number of divergent IFs mapped to each ENDIF. Because the ENDIF L2 (identified in the stack at line # 25) has two divergent IFs mapped to it (as indicated by 25:**2** in the stack), the global counter is decremented by 2 (i.e., from 3 to 1). By contrast, the second ENDIF L1 is associated in the stack 2101 with a single IF (as indicated by 30:1). As such the global counter is decremented from 1 to 0 and the associated per-channel counter 1900 is reset to 0, as illustrated.

[00176] In one embodiment, the below rules are used to compute the counters, with pertinent updates bolded to contrast with the prior embodiments:

```
global_counter = 0, counter[I] = 0, endif_stack = empty at program start
On divergent IF:
        global_counter++;
        If channel I is disabled by IF
                counter[I] = global_counter
        If label == endif_stack.top().label
                endif_stack.top().counter++
        else
```

endif_stack.push(endif_label, 1)

On divergent ELSE:

If counter[I] == 0

counter[I] = global_counter

Else if counter[I] == global_counter

counter[I] = 0

On divergent ENDIF:

global_counter -= endif_stack.top().counter

endif_stack.pop()

If counter[I] > global_counter

counter[I] = 0

[00177] Thus, the embodiments of the invention ensure that on an ENDIF, the global counter 1910 is decremented by the number of IFs that map to it. While the example shown in **Figure 22** uses two fused ENDIFs for the purpose of explanation, the underlying principles of the invention may be implemented with a larger number of fused ENDIFs. To reduce the number of bits needed for the stack 2101 counter, ENDIF fusing may be disabled when the number of consecutive ENDIFs exceeds a predetermined threshold. For example, the stack counter may be limited to 3 bits and the number of fused ENDIFs may be set to 8. In one embodiment, the stack 2101 depth is bounded by the maximum number of divergent branches and must therefore be at most the SIMD width, since at least one channel must be disabled for each such IF.

[00178] A method in accordance with one embodiment of the invention is illustrated in **Figure 23**. The method may be implemented within the context of the architectures described above, but is not limited to any particular architecture.

[00179] At 2301, N consecutive control flow instructions (e.g., ENDIFs) are detected within program code and, in response, the control flow instructions are fused to generate program code containing the fused control flow instructions. In one particular embodiment, the fused control flow instructions are converging control flow instructions such as ENDIFs. As mentioned, the operation of fusing the control flow instructions may be performed by software (e.g., a compiler or kernel). However, the underlying principles of the invention are not limited to a software-based implementation.

[00180] At 2302, the program code is executed and a counter value of N is stored in a stack and associated with the fused control flow instruction. As mentioned, the control flow instruction may be identified with a label (e.g., a line #). At 2303, the fused control flow instruction is executed and, in response, the stack is consulted to determine the value of N. The

global counter is then decremented by N. At 2304, the per channel counters are updated in accordance with the new global counter value (e.g., per-channel counters greater than the global counter may be set to 0 as described above). In addition, the channel enable mask may be updated in accordance with the new per-channel counter values as described herein (e.g., enabling channels with a per-channel counter value of 0).

[00181] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[00182] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals, etc.). In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that

the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

CLAIMS

What is claimed is:

1. A processor comprising:

an execution unit having a plurality of channels to execute control flow instructions including fused control flow instructions comprising two or more consecutive control flow instructions fused into a single fused control flow instruction; and

a branch unit to process the control flow instructions and to maintain a global counter indicating a nesting level of the control flow instructions, wherein to process a fused control flow instruction, the branch unit is to store a value N in a stack indicating a number of control flow instructions fused into the fused control flow instruction, the branch unit to subsequently read the value N from the stack upon execution of the fused control flow instruction and decrement the global counter by a value of N responsive to execution of the fused control flow instruction.

2. The processor as in claim 1 wherein the branch unit is to further maintain a per-channel count value for each of the plurality of channels, the per-channel count value to indicate whether each corresponding channel is enabled or disabled.

3. The processor as in claim 2 wherein the branch unit is to compare the per-channel count value to the global counter value and reset any per-channel counters having a value greater than the global counter value to 0.

4. The processor as in claim 3 wherein the branch unit is to generate a channel enable mask comprising a bit associated with each channel to indicate whether the channel is enabled or disabled, the bits of the channel enable mask to be set based on the values in the corresponding per-channel counters.

5. The processor as in claim 1 wherein the fused control flow instruction comprises a fused ENDIF instruction generated by fusing N consecutive ENDIF instructions.

6. The processor as in claim 1 wherein the stack comprises a stack counter to store the value N associated with the fused control flow instruction, wherein a threshold number of consecutive control flow instructions capable of being fused is set based on a number of bits used for the stack counter.

7. The processor as in claim 6 wherein the stack counter comprises three bits and the threshold number of consecutive control flow instructions capable of being fused comprises eight.

8. The processor as in claim 7 wherein the stack comprises a depth which is bounded by a maximum number of divergent branches comprising a single instruction multiple data (SIMD) width of the processor.

9. The processor as in claim 1 wherein the execution unit comprises one of a plurality of execution units within a graphics processing unit (GPU).

10. The processor as in claim 1 wherein a software kernel and/or compiler is to analyze the control flow instructions and to fuse the consecutive control flow instructions to generate the fused control flow instruction.

11. A method comprising:

fusing two or more consecutive control flow instructions into a single fused control flow instruction;

processing a sequence of control flow instructions including the fused control flow instruction and maintaining a global counter indicating a nesting level of the control flow instructions, wherein to process the fused control flow instruction:

storing a value N in a stack indicating a number of control flow instructions fused into the fused control flow instruction,

subsequently reading the value N from the stack upon execution of the fused control flow instruction and decrementing the global counter by a value of N responsive to execution of the fused control flow instruction.

12. The method as in claim 11 wherein the branch unit is to further maintain a per-channel count value for each of the plurality of channels, the per-channel count value to indicate whether each corresponding channel is enabled or disabled.

13. The method as in claim 12 wherein the branch unit is to compare the per-channel count value to the global counter value and reset any per-channel counters having a value greater than the global counter value to 0.

14. The method as in claim 13 wherein the branch unit is to generate a channel enable mask comprising a bit associated with each channel to indicate whether the channel is enabled or disabled, the bits of the channel enable mask to be set based on the values in the corresponding per-channel counters.

15. The method as in claim 11 wherein the fused control flow instruction comprises a fused ENDIF instruction generated by fusing N consecutive ENDIF instructions.

16. The method as in claim 11 wherein the stack comprises a stack counter to store the value N associated with the fused control flow instruction, wherein a threshold number of consecutive control flow instructions capable of being fused is set based on a number of bits used for the stack counter.

17. The method as in claim 16 wherein the stack counter comprises three bits and the threshold number of consecutive control flow instructions capable of being fused comprises eight.

18. The method as in claim 17 wherein the stack comprises a depth which is bounded by a maximum number of divergent branches comprising a single instruction multiple data (SIMD) width of a processor.

19. The method as in claim 11 implemented within an execution unit of a graphics processing unit (GPU).

20. The method as in claim 11 wherein a software kernel and/or compiler is to analyze the control flow instructions and to fuse the consecutive control flow instructions to generate the fused control flow instruction.

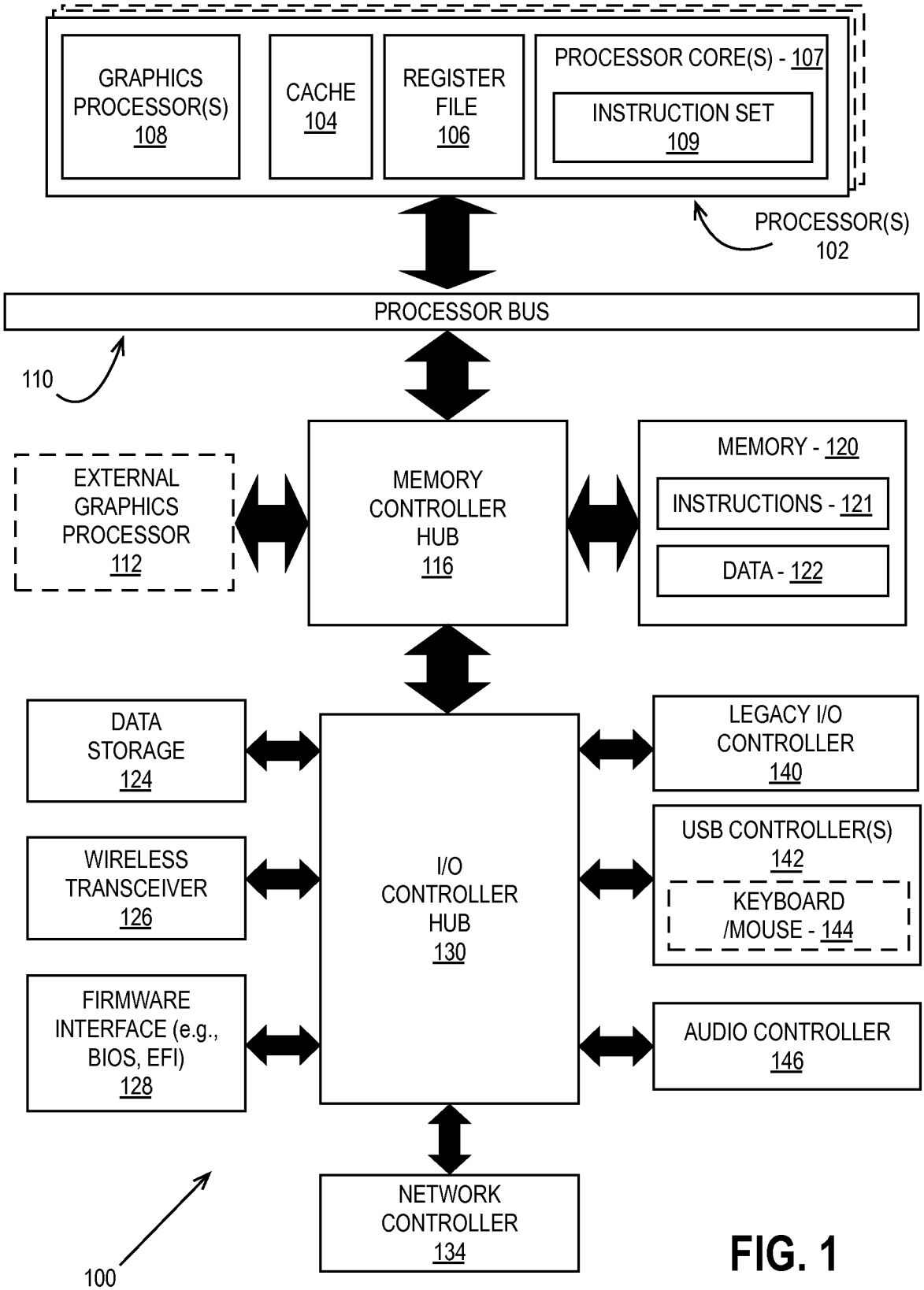

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) - 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY - 120
INSTRUCTIONS - 121
DATA - 122
DATA STORAGE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE (e.g., BIOS, EFI) 128
I/O CONTROLLER HUB 130
LEGACY I/O CONTROLLER 140
USB CONTROLLER(S) 142
KEYBOARD /MOUSE - 144
AUDIO CONTROLLER 146
NETWORK CONTROLLER 134
100

