(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 2016/105763 A1

(51) International Patent Classification:
G06T 1/20 (2006.01)

(21) International Application Number:
PCT/US2015/062068

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

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/581,701 23 December 2014 (23.12.2014) US

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

(72) Inventors: **MAIYURAN, Subramaniam**; 1915 Black Slate Court, Gold River, California 95670 (US). **PIAZZA, Thomas A.**; 9005 Oak Leaf Way, Granite Bay, California 95746 (US). **GARCIA PABON, Jorge F.**; 1900 Prairie City Road, Folsom, California 95630 (US). **SHAH, Shubh B.**; 1133 Buckbrush Drive, Folsom, California 95630 (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))*

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

(54) Title: METHOD AND APPARATUS FOR A HIGH THROUGHPUT RASTERIZER

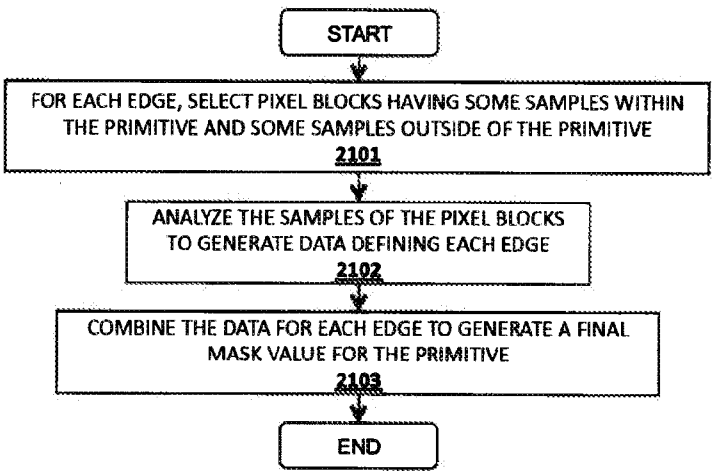


FIG. 21

(57) Abstract: An apparatus and method are described for a high throughput rasterizer. For example, one embodiment of an apparatus comprises: block selection logic to select a plurality of pixel blocks associated with edges of a primitive, the plurality of pixel blocks selected based on the pixel blocks having samples which are both inside and outside of the primitive; and edge determination logic to analyze samples of the plurality of pixel blocks selected by the block selection logic and responsively generate data identifying each edge of the primitive; and final mask determination logic to combine the data identifying each edge and generate a final mask representing the primitive.

WO 2016/105763 A1

METHOD AND APPARATUS FOR A HIGH THROUGHPUT RASTERIZER

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 a high throughput rasterizer.

Description of the Related Art

[0002] Rasterizing is a critical task of a graphics processing unit (GPU) in which an image described in a vector graphics format is converted into a raster image (e.g., pixels) for output on a display device. Rasterization often requires determining a list of lit samples of a primitive, which is a recurrent task that must be done efficiently to consume a minimum amount of power. Graphics techniques such as multisample anti-aliasing (MSAA) require a significant amount of graphics data to be processed without impacting the rendering time. Techniques have been applied to this problem but the throughout available using current solutions is insufficient to support new and future requirements.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0004] **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;

[0005] **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;

[0006] **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;

[0007] **FIG. 4** is a block diagram of an embodiment of a graphics-processing engine for a graphics processor;

[0008] **FIG. 5** is a block diagram of another embodiment of a graphics processor;

[0009] **FIG. 6** is a block diagram of thread execution logic including an array of processing elements;

[0010] **FIG. 7** illustrates a graphics processor execution unit instruction format according to an embodiment;

[0011] **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;

[0012] **FIG. 9A** is a block diagram illustrating a graphics processor command format according to an embodiment;

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

[0014] **FIG. 10** illustrates exemplary graphics software architecture for a data processing system according to an embodiment;

[0015] **FIG. 11** illustrates a graphics processing architecture within which one embodiment of the high throughput rasterizer may be employed;

[0016] **FIGS. 12-13** illustrate how primitives may be processed to determine edges during rasterization;

[0017] **FIG. 14** illustrates an implementation in which a 16x16 pixel is subdivided into smaller blocks across multiple processing stages;

[0018] **FIG. 15** illustrates one embodiment in which parallel processing is performed on each edge of a primitive;

[0019] **FIG. 16** illustrates additional details for one embodiment in which parallel processing is performed on each edge of a primitive;

[0020] **FIG. 17** illustrates symmetry which is utilized to perform more efficient rasterization in one embodiment of the invention;

[0021] **FIGS. 18A-B** illustrates additional logic including a set of comparators employed in one embodiment of the invention;

[0022] **FIG. 19** illustrates additional layers of symmetry utilized to perform more efficient rasterization in one embodiment of the invention;

[0023] **FIG. 20** illustrates additional layers of symmetry at the pixel level utilized to perform more efficient rasterization in one embodiment of the invention; and

[0024] **FIG. 21** illustrates a method in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

[0025] 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

[0026] System Overview

[0027] **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.

[0028] 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.

[0029] 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).

[0030] 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.

[0031] 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.

[0032] 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.

[0033] 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.

[0034] **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.

[0035] 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.

[0036] 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).

[0037] 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.

[0038] 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.

[0039] 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.

[0040] 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.

[0041] 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.

[0042] 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.

[0043] **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.

[0044] 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.

[0045] 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.

[0046] 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.

[0047] 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.

[0048] 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.

[0049] **3D/Media Processing**

[0050] **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.

[0051] 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.

[0052] 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.

[0053] 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).

[0054] 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.

[0055] 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.

[0056] 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.

[0057] **Execution Units**

[0058] **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.

[0059] 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.

[0060] 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.

[0061] 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.

[0062] **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.

[0063] 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.

[0064] 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).

[0065] 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.

[0066] 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.

[0067] 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.

[0068] 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.

[0069] 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.

[0070] 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.

[0071] **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.

[0072] 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.

[0073] 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.

[0074] 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.

[0075] 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.

[0076] 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.

[0077] **Graphics Pipeline**

[0078] **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.

[0079] 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.

[0080] 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.

[0081] 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.

[0082] 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.

[0083] 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.

[0084] 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.

[0085] 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.

[0086] 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.

[0087] 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.

[0088] 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.

[0089] 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.

[0090] **Graphics Pipeline Programming**

[0091] **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.

[0092] 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.

[0093] 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.

[0094] 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.

[0095] 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.

[0096] 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.

[0097] 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.

[0098] 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.

[0099] 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.

[00100] 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.

[00101] 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.

[00102] 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.

[00103] 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.

[00104] 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.

[00105] **Graphics Software Architecture**

[00106] **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.

[00107] 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.

[00108] 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.

[00109] 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 A HIGH THROUGHPUT RASTERIZER

[00110] Embodiments of the invention may be implemented within a rasterizer of a graphics pipeline. **Figure 11** illustrates an exemplary graphics architecture which includes a high throughput rasterizer 1100 within a render output pipeline 1170 for implementing the techniques described below. The other components of the graphics processor architecture shown in **Figure 11** include a graphics pipeline 820, a media pipeline 830, a display engine 840, and thread execution logic 850, embodiments of which are described above with respect to **Figure 8**. It should be noted, however, that the underlying principles of the invention are not limited to any specific graphics processor architecture.

[00111] As mentioned above, rasterizing is a critical task of a GPU in which an image described in a vector graphics format is converted into a raster image (e.g., pixels) for output on

a display device. Rasterization often requires determining a list of lit samples of a primitive, which is a recurrent task that must be done efficiently to consume a minimum amount of power. Graphics techniques such as multisample anti-aliasing (MSAA) require a significant amount of graphics data to be processed without impacting the rendering time. Techniques have been applied to this problem but the throughout available using current solutions is insufficient to support current and future requirements.

[00112] In operation, to produce the final sample mask, a primitive is divided into different edges. By way of example, **Figure 12** shows how a triangle primitive 1200 may be divided into three edges 1201-1203. Each edge 1201-1203 may be calculated independently and then the results may be merged to produce the final mask. As used herein, a bounding box 1205 comprises a region containing the entire primitive 1200 (i.e., pixels outside of the bounding box do not include any portion of the primitive).

[00113] By way of another example, **Figure 13** shows how a wide line primitive 1300 may be divided into four different edges 1301-1304. Once again, each edge 1301-1304 is typically calculated independently, followed by a merging of the results to produce the mask. In the specific examples in **Figures 12-13** and in certain embodiments described below, the primitive and results are encoded within a 16x16 block. However, the underlying principles of the invention are not limited to any particular block size.

[00114] In one embodiment, to determine whether a sample is inside ("lit") or outside the primitive ("not lit"), the following equation is used:

$$L_0(x,y) = Lo_{16x16} + \Delta L_x + \Delta L_y \geq 0.$$

where Lo_{16x16} is the Lo value of the center of the current 16x16 pixel block and ΔL_x and ΔL_y specify the distance from the center of the 16x16 block to the sample (using X and Y coordinates, respectively). In one embodiment, if $Lo(x, y)$ is zero or positive then the sample is lit and if it is negative then the sample is not lit.

[00115] Existing solutions use a divide and conquer approach such as shown in **Figure 14** which take a 16x16 pixel block 1400 and divide it into four 8x8 blocks within a "superspan" stage 1401. Each 8x8 block is further divided into four 4x4 blocks within a "span" stage 1402 and each 4x4 block is divided into four 2x2 blocks within a "subspan" stage 1403. A final pixel stage 1404 then generates the mask 1405. One limitation of this approach is that it is not readily scalable and, as such, suffers from throughput problems.

[00116] In contrast, the embodiments of the invention described below use a parallel approach and calculate all the 4x4 blocks (sometimes referred to as "spans") at the same time. This may be accomplished, for example, by only calculating partial 4x4 blocks. In particular, if all samples within a 4x4 block are lit (the block is "fully covered") or if all samples are not lit

(the block is "void") these spans are ignored. This may be done because fully lit blocks and void blocks do not contribute to an edge. This parallel approach is more scalable than prior implementations because more span calculators can be added.

[00117] One embodiment of the invention employs a span render scheme to improve the current windower throughput and power management. The high throughput rasterizer determines the status of all 16 spans (4x4 pixel blocks) of the 16x16 pixels input in parallel. After this first operation the rasterizer determines the partial spans (some samples lit and some not lit), fully covered spans (all samples lit), and void spans (no samples lit). Because fully covered and void span output values are known, the focus of subsequent processing stages is then on the partials, which are used to determine the primitive edges. In one embodiment, there may be up to 7 partials per edge.

[00118] As mentioned above, the number of edges depends on the primitive (e.g., triangle = 3, rectangle = 4). The rasterizer may support different MSAA including 1x, 2x, 4x, 8x and 16x. The circuits of one embodiment are capable of calculating 256 samples/clock. Based on the MSAA and the list of partial and fully covered spans, the rasterizer determines the list of lit and unlit samples.

[00119] In one embodiment of the invention, the windower analyzes only the partial spans and does it in a parallel way. **Figure 15** illustrates a block diagram of one embodiment of the invention. The first stage comprises a set of span select modules 1500-1503 to select a different set of spans for each edge. For example, span select module 1500 selects spans for edge 0; span select module 1501 selects spans for edge 1; span select module 1502 selects spans for edge 2; and span select module 1503 selects spans for edge 3. In one embodiment, each span select module 1500-1503 selects the set of partial spans having both lit and unlit samples (e.g., ignoring fully lit and void spans). In one embodiment, the span select modules 1500-1503 evaluate all the 4x4 spans in parallel.

[00120] Once the spans have been selected by the span select modules 1500-1503, the spans for each edge are passed to edge determination modules 1510-1513, respectively, which determine each edge and generate a mask for each edge. For example, edge determination module 1510 determines edge 0 and generates mask 0; edge determination module 1511 determines edge 1 and generates mask 1; edge determination module 1512 determines edge 2 and generates mask 2; and edge determination module 1513 determines edge 3 (if there is an edge 3) and generates mask 3. In one embodiment, each edge determination modules 1510-1513 operate on up to 16 spans x 16 pixels (256 samples) per clock to determine the location of each edge. In one embodiment, the mask determination modules 1510-1513 may implement different MSAA techniques including 1x, 2x, 4x, 8x and 16x (each of which may require different

numbers of pixels for each edge). Based on the MSAA and the list of partial and fully covered spans, the edge determination modules 1510-1513 determine the list of lit and unlit samples for each edge. A final mask determination module 1520 then merges all masks in the last stage to obtain the final mask in which all edges have been identified (i.e., where all of the lit samples of the primitive are known).

[00121] The new techniques described above use 7 pipe stages to produce the final mask, while existing algorithms require 21 pipe stages. Table 1 shows the improvement in performance between the old and the new techniques, highlighting the pixels per clock processed using each MSAA. The table illustrates that the new techniques result in a 64x improvement for some cases.

	Old Techniques Pixels/Clock	New Techniques Pixels/Clock
1 x MSAA	16	256
2 x MSAA	16	128
4 x MSAA	16	64
8 x MSAA	8	32
16 x MSAA	4	16
Conservative Rast	16	128
Diamond Line	4	256
Wide Lines	16	256

TABLE 1

[00122] With respect to power usage, the distribution for 16x16, 8x8, 4x4 and pixels are illustrated in Table 2 below. Only 5% of the 16x16 are void or fully covered. Out of the 16x16 partial, 55% of the 8x8 are partial and then out of the partial 8x8, 51% of the 4x4 are partial. This means that the average triangle will have a partial 16x16, then there will be 2 partial 8x8 and in

each of the partial 8x8 there will be 2 partial 4x4. On average there will be 4 iterations to produce the 4x4 result. Based on Table 2 the usage of the different stages are: the 16x16 is used 94% of the time, then the 8x8 is used 55% of the time and 4x4 is used 51% of the time.

