



- (51) International Patent Classification:
G06F 9/30 (2006.01) *G06T 1/60* (2006.01)
- (21) International Application Number:
PCT/US2016/048055
- (22) International Filing Date:
22 August 2016 (22.08.2016)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
14/862,724 23 September 2015 (23.09.2015) US
- (71) Applicant: INTEL CORPORATION [US/US]; 2200
Mission College Boulevard, Santa Clara, CA 95054 (US).
- (72) Inventor: SEILER, Larry; 198 Linden Street, Boylston,
MA 01505 (US).
- (74) Agents: TROP, Timothy, N. et al.; Trop, Pruner & Hu,
P.C., 1616 S. Voss Rd., Ste. 750, Houston, TX 77057-
2631 (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).

[Continued on next page]

(54) Title: SUPPORTING DATA CONVERSION AND META-DATA IN A PAGING SYSTEM

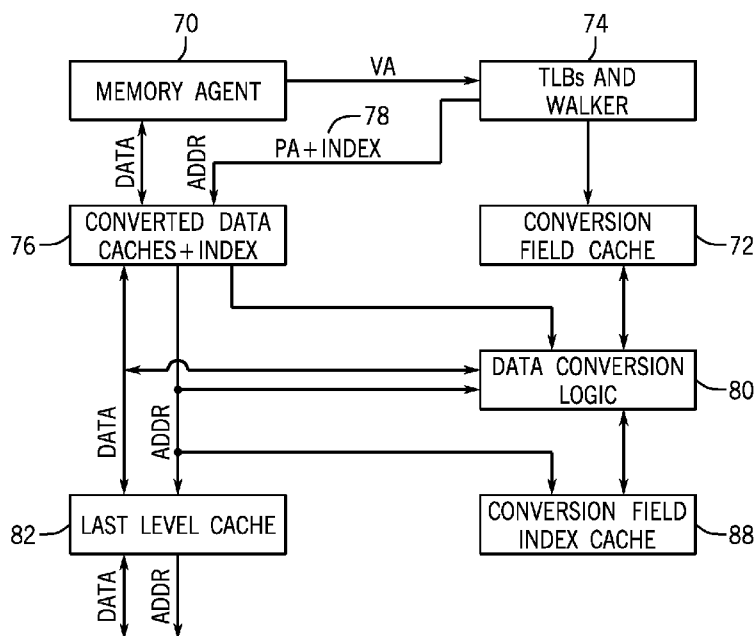


FIG. 7

(57) Abstract: Some embodiments allow data conversions and meta-data to be specified using the page tables. As a result, individual memory resources can use their own compression methods or other conversions, and meta-data need only be specified for memory resources as necessary. Hardware engines and application code can access the memory resources as if they are stored in memory in the post-conversion, uncompressed form.

**Declarations under Rule 4.17:**

- as to the identity of the inventor (Rule 4.17(i))
- as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))

- as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))

Published:

- with international search report (Art. 21(3))

SUPPORTING DATA CONVERSION AND META-DATA IN A PAGING SYSTEM

Background

[0001] It is common, especially with graphics processing units (GPUs), to compress memory resources in order to save bandwidth and power. Dedicated logic decompresses the data on reads and compresses it on writes. Such compression may require meta-data, that is, additional bits that are used to interpret the compressed base resource data. Meta-data may be used for other purposes, including debugging and security. Other kinds of data conversions may be used, for example, to rearrange the data for cache efficiency or to implement error checking and correction.

[0002] In current systems, memory controllers may convert the data accessed in certain ways, including by performing error correction or bank swapping. It is not possible to select different operations for individual memory resources. For example, if an application allocates two different arrays, the same error correction or bank swapping applies to both unless some explicit means is provided so that the memory controllers can distinguish them. This is sometimes done using registers that define address ranges. At best, it is only practical to define a few physical memory ranges that have different properties.

[0003] GPUs typically use descriptors that specify the size and location of the base data array and the location of control planes that store associated meta-data. Dedicated logic uses a descriptor together with array indices, e.g. (X,Y) or (X,Y,Z), to compute memory addresses in the control plane(s) and the base resource array. This means that code or logic that accesses these arrays must produce array indices and a descriptor reference, instead of computing a memory address and directly accessing memory. Some memory accesses on GPUs also use memory addresses instead of descriptors.

[0004] Devices such as central processing units (CPUs) that do not use descriptors require a means to directly address the data in uncompressed form. The general solution for allowing CPU access to arrays of compressed data is to first decompress the arrays. This is a high latency operation that requires synchronization. It also

prevents mixing operations that can access the array when it is compressed with those that require it to be uncompressed. For some compression methods, decompressing the data discards information, so that recompression to the original form is not practical.

[0005] Other usages for meta-data are currently supported by providing that meta-data for the entire memory range, or for coarsely defined subsets of the memory range. Similarly, other kinds of data conversions are generally performed for all of memory or for coarsely defined subsets, instead of being able to specify individual memory resources.

Brief Description Of The Drawings

[0006] Some embodiments are described with respect to the following figures:

Figure 1 is a depiction of four compressed data blocks with meta-data on the left and four uncompressed data blocks on the right according to one embodiment, where shaded squares represent the stored data;

Figure 2 is a depiction of a page table according to one embodiment;

Figure 3 is a depiction of a set of blocks with 4x lossy compression according to one embodiment;

Figure 4 is a depiction of a page table entry with three memory pointers according to one embodiment;

Figure 5 illustrates a page table entry that can support varying kinds of page walks according to one embodiment;

Figure 6 shows a way to provide additional space for a compression mechanism that accesses more than one meta-data array according to one embodiment;

Figure 7 depicts a method to implement compression and other data conversions defined in page tables according to one embodiment;

Figure 8 is a block diagram of a processing system according to one embodiment;

Figure 9 is a block diagram of a processor according to one embodiment;

Figure 10 is a block diagram of a graphics processor according to one embodiment;

Figure 11 is a block diagram of a graphics processing engine according to one embodiment;

Figure 12 is a block diagram of another embodiment of a graphics processor;

Figure 13 is a depiction of thread execution logic according to one embodiment;

Figure 14 is a block diagram of a graphics processor instruction format according to some embodiments;

Figure 15 is a block diagram of another embodiment of a graphics processor;

Figure 16A is a block diagram of a graphics processor command format according to some embodiments;

Figure 16B is a block diagram illustrating a graphics processor command sequence according to some embodiments;

Figure 17 is a depiction of an exemplary graphics software architecture according to some embodiments;

Figure 18 is a block diagram illustrating an IP core development system according to some embodiments; and

Figure 19 is a block diagram showing an exemplary system on chip integrated circuit according to some embodiments.

Detailed Description

[0007] Some embodiments allow such data conversions and meta-data to be specified using the page tables. As a result, individual memory resources can use their own compression methods or other conversions, and meta-data need only be specified for memory resources as necessary. A memory resource is a range of memory that an application has allocated for some purpose such as a two dimensional (2D) array. Hardware engines and application code can access the memory resources as if they are stored in memory in the post-conversion, uncompressed form.

[0008] Information about data conversions (such as compression) and meta-data can be provided in the page tables. Each individual page contains information on the type of data conversion to perform, as well as pointers to the meta-data required

for that page, if any. This allows different conversions and meta-data to be selected for individual memory resources.

[0009] Such additional information is accessed during the page walk and may be stored, for example, in the translation lookaside buffer (TLB) or in a separate cache associated with the TLB. As a result, converting a virtual address to a physical address provides access to that information. Typically, each converted memory resource is mapped using two virtual address ranges: one for the data as stored in memory and one to access the converted data. Only one virtual address range may be required if the data is always accessed in uncompressed form.

[0010] The cache hierarchy is divided into levels nearer the client that store converted (e.g. uncompressed) data and levels nearer the memory that store the data as it is stored in memory. The compression or other operation is performed for accesses that miss in the upper level caches, so that it is necessary to access a cache level that stores data in memory format. This method supports multiple kinds of conversions, which can be applied at different levels of the cache hierarchy.

[0011] These conversions can be performed in ways that are completely cache coherent. That is, multiple central processing unit (CPU) and graphics processing unit (GPU) memory agents may interleave accesses to converted arrays (using either address range) while guaranteeing cache coherency, according to the same rules that apply to normal memory accesses.

[0012] As a result, Shared Virtual Memory (SVM) access to applications may be provided on the CPU for a compressed array used by the GPU. Efficient random access by CPU applications to compressed or otherwise converted memory resources may also be provided in some embodiments.

[0013] In general, it is impractical for applications to compress or decompress data in software during random accesses, due to code complexity. Software compression or decompression is usually only practical when compressing or decompressing the whole memory resource. Some embodiments allow compression and decompression on access for any compression format that supports random access.