FIG. 1

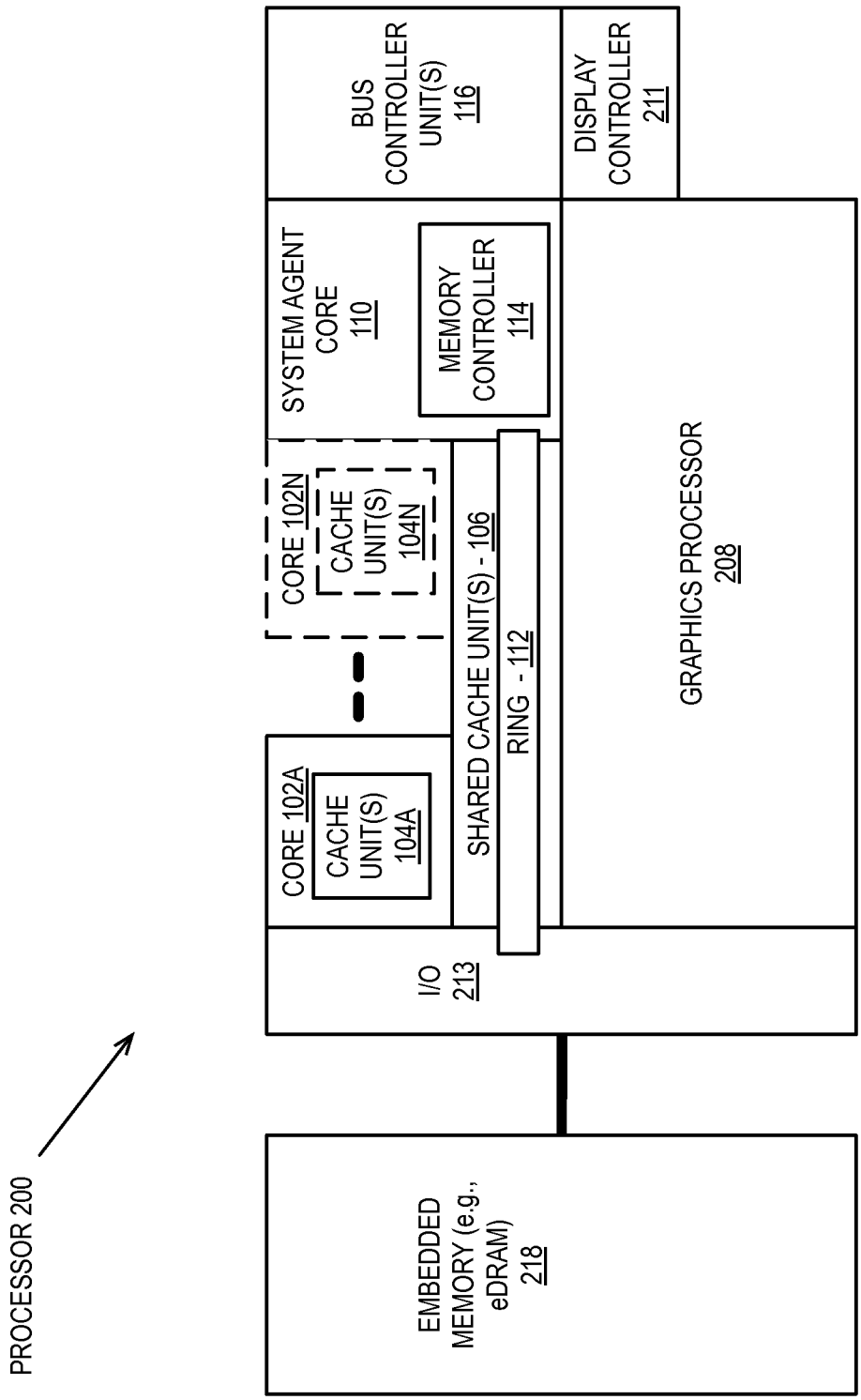

PROCESSOR 200
EMBEDDED MEMORY (e.g., eDRAM) 218
I/O 213
CORE 102A
CACHE UNIT(S) 104A
CORE 102N
CACHE UNIT(S) 104N
SHARED CACHE UNIT(S) - 106
RING - 112
GRAPHICS PROCESSOR 208
SYSTEM AGENT CORE 110
MEMORY CONTROLLER 114
BUS CONTROLLER UNIT(S) 116
DISPLAY CONTROLLER 211

FIG. 2

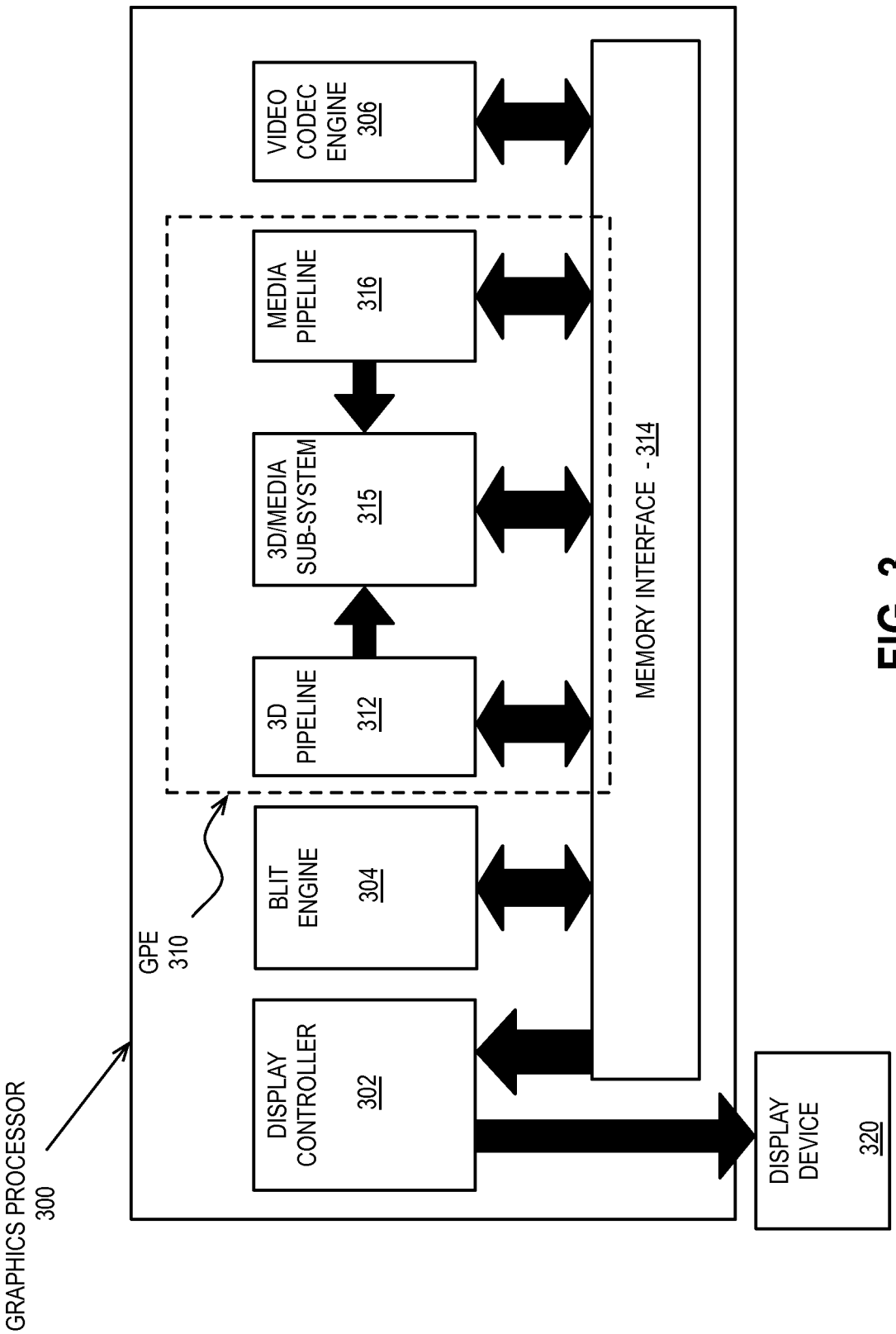

GRAPHICS PROCESSOR 300
GPE 310
DISPLAY CONTROLLER 302
BLIT ENGINE 304
3D PIPELINE 312
3D/MEDIA SUB-SYSTEM 315
MEDIA PIPELINE 316
VIDEO CODEC ENGINE 306
MEMORY INTERFACE - 314
DISPLAY DEVICE 320

FIG. 3

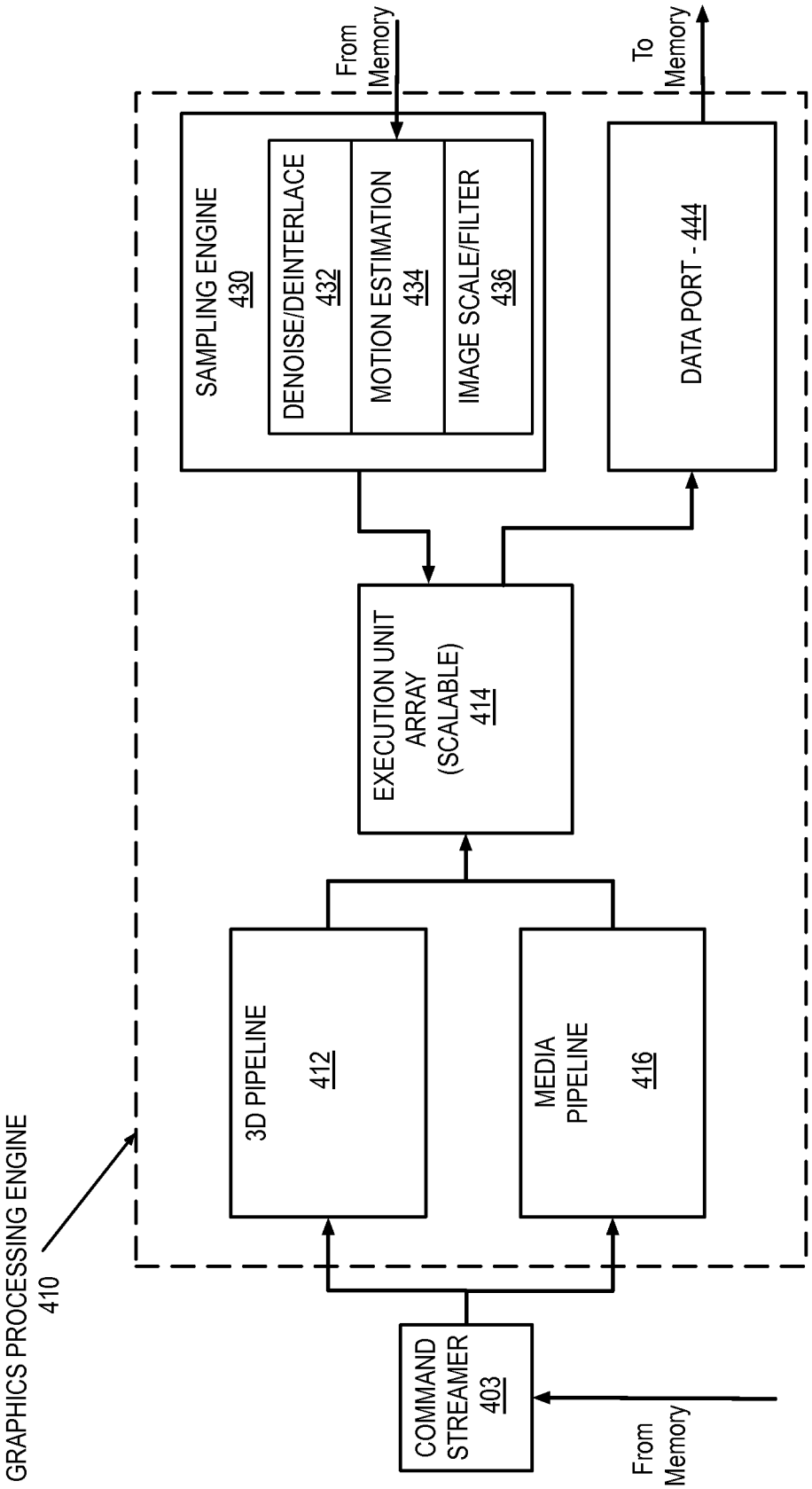

GRAPHICS PROCESSING ENGINE
410
From Memory
To Memory
SAMPLING ENGINE
430
DENOISE/DEINTERLACE
432
MOTION ESTIMATION
434
IMAGE SCALE/FILTER
436
DATA PORT - 444
EXECUTION UNIT ARRAY (SCALABLE)
414
3D PIPELINE
412
MEDIA PIPELINE
416
COMMAND STREAMER
403
From Memory

FIG. 4

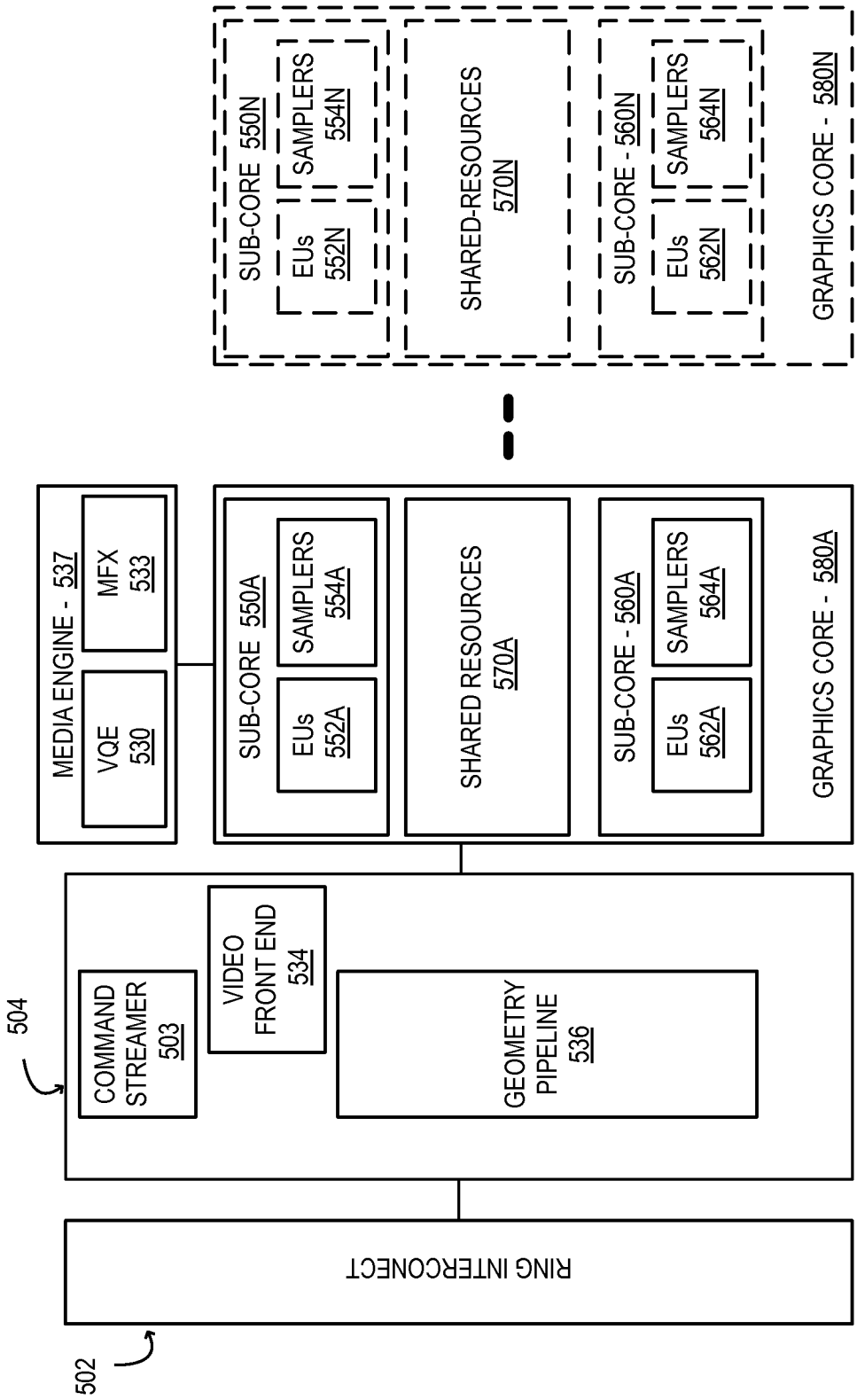

502
RING INTERCONNECT
504
COMMAND STREAMER 503
VIDEO FRONT END 534
GEOMETRY PIPELINE 536
MEDIA ENGINE - 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE - 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE - 580A
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE - 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE - 580N