	16x16	8x8	4x4	Pixel
Void/Fully	5.15%	44.24%	48.27%	89.32%
Partial	94.85%	55.76%	51.73%	10.68%
1 Edge	94.85%	55.76%	51.73%	10.68%
2 Edges	82.39%	51.44%	49.43%	10.58%
3 Edges	12.46%	4.31%	2.31%	0.10%

TABLE 2

[00123] Table 3 illustrates results using the new parallel techniques described herein. It evaluates all 16 4x4 spans, to see if they are fully covered, partial covered or void. The fully covered and void cases do not require further calculations and the masks are known. Only the partial 4x4 require more calculations. Table 3 shows the average distribution of the 4x4 spans; it uses the same inputs of Table 2. Table 3 shows that 72% of the 4x4 the mask is known; 10% will be required to calculate two edges; and only 1.2% will be required to calculate 3 edges. Thus, edge 1 is on 28% of the time, edge 2 is on 10% and edge 3 is on only 1.2% of the time.

	16x16	4x4
Void/Fully	5.15%	72.64%
Partial	94.85%	27.36%
1 Edge	94.85%	27.36%
2 Edges	63.39%	10.16%
3 Edges	12.46%	1.22%

TABLE 3

[00124] **Figure 16** illustrates additional details of one embodiment of the invention which includes the span select modules 1500-1503 and edge determination modules 1510-1513 described above. In addition, bounding box logic 1605 contains the information about the bounding box rectangle that contains the primitive; anything outside of the bounding box is void (exemplary bounding boxes 1205, 1305 are illustrated in **Figures 12** and **13**, respectively). In one embodiment, the bounding box logic 1605 determines the spans that are out of the bounding box and mark them as void. The span select modules 1500-1503 determine the partial, void and fully covered spans. In one embodiment, they use the same equation:

$$L_0(x,y) = Lo_{16x16} + \Delta L_x + \Delta L_y \geq 0.$$

In one embodiment, the span select modules 1500-1503 check the 4 corners of the 4x4 spans and use the symmetry to reduce the number of compare circuits required for rasterization. Span, Edge, and MSAA control logic 1600 implement span/pixel center calculator 1602 which calculates the values for nLx + mLy and module 1606 determines values for Lo_{16x16}+ Lo_{4x4} + nLx+mLy according to a specified MSAA. The output of module 1606 is used to control each of the edge determination modules 1510-1513.

[00125] **Figure 17** illustrates the symmetry that exists in a 16x16 block and which is utilized to improve efficiency in one embodiment of the invention. In particular, only 12 compares are required per edge to evaluate the entire 16x16 because of the following known equalities between Lo values:

[00126] $Lo_0 = -Lo_{24} = -8Lx - 8Ly$

[00127] $Lo_1 = -Lo_{23} = -4Lx - 8Ly$

[00128] $Lo_2 = -Lo_{22} = -8Ly$

[00129] $Lo_3 = -Lo_{21} = 4Lx - 8Ly$

[00130] $Lo_4 = -Lo_{20} = 8Lx - 8Ly$

[00131] $Lo_5 = -Lo_{19} = -8Lx - 4Ly$

[00132] $Lo_6 = -Lo_{18} = Lo_0/2$

[00133] $Lo_7 = -Lo_{17} = -4Ly$

[00134] $Lo_8 = -Lo_{16} = -Lo_4/2$

[00135] $Lo_9 = -Lo_{15} = 8Lx - 4Ly$

[00136] $Lo_{10} = -Lo_{14} = -8Lx$

[00137] $Lo_{11} = -Lo_{13} = -4Lx$

[00138] $Lo_{12} = 0$

[00139] Thus, $L_0(x,y) = Lo_{16x16} - 8Lx - 8Ly \geq 0$ for both Lo_0 and $-Lo_{24}$. The values for -8Lx and -8Ly could be positive or negative. If both are positive the corner is lit and if both are negative the corner is not lit, and there is no need to compare. But the complementary corner will have different signs and the two values need to be compared. This reduces the number of compares in half.

[00140] Then, in one embodiment, to calculate the Lo for a specific sample the equation is:

$$L_0(x,y) = Lo_{16x16} + Lo_{4x4} + \Delta L_{MSAA} + \Delta Lo_{pixel} > 0.$$

Where Lo_{16x16} is an input of the block and is the Lo value for the center of the 16x16 block. The Lo_{4x4} value is the Lo from the 16x16 center to the 4x4 center. The Lo_{pixel} value is the Lo from the 4x4 center to the pixel center. The L_{MSAA} value is the distance from the pixel center to the sample.

[00141] **Figures 18A-B** illustrate additional details of a circuit for performing the foregoing calculations. In particular, circuit block 1801 performs a plurality of addition and multiplication operations to arrive at the values of nLx + mLy. A first set of adders 1802 then adds these results to Lo_{16x16} to arrive at the result: Lo_{16x16} + nLx + mLy. A second set of adders 1803 then adds these results to Lo_{4x4} to arrive at the result: $Lo_{16x16} + Lo_{4x4}$ + nLx + mLy. A set of 64 comparators 1804 then perform comparisons with these results and one of the Lo_{pixel} values (Lx+Ly, Lx+3Ly, 3Lx+Ly, Lx-Ly, Lx-3Ly, 3Lx-Ly) to determine whether $Lo_{16x16} + Lo_{4x4} + Lo_{pixel}$ + nLx + mLy > 0.

[00142] **Figure 19** shows that the Lo_{4x4} symmetry which is used to reduce the number of multipliers needed to calculate all 16 values. In particular, because of the symmetry, the following equalities may be utilized (where Lo_{4x4-i} is the 4x4 center Lo, relative to the Lo_{16x16}):

[00143] $Lo_{4x4-o} = -Lo_{4x4-15} = -6Lx - 6Ly$

[00144] $Lo_{4x4-1} = -Lo_{4x4-14} = -2Lx - 6Ly$

[00145] $Lo_{4x4-2} = -Lo_{4x4-13} = -6Ly - 2Ly$

[00146] $Lo_{4x4-3} = -Lo_{4x4-12} = L4x4\text{-}o / 2$

[00147] $Lo_{4x4-4} = -Lo_{4x4-11} = 2Lx - 6Ly$

[00148] $Lo_{4x4-5} = -Lo_{4x4-10} = 6Lx - 6Ly$

[00149] $Lo_{4x4-6} = -Lo_{4x4-9} = L4x4\text{-}5 /2$

[00150] $Lo_{4x4-7} = -Lo_{4x4-8} = 6Lx - 2Ly$

[00151] **Figure 20** shows another property which is that $Lo_{pixel} = Lo_{4x4}/4$. **Figure 18A** illustrates how these values are used for Lo_{pixel} as input to the comparators 1804. In addition, these values may be used inputs to the adders for the equation $Lo_{16x16} + \Delta Lo_{4x4} + \Delta L_{MSAA} > \Delta L0_{Pixel}$. The same symmetry property that is used in the first stage to calculate the partial spans

may be used to reduce the number of comparators to 8. Thus, **Figure 18A** illustrates that only 8 columns of comparators 1804 may be used to arrive at the 16 pixel values. Note that while 64 comparators 1804 are illustrated in **Figure 18A**, one set of 8 comparators may be used 8 times in succession to perform the calculations described herein.

[00152] In one embodiment, the final mask calculation block 1805 executes a logical AND to all the outputs and generates the final mask which may be stored in a register 1806 as illustrated in **Figure 18A**.

[00153] A method in accordance with one embodiment of the invention is illustrated in **Figure 21**. The method may be implemented with the context of the architectures described above, but is not limited to any particular architecture.

[00154] At 2101, a plurality of pixel blocks associated with edges of a primitive are selected based on the pixel blocks having samples which are both inside and outside of the primitive. As mentioned above, for example, in one embodiment 4x4 spans which are fully covered and void may be identified and ignored. At 2102, the samples within each selected plurality of pixel blocks are analyzed to generate data identifying each edge of the primitive. In the embodiments described above, for example, the specific location of each edge may be determined by edge determination logic 1510-1513. Finally, at 2103, the edge data determined for each edge of the primitive is combined and a final mask is generated representing the primitive.

[00155] 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.

[00156] 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. An apparatus comprising:

block selection logic to select a plurality of pixel blocks associated with edges of a primitive, the plurality of pixel blocks selected based on the pixel blocks having samples which are both inside and outside of the primitive; and

edge determination logic to analyze samples of the plurality of pixel blocks selected by the block selection logic and responsively generate data identifying each edge of the primitive; and

final mask determination logic to combine the data identifying each edge and generate a final mask representing the primitive.

2. The apparatus as in claim 1 wherein the pixel blocks selected by the block selection logic comprise 4x4 spans.

3. The apparatus as in claim 2 wherein the block selection logic comprises:

a first span selection module to select a first set of spans for a first edge;

a second span selection module to select a second set of spans for a second edge;

a third span selection module to select a third set of spans for a third edge; and

a fourth span selection module to select a fourth set of spans for a fourth edge.

4. The apparatus as in claim 3 wherein the primitive is encoded within a 16 x 16 pixel block comprising 16 4x4 spans from which the first, second, third, and fourth span selection logic is to select.

5. The apparatus as in claim 3 wherein the edge determination logic comprises:

a first edge determination module to determine a first edge associated with the first set of spans;

a second edge determination module to determine a second edge associated with the second set of spans;

a third edge determination module to determine a third edge associated with the third set of spans; and

a fourth edge determination module to determine a fourth edge associated with the fourth set of spans.

6. The apparatus as in claim 1 wherein the edge determination logic further comprises multi-sample anti-aliasing (MSAA) logic to perform MSAA when identifying each edge of the primitive.

7. The apparatus as in claim 6 wherein the MSAA logic is configurable to perform either 1x, 2x, 4x, 8x, or 16x MSAA.

8. The apparatus as in claim 1 further comprising:

bounding box logic to determine a bounding box for the primitive, the bounding box logic to generate data defining a rectangle which contains the primitive, wherein any samples outside of the bounding box are void.

9. The apparatus as in claim 1 wherein each pixel block comprises a 4x4 span and wherein the block selection logic is to determine void spans which contain no samples of the primitive, fully covered spans with all samples from the primitive, and partial spans which include samples which are inside the primitive and samples which are outside of the primitive.

10. The apparatus as in claim 1 wherein the block selection logic selects the plurality of pixel blocks from a 16x16 sample block, the block selection logic to utilize symmetry within the 16x16 sample block to reduce a number of compare operations required to select the plurality of pixel blocks and identify each edge.

11. The apparatus as in claim 10 wherein 12 compare operations are required per edge to identify edges for the 16x16 sample block.

12. The apparatus as in claim 11 wherein the edges are identified using the equation: $L_0(x,y) = Lo_{16x16} + \Delta L_x + \Delta L_y \geq 0$ which, if ≥ 0 identifies a sample within the primitive and, if not, identifies a sample outside of the primitive.

13. The apparatus as in claim 12 wherein the number of compare operations are to be reduced in accordance with the following equalities within the 16x16 sample block:

$Lo_0 = -Lo_{24} = -8Lx - 8Ly;$

$Lo_1 = -Lo_{23} = -4Lx - 8Ly;$

$Lo_2 = -Lo_{22} = -8Ly;$

$Lo_3 = -Lo_{21} = 4Lx - 8Ly;$

$Lo_4 = -Lo_{20} = 8Lx - 8Ly;$

$Lo_5 = -Lo_{19} = -8Lx - 4Ly;$

$Lo_6 = -Lo_{18} = Lo_0/2;$

$Lo_7 = -Lo_{17} = -4Ly;$

$Lo_8 = -Lo_{16} = -Lo_4/2;$

$Lo_9 = -Lo_{15} = 8Lx - 4Ly;$

$Lo_{10} = -Lo_{14} = -8Lx;$

$Lo_{11} = -Lo_{13} = -4Lx;$ and

$Lo_{12} = 0.$

14. The apparatus as in claim 13 wherein the following equation is to be used to determine Lo to a specific sample $L_0(x,y) = Lo_{16x16} + Lo_{4x4} + \Delta L_{MSAA} + \Delta Lo_{pixel} > 0$, where Lo_{16x16} is the Lo value for the center of the 16x16 sample block; the Lo_{4x4} value is the Lo from the 16x16 center to a 4x4 span center; the Lo_{pixel} value is the Lo from the 4x4 span center to a pixel center; and the L_{MSAA} value is a distance from the pixel center to a sample.

15. A method comprising:

selecting a plurality of pixel blocks associated with edges of a primitive, the plurality of pixel blocks selected based on the pixel blocks having samples which are both inside and outside of the primitive; and

analyzing the samples within each selected plurality of pixel blocks to generate data identifying each edge of the primitive; and

combining the data identifying each edge and generating a final mask representing the primitive.

16. The method as in claim 15 wherein the pixel blocks selected by the block selection logic comprise 4x4 spans.

17. The method as in claim 16 wherein selecting further comprises:

selecting a first set of spans for a first edge;

selecting a second set of spans for a second edge;

selecting a third set of spans for a third edge; and

selecting a fourth set of spans for a fourth edge.

18. The method as in claim 17 wherein the primitive is encoded within a 16 x 16 pixel block comprising 16 4x4 spans from which the first, second, third, and fourth sets of spans are to be selected.

19. The method as in claim 17 wherein analyzing the samples and generating data identifying each edge of the primitive comprises:

determining a first edge associated with the first set of spans;

determining a second edge associated with the second set of spans;

determining a third edge associated with the third set of spans; and

determining a fourth edge associated with the fourth set of spans.

20. The method as in claim 15 further comprising:

performing multi-sample anti-aliasing (MSAA) when generating the data identifying each edge of the primitive.

21. The method as in claim 20 further comprising:

performing either 1x, 2x, 4x, 8x, or 16x MSAA.