[0014] At present, meta-data generally can only be used if it is provided for all of physical memory or for a defined sub-range of physical or virtual memory. Embodiments allow meta-data to be specified in different ways for individual memory resources. For example, meta-data may be used to mark uninitialized data for a particular memory resource, without providing those meta-data bits for other memory resources.

[0015] Finally, embodiments may allow GPU engines to perform normal memory accesses without use of descriptors. In many cases, it is beneficial for GPU engines to use descriptors and directly access meta-data, so that compressed data can be stored in a local cache. In other cases this is a complexity that does not provide enough benefit to justify the cost. This technique can be used to provide additional options for GPU implementation.

[0016] A variety of data conversions may be specified in the page table entries. One class of data conversions used by GPUs includes device-specific lossless compression formats that are optimized for specific types of memory resources, such as media data, single-sample color buffers, multi-sample color buffers, or depth buffers. Another class includes standard lossy compression formats, such as adaptive scalable texture compression (ASTC) or block compression (BC), or converting between floating point and specialized GPU numeric types such as unnormalized, non-standard floating point, or sRGB. Normalized and sRGB use short integers, e.g. 8-bit, to represent values in the range [0.0..1.0], using equal steps for unnormalized and using an exponential function for sRGB. These are standard numeric types in graphics application program interfaces (APIs).

[0017] The table entries can also support data conversions useful for CPUs or other compute engines, as opposed to media and 3D graphics engines. One category includes compression formats designed to support sparse matrices or other data formats common on CPUs. Another category provides debug, security or consistency checking information at a fine memory access granularity for specific memory resources. **Figure 1** shows a set of four compressed data blocks on the left, each containing four 64B cache lines. On the right they are decompressed. On the left they are compressed using a 2-bit code per block that is interpreted as follows.

0: the block uses a clear color (i.e. color used to clear a memory region to a constant value) and stores no data, 1: the block stores one compressed cache line, 2: the block stores two compressed cache lines, or 3: the block is not compressed and stores four cache lines. So there are 2 bits of meta-data for every 256B of compression data. To represent an entire two-dimensional (2D) image that occupies N pages if uncompressed, the memory stores a page that contains the clear color, N pages that store the compressed data, and N/1K or more pages of meta-data. A single page of meta-data covers 1K pages of compressed data.

[0018] To represent this in a page table entry, three memory addresses are stored: the page of compressed data, the page containing the clear color, and the page of meta-data, together with a 10-bit index to the portion of the meta-data page that corresponds to this particular compressed data page. Schematically, the page table entry is shown in **Figure 2**. It includes a page table entry 10, resource data 12, a header block 14 and a meta-data page 16. The shaded box 18 indicates the particular subset of the meta-data that is needed for this compressed resource data page.

[0019] Related cases are data conversions that do not require a header block. These could be additional lossless compression algorithms, or they could be other uses for meta-data. In such a case the page table entry only needs to store two pointers.

[0020] Another case is lossy compression with a fixed block size. **Figure 3** shows a set of blocks that achieve 4x compression. So if there are N compressed pages, there are 4N uncompressed pages. Representing this in the page table uses only one address pointer. It is much like the meta-data pointer above, in that it specifies the subset of the compression page that corresponds to the page of uncompressed data. Note that this same method works if the unconverted and converted data are the same size, e.g. if data is being reordered or otherwise altered without compression. This is the case when the meta-data specifies debug, protection, or consistency checking information, rather than a compression method.

[0021] Typically the compression ratio and number of meta-data bits per block (if any) are both powers of two. In these cases, the pointers can specify physical addresses. That requires the meta-data to be swizzled so that a single uncompressed page uses meta-data bits that are all in the same page. It also involves operating system (OS) kernel support for managing the physical memory mapping.

[0022] Another alternative is to store virtual addresses for one or more of the pointers in the page table entry. This requires another level of page walk, but it also allows non-power-of-two compression ratios, e.g. ASTC 5x5 block compression. If the page tables are also specified with virtual addresses, the entire data conversion method can be implemented in user mode, with no operating system (OS) kernel changes. This user mode page walk may precede the normal OS-managed page walk.

[0023] **Figure 4** illustrates a way to store three memory pointers in a 128-bit page table entry. This example supports mapping a 48-bit virtual address to 64KB pages. This is useful, for example, to allow a user mode page walk (using virtual addresses for the tables) that precedes a standard operating system (OS)-managed virtual to physical page walk. This extra page walk works similarly to the user-mode Tiled Resource Translation described in Windows 8.1, and may be combined with it. Both the compressed data mapping and the uncompressed data mapping address ranges can be shared using SVM.

[0024] The upper 64-bits in the example above store pointers to a 64KB header block page 20 and a page of meta-data 22. The Meta-Data Control field 24 provides space to store an offset within the meta-data page 22, as well as extra bits that are part of the Meta-Data Control field 24 that select the specific type of compression or other conversion. The V bit 30 indicates whether the addresses are virtual or physical, so that the compression page walk can either use physical addresses (OS-managed) or virtual addresses (user-managed). Other fields contain control bits required by existing paging systems, such as P (present), A (accessed) and D (dirty) bits. The reserved field (res.) 28 provides room for additional page table bits to be specified.

[0025] A more general solution supports varying numbers of extra memory addresses and does not require storing this information directly in the page table entries. This would make it easier to use a single page table format for all walks.

Figure 5 illustrates one example of such a format. The Walk field 40 is identical to the bottom four bits (Walk Mode Bits) of Figure 4. The extended hardware bits field 42 is available to specify compression formats that do not require meta-data or header block addresses. The hardware bits field 44 stores other control bits available in Intel architecture paging method IA-32e, as well as a field to select compression. For a normal page walk the PageAddr field 46 points to a page table page or a page of memory, but depending on the compression bits in 44, it may point to a special compression data block.

[0026] **Figure 6** shows an example of a compression data block format that can be used to provide varying amounts of compression information in the page walk. Compression methods could potentially access more than one meta-data array, so it is desirable to not be limited to the data that can fit into a single page table entry. Figure 6 shows two different sizes for such a block, in one embodiment a 16 byte block 52 or a 32 byte block 54. Larger sizes are possible. When a compression mode bit is enabled in hardware (HW) bits field 44 in Figure 5, the PageAddr field 46 in Figure 5 points to a compression block of a selected size. All except the first 8 bytes of the compression block stores compression data, e.g. pointers to a header block and/or one or more pages of meta-data. The bottom eight bytes store the Page Address 50 that would normally be stored in the page table entry that selected this header block.

[0027] Compression and other data conversions that are defined in the page tables may be implemented in several ways. One method is schematically shown in **Figure 7**. When the memory agent 70 performs a virtual to physical address translation, this causes the fields in the page table (if any) that represent how to convert the data to be loaded into a Conversion Field Cache 72 that is loaded at the same time as the TLB 74. An index 78 into this cache (or an identifier for simple conversions) is loaded into the data cache 76 along with the physical address.

[0028] Accesses to converted data (e.g. uncompressed) that hit in the data cach 76 are processed normally. Accesses that miss in that data cach require data conversion. This is accomplished by passing the address and index to the data conversion logic 80, which performs the compression/decompression or other conversion, serializing accesses to the same data block so that data and meta-data are updated consistently, rather than allowing the data from one transaction to be stored with the meta-data from another.

[0029] When an invalidate is issued to the lower cache 82, the conversion field index cache 78 may be used to determine how to convert that address into the address(es) that must be invalidated in the caches that store converted data. Finally, if a conversion field is invalidated in the conversion field cache 72, all converted cache lines that use it must also be invalidated.

[0030] **Figure 8** is a block diagram of a processing system 100, according to an embodiment. In various embodiments the 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 system 100 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

[0031] An embodiment of 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 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.

[0032] 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).

[0033] 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.

[0034] In some embodiments, processor 102 is coupled to a processor bus 110 to transmit communication signals such as address, data, or control signals between processor 102 and other components in system 100. In one embodiment the system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116 and an Input Output (I/O) controller hub 130. A memory controller hub 116 facilitates communication between a memory device and other components of system 100, while an I/O Controller Hub (ICH) 130 provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub 116 is integrated within the processor.

[0035] Memory device 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 120 can operate as system memory for the system 100, to store data 122 and instructions 121 for use when the one or more processors 102 executes an application or 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.

[0036] In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, 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 140 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. It will be appreciated that the system 100 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub 130 may be integrated within the one or more processor 102, or the memory controller hub 116 and I/O controller hub 130 may be integrated into a discreet external graphics processor, such as the external graphics processor 112.