FIG. 5

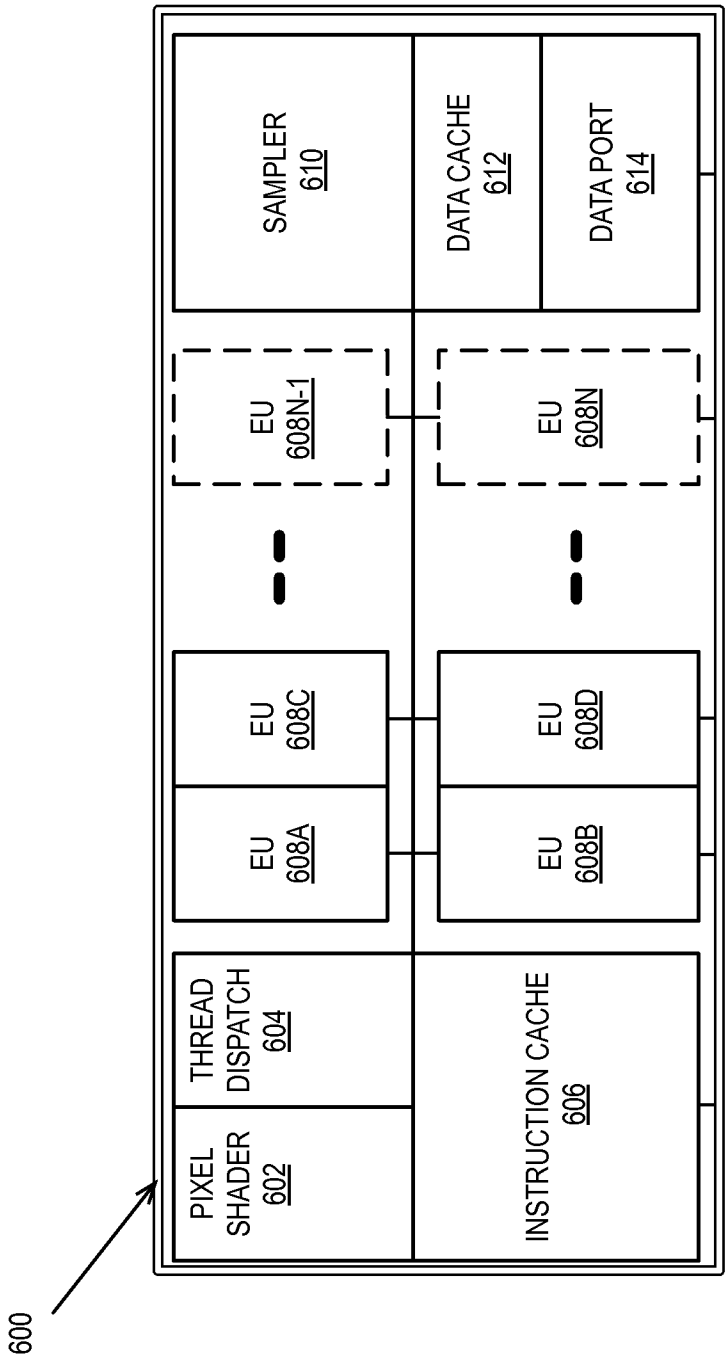

600
PIXEL SHADER 602
THREAD DISPATCH 604
INSTRUCTION CACHE 606
EU 608A
EU 608C
EU 608B
EU 608D
EU 608N-1
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 6

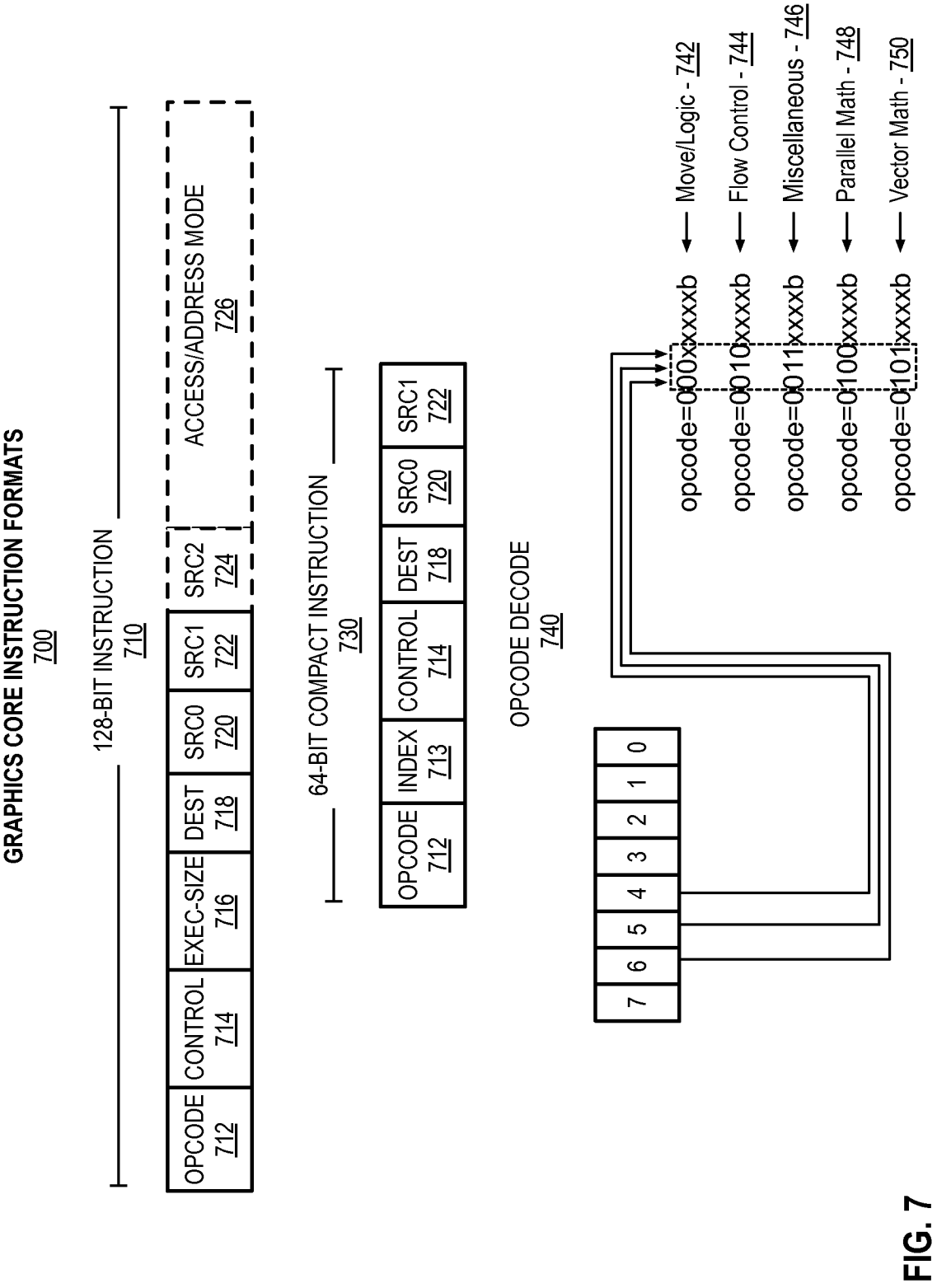

GRAPHICS CORE INSTRUCTION FORMATS
700
128-BIT INSTRUCTION
710
OPCODE 712
CONTROL 714
EXEC-SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS/ADDRESS MODE 726
64-BIT COMPACT INSTRUCTION
730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE
740
7 6 5 4 3 2 1 0
opcode=0000xxxxb
opcode=0010xxxxb
opcode=0011xxxxb
opcode=0100xxxxb
opcode=0101xxxxb
Move/Logic - 742
Flow Control - 744
Miscellaneous - 746
Parallel Math - 748
Vector Math - 750

FIG. 7

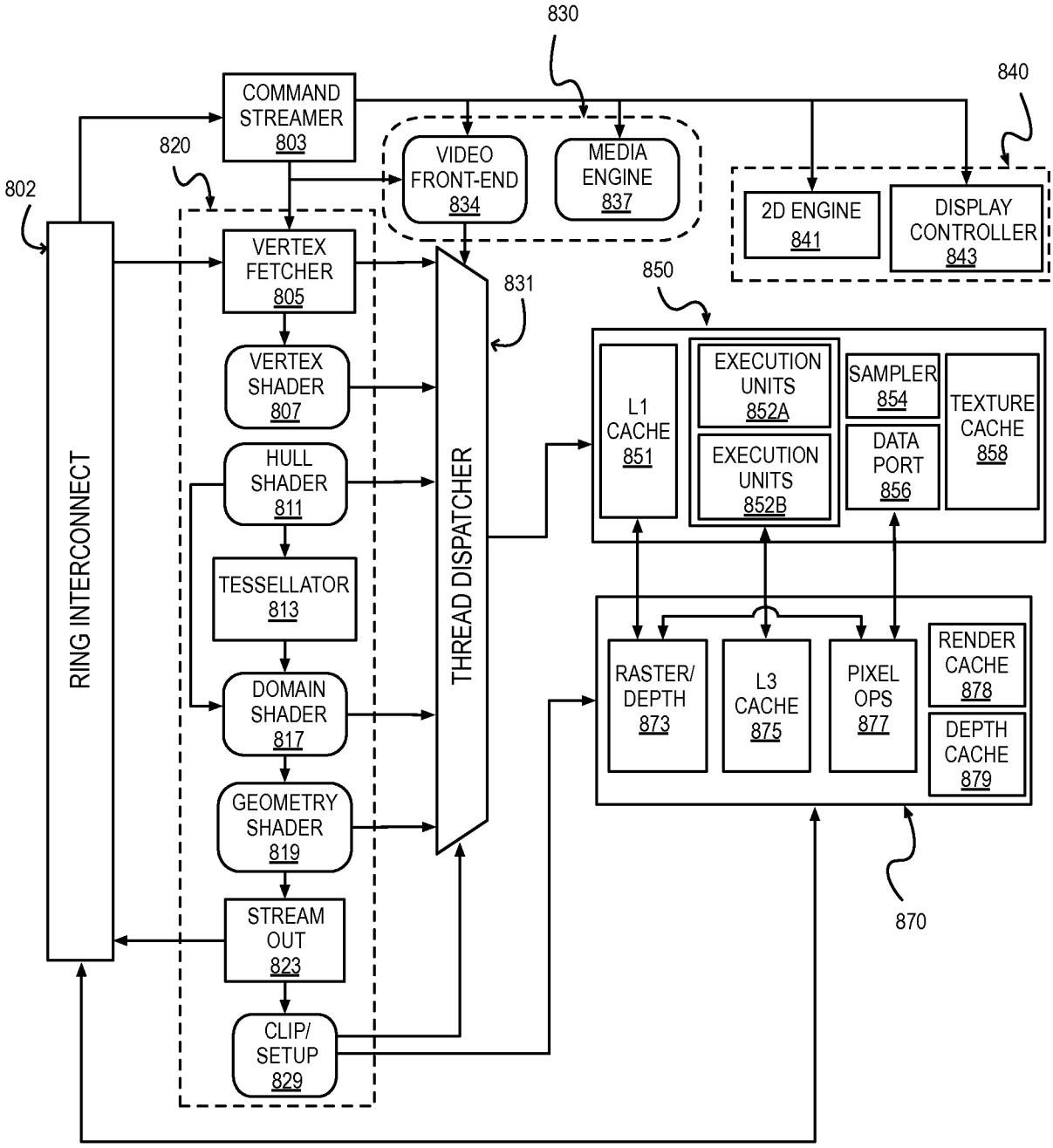

830
840
COMMAND STREAMER 803
820
802
VIDEO FRONT-END 834
MEDIA ENGINE 837
2D ENGINE 841
DISPLAY CONTROLLER 843
VERTEX FETCHER 805
831
850
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP/ SETUP 829
RING INTERCONNECT
THREAD DISPATCHER
L1 CACHE 851
EXECUTION UNITS 852A
EXECUTION UNITS 852B
SAMPLER 854
DATA PORT 856
TEXTURE CACHE 858
RASTER/ DEPTH 873
L3 CACHE 875
PIXEL OPS 877
RENDER CACHE 878
DEPTH CACHE 879
870