22. The method as in claim 15 further comprising:

determining a bounding box for the primitive, the bounding box logic to generate data defining a rectangle which contains the primitive, wherein any samples outside of the bounding box are void.

23. The method as in claim 15 wherein each pixel block comprises a 4x4 span, the method further comprising:

determining void spans which contain no samples of the primitive, fully covered spans with all samples from the primitive, and partial spans which include samples which are inside the primitive and samples which are outside of the primitive, the partial spans comprising the plurality of pixel blocks selected based on the pixel blocks having samples which are both inside and outside of the primitive.

24. The method as in claim 15 wherein the plurality of pixel blocks are selected from a 16x16 sample block and wherein symmetry within the 16x16 sample block is utilized to reduce a number of compare operations required to select the plurality of pixel blocks and identify each edge.

25. The method as in claim 24 wherein 12 compare operations are required per edge to identify edges for the 16x16 sample block.

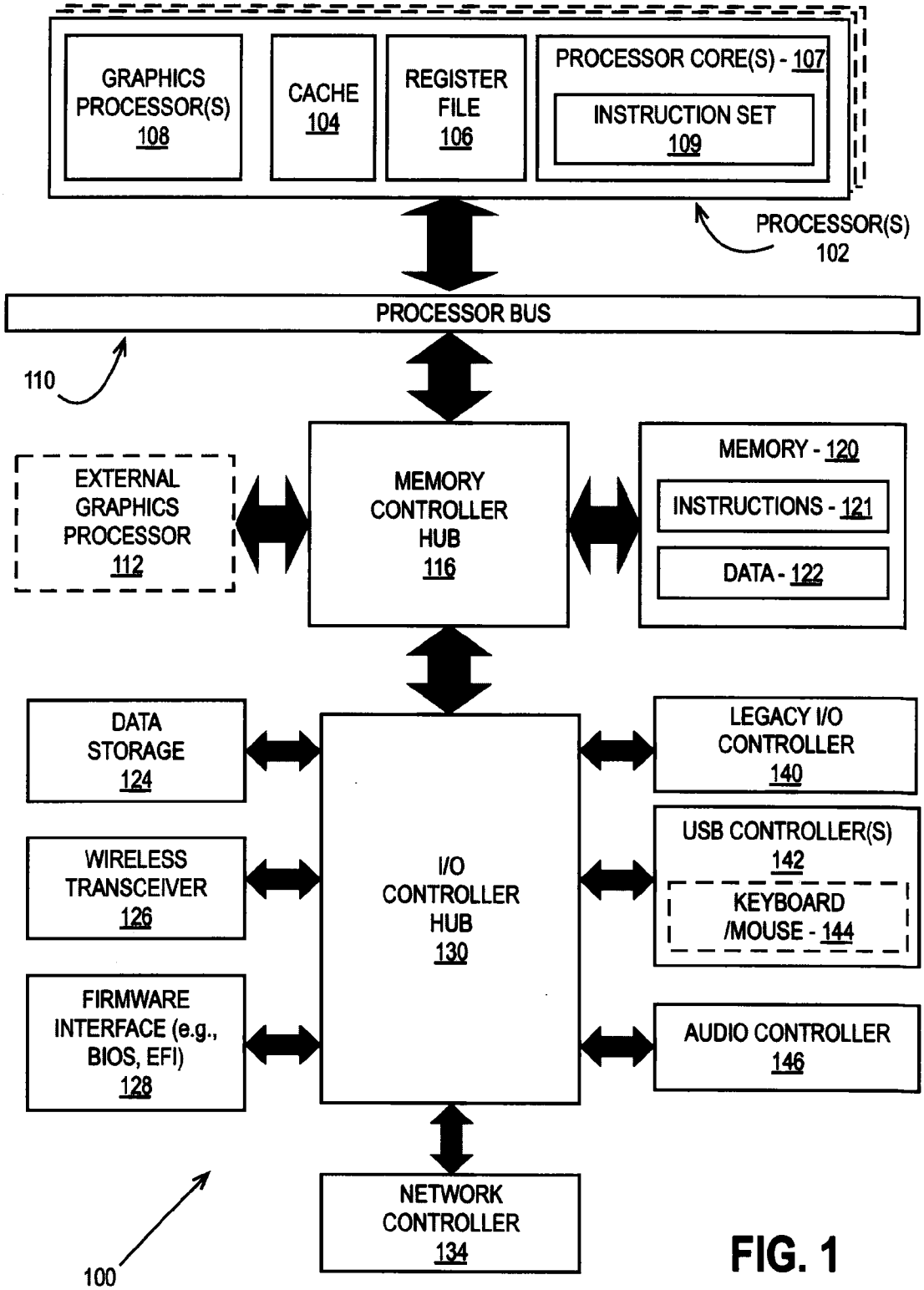

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) - 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY - 120
INSTRUCTIONS - 121
DATA - 122
DATA STORAGE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE (e.g., 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

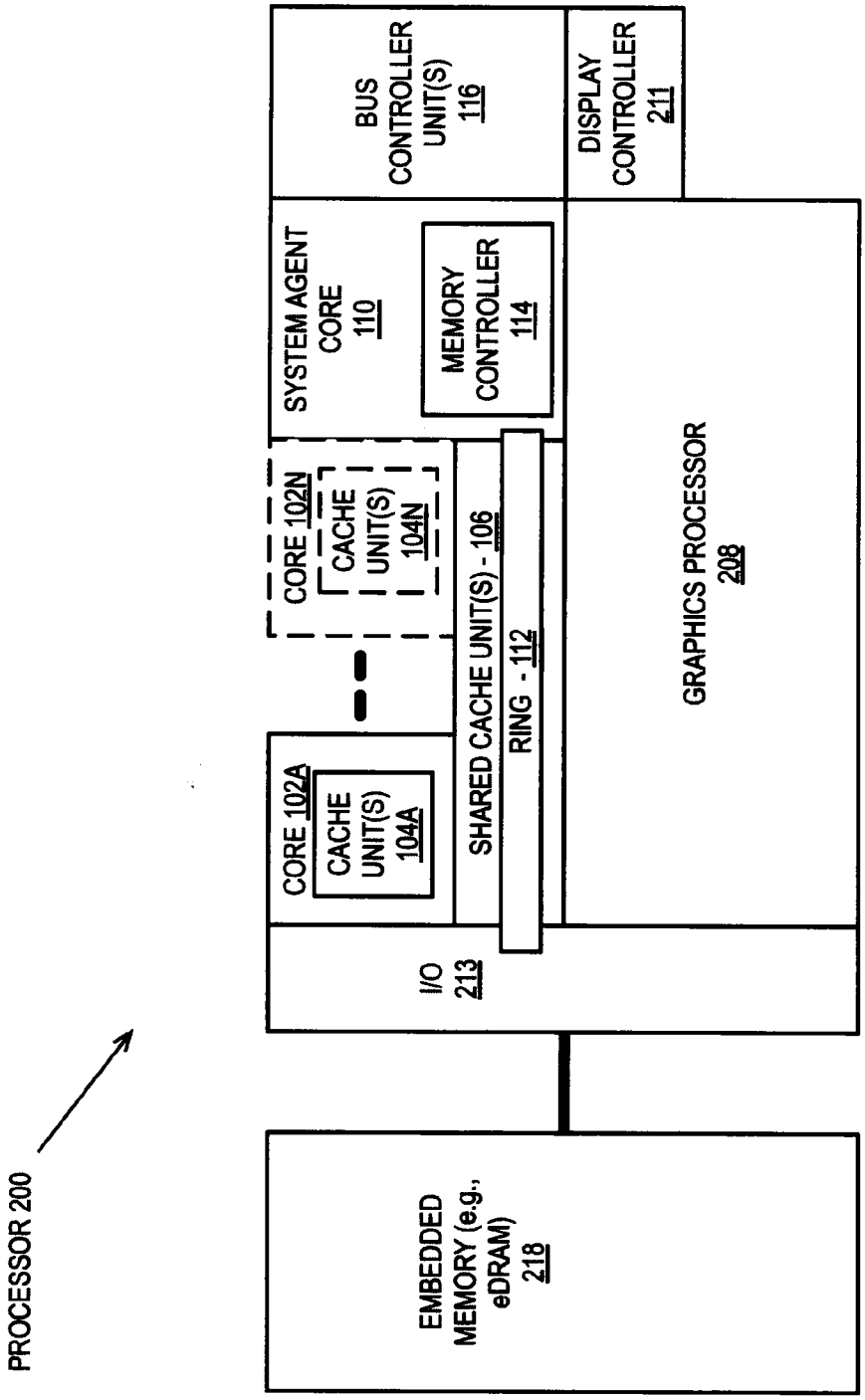

PROCESSOR 200
EMBEDDED MEMORY (e.g., eDRAM) 218
I/O 213
CORE 102A
CACHE UNIT(S) 104A
CORE 102N
CACHE UNIT(S) 104N
SHARED CACHE UNIT(S) - 106
RING - 112
SYSTEM AGENT CORE 110
MEMORY CONTROLLER 114
GRAPHICS PROCESSOR 208
BUS CONTROLLER UNIT(S) 116
DISPLAY CONTROLLER 211

FIG. 2

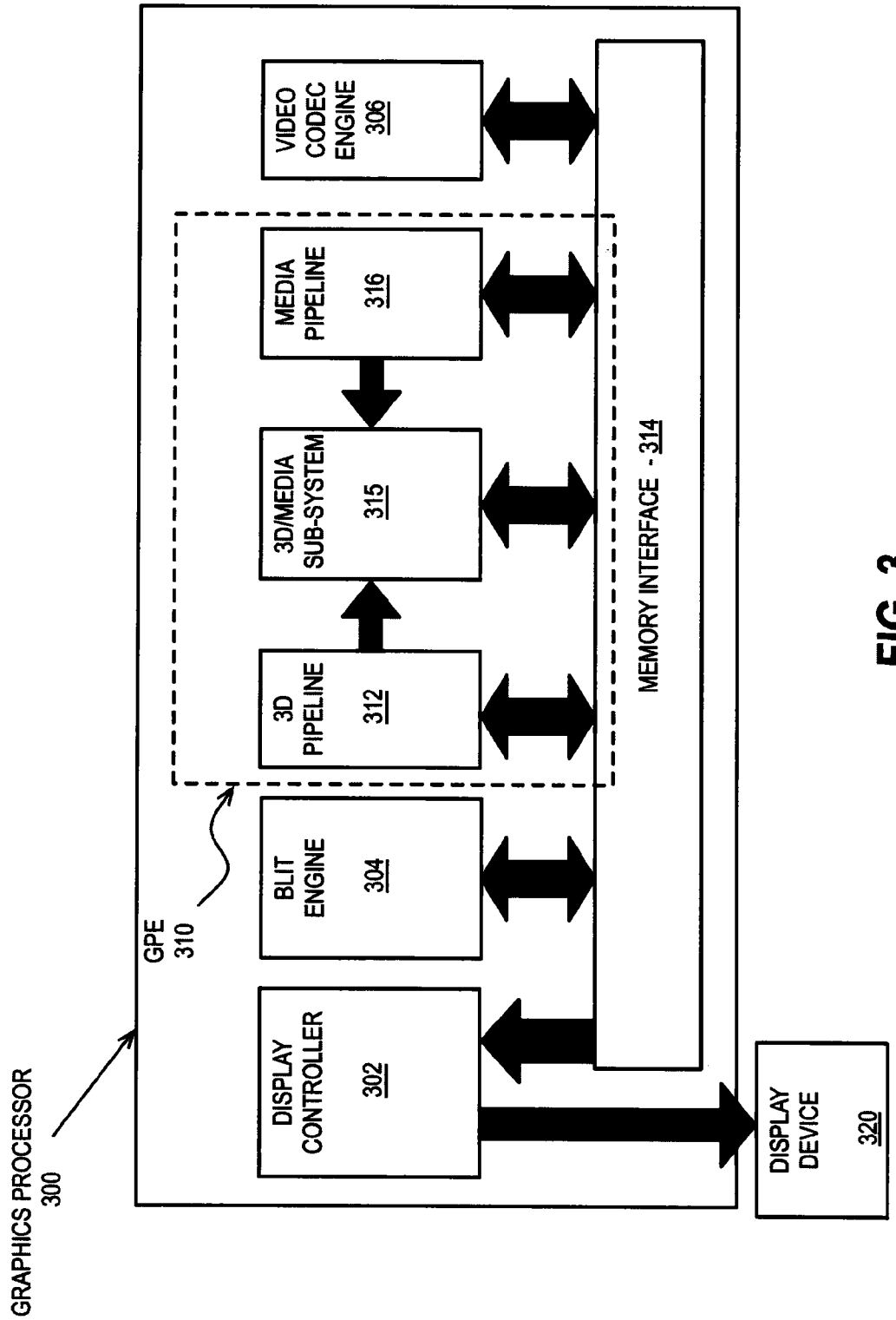

GRAPHICS PROCESSOR
300
GPE
310
DISPLAY CONTROLLER
302
BLIT ENGINE
304
3D PIPELINE
312
3D/MEDIA SUB-SYSTEM
315
MEDIA PIPELINE
316
VIDEO CODEC ENGINE
306
MEMORY INTERFACE - 314
DISPLAY DEVICE
320