[0037] **Figure 9** is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-202N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of **Figure 9** 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 processor cores 202A-202N includes one or more internal cache units 204A-204N. In some embodiments each processor core also has access to one or more shared cache units 206.

[0038] The internal cache units 204A-204N 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 processor 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-204N.

[0039] In some embodiments, processor 200 may also include a set of one or more bus controller units 216 and a system agent core 210. The one or more bus controller units 216 manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core 210 provides management functionality for the various processor components. In some embodiments, system agent core 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

[0040] In some embodiments, one or more of the processor cores 202A-202N include support for simultaneous multi-threading. In such embodiment, the system agent core 210 includes components for coordinating and operating cores 202A-202N during multi-threaded processing. System agent core 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores 202A-202N and graphics processor 208.

[0041] 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 core 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 a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent core 210.

[0042] 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.

[0043] 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 processor cores 202-202N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

[0044] In some embodiments, processor cores 202A-202N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 202A-202N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor 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. In one embodiment processor cores 202A-202N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. 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.

[0045] **Figure 10** 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.

[0046] 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.

[0047] 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 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.

[0048] 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.

[0049] 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.

[0050] 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.

[0051] **Figure 11** 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 10**. Elements of **Figure 11** 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.

[0052] 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 commands are directives fetched from a ring buffer, which stores commands for

the 3D and media pipelines 412, 416. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The 3D and media pipelines 412, 416 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.

[0053] 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.

[0054] 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 frame 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).

[0055] 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.

[0056] 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.

[0057] 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.

[0058] **Figure 12** is a block diagram of another embodiment of a graphics processor 500. Elements of **Figure 12** 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-580N. 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-580N. 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-580N (sometimes referred to as core slices), each having multiple sub-cores 550A-550N, 560A-560N (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-580N, each including a set of first sub-cores 550A-550N and a set of second sub-cores 560A-560N. Each sub-core in the set of first sub-cores 550A-550N includes at least a first set of execution units 552A-552N and media/texture samplers 554A-554N. Each sub-core in the set of second sub-cores 560A-560N includes at least a second set of execution units 562A-562N and samplers 564A-564N. In some embodiments, each sub-core 550A-550N, 560A-560N shares a set of shared resources 570A-570N. 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 13** illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of **Figure 13** 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-608N, 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-608N. 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-608N includes any number individual execution units.

[0064] In some embodiments, execution unit array 608A-608N is primarily used to execute “shader” programs. In some embodiments, the execution units in array 608A-608N 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-608N 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-608N 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-608N. For example, the geometry pipeline (e.g., 536 of **Figure 12**) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (**Figure 13**). 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 application programming interface (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 14** 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 714 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 an 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.

[0076] In some embodiments, the 128-bit instruction format 710 includes an access/address mode field 726, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. 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 addressing for all source and destination operands.

[0077] In one embodiment, the address mode portion of the access/address mode field 726 determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction 710 directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

[0078] In some embodiments instructions are grouped based on opcode 712 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 and logic instructions are in the form of 0001xxxxb. 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.

[0079] **Figure 15** is a block diagram of another embodiment of a graphics processor 800. Elements of **Figure 15** 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.

[0080] 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.

[0081] 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.

[0082] 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.

[0083] 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.

[0084] 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.

[0085] 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/depth 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.

[0086] 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.

[0087] 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. An associated render cache 878 and depth cache 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.

[0088] 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.

[0089] 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.

[0090] 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.

[0091] **Figure 16A** is a block diagram illustrating a graphics processor command format 900 according to some embodiments. **Figure 16B** is a block diagram illustrating a graphics processor command sequence 910 according to an embodiment. The solid lined boxes in **Figure 16A** 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 16A** 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 16B** shows an exemplary graphics processor 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, the graphics processor 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 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.

[0100] 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.

[0101] 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.

[0102] In some embodiments, the graphics processor 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.

[0103] 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.

[0104] 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.

[0105] **Figure 17** illustrates exemplary graphics software architecture for a data processing system 1000 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.

[0106] 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.

[0107] 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.

[0108] 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.

[0109] One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the

machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as “IP cores,” are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

[0110] **Figure 18** is a block diagram illustrating an IP core development system 1100 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 1100 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 1130 can generate a software simulation 1110 of an IP core design in a high level programming language (e.g., C/C++). The software simulation 1110 can be used to design, test, and verify the behavior of the IP core using a simulation model 1112. The simulation model 1112 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design can then be created or synthesized from the simulation model 1112. The RTL design 1115 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 1115, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

[0111] The RTL design 1115 or equivalent may be further synthesized by the design facility into a hardware model 1120, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility 1165 using non-volatile

memory 1140 (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection 1150 or wireless connection 1160. The fabrication facility 1165 may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

[0112] **Figure 19** is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. The exemplary integrated circuit includes one or more application processors 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. The integrated circuit includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I²S/I²C controller 1240. Additionally, the integrated circuit can include a display device 1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

[0113] Additionally, other logic and circuits may be included in the processor of integrated circuit 1200, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

[0114] The following clauses and/or examples pertain to further embodiments:

One example embodiment may be a method comprising using a page table to convert an address in order to access data stored in memory, and converting said data according to information stored in the page table. The method may also include allowing shared access by multiple different processors to the same data in memory, and accessing that data in converted form in one or more processors and in

unconverted form in another processor. The method may also include referencing meta-data for said array using a page table entry. The method may also include providing an address for a page containing a clear color. The method may also include providing a page table entry that includes addresses for a page of compressed data together with a pointer to a portion of the meta-data page that corresponds to a particular compressed data page. The method may also include accessing said array as if it were uncompressed. The method may also include accessing information for data conversions and meta-data during a page walk. The method may also include using the page table to expand a smaller data array to a larger data array. The method may also include loading into cache fields that represent how to convert data when an agent performs a physical address translation. The method may also include using meta-data to provide additional information about data in memory for one of compression format, debug, security or consistency checking.

[0115] Another example embodiment may be one or more non-transitory computer readable media storing instructions to perform a sequence comprising using a page table to convert an address in order to access data stored in memory, and converting said data according to information stored in the page table. The media may include allowing shared access by multiple different processors to the same data in memory, and accessing that data in converted form in one or more processors and in unconverted form in another processor. The media may include referencing meta-data for said array using a page table entry. The media may include providing an address for a page containing a clear color. The media may include providing a page table entry that includes addresses for a page of compressed data together with a pointer to a portion of the meta-data page that corresponds to a particular compressed data page. The media may include accessing said array as if it were uncompressed. The media may include accessing information for data conversions and meta-data during a page walk. The media may include using the page table to expand a smaller data array to a larger data array. The media may include loading into cache fields that represent how to convert data when an agent performs a physical address translation. The media may include using meta-data to provide

additional information about data in memory for one of compression format, debug, security or consistency checking.

[0116] In another example embodiment may be an apparatus comprising a memory, and a processor, coupled to said memory, to use a page table to convert an address in order to access data stored in the memory and to convert said data according to information stored in the page table. The apparatus may include said processor to allow shared access by multiple different processors to the same data in memory, and accessing that data in converted form in one or more processors and in unconverted form in another processor. The apparatus may include said processor to reference meta-data for said array using a page table entry. The apparatus may include said processor to provide an address for a page containing a clear color. The apparatus may include said processor to provide a page table entry that includes addresses for a page of compressed data together with a pointer to a portion of the meta-data page that corresponds to a particular compressed data page. The apparatus may include said processor to access said array as if it were uncompressed. The apparatus may include said processor to access information for data conversions and meta-data during a page walk. The apparatus may include said processor to use the page table to expand a smaller data array to a larger data array. The apparatus may include said processor to load into cache fields that represent how to convert data when an agent performs a physical address translation. The apparatus may include said processor to use meta-data to provide additional information about data in memory for one of compression format, debug, security or consistency checking.

[0117] The graphics processing techniques described herein may be implemented in various hardware architectures. For example, graphics functionality may be integrated within a chipset. Alternatively, a discrete graphics processor may be used. As still another embodiment, the graphics functions may be implemented by a general purpose processor, including a multicore processor.

[0118] References throughout this specification to “one embodiment” or “an embodiment” mean that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one implementation

encompassed within the present disclosure. Thus, appearances of the phrase “one embodiment” or “in an embodiment” are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be instituted in other suitable forms other than the particular embodiment illustrated and all such forms may be encompassed within the claims of the present application.

[0119] While a limited number of embodiments have been described, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this disclosure.