FIG. 8

FIG. 9A

SAMPLE COMMAND FORMAT
900

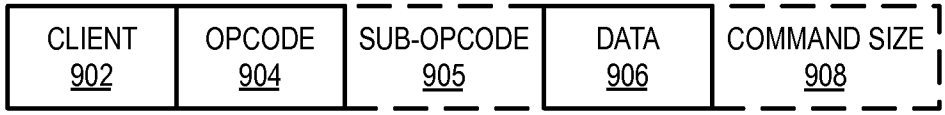
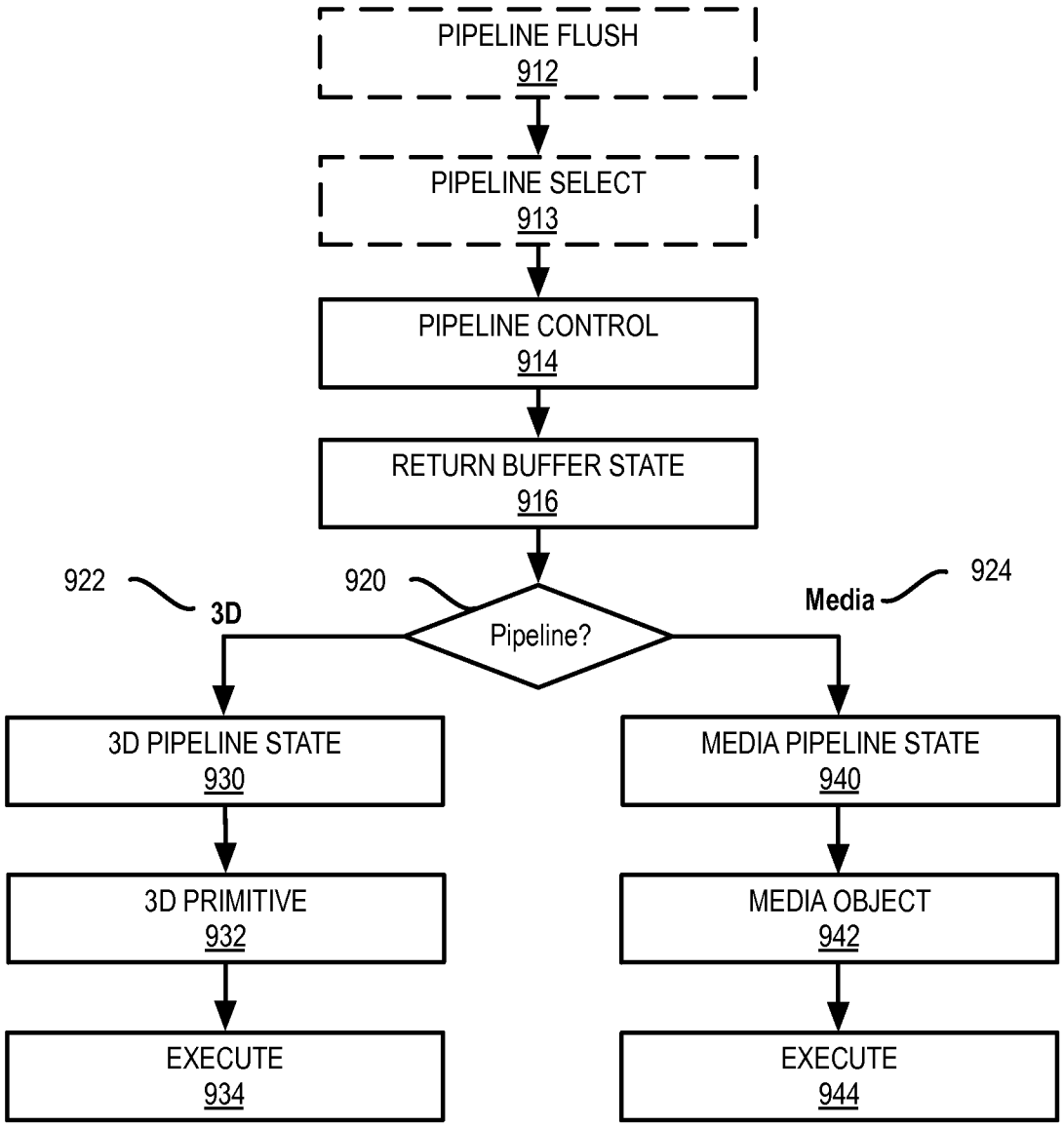


FIG. 9B

SAMPLE COMMAND SEQUENCE
910



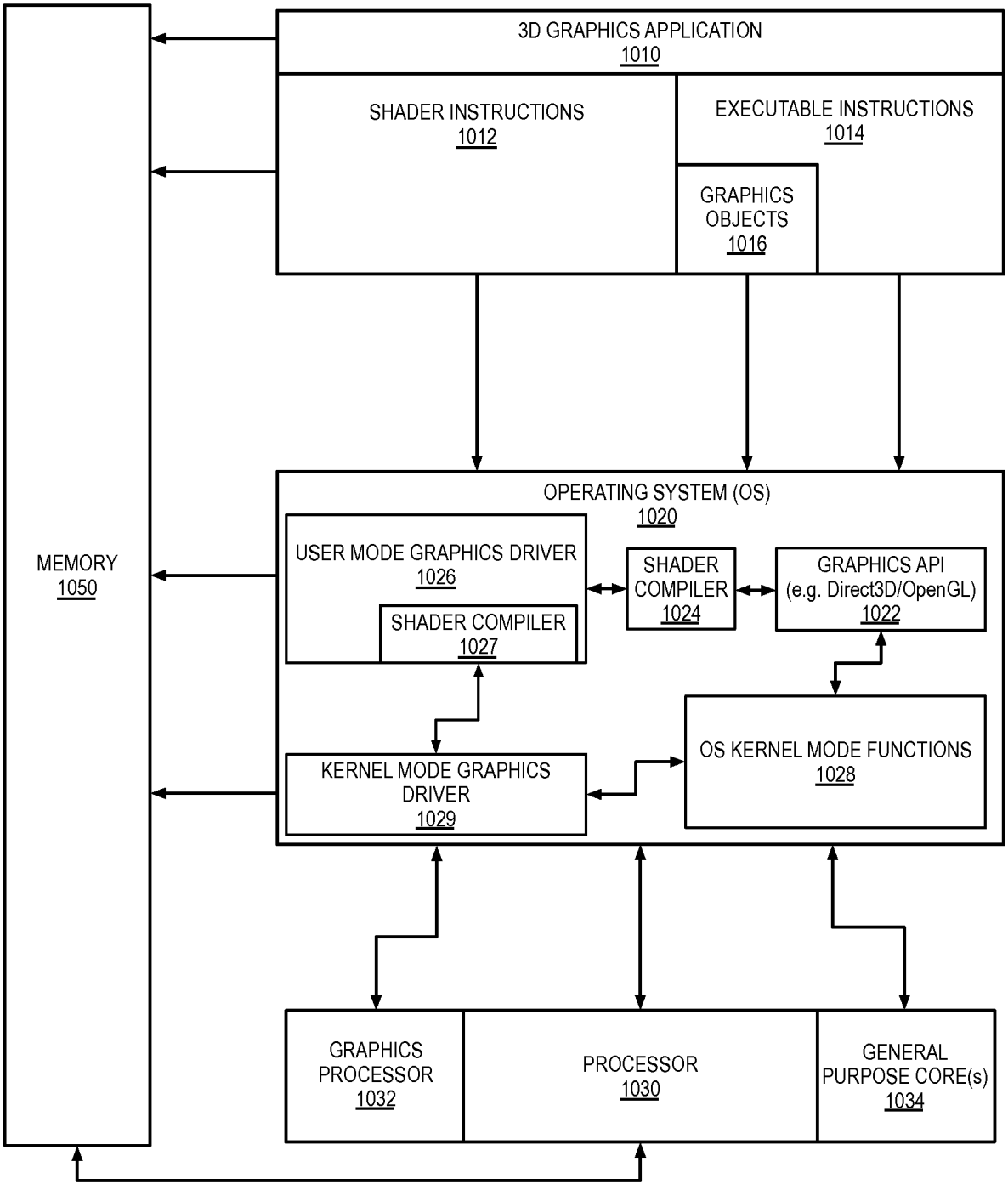

3D GRAPHICS APPLICATION
1010
SHADER INSTRUCTIONS
1012
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS OBJECTS
1016
MEMORY
1050
OPERATING SYSTEM (OS)
1020
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
SHADER COMPILER
1024
GRAPHICS API
(e.g. Direct3D/OpenGL)
1022
KERNEL MODE GRAPHICS DRIVER
1029
OS KERNEL MODE FUNCTIONS
1028
GRAPHICS PROCESSOR
1032
PROCESSOR
1030
GENERAL PURPOSE CORE(s)
1034

FIG. 10

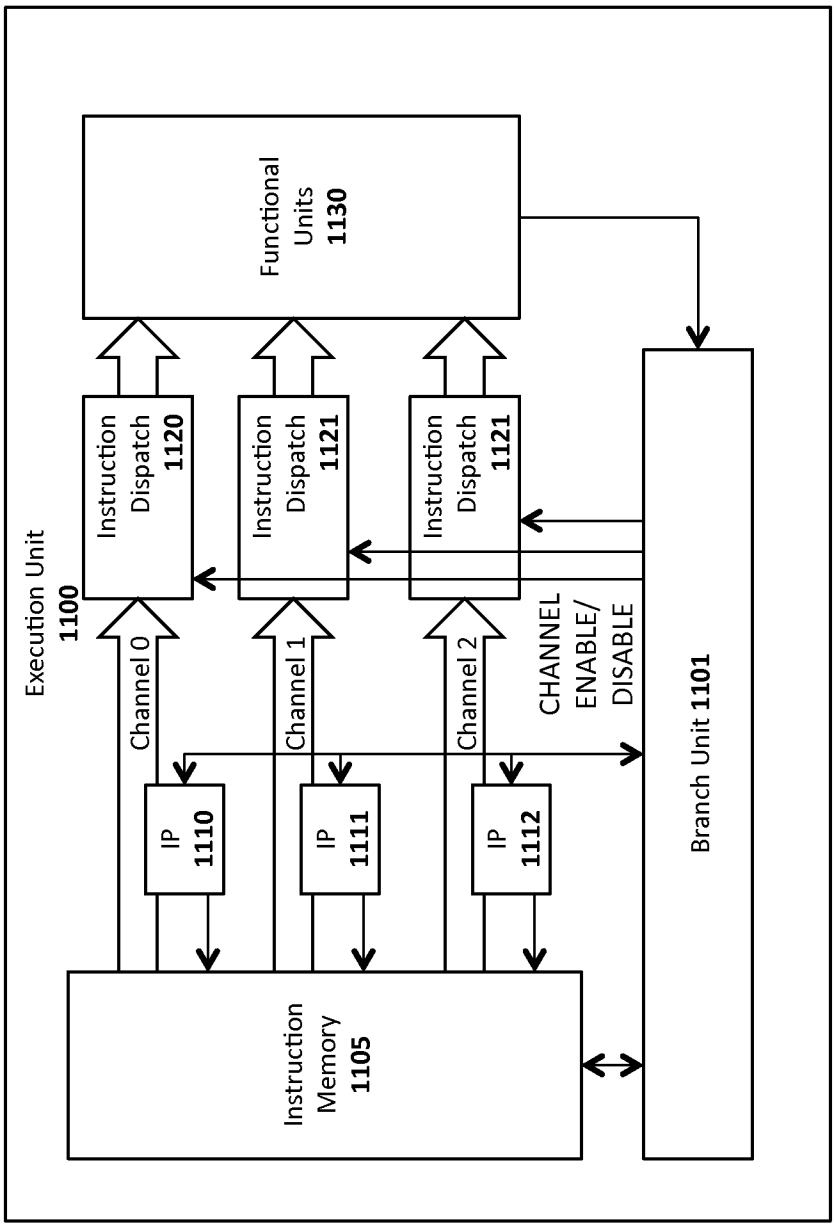

Execution Unit
1100
Functional Units
1130
Instruction Dispatch
1120
Instruction Dispatch
1121
Instruction Dispatch
1121
Channel 0
Channel 1
Channel 2
CHANNEL ENABLE/ DISABLE
IP
1110
IP
1111
IP
1112
Instruction Memory
1105
Branch Unit 1101