FIG. 3

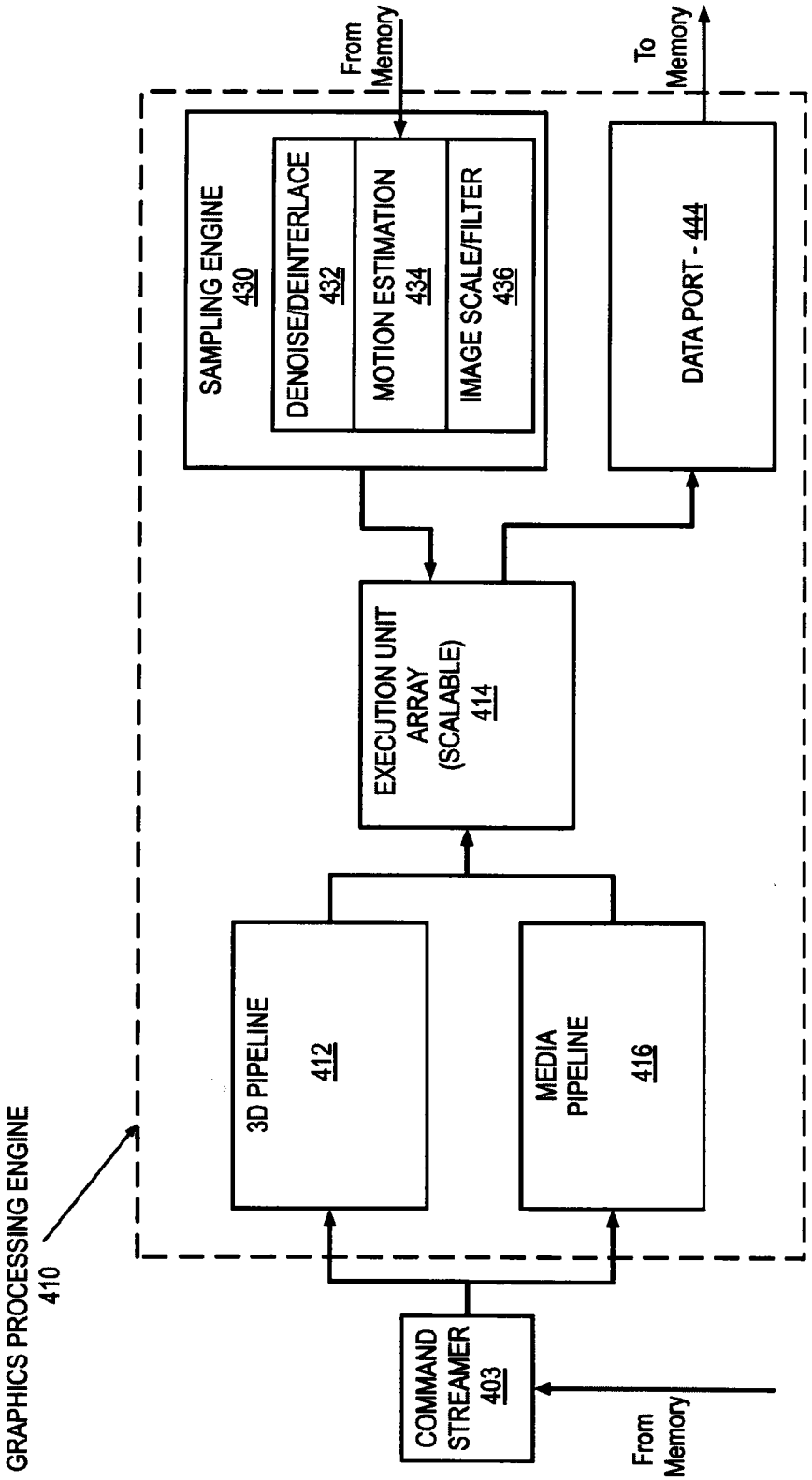

GRAPHICS PROCESSING ENGINE
410
From Memory
To Memory
SAMPLING ENGINE
430
DENOISE/DEINTERLACE
432
MOTION ESTIMATION
434
IMAGE SCALE/FILTER
436
DATA PORT - 444
EXECUTION UNIT ARRAY (SCALABLE)
414
3D PIPELINE
412
MEDIA PIPELINE
416
COMMAND STREAMER
403
From Memory

FIG. 4

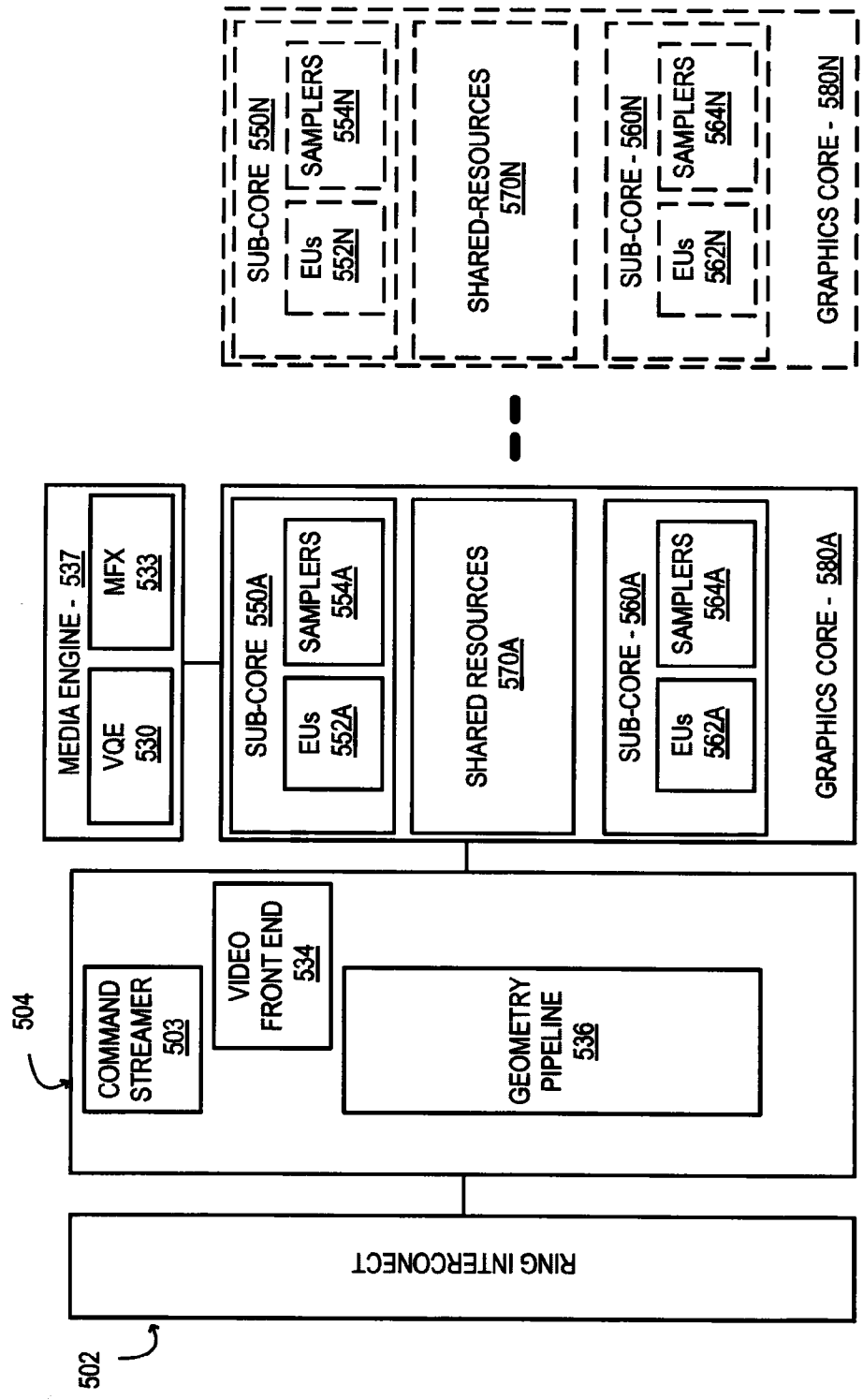

RING INTERCONECT
502
504
COMMAND STREAMER 503
VIDEO FRONT END 534
GEOMETRY PIPELINE 536
MEDIA ENGINE - 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED RESOURCES 570A
SUB-CORE - 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE - 580A
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N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

FIG. 5

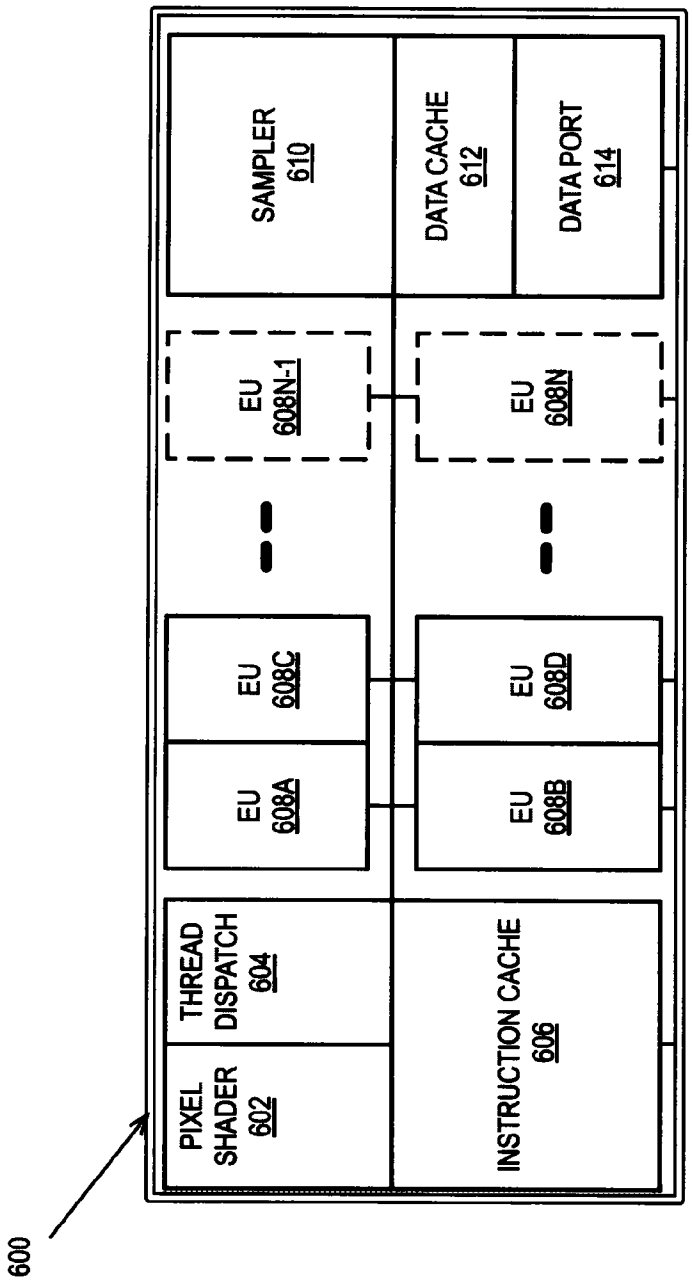

600
PIXEL SHADER 602
THREAD DISPATCH 604
INSTRUCTION CACHE 606
EU 608A
EU 608B
EU 608C
EU 608D
EU 608N-1
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 6

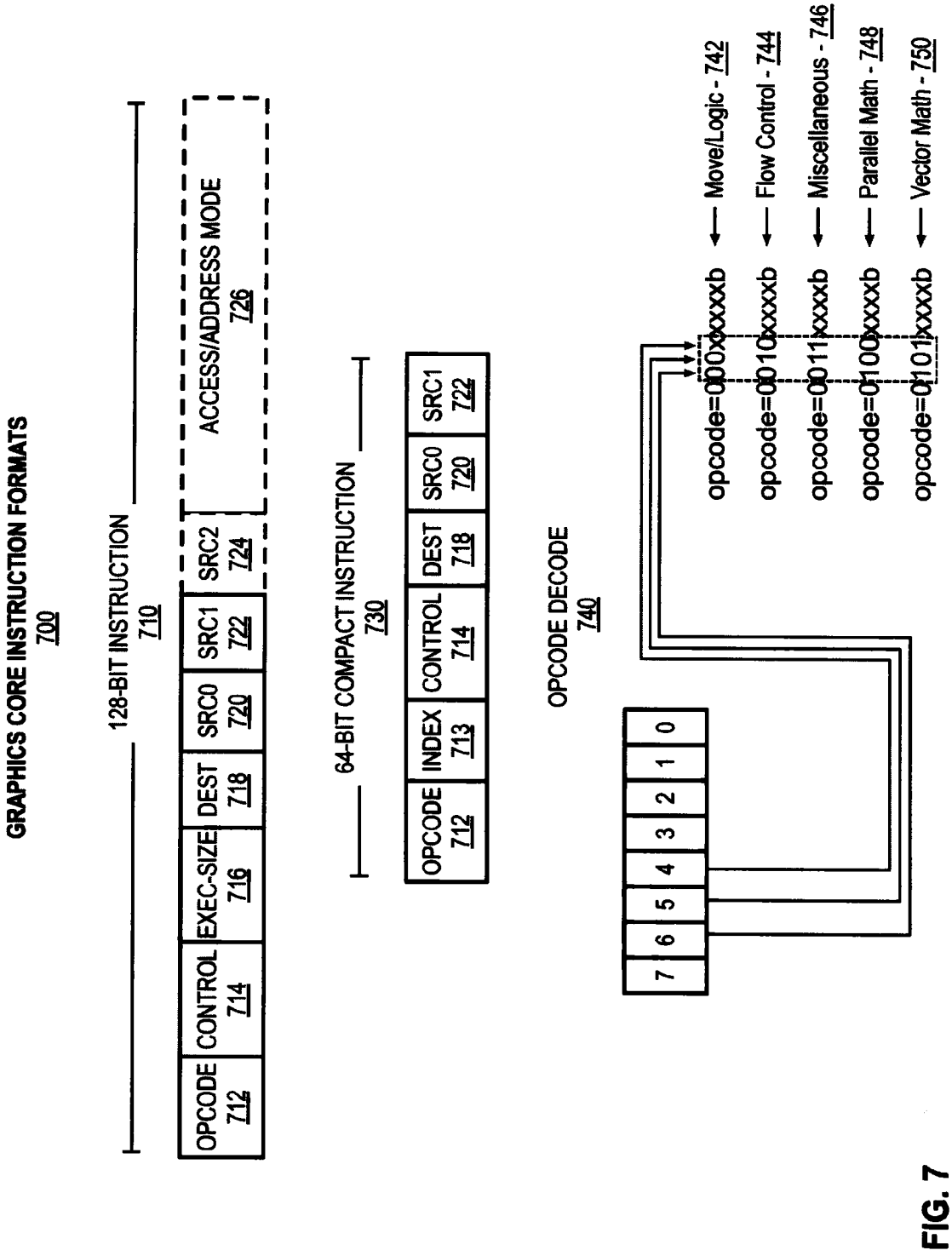

GRAPHICS CORE INSTRUCTION FORMATS
700
128-BIT INSTRUCTION
710
OPCODE 712
CONTROL 714
EXEC-SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS/ADDRESS MODE 726
64-BIT COMPACT INSTRUCTION
730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE
740
7 6 5 4 3 2 1 0
opcode=0000xxxxb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