What is claimed is:

- 1 1. A method comprising:
2 accessing memory resources stored in graphics processor using an
3 application running on a central processing unit; and
4 performing swizzling translations of the graphics processor-specific formats
5 on said central processing unit.
- 1 2. The method of claim 1 including performing a first page table walk to map a
2 virtual address of the resource to a virtual page that contains swizzled data.
- 1 3. The method of claim 1 including storing the data in caches with a physical
2 address tagged with a swizzle-format field.
- 1 4. The method of claim 1 including using tag bits on non-linearized entries to
2 indicate whether there is a linear mapping for the virtual address.
- 1 5. The method of claim 3 including swizzling before any write-back caches.
- 1 6. The method of claim 5 including using cacheline tags to perform data
2 conversion.
- 1 7. The method of claim 2 including supporting user-mode for tiled resource
2 translation using a second page table walk after the first page table walk to specify
3 swizzle translation.
- 1 8. The method of claim 1 including invalidating all entries in caches closer to
2 central processing unit that use any corresponding swizzled address.
- 1 9. The method of claim 1 including invalidating the corresponding linearized
2 addresses in all caches closer to the central processing unit when writing a non-
3 linearized address that has associated linearized addresses.

1 10. The method of claim 1 including using a translation lookaside buffer content
2 addressable memory that matches on either a virtual or physical address.

1 11. One or more non-transitory computer readable media storing instructions
2 executed to perform a sequence comprising:
3 accessing memory resources stored in graphics processor using an
4 application running on a central processing unit; and
5 performing swizzling and tiled rendering translators of the GPU-specific
6 formats on said central processing unit.

1 12. The media of claim 11, said sequence including performing a first page table
2 walk to map a virtual address of the resource to a virtual page that contains swizzled
3 data.

1 13. The media of claim 12, said sequence including storing the data with a
2 physical address tagged with a swizzle-format field.

1 14. The media of claim 13, said sequence including using tag bits on non-
2 linearized entries to indicate whether there is a linear mapping for the virtual
3 address.

1 15. The media of claim 12, said sequence including swizzling before any write-
2 back caches.

1 16. The media of claim 15, said sequence including using cacheline tags to
2 perform data conversion.

1 17. The media of claim 12, said sequence including support user-mode tiled
2 resource translation using a second page table walk after the first page table walk to
3 produce a virtual address.

1 18. The media of claim 11, said sequence including invalidating all entries in
2 caches closer to central processing unit that use any corresponding swizzled
3 address.

1 19. The media of claim 11, said sequence including invalidating the
2 corresponding linearized addresses in all caches closer to the central processing unit
3 when writing a non-linearized address that has associated linearized addresses.

1 20. The media of claim 11, said sequence including accessing data from a shared
2 virtual memory used by the central processing unit and a graphics processing unit.

1 21. An apparatus comprising:
2 a processor to access memory resources stored in graphics processor using
3 an application running on a central processing unit, perform swizzling and tiled
4 rendering translators of the GPU-specific formats on said central processing unit;
5 and
6 a storage coupled to said processor.

1 22. The apparatus of claim 21, said processor to perform a first page table walk to
2 map a virtual address of the resource to a virtual page that contains swizzled data.

1 23. The apparatus of claim 22, said processor to store the data with a physical
2 address tagged with a swizzle-format field.

1 24. The apparatus of claim 23, said processor to use tag bits on non-linearized
2 entries to indicate whether there is a linear mapping for the virtual address.

1 25. The apparatus of claim 22, said processor to swizzle before any write-back
2 caches.

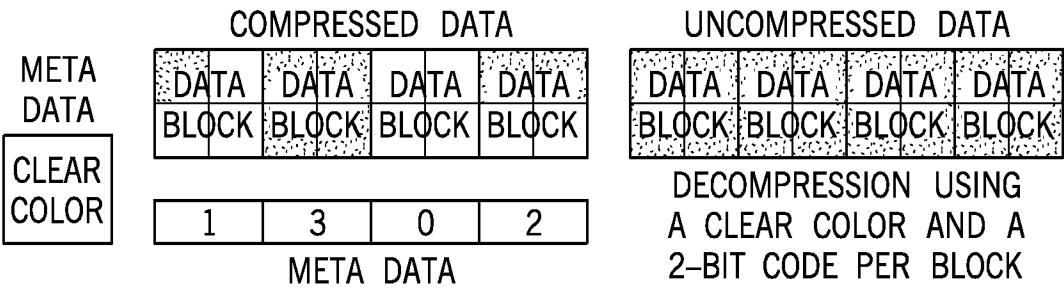


FIG. 1

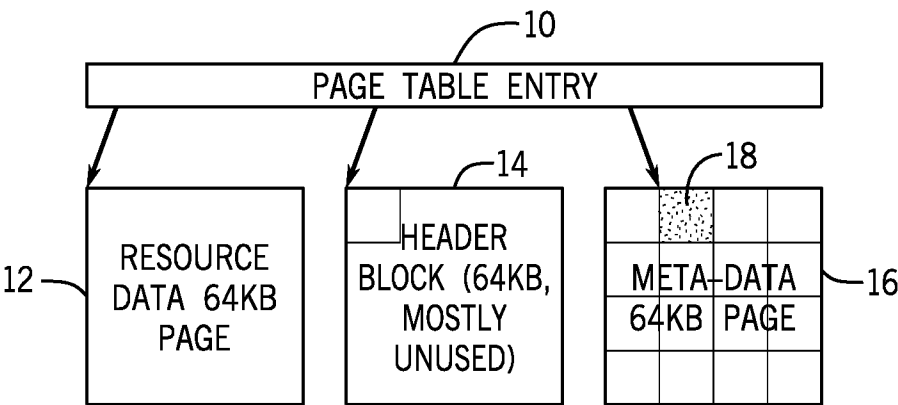


FIG. 2

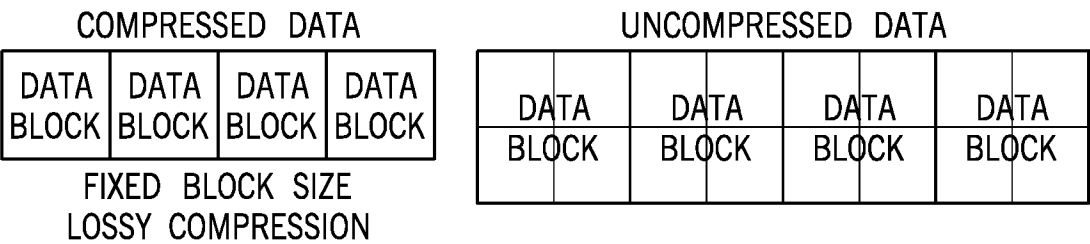
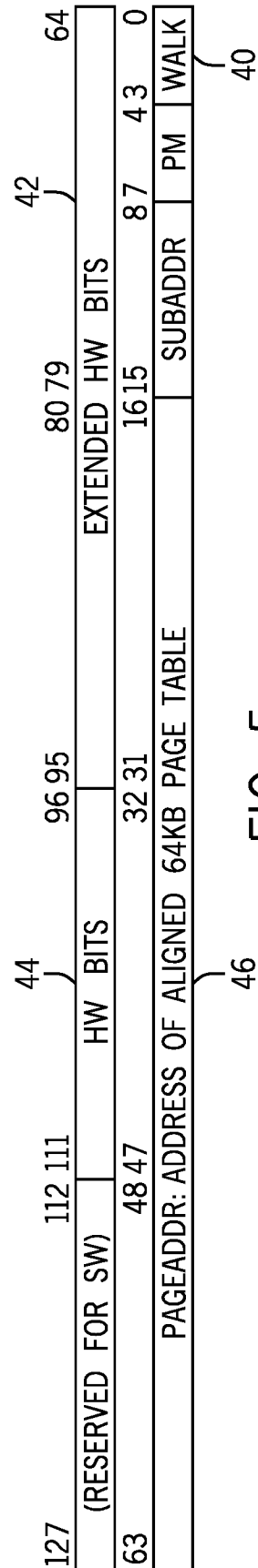
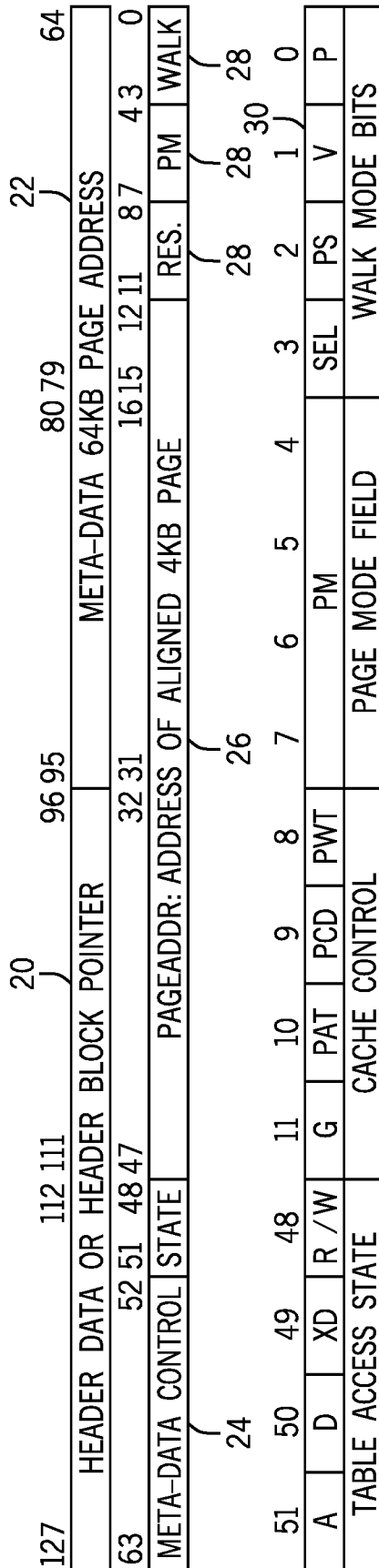