FIG. 11

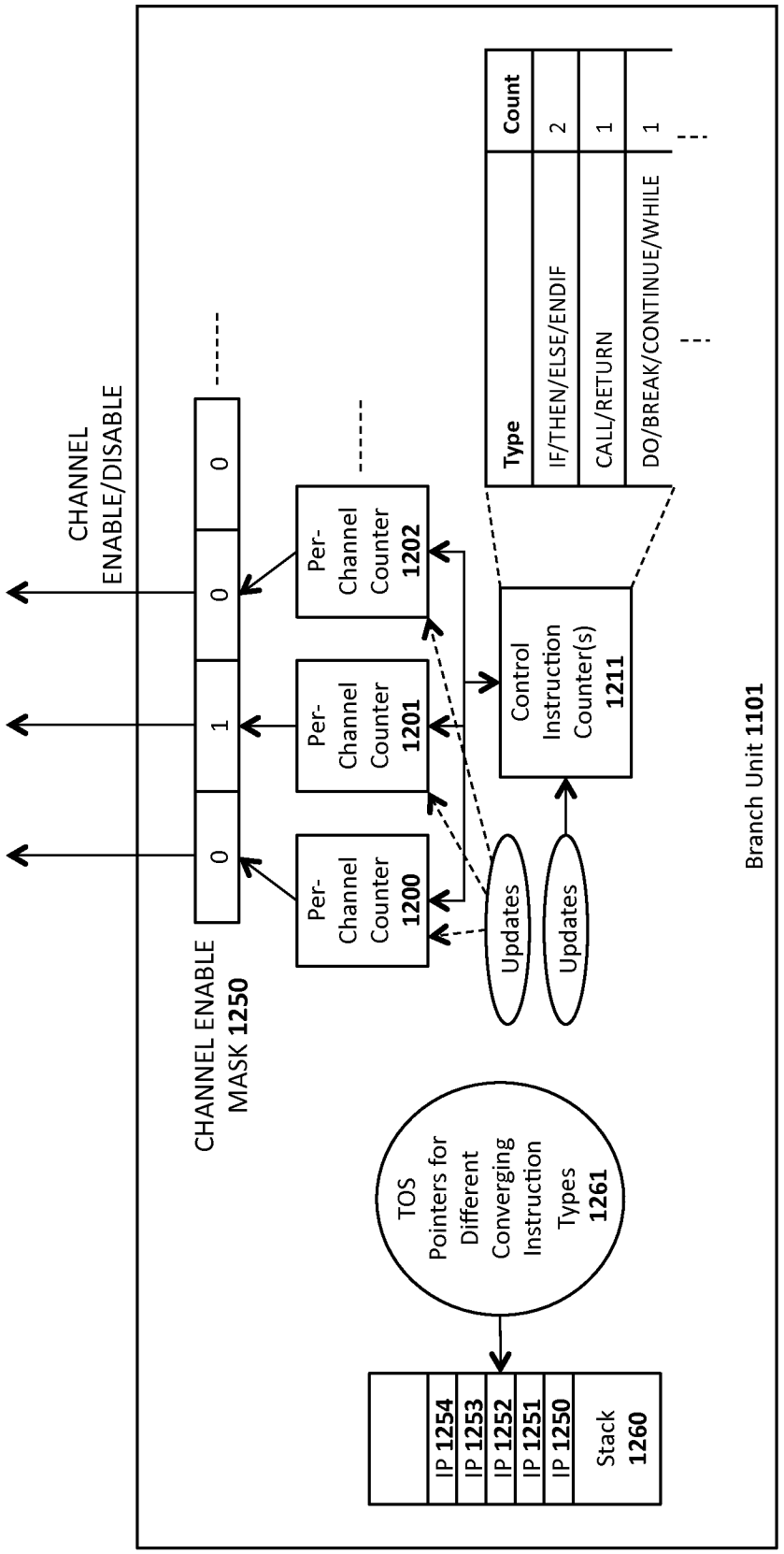

Branch Unit 1101
CHANNEL ENABLE/DISABLE
CHANNEL ENABLE MASK 1250
0
1
0
0
0
0
Per-Channel Counter 1200
Per-Channel Counter 1201
Per-Channel Counter 1202
Updates
Updates
Control Instruction Counter(s) 1211
Type
IF/THEN/ELSE/ENDIF
CALL/RETURN
DO/BREAK/CONTINUE/WHILE
Count
2
1
1
TOS Pointers for Different Converging Instruction Types 1261
Stack 1260
IP 1254
IP 1253
IP 1252
IP 1251
IP 1250

FIG. 12

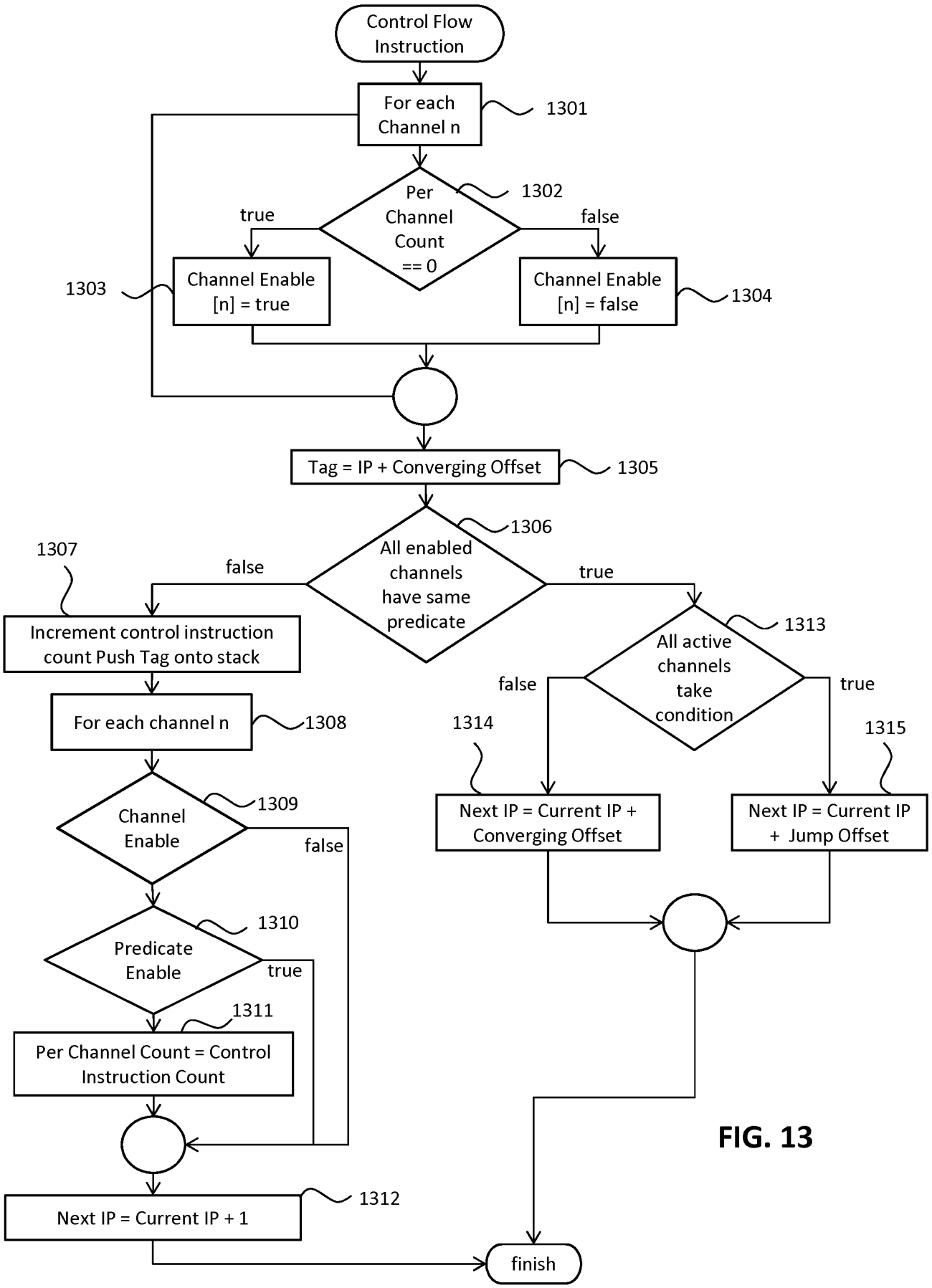

Control Flow Instruction
For each Channel n
1301
Per Channel Count == 0
1302
true
false
Channel Enable [n] = true
1303
Channel Enable [n] = false
1304
Tag = IP + Converging Offset
1305
All enabled channels have same predicate
1306
1307
false
true
Increment control instruction count Push Tag onto stack
For each channel n
1308
Channel Enable
1309
false
Predicate Enable
1310
true
Per Channel Count = Control Instruction Count
1311
Next IP = Current IP + 1
1312
All active channels take condition
1313
false
true
1314
1315
Next IP = Current IP + Converging Offset
Next IP = Current IP + Jump Offset
finish

FIG. 13

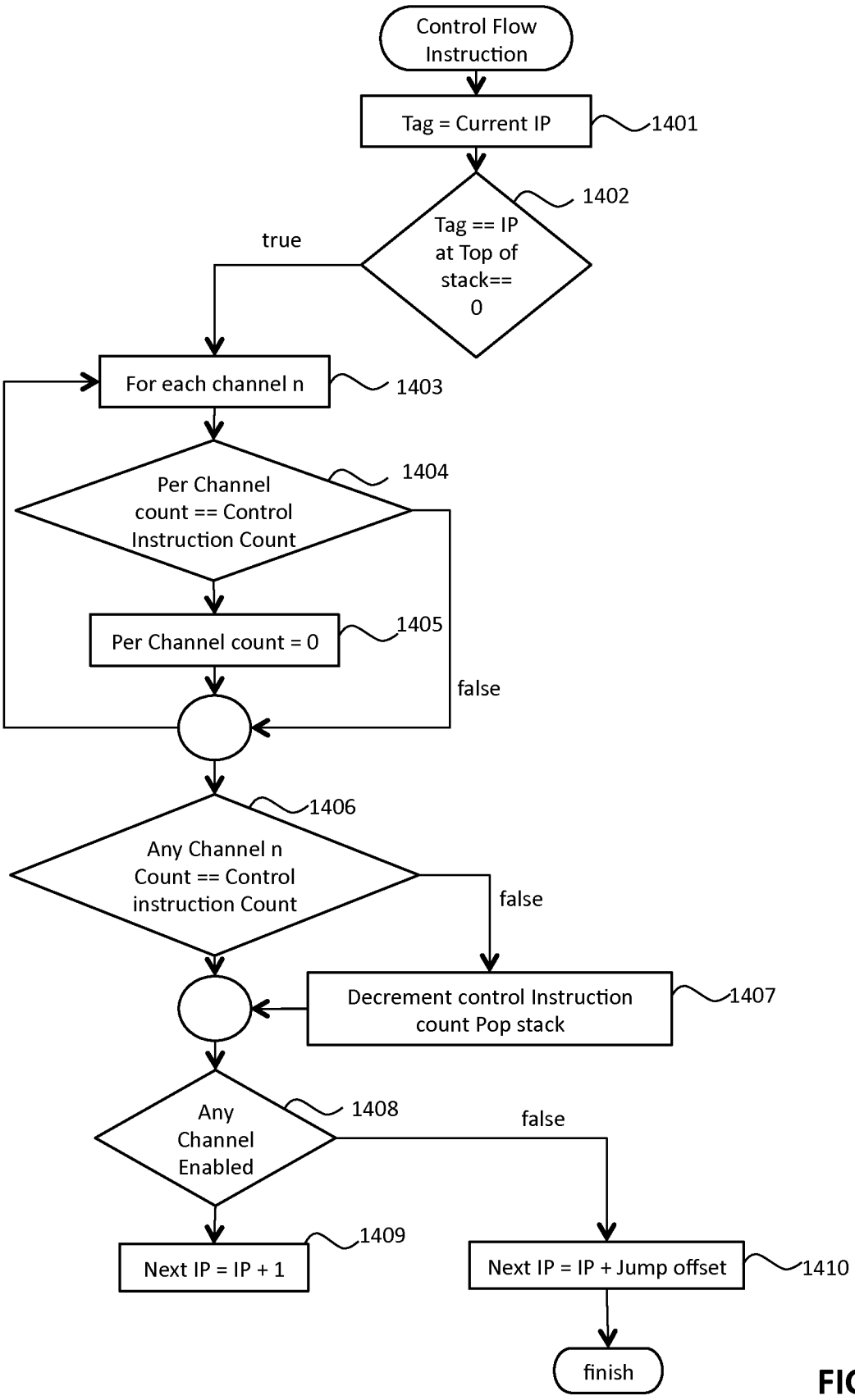

Control Flow Instruction
Tag = Current IP
1401
1402
Tag == IP at Top of stack== 0
true
For each channel n
1403
1404
Per Channel count == Control Instruction Count
Per Channel count = 0
1405
false
1406
Any Channel n Count == Control instruction Count
false
Decrement control Instruction count Pop stack
1407
Any Channel Enabled
1408
false
1409
Next IP = IP + 1
Next IP = IP + Jump offset
1410
finish