FIG. 7

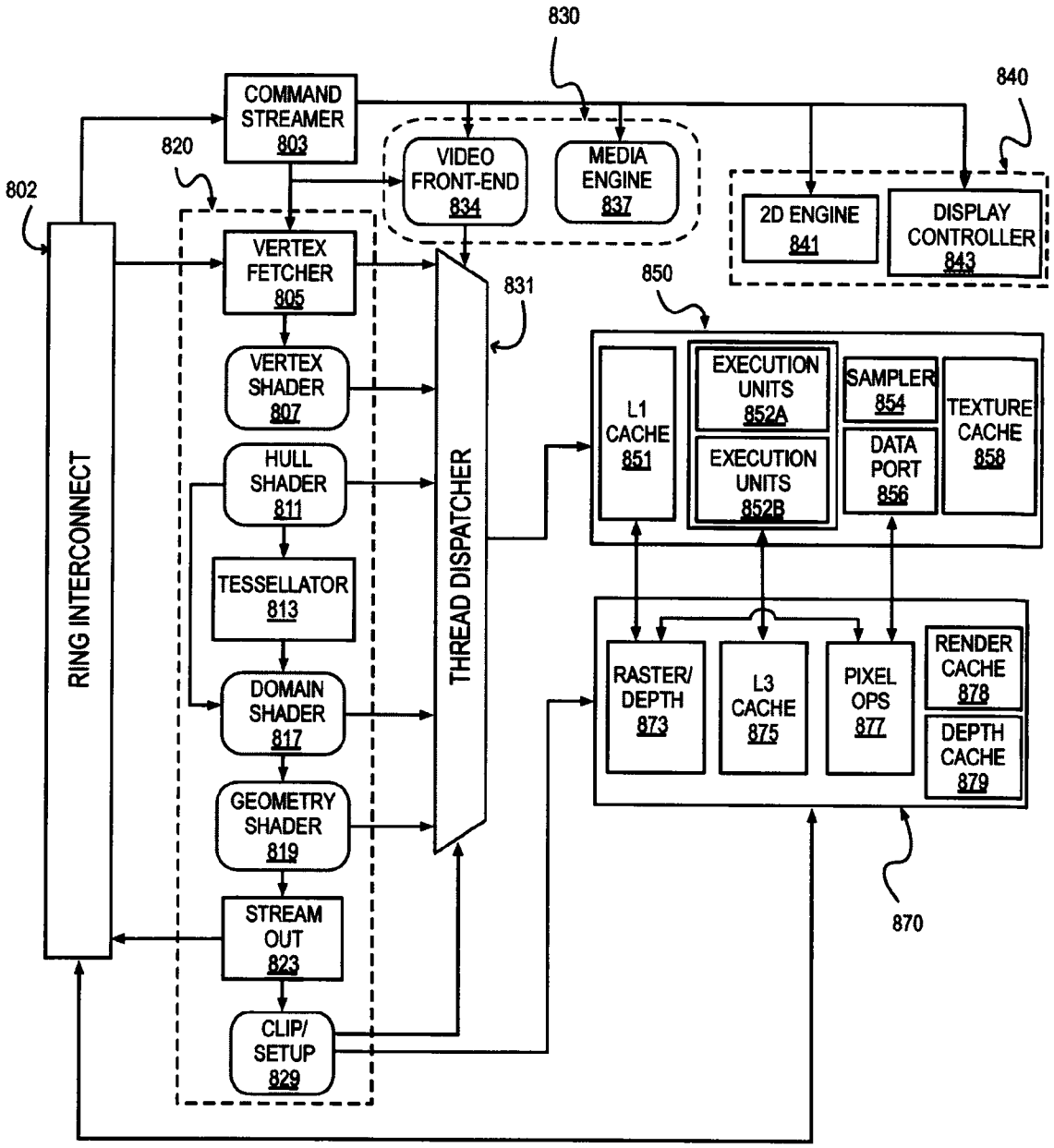

830
840
COMMAND STREAMER 803
820
802
VIDEO FRONT-END 834
MEDIA ENGINE 837
2D ENGINE 841
DISPLAY CONTROLLER 843
VERTEX FETCHER 805
831
850
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP/ SETUP 829
RING INTERCONNECT
THREAD DISPATCHER
L1 CACHE 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

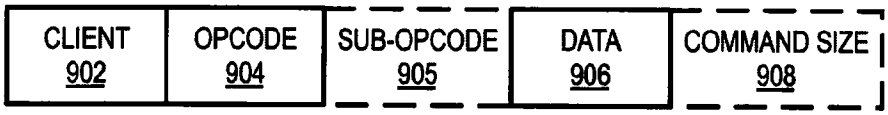
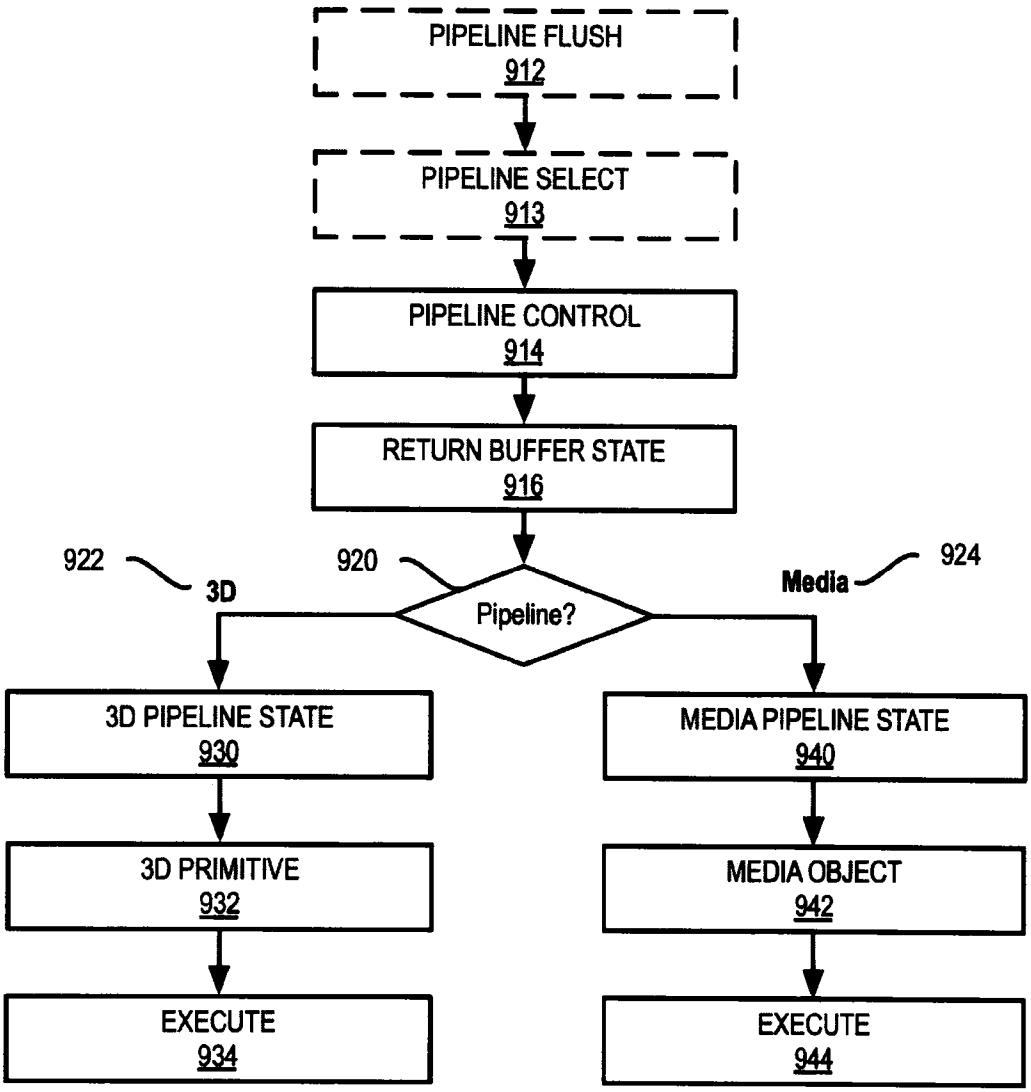


FIG. 9B

SAMPLE COMMAND SEQUENCE
910



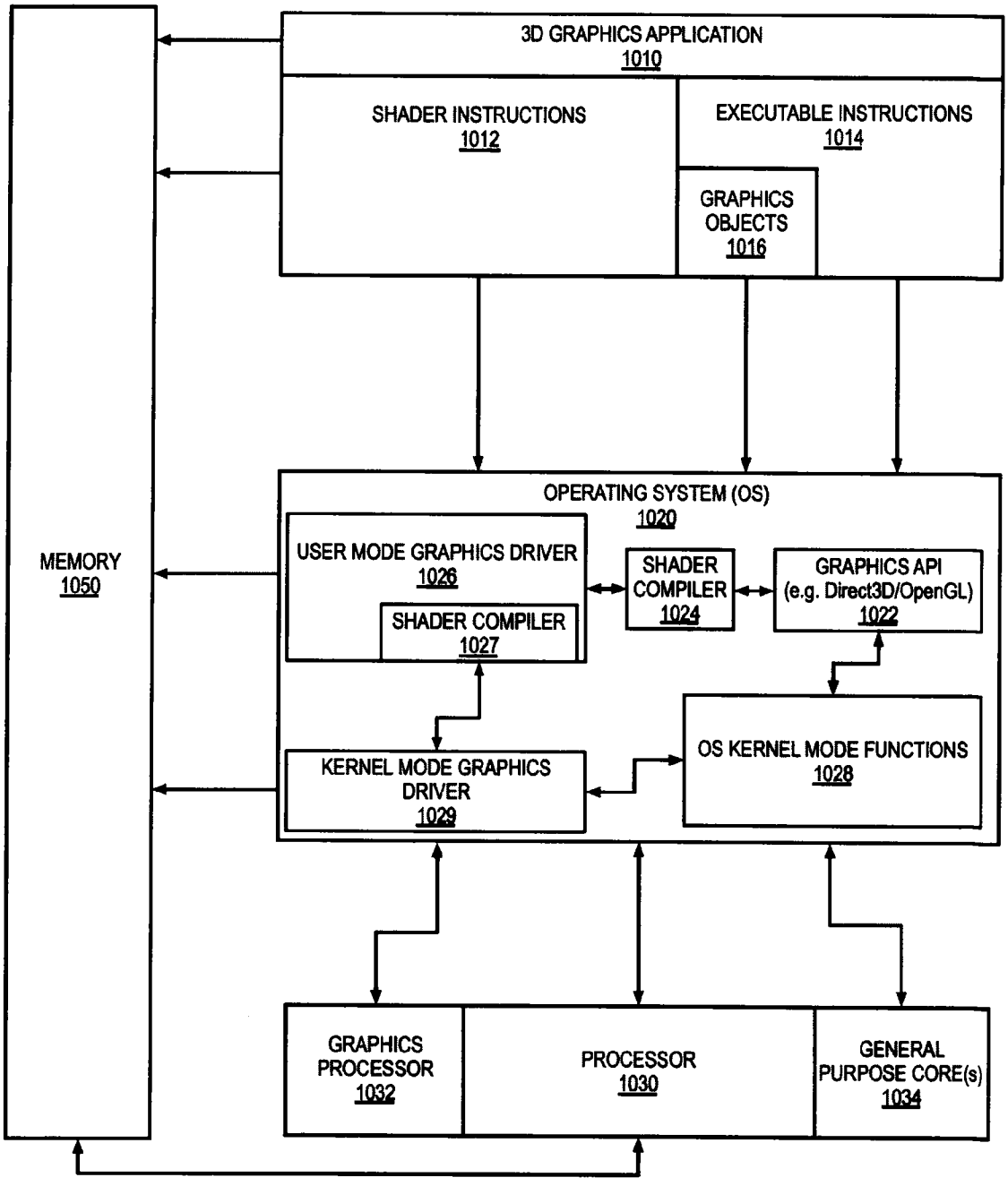

3D GRAPHICS APPLICATION
1010
SHADER INSTRUCTIONS
1012
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS OBJECTS
1016
MEMORY
1050
OPERATING SYSTEM (OS)
1020
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
SHADER COMPILER
1024
GRAPHICS API
(e.g. Direct3D/OpenGL)
1022
KERNEL MODE GRAPHICS DRIVER
1029
OS KERNEL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