FIG. 3



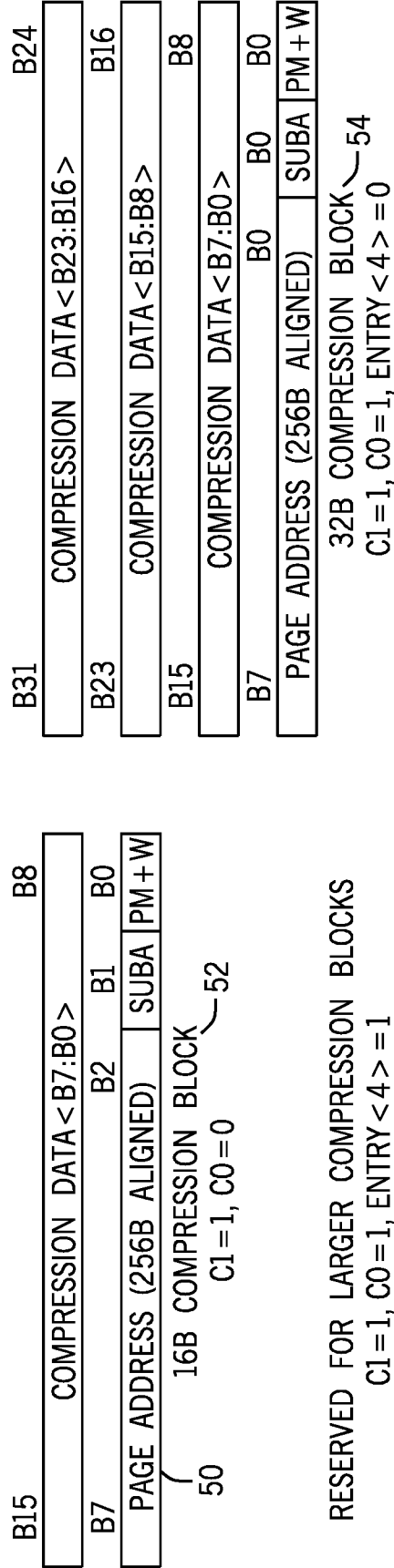


FIG. 6

4 / 16

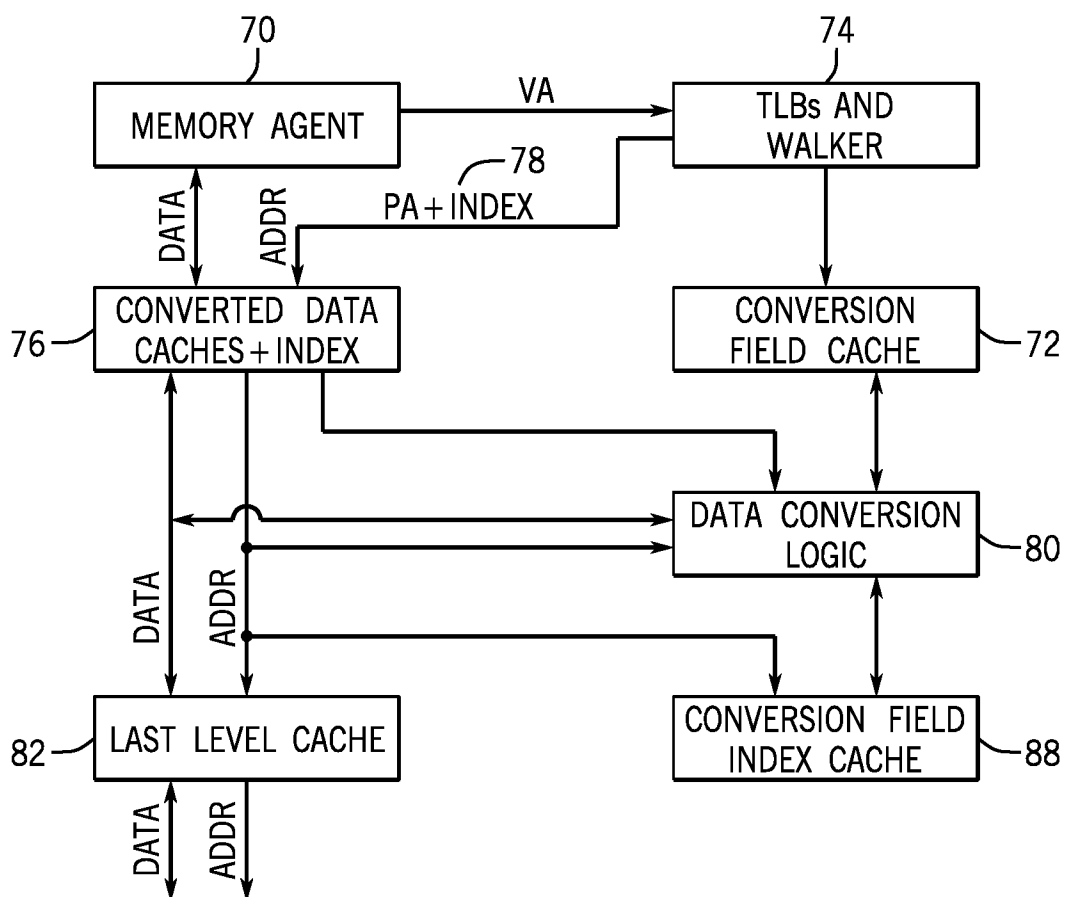


FIG. 7

5 / 16

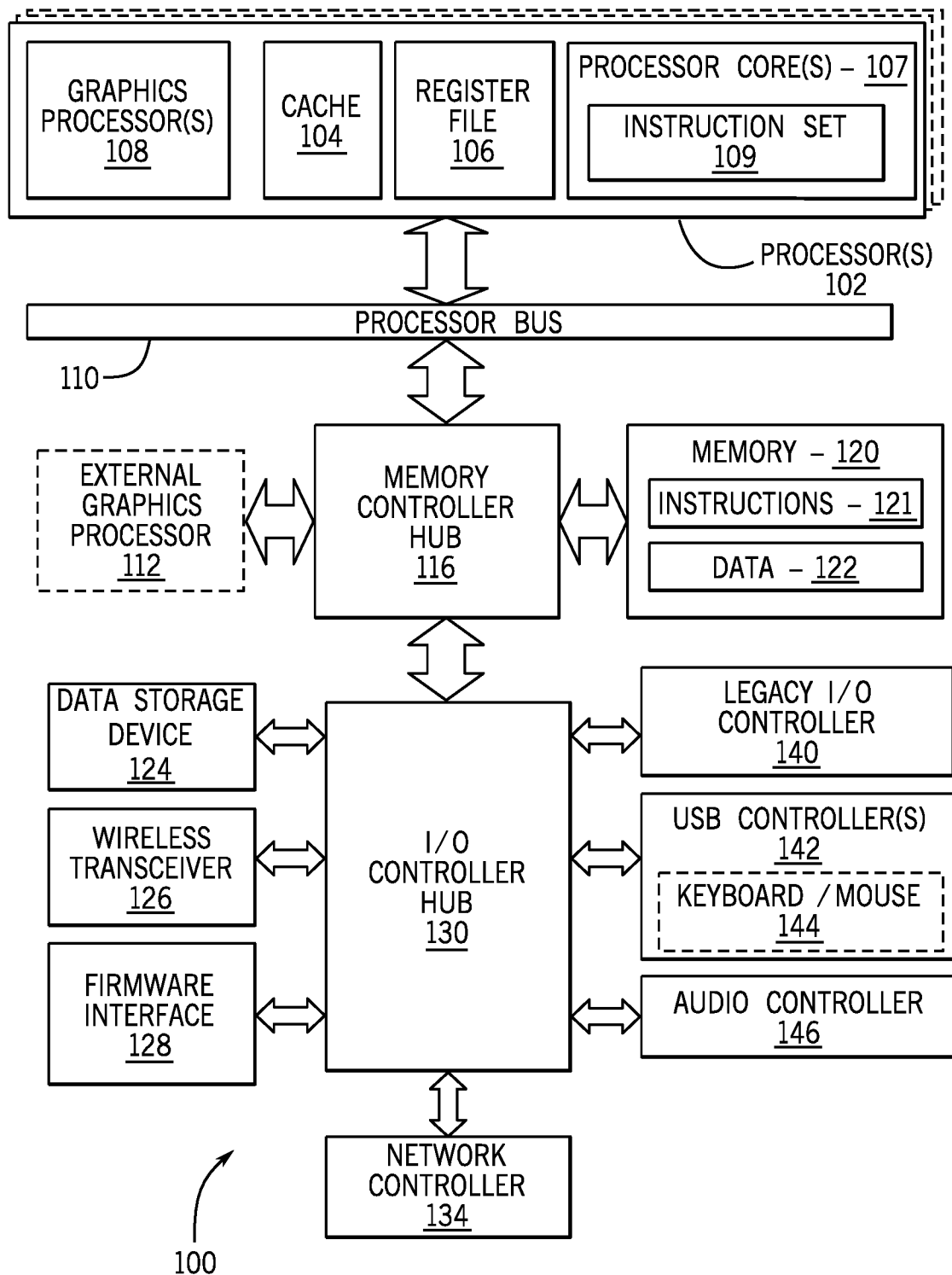


FIG. 8

PROCESSOR 200 →