FIG. 14

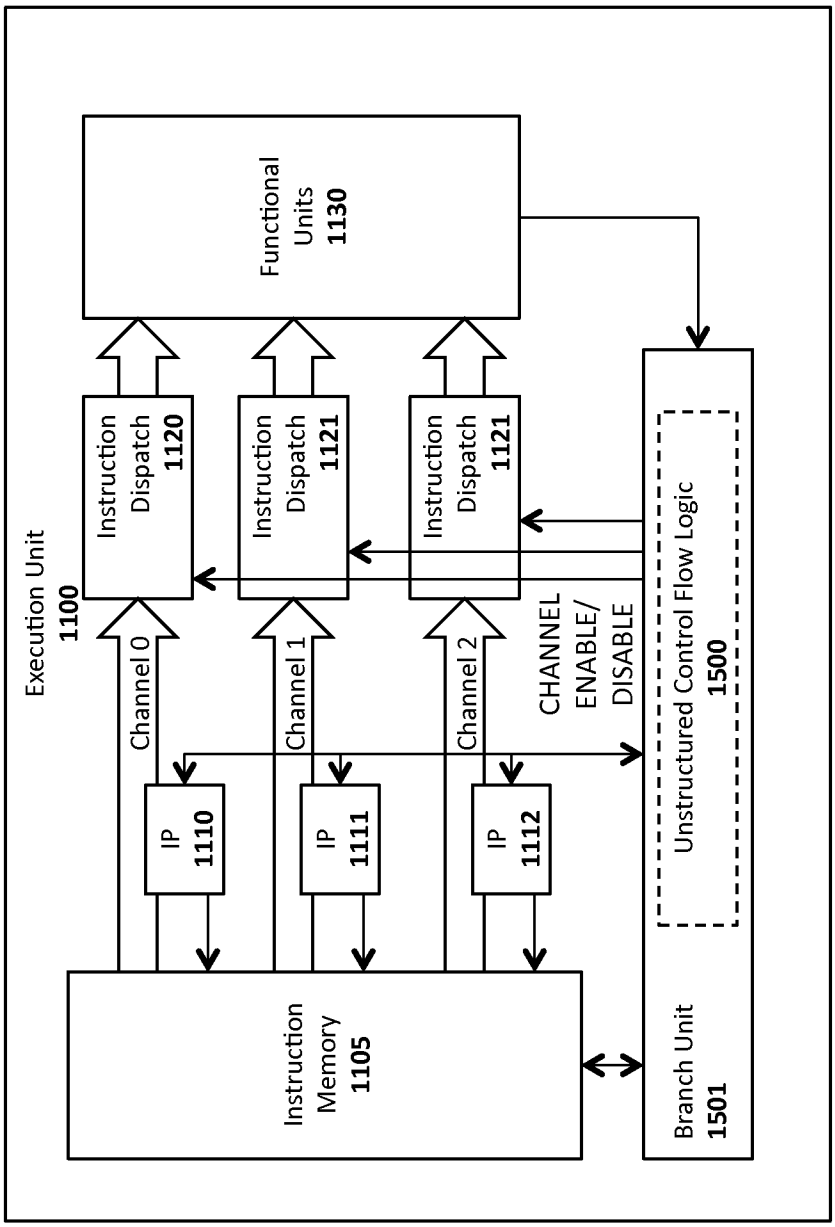

Execution Unit
1100
Functional Units
1130
Instruction Dispatch
1120
Instruction Dispatch
1121
Instruction Dispatch
1121
Channel 0
Channel 1
Channel 2
CHANNEL ENABLE/ DISABLE
IP
1110
IP
1111
IP
1112
Instruction Memory
1105
Unstructured Control Flow Logic
1500
Branch Unit
1501

FIG. 15

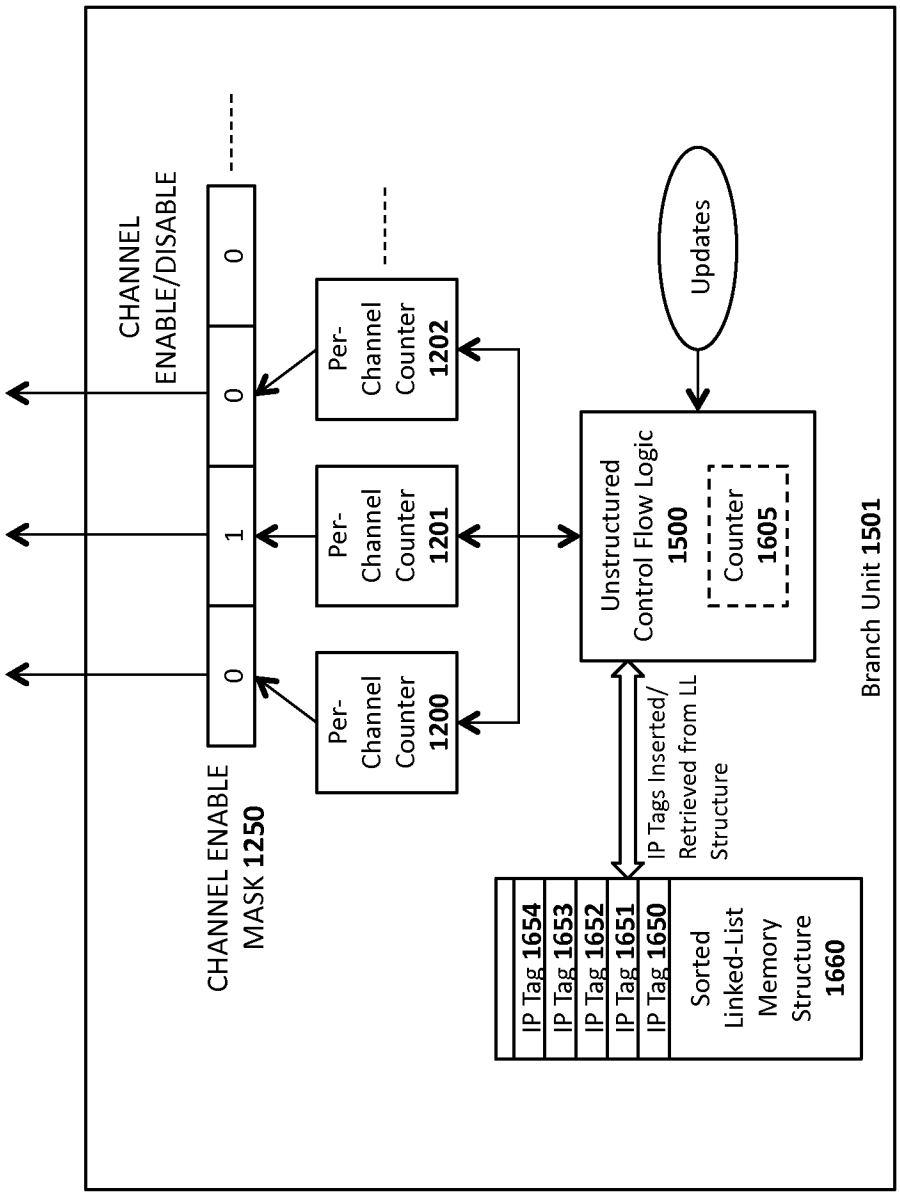

Branch Unit 1501
Unstructured Control Flow Logic 1500
Counter 1605
Updates
Per-Channel Counter 1202
Per-Channel Counter 1201
Per-Channel Counter 1200
CHANNEL ENABLE/DISABLE
CHANNEL ENABLE MASK 1250
0
1
0
0
IP Tags Inserted/ Retrieved from LL Structure
Sorted Linked-List Memory Structure 1660
IP Tag 1650
IP Tag 1651
IP Tag 1652
IP Tag 1653
IP Tag 1654

FIG. 16

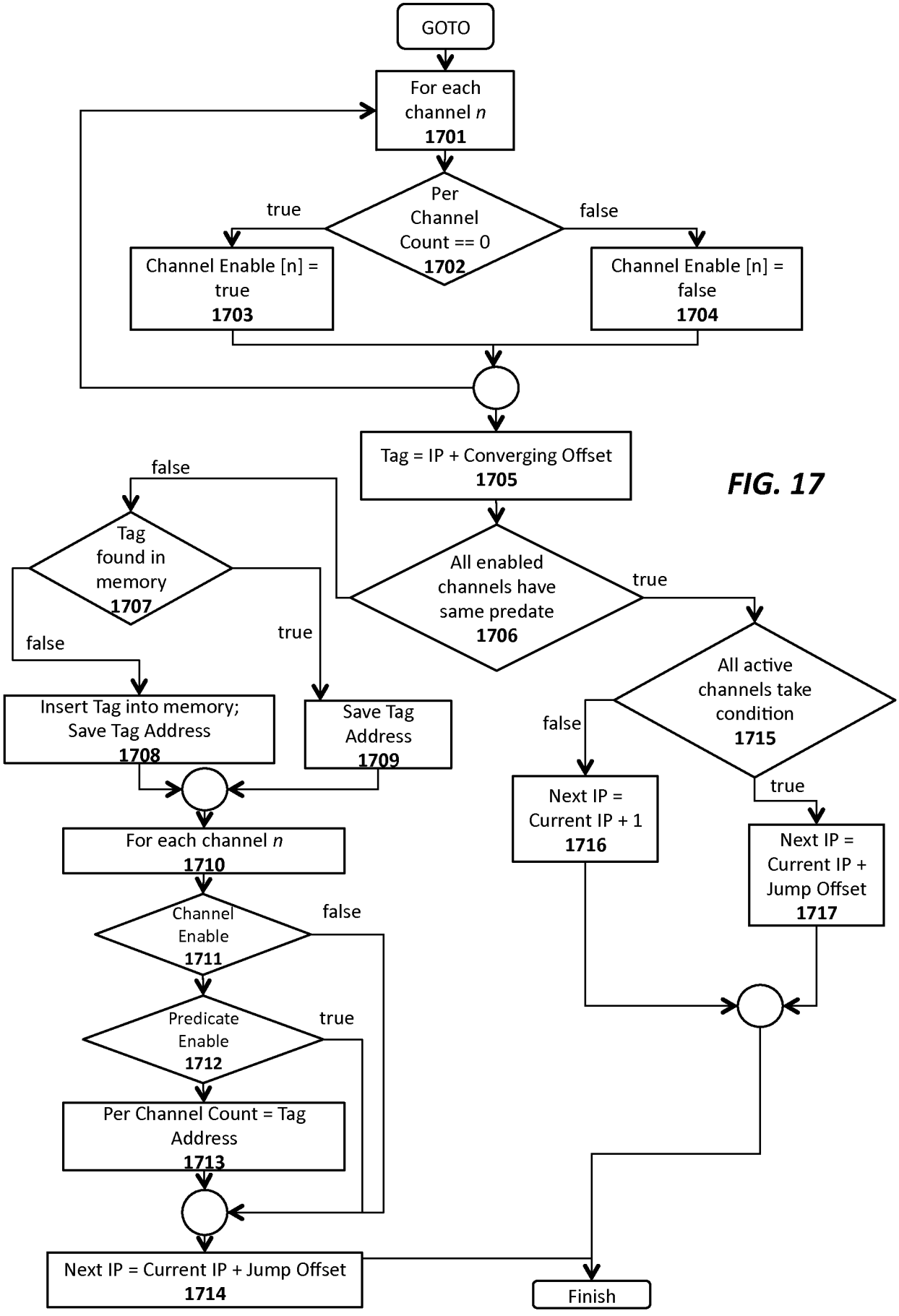

GOTO
For each channel n 1701
Per Channel Count == 0 1702
true
false
Channel Enable [n] = true 1703
Channel Enable [n] = false 1704
Tag = IP + Converging Offset 1705
All enabled channels have same predate 1706
true
false
All active channels take condition 1715
true
false
Next IP = Current IP + 1 1716
Next IP = Current IP + Jump Offset 1717
Tag found in memory 1707
false
true
Insert Tag into memory; Save Tag Address 1708
Save Tag Address 1709
For each channel n 1710
Channel Enable 1711
false
Predicate Enable 1712
true
Per Channel Count = Tag Address 1713
Next IP = Current IP + Jump Offset 1714
Finish