FIG. 10

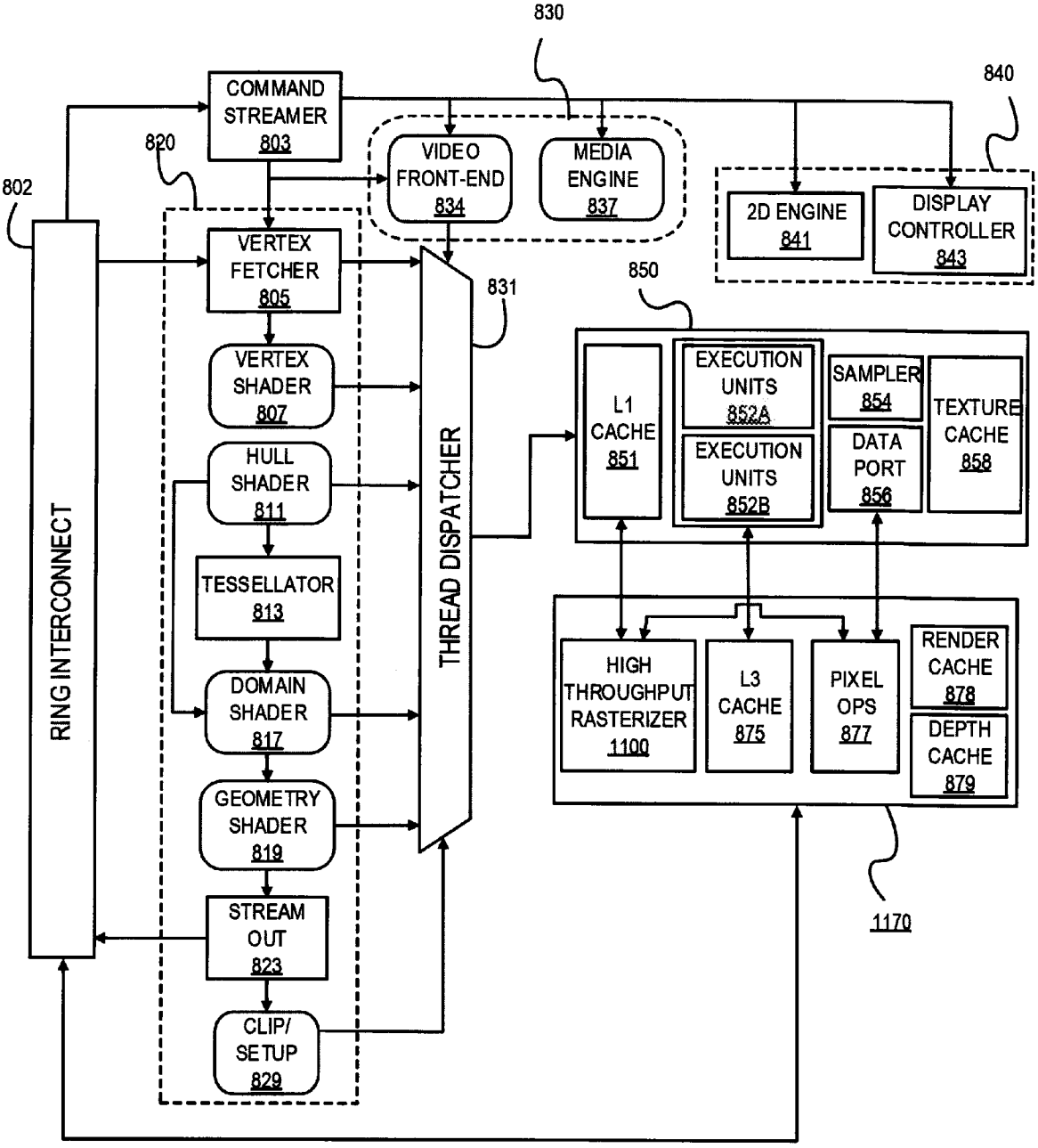

830
840
COMMAND STREAMER 803
820
802
VIDEO FRONT-END 834
MEDIA ENGINE 837
2D ENGINE 841
DISPLAY CONTROLLER 843
VERTEX FETCHER 805
831
850
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP/ SETUP 829
RING INTERCONNECT
THREAD DISPATCHER
L1 CACHE 851
EXECUTION UNITS 852A
EXECUTION UNITS 852B
SAMPLER 854
DATA PORT 856
TEXTURE CACHE 858
HIGH THROUGHPUT RASTERIZER 1100
L3 CACHE 875
PIXEL OPS 877
RENDER CACHE 878
DEPTH CACHE 879
1170

FIG. 11

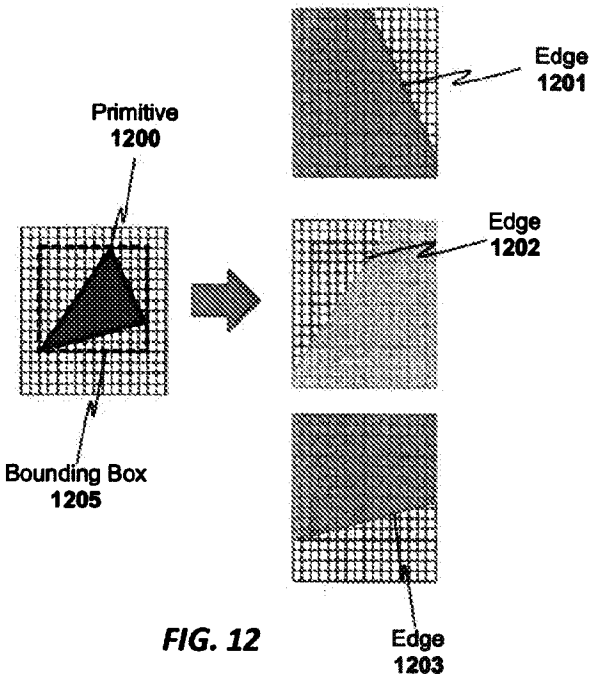

Primitive
1200
Bounding Box
1205
Edge
1201
Edge
1202
Edge
1203

FIG. 12

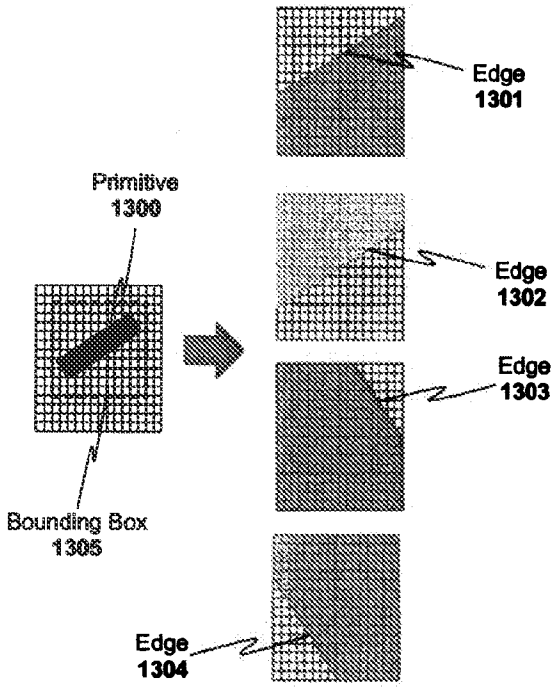

Edge
1301
Primitive
1300
Edge
1302
Edge
1303
Bounding Box
1305
Edge
1304

FIG. 13

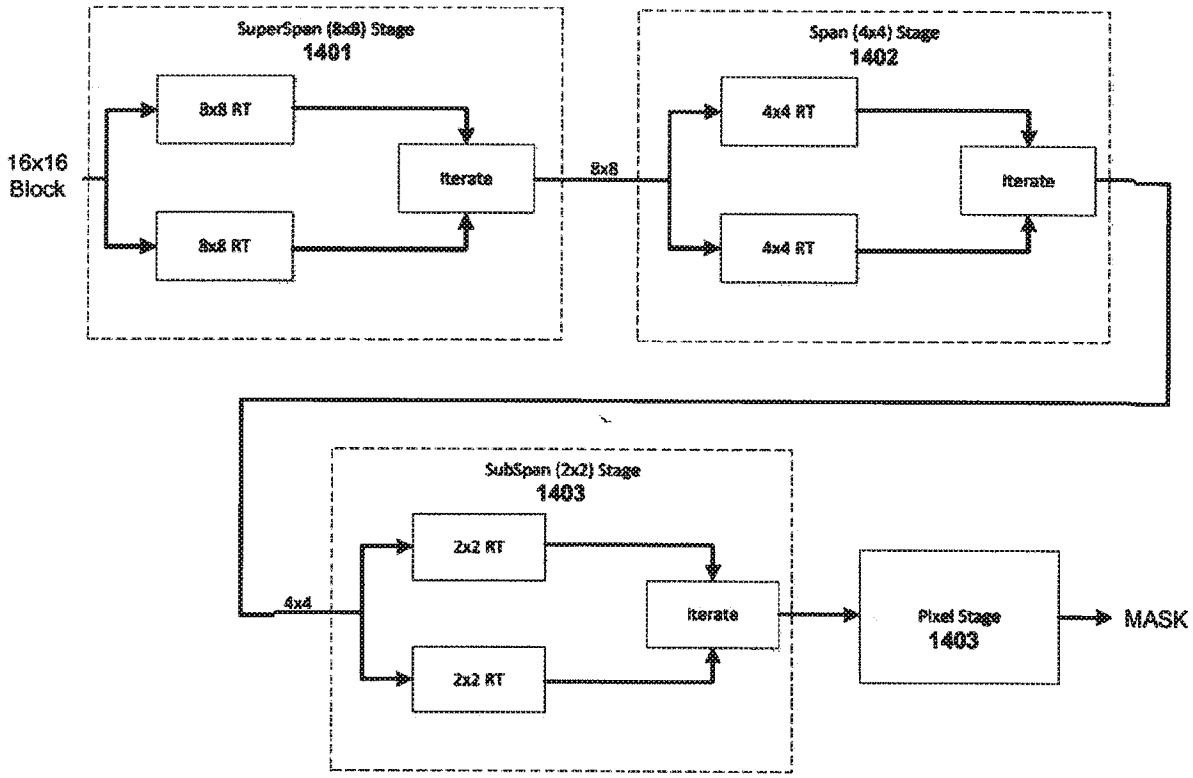

SuperSpan (8x8) Stage
1401
8x8 RT
8x8 RT
Iterate
16x16
Block
8x8
Span (4x4) Stage
1402
4x4 RT
4x4 RT
Iterate
SubSpan (2x2) Stage
1403
2x2 RT
2x2 RT
Iterate
4x4
Pixel Stage
1403
MASK

FIG. 14

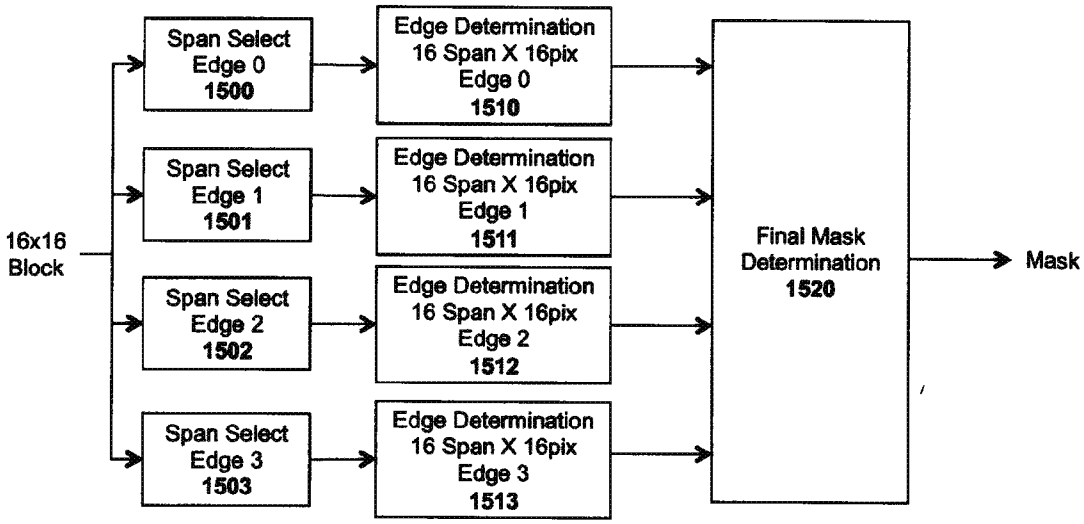

16x16
Block
Span Select
Edge 0
1500
Span Select
Edge 1
1501
Span Select
Edge 2
1502
Span Select
Edge 3
1503
Edge Determination
16 Span X 16pix
Edge 0
1510
Edge Determination
16 Span X 16pix
Edge 1
1511
Edge Determination
16 Span X 16pix
Edge 2
1512
Edge Determination
16 Span X 16pix
Edge 3
1513
Final Mask
Determination
1520
Mask