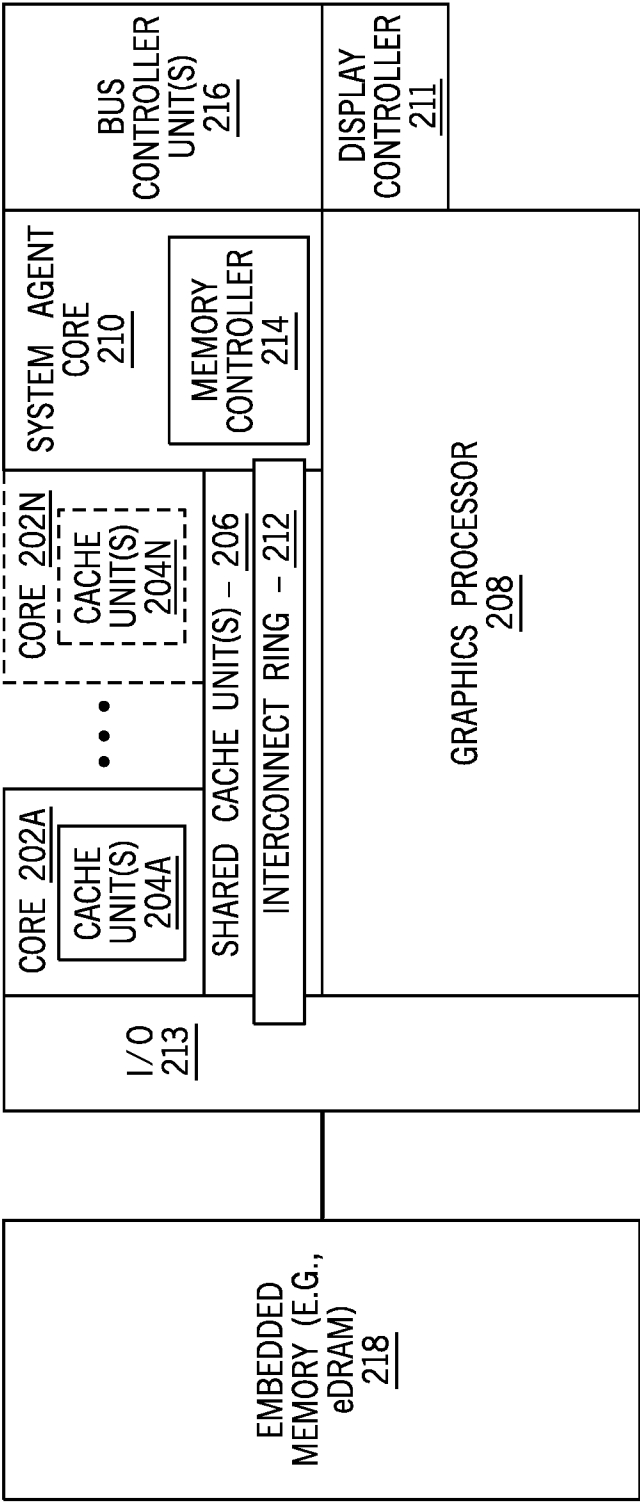


FIG. 9

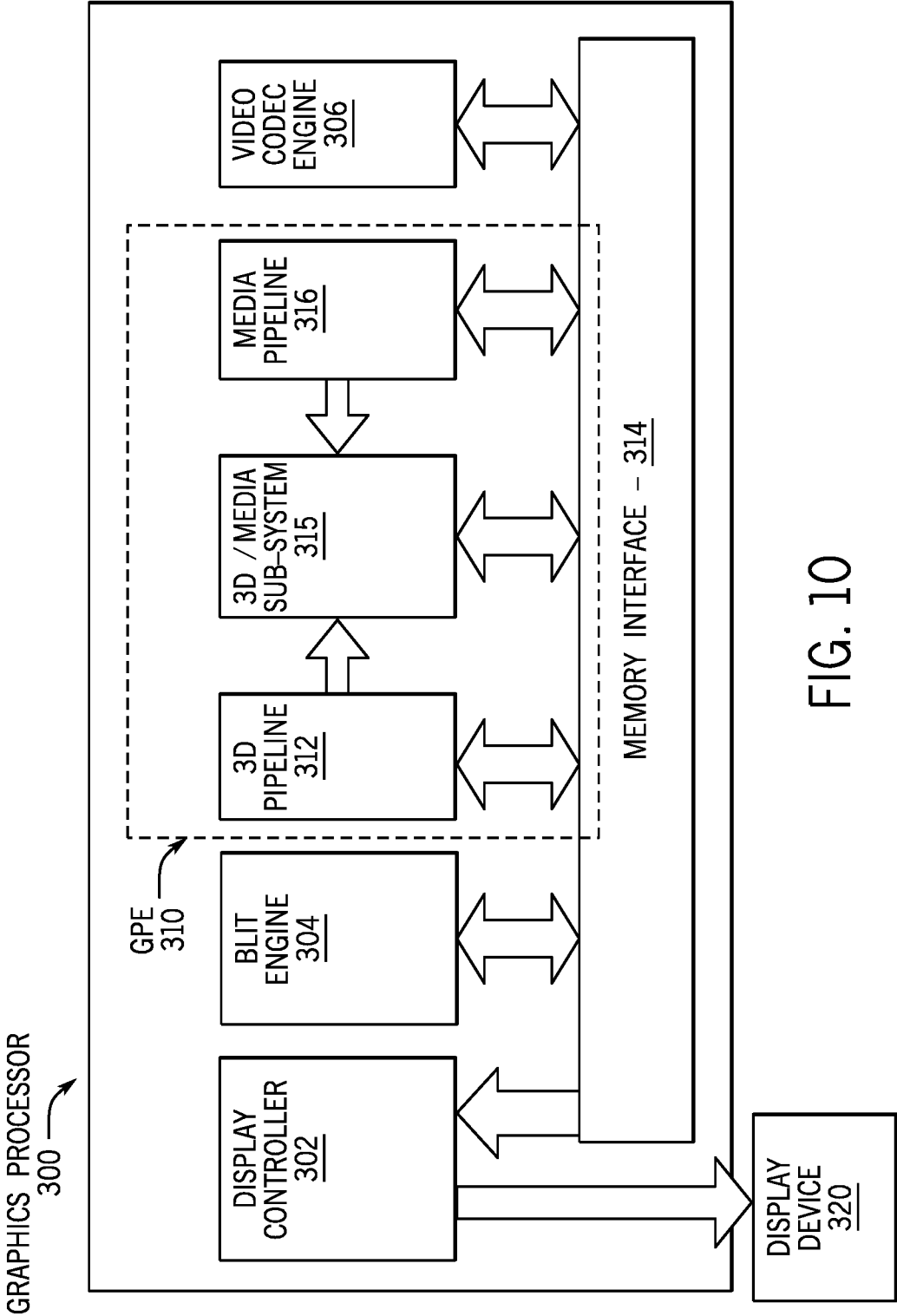


FIG. 10

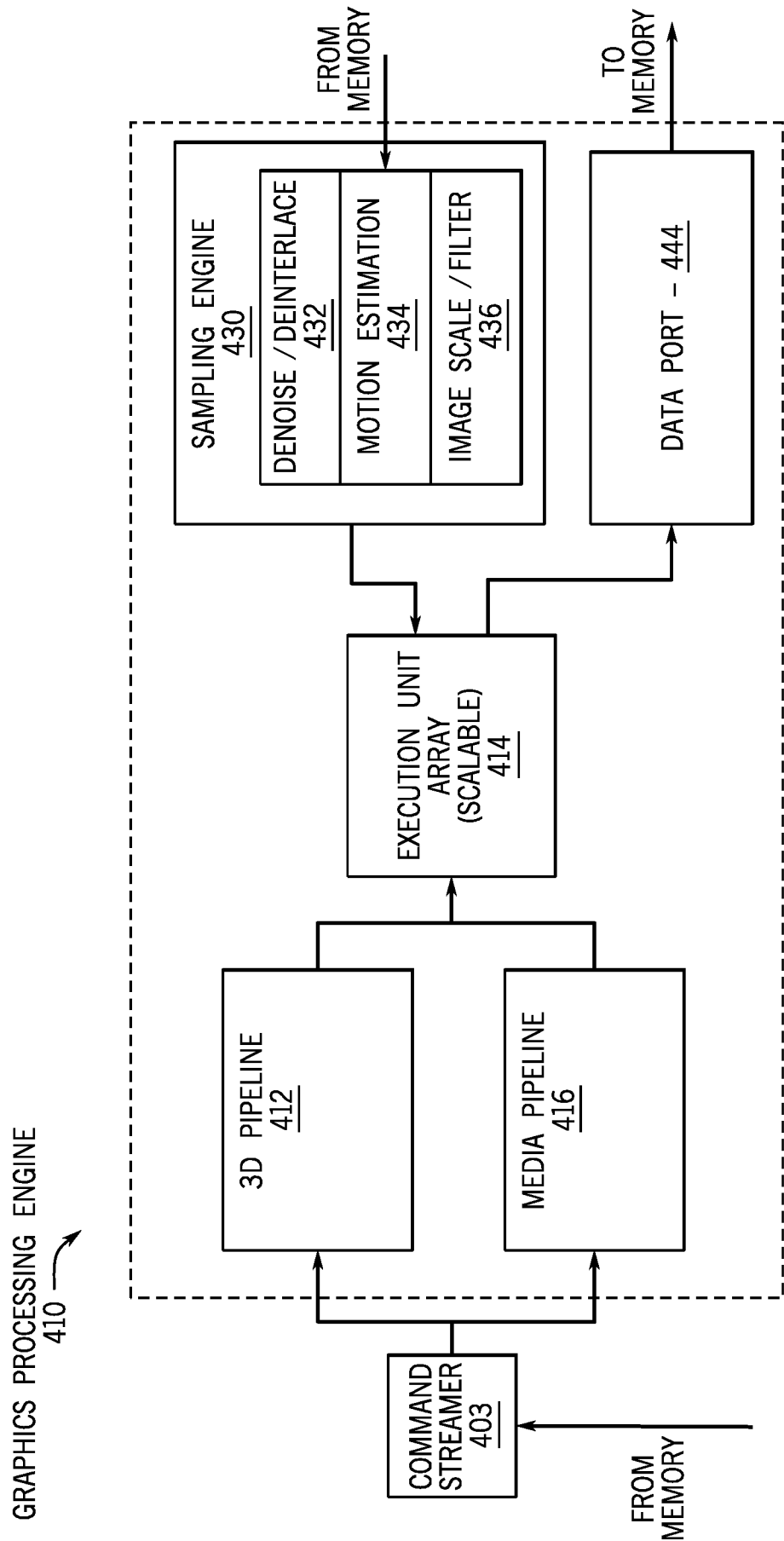


FIG. 11

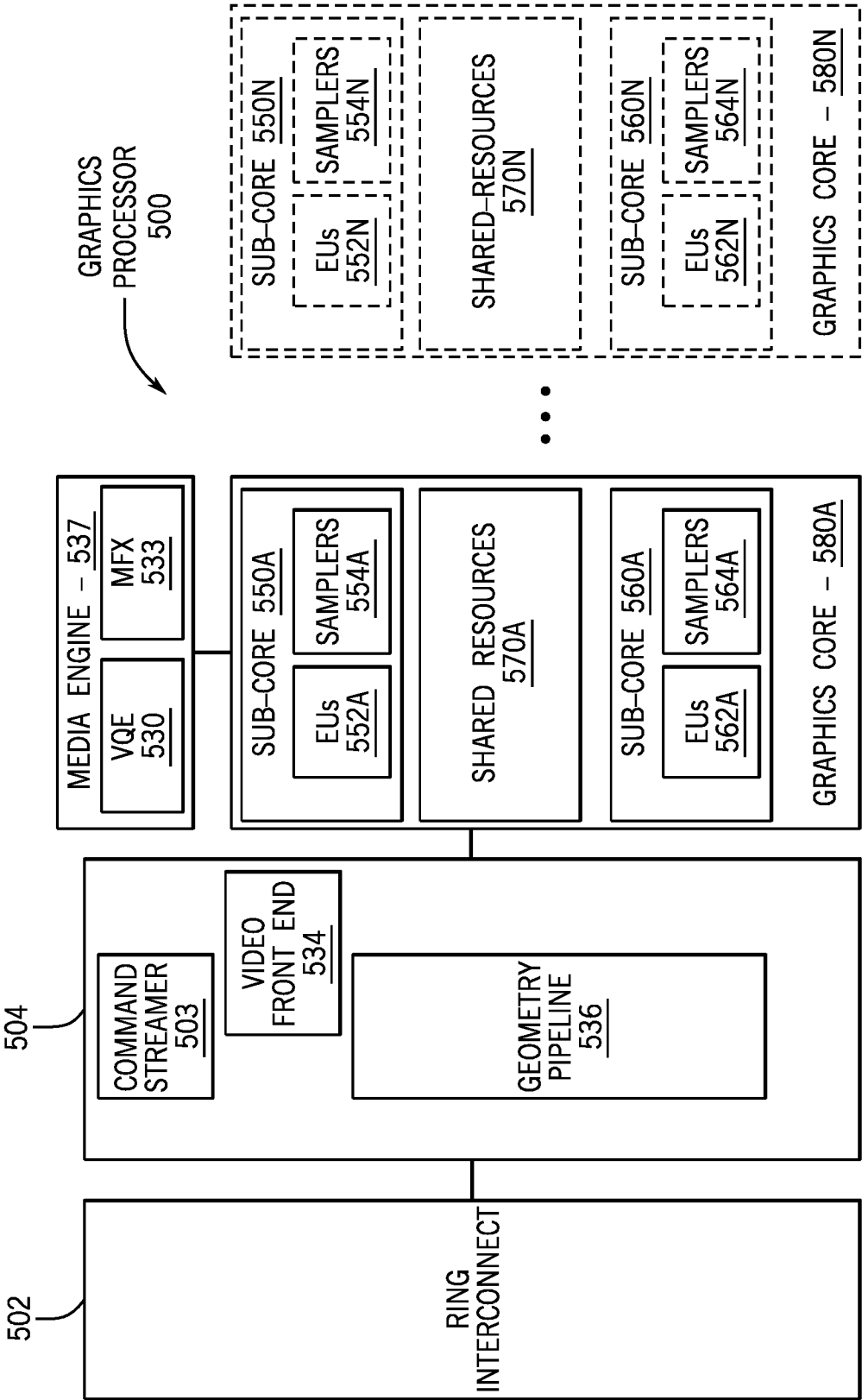


FIG. 12