FIG. 17

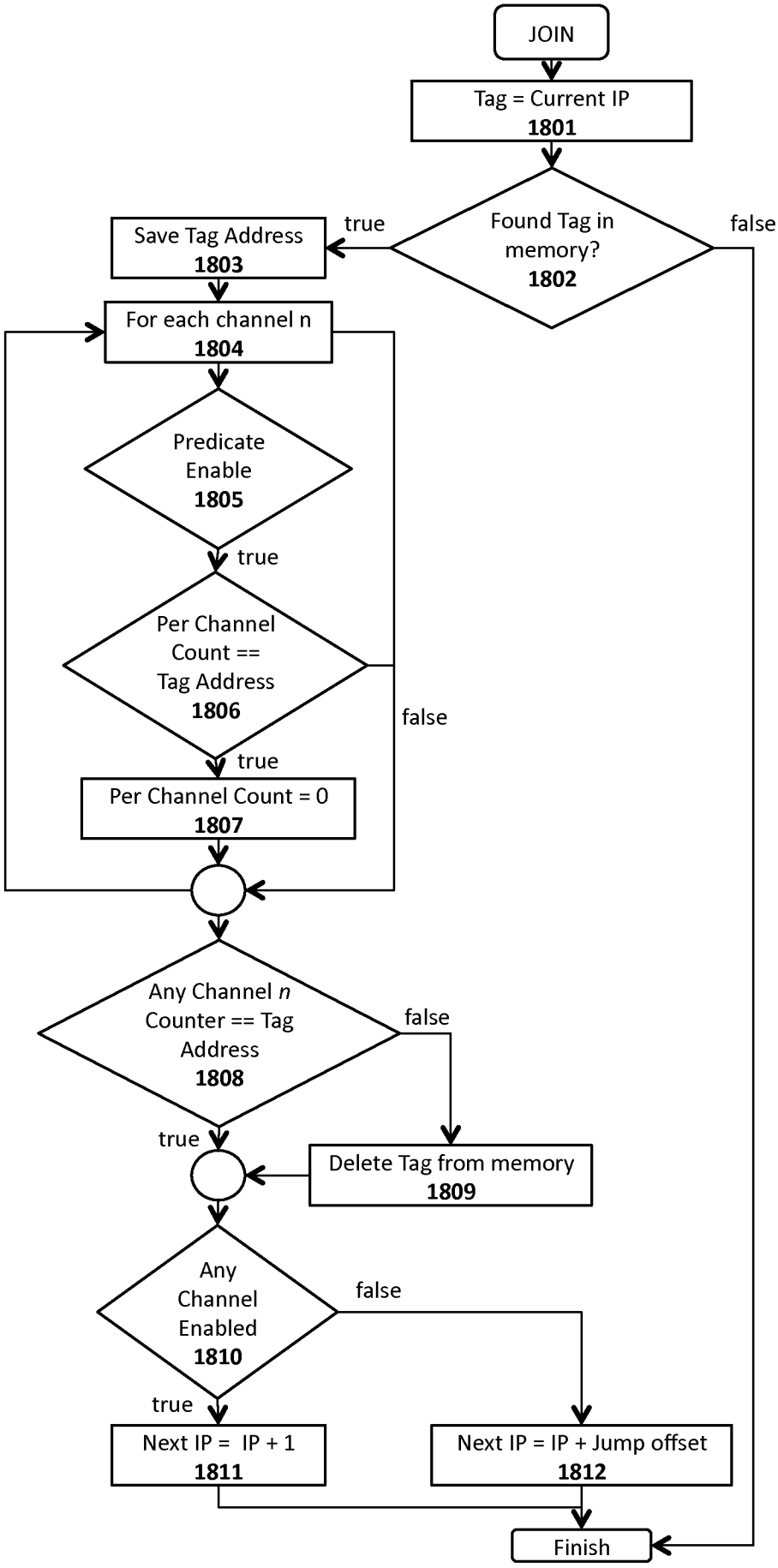

JOIN
Tag = Current IP
1801
Found Tag in memory?
1802
true
false
Save Tag Address
1803
For each channel n
1804
Predicate Enable
1805
true
Per Channel Count == Tag Address
1806
false
true
Per Channel Count = 0
1807
Any Channel n Counter == Tag Address
1808
false
true
Delete Tag from memory
1809
Any Channel Enabled
1810
false
true
Next IP = IP + 1
1811
Next IP = IP + Jump offset
1812
Finish

FIG. 18

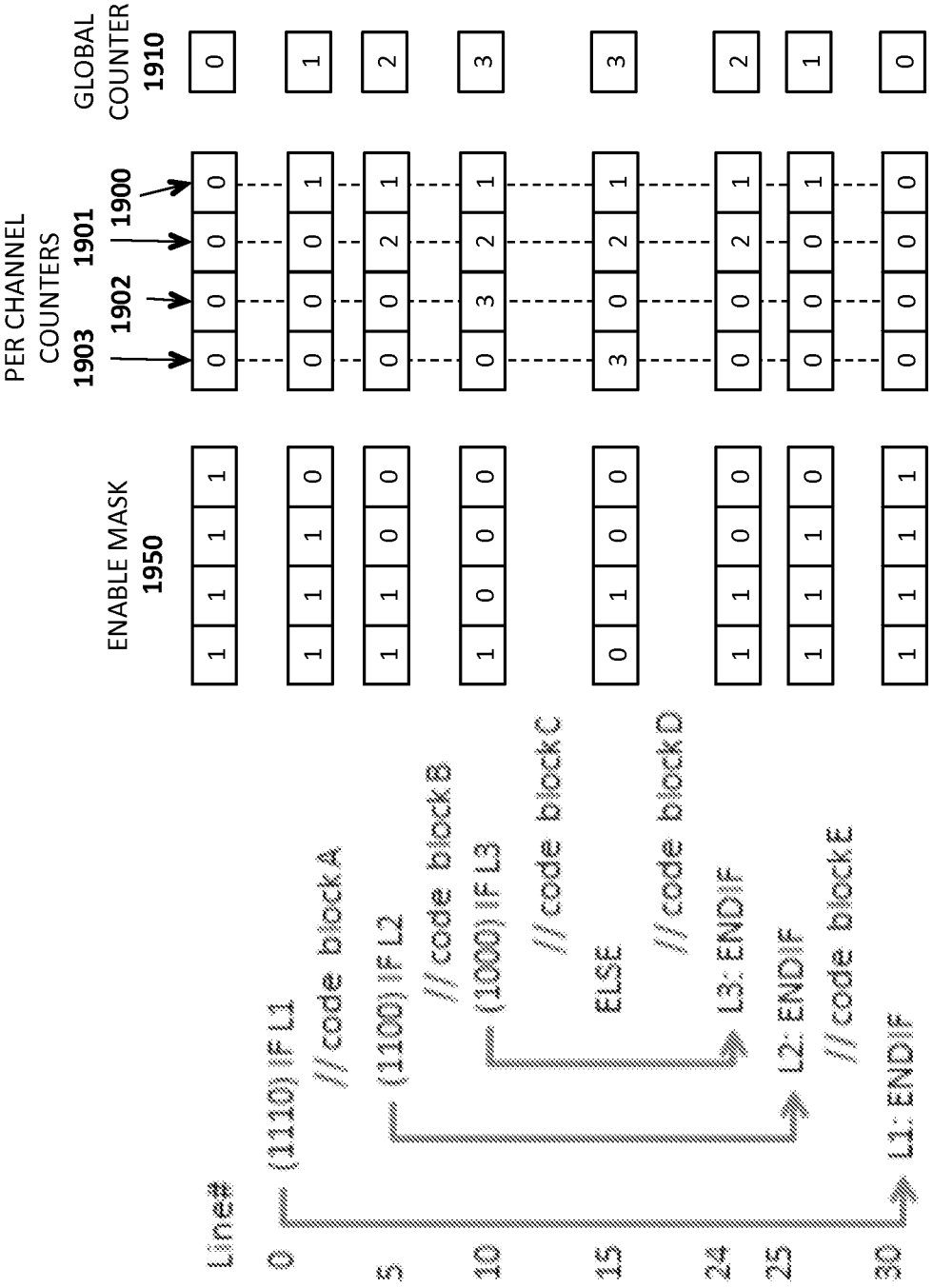

Line#
0
5
10
15
24
25
30
(1110) IF L1
// code block A
(1100) IF L2
// code block B
(1000) IF L3
// code block C
ELSE
// code block D
L3: ENDIF
L2: ENDIF
// code block E
L1: ENDIF
ENABLE MASK
1950
PER CHANNEL
COUNTERS
1903
1902
1901
1900
GLOBAL
COUNTER
1910

FIG. 19

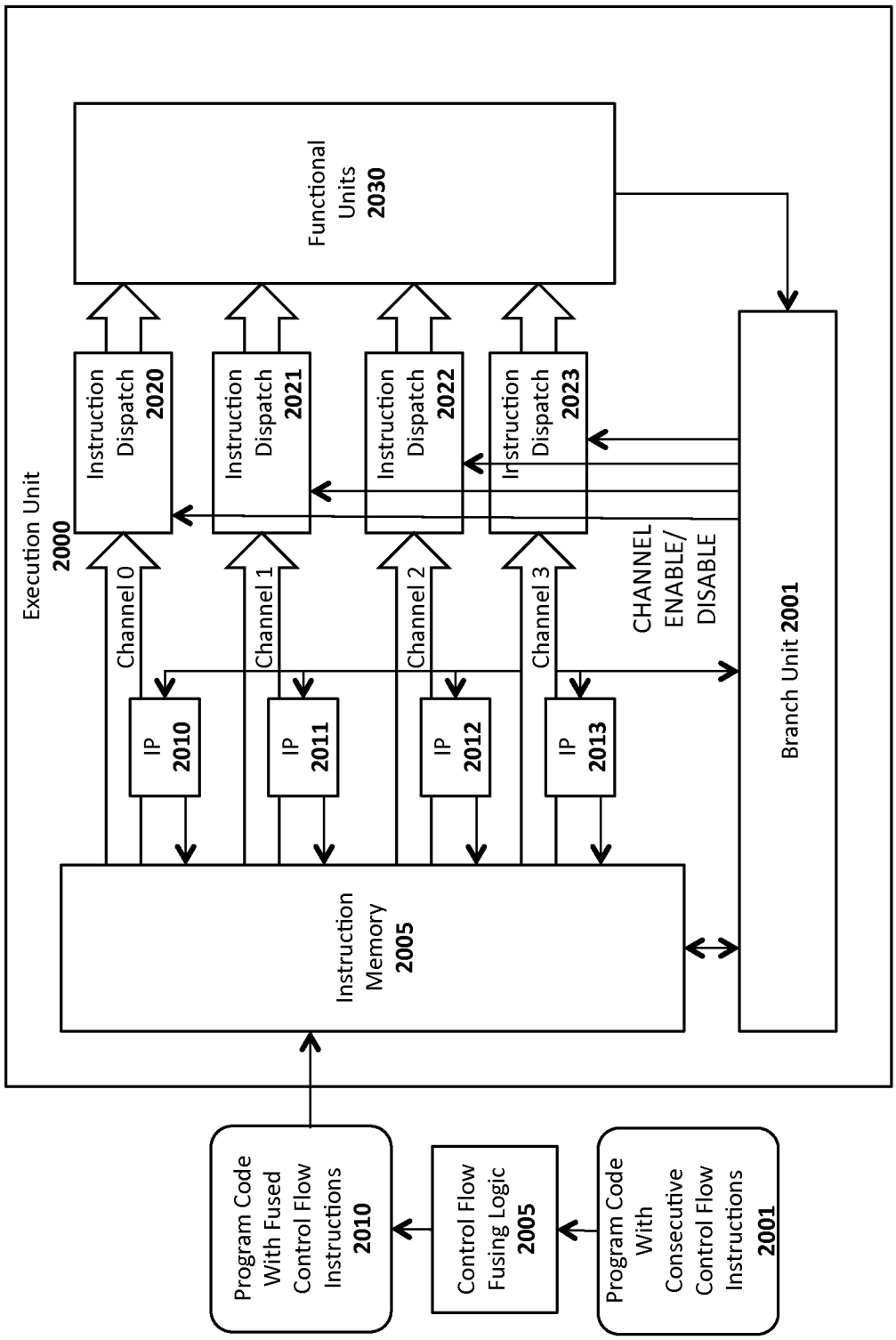

Execution Unit
2000
Functional Units
2030
Instruction Dispatch
2020
Instruction Dispatch
2021
Instruction Dispatch
2022
Instruction Dispatch
2023
Channel 0
Channel 1
Channel 2
Channel 3
IP
2010
IP
2011
IP
2012
IP
2013
CHANNEL ENABLE/ DISABLE
Branch Unit 2001
Instruction Memory
2005
Program Code With Fused Control Flow Instructions
2010
Control Flow Fusing Logic
2005
Program Code With Consecutive Control Flow Instructions
2001

FIG. 20

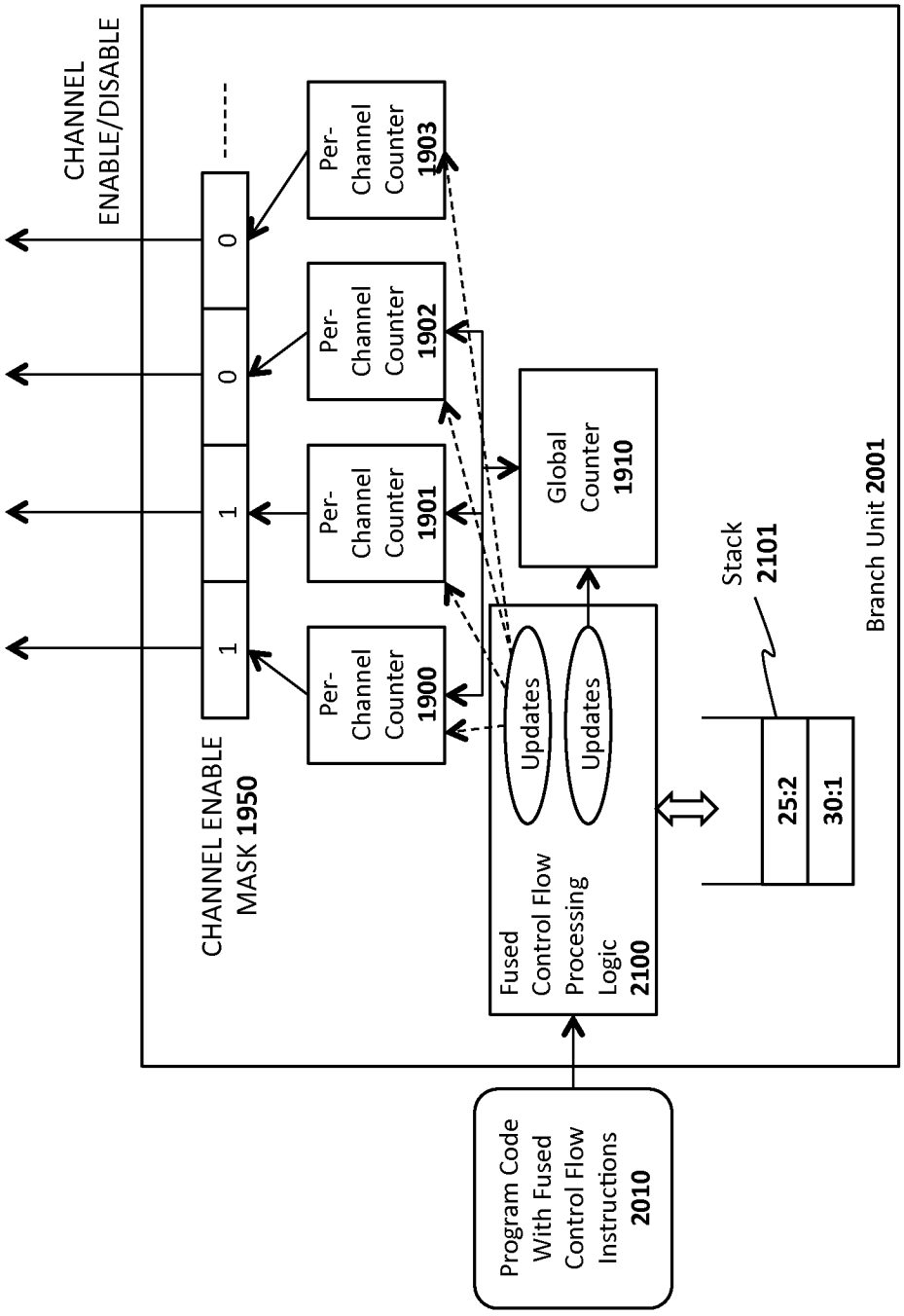

CHANNEL ENABLE/DISABLE
CHANNEL ENABLE MASK 1950
1
1
0
0
Per-Channel Counter 1900
Per-Channel Counter 1901
Per-Channel Counter 1902
Per-Channel Counter 1903
Global Counter 1910
Updates
Updates
Fused Control Flow Processing Logic 2100
Stack 2101
25:2
30:1
Branch Unit 2001
Program Code With Fused Control Flow Instructions 2010