FIG. 15

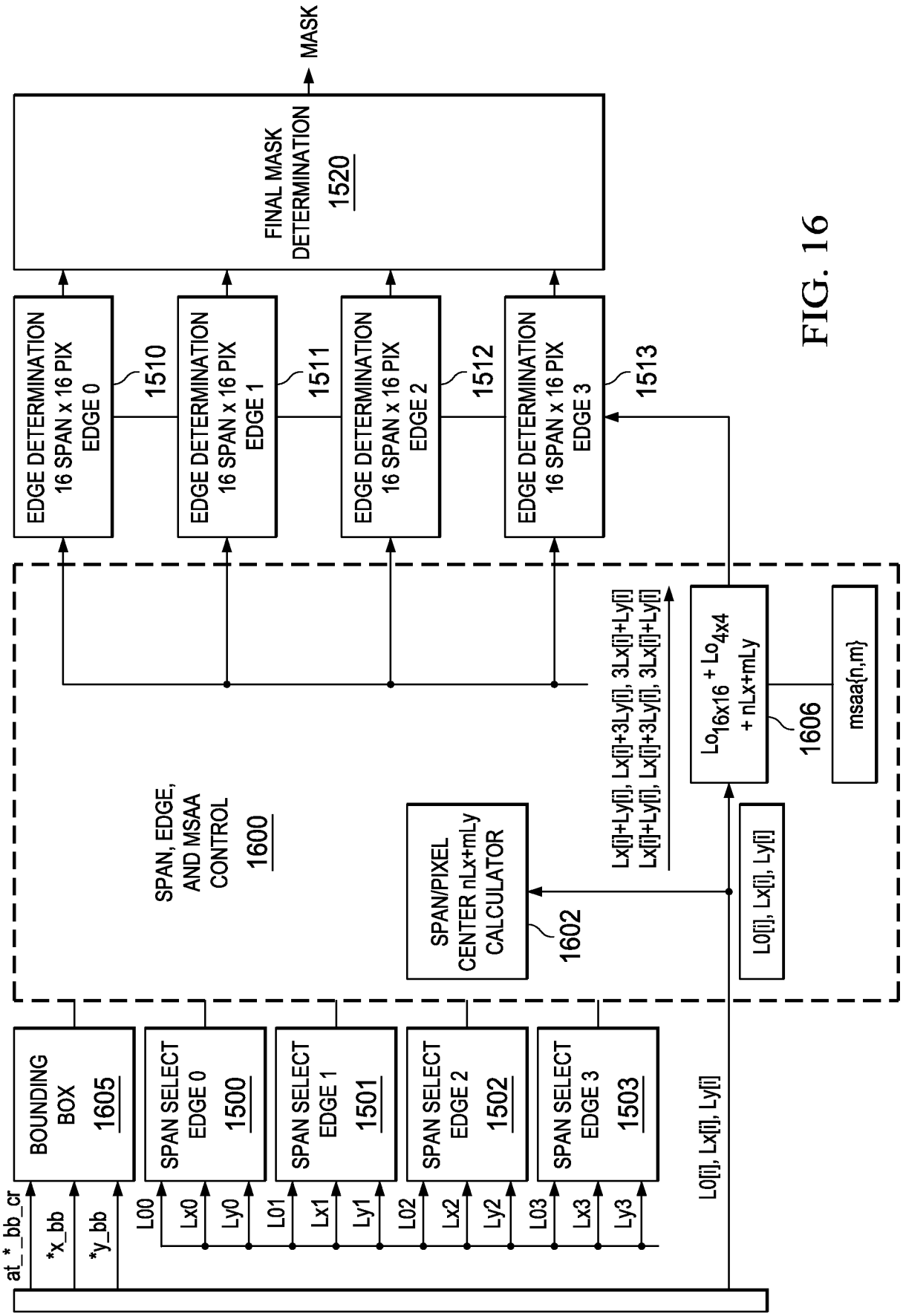
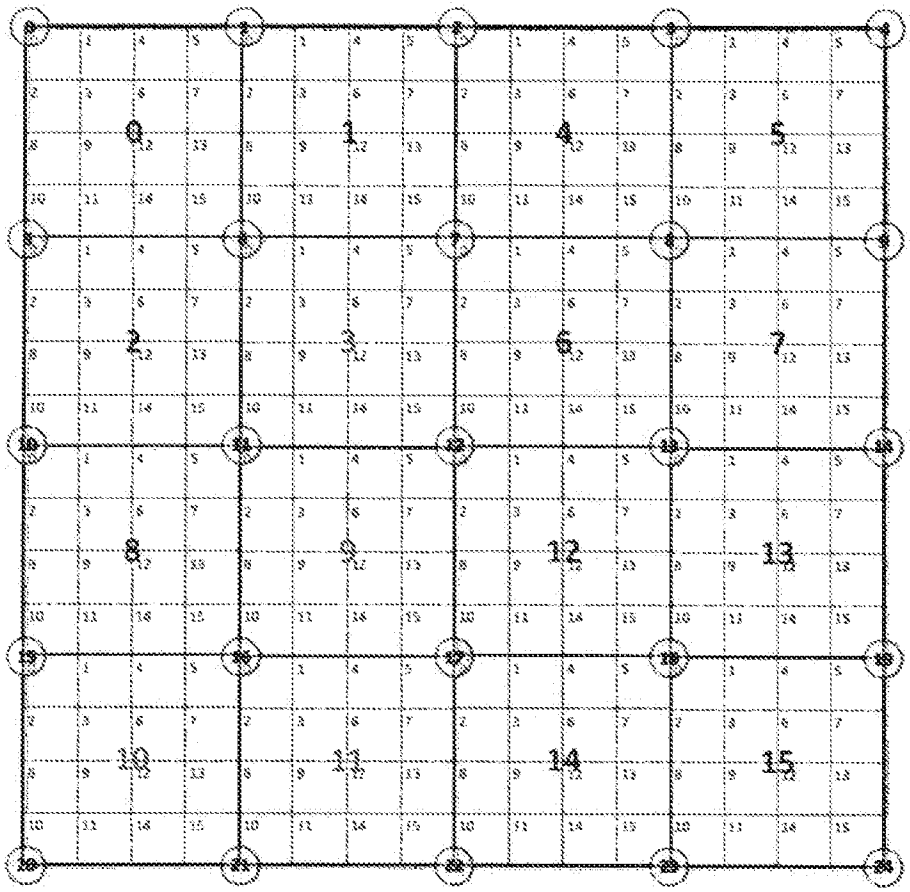

MASK
FINAL MASK DETERMINATION 1520
EDGE DETERMINATION 16 SPAN x 16 PIX EDGE 0
EDGE DETERMINATION 16 SPAN x 16 PIX EDGE 1
EDGE DETERMINATION 16 SPAN x 16 PIX EDGE 2
EDGE DETERMINATION 16 SPAN x 16 PIX EDGE 3
1510
1511
1512
1513
SPAN, EDGE, AND MSAA CONTROL 1600
SPAN/PIXEL CENTER nLx+mLy CALCULATOR 1602
Lx[i]+Ly[i], Lx[i]+3Ly[i], 3Lx[i]+Ly[i]
Lx[i]+Ly[i], Lx[i]+3Ly[i], 3Lx[i]+Ly[i]
Lo16x16 + Lo4x4 + nLx+mLy
1606
msaa{n,m}
L0[i], Lx[i], Ly[i]
BOUNDING BOX 1605
SPAN SELECT EDGE 0 1500
SPAN SELECT EDGE 1 1501
SPAN SELECT EDGE 2 1502
SPAN SELECT EDGE 3 1503
at_*_bb_cr
*x_bb
*y_bb
L00
Lx0
Ly0
L01
Lx1
Ly1
L02
Lx2
Ly2
L03
Lx3
Ly3
L0[i], Lx[i], Ly[i]

FIG. 16



$L_{O0} = -L_{O24} = -8L_x - 8L_y$
$L_{O1} = -L_{O23} = -4L_x - 8L_y$
$L_{O2} = -L_{O22} = -8L_y$
$L_{O3} = -L_{O21} = 4L_x - 8L_y$
$L_{O4} = -L_{O20} = 8L_x - 8L_y$
$L_{O5} = -L_{O19} = -8L_x - 4L_y$
$L_{O6} = -L_{O18} = L_{O0}/2$
$L_{O7} = -L_{O17} = -4L_y$
$L_{O8} = -L_{O16} = -L_{O4}/2$
$L_{O9} = -L_{O15} = 8L_x - 4L_y$
$L_{O10} = -L_{O14} = -8L_x$
$L_{O11} = -L_{O13} = -4L_x$
$L_{O12} = 0$

FIG. 17

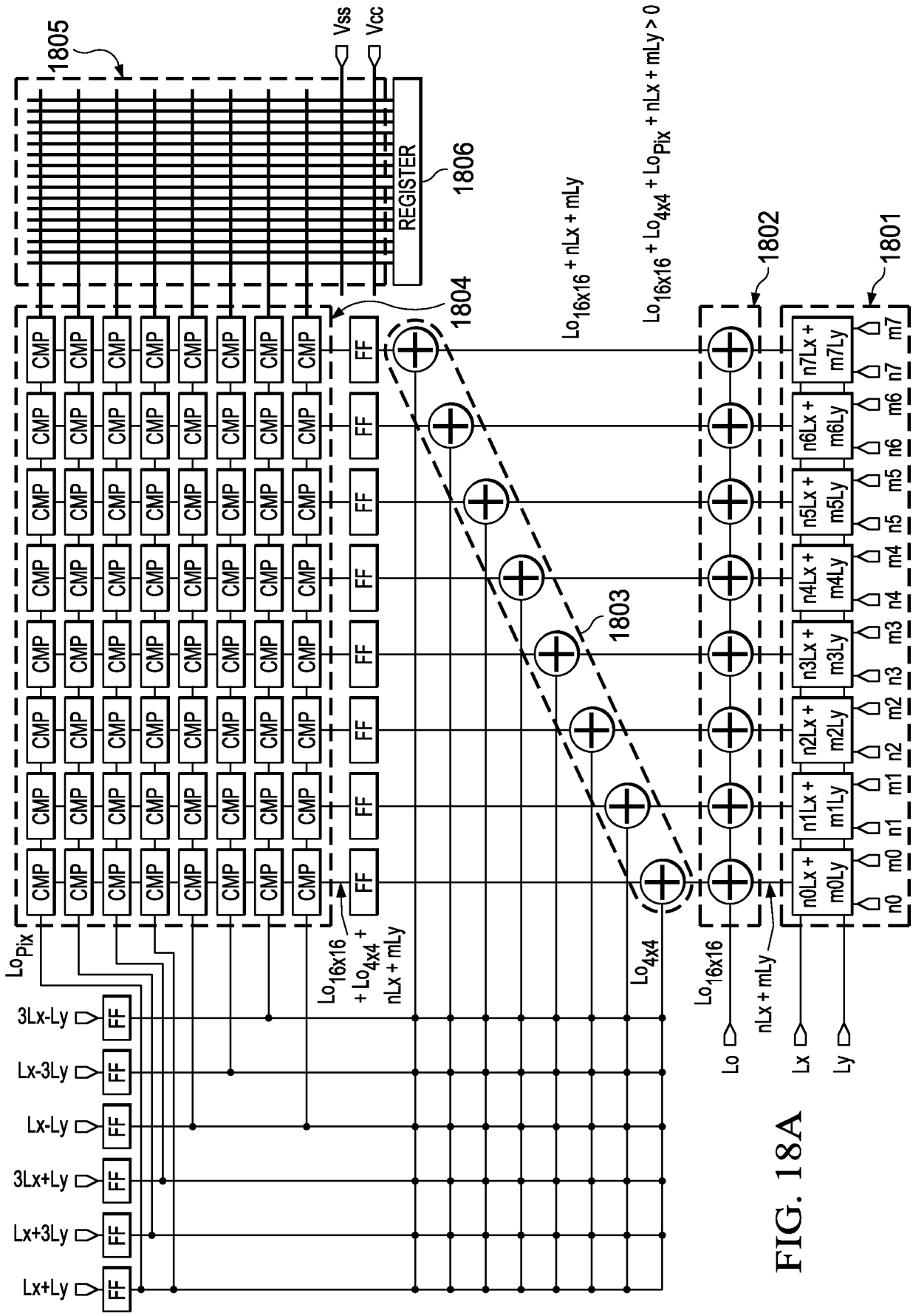

1805
Vss
Vcc
REGISTER
1806
1804
1803
1802
1801
CMP
FF
Lo_{Pix}
3Lx-Ly
Lx-3Ly
Lx-Ly
3Lx+Ly
Lx+3Ly
Lx+Ly
$Lo_{16x16} + Lo_{4x4} + nLx + mLy$
Lo_{4x4}
$Lo_{16x16} + nLx + mLy$
$Lo_{16x16} + Lo_{4x4} + Lo_{Pix} + nLx + mLy > 0$
Lo_{16x16}
$nLx + mLy$
Lo
Lx
Ly
n0Lx + m0Ly
n1Lx + m1Ly
n2Lx + m2Ly
n3Lx + m3Ly
n4Lx + m4Ly
n5Lx + m5Ly
n6Lx + m6Ly
n7Lx + m7Ly
n0 m0 n1 m1 n2 m2 n3 m3 n4 m4 n5 m5 n6 m6 n7 m7