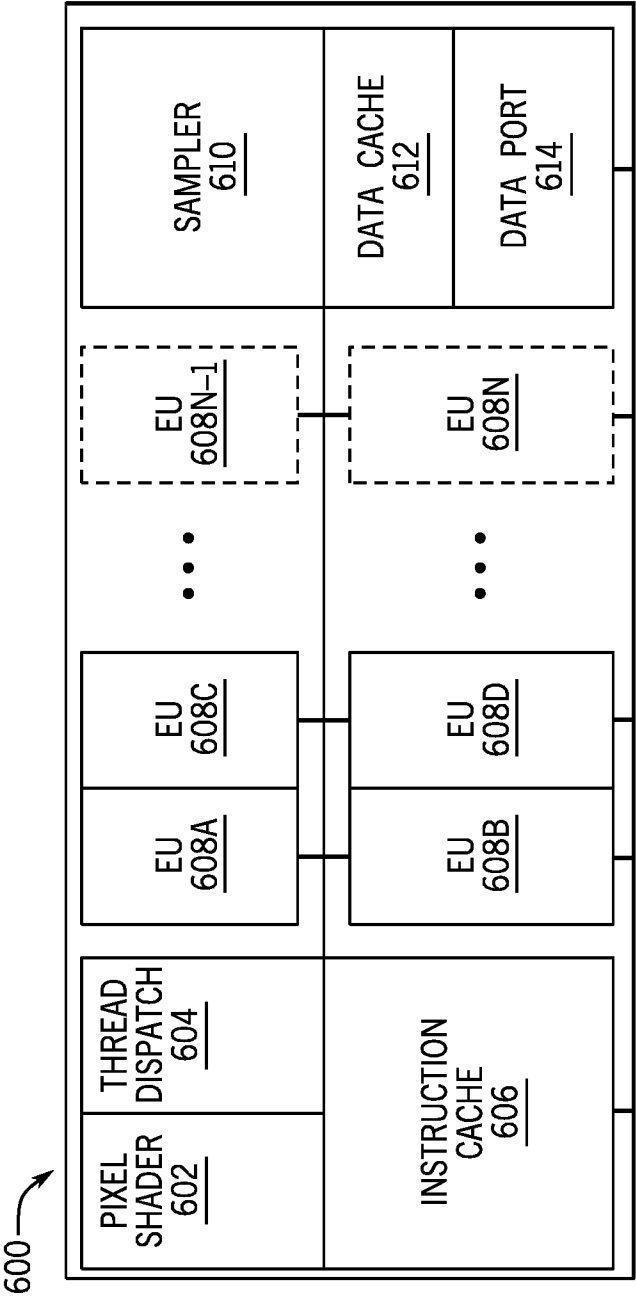
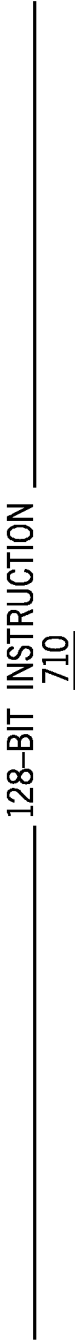


FIG. 13

GRAPHICS CORE INSTRUCTION
FORMATS

700



64-BIT COMPACT



OPCODE DECODE

740

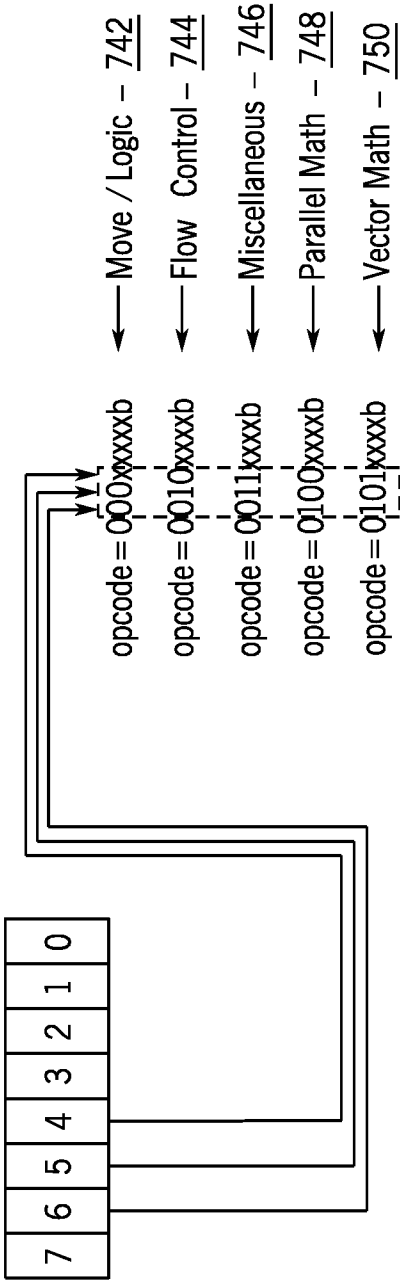


FIG. 14

12 / 16

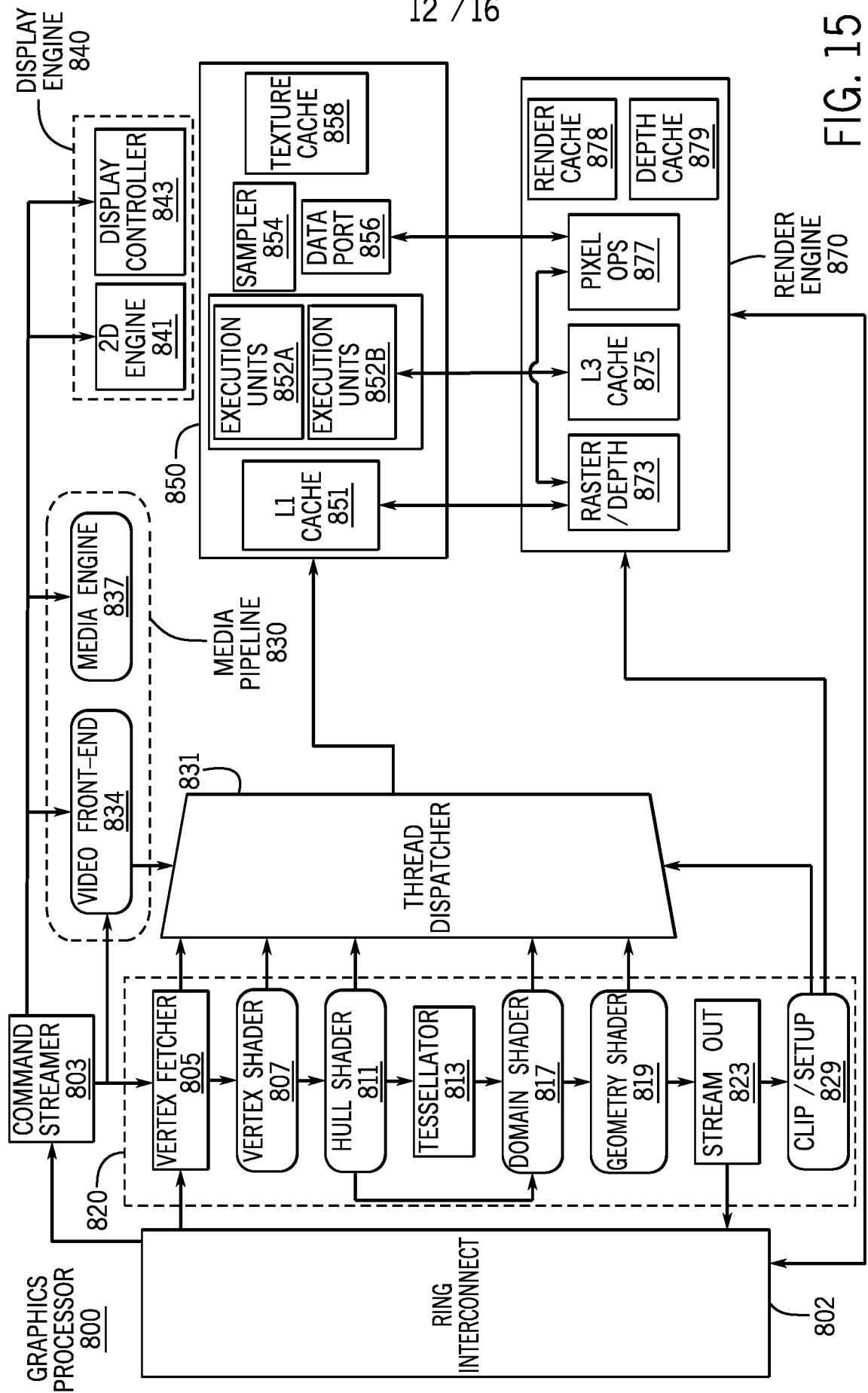


FIG. 15

13 / 16

GRAPHICS PROCESSOR COMMAND FORMAT

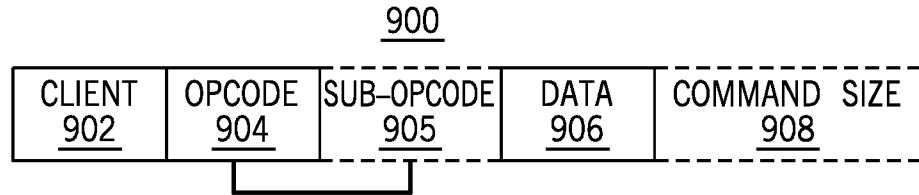


FIG. 16A

GRAPHICS PROCESSOR COMMAND SEQUENCE