FIG. 21

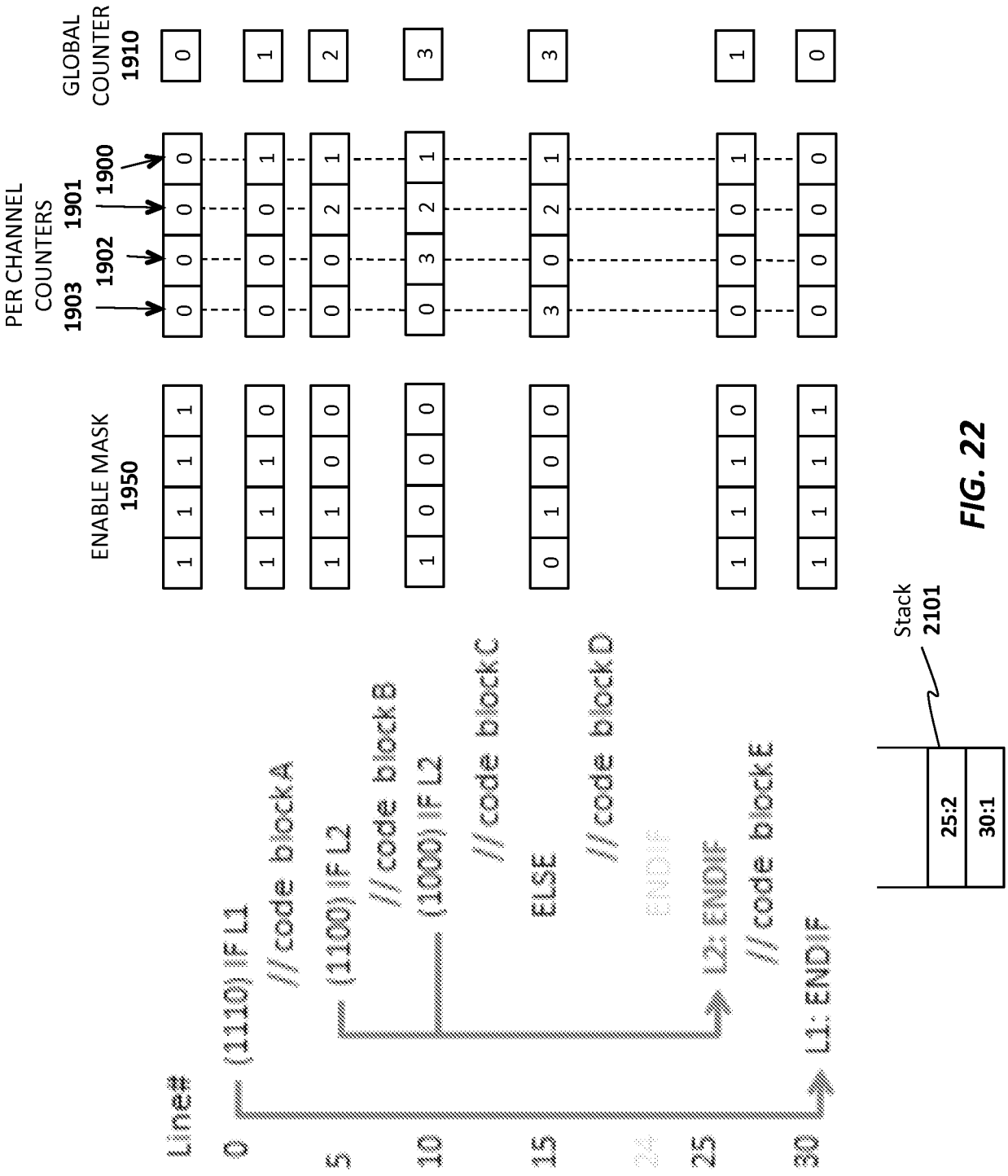

PER CHANNEL COUNTERS
1903
1902
1901
1900
GLOBAL COUNTER
1910
ENABLE MASK
1950
Line#
0
5
10
15
24
25
30
(1110) IF L1
// code block A
(1100) IF L2
// code block B
(1000) IF L2
// code block C
ELSE
// code block D
ENDIF
L2: ENDIF
// code block E
L1: ENDIF
Stack
2101
25:2
30:1

FIG. 22

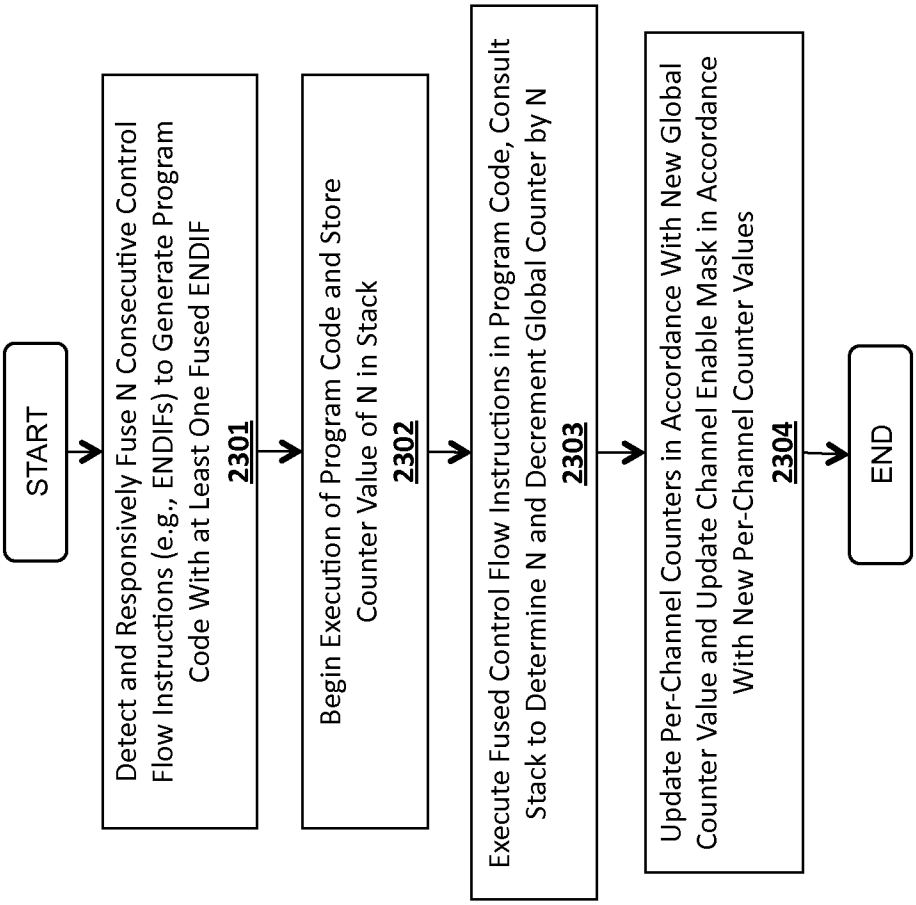

START
Detect and Responsively Fuse N Consecutive Control Flow Instructions (e.g., ENDIFs) to Generate Program Code With at Least One Fused ENDIF
2301
Begin Execution of Program Code and Store Counter Value of N in Stack
2302
Execute Fused Control Flow Instructions in Program Code, Consult Stack to Determine N and Decrement Global Counter by N
2303
Update Per-Channel Counters in Accordance With New Global Counter Value and Update Channel Enable Mask in Accordance With New Per-Channel Counter Values
2304
END

FIG. 23

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/062066

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/38(2006.01)i, G06F 15/80(2006.01)i, G06T 1/20(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 9/38; G06F 9/30; G06F 9/02; G06F 9/315; G06F 9/00; G06F 15/80; G06T 1/20

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: control flow instruction, fuse, global counter, branch, stack, nested, GPU, ENDIF, and similar terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2008-0114975 A1 (HSUEH-BING YEN) 15 May 2008 See paragraphs [0020]-[0025]; claims 1-9; and figure 1.	1-20
A	US 2013-0290674 A1 (BIJU GEORGE et al.) 31 October 2013 See paragraphs [0011]-[0039]; and claims 1-4.	1-20
A	US 2005-0154864 A1 (NORMAN RUBIN et al.) 14 July 2005 See paragraphs [0015]-[0035]; and figures 1-3.	1-20
A	US 2008-0052492 A1 (DAVE STUTTARD et al.) 28 February 2008 See paragraphs [0119], [0217]; and claim 1.	1-20
A	US 2014-0181477 A1 (ANIRUDDHA S. VAIDYA et al.) 26 June 2014 See paragraphs [0019]-[0039]; and figures 1-4.	1-20

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

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
18 March 2016 (18.03.2016)	**21 March 2016 (21.03.2016)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-472-7140

Authorized officer
CHIN, Sang Bum
Telephone No. +82-42-481-8398

Form PCT/ISA/210 (second sheet) (January 2015)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2015/062066

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0114975 A1	15/05/2008	None	
US 2013-0290674 A1	31/10/2013	None	
US 2005-0154864 A1	14/07/2005	US 7281122 B2	09/10/2007
US 2008-0052492 A1	28/02/2008	AU 2000-38295 A1	14/11/2000
		EP 1181648 A1	27/02/2002
		GB 2348971 A	18/10/2000
		GB 2348972 A	18/10/2000
		GB 2348973 A	18/10/2000
		GB 2348974 A	18/10/2000
		GB 2348975 A	18/10/2000
		GB 2348976 A	18/10/2000
		GB 2348977 A	18/10/2000
		GB 2348978 A	18/10/2000
		GB 2348979 A	18/10/2000
		GB 2348980 A	18/10/2000
		GB 2348981 A	18/10/2000
		GB 2348982 A	18/10/2000
		GB 2348983 A	18/10/2000
		GB 2348984 A	18/10/2000
		GB 2349484 A	01/11/2000
		GB 2352306 A	24/01/2001
		JP 05285828 B2	11/09/2013
		JP 2002-541586 A	03/12/2002
		JP 2011-023036 A	03/02/2011
		JP 2011-023037 A	03/02/2011
		US 2002-0174318 A1	21/11/2002
		US 2007-0226458 A1	27/09/2007
		US 2007-0242074 A1	18/10/2007
		US 2007-0245123 A1	18/10/2007
		US 2007-0245130 A1	18/10/2007
		US 2007-0245132 A1	18/10/2007
		US 2007-0294510 A1	20/12/2007
		US 2008-0007562 A1	10/01/2008
		US 2008-0008393 A1	10/01/2008
		US 2008-0010436 A1	10/01/2008
		US 2008-0016318 A1	17/01/2008
		US 2008-0028184 A1	31/01/2008
		US 2008-0034185 A1	07/02/2008
		US 2008-0034186 A1	07/02/2008
		US 2008-0040575 A1	14/02/2008
		US 2008-0098201 A1	24/04/2008
		US 2008-0162874 A1	03/07/2008
		US 2008-0162875 A1	03/07/2008
		US 2008-0184017 A1	31/07/2008
		US 2009-0198898 A1	06/08/2009
		US 2009-0228683 A1	10/09/2009

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2015/062066

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
		US 7363472 B2	22/04/2008
		US 7506136 B2	17/03/2009
		US 7526630 B2	28/04/2009
		US 7627736 B2	01/12/2009
		US 7802079 B2	21/09/2010
		US 7925861 B2	12/04/2011
		US 7958332 B2	07/06/2011
		US 7966475 B2	21/06/2011
		US 8169440 B2	01/05/2012
		US 8171263 B2	01/05/2012
		US 8174530 B2	08/05/2012
		US 8762691 B2	24/06/2014
		WO 00-62182 A2	19/10/2000
		WO 00-62182 A3	12/04/2001
US 2014-0181477 A1	26/06/2014	None	