FIG. 18A

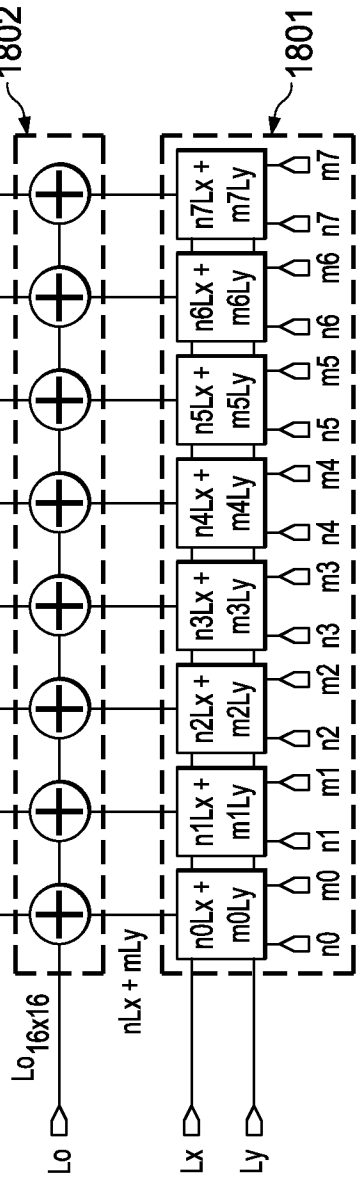
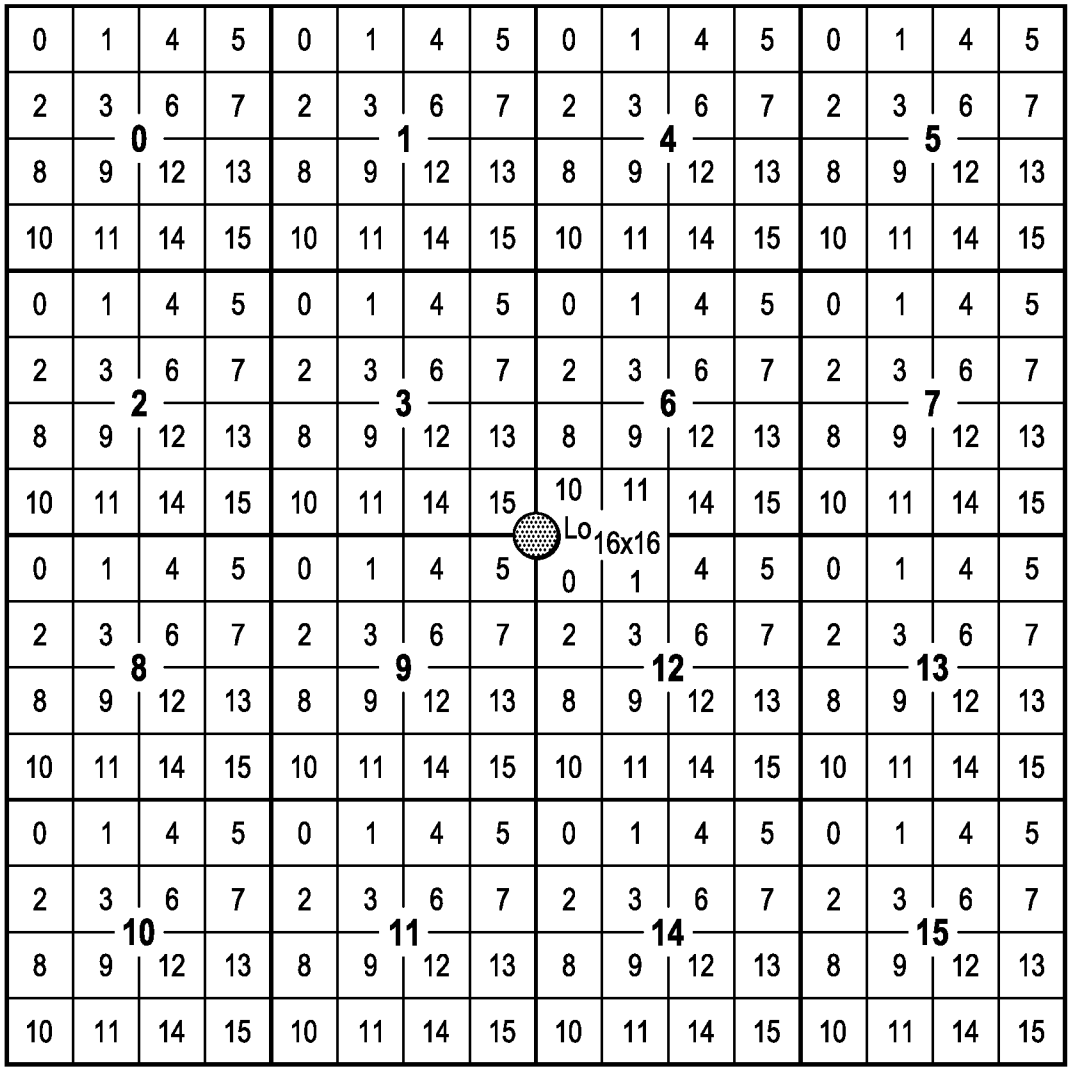

1802
1801
Lo16x16
nLx + mLy
Lo
Lx
Ly
n0Lx + m0Ly
n1Lx + m1Ly
n2Lx + m2Ly
n3Lx + m3Ly
n4Lx + m4Ly
n5Lx + m5Ly
n6Lx + m6Ly
n7Lx + m7Ly
n0 m0 n1 m1 n2 m2 n3 m3 n4 m4 n5 m5 n6 m6 n7 m7

FIG. 18B

Lo_{4x4-i} IS THE 4x4 CENTER Lo, RELATIVE TO THE Lo_{16x16}



$Lo_{4x4-0} = Lo_{4x4-15} = -6Lx - 6Ly$
$Lo_{4x4-1} = Lo_{4x4-14} = -2Lx - 6Ly$
$Lo_{4x4-2} = Lo_{4x4-13} = -6Ly - 2Ly$
$Lo_{4x4-3} = Lo_{4x4-12} = L_{4x4-0} / 2$
$Lo_{4x4-4} = Lo_{4x4-11} = 2Lx - 6Ly$
$Lo_{4x4-5} = Lo_{4x4-10} = 6Lx - 6Ly$
$Lo_{4x4-6} = Lo_{4x4-9} = L_{4x4-5} / 2$
$Lo_{4x4-7} = Lo_{4x4-8} = 6Lx - 2Ly$

FIG. 19

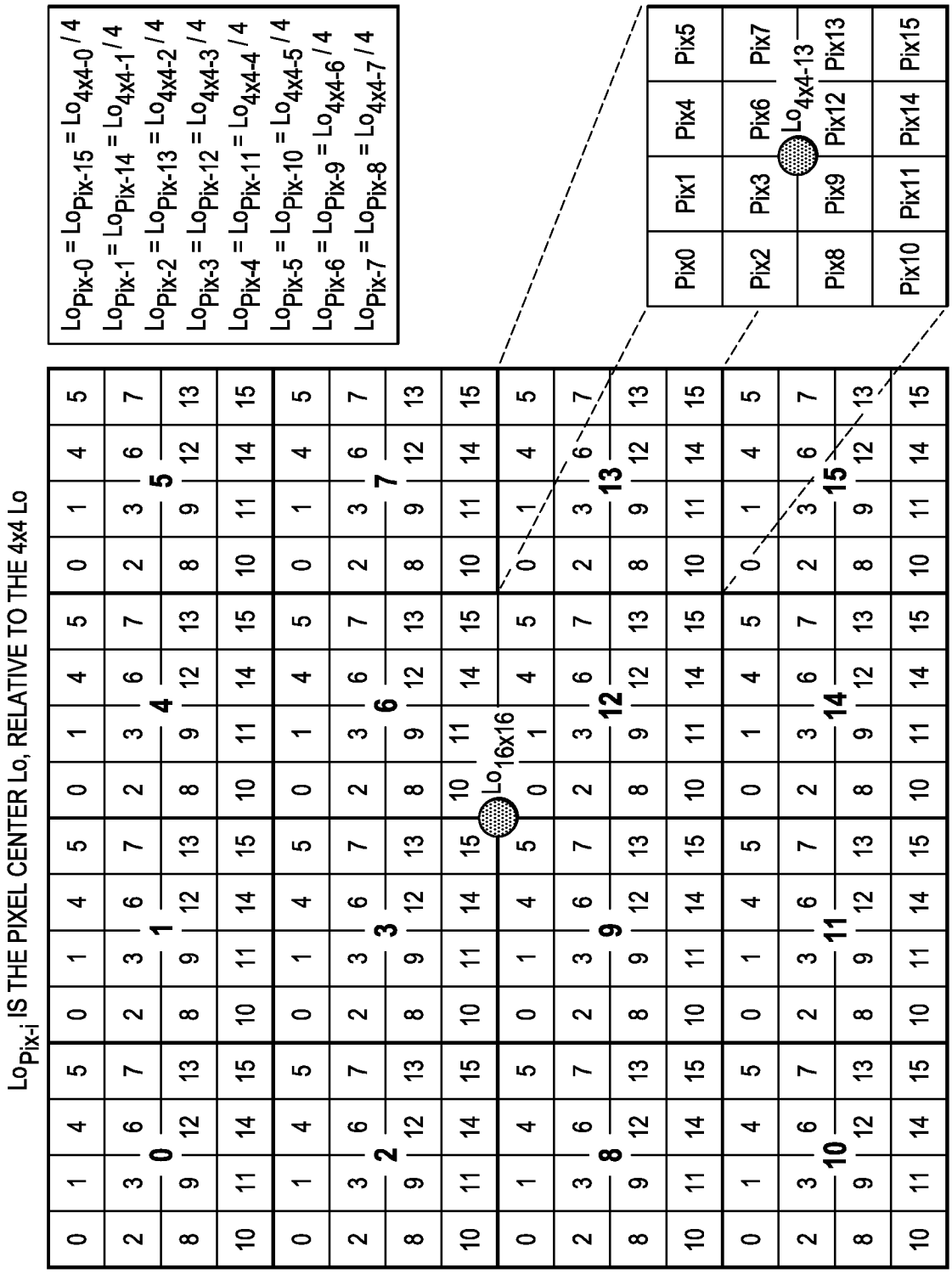

Lo_{Pix-i} IS THE PIXEL CENTER Lo, RELATIVE TO THE 4x4 Lo
$Lo_{Pix-0} = Lo_{Pix-15} = Lo_{4x4-0} / 4$
$Lo_{Pix-1} = Lo_{Pix-14} = Lo_{4x4-1} / 4$
$Lo_{Pix-2} = Lo_{Pix-13} = Lo_{4x4-2} / 4$
$Lo_{Pix-3} = Lo_{Pix-12} = Lo_{4x4-3} / 4$
$Lo_{Pix-4} = Lo_{Pix-11} = Lo_{4x4-4} / 4$
$Lo_{Pix-5} = Lo_{Pix-10} = Lo_{4x4-5} / 4$
$Lo_{Pix-6} = Lo_{Pix-9} = Lo_{4x4-6} / 4$
$Lo_{Pix-7} = Lo_{Pix-8} = Lo_{4x4-7} / 4$
Lo_{16x16}
Lo_{4x4-13}
Pix0 Pix1 Pix4 Pix5
Pix2 Pix3 Pix6 Pix7
Pix8 Pix9 Pix12 Pix13
Pix10 Pix11 Pix14 Pix15

FIG. 20

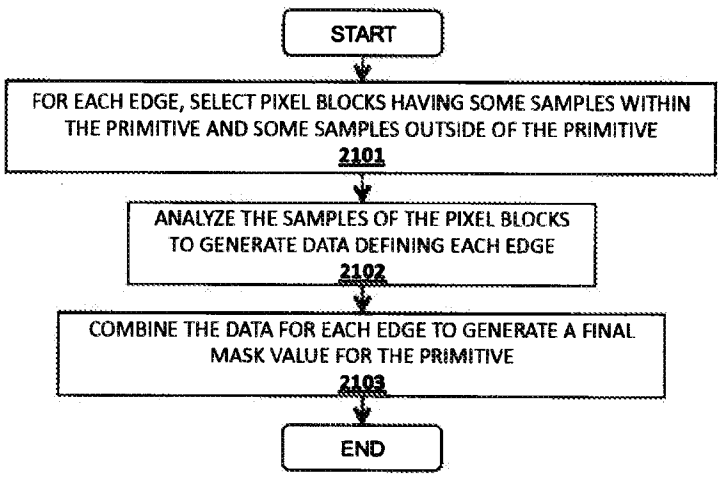

START
FOR EACH EDGE, SELECT PIXEL BLOCKS HAVING SOME SAMPLES WITHIN THE PRIMITIVE AND SOME SAMPLES OUTSIDE OF THE PRIMITIVE
2101
ANALYZE THE SAMPLES OF THE PIXEL BLOCKS TO GENERATE DATA DEFINING EACH EDGE
2102
COMBINE THE DATA FOR EACH EDGE TO GENERATE A FINAL MASK VALUE FOR THE PRIMITIVE
2103
END

FIG. 21

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/062068

A. CLASSIFICATION OF SUBJECT MATTER

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)
G06T 1/20; G06T 15/40; G06F 15/00; G06T 15/30; G06T 11/20; G09G 5/00

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: rasterizer, pixel block, edge determination, block selection logic

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y A	US 7804499 B1 (STEVEN E. MOLNAR et al.) 28 September 2010 See column 8, lines 7-10, column 11, lines 26-49; and figures 3A, 5A.	1-2,6-9,15-16,20-23 3-5,10-14,17-19,24-25
Y	US 2009-0027416 A1 (MASSIMILIANO BARONE et al.) 29 January 2009 See paragraphs [0011], [0063]; and figure 5.	1-2,6-9,15-16,20-23
Y	US 7119809 B1 (DANIEL H. MCCABE) 10 October 2006 See column 3, line 64 - column 4, line 10; and figures 2, 3.	8,22
A	US 2005-0007365 A1 (JIANGEN CAO) 13 January 2005 See paragraphs [0018]-[0107]; and figures 1, 2A.	1-25
A	US 5982384 A (BRYAN G. PROUTY et al.) 09 November 1999 See column 8, line 62 - column 9, line 20; and figure 11.	1-25

☐ 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
21 April 2016 (21.04.2016)	**22 April 2016 (22.04.2016)**

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. +82-42-481-8578

Authorized officer
LEE, Dong Yun
Telephone No. +82-42-481-8734

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2015/062068

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 7804499 B1	28/09/2010	None	
US 2009-0027416 A1	29/01/2009	EP 2026283 A2	18/02/2009
		EP 2026283 A3	28/07/2010
		EP 2026283 B1	16/11/2011
		US 7852350 B2	14/12/2010
US 7119809 B1	10/10/2006	None	
US 2005-0007365 A1	13/01/2005	US 7161597 B2	09/01/2007
US 05982384 A	09/11/1999	GB 2302002 A	18/12/1996
		JP 09-016144 A	17/01/1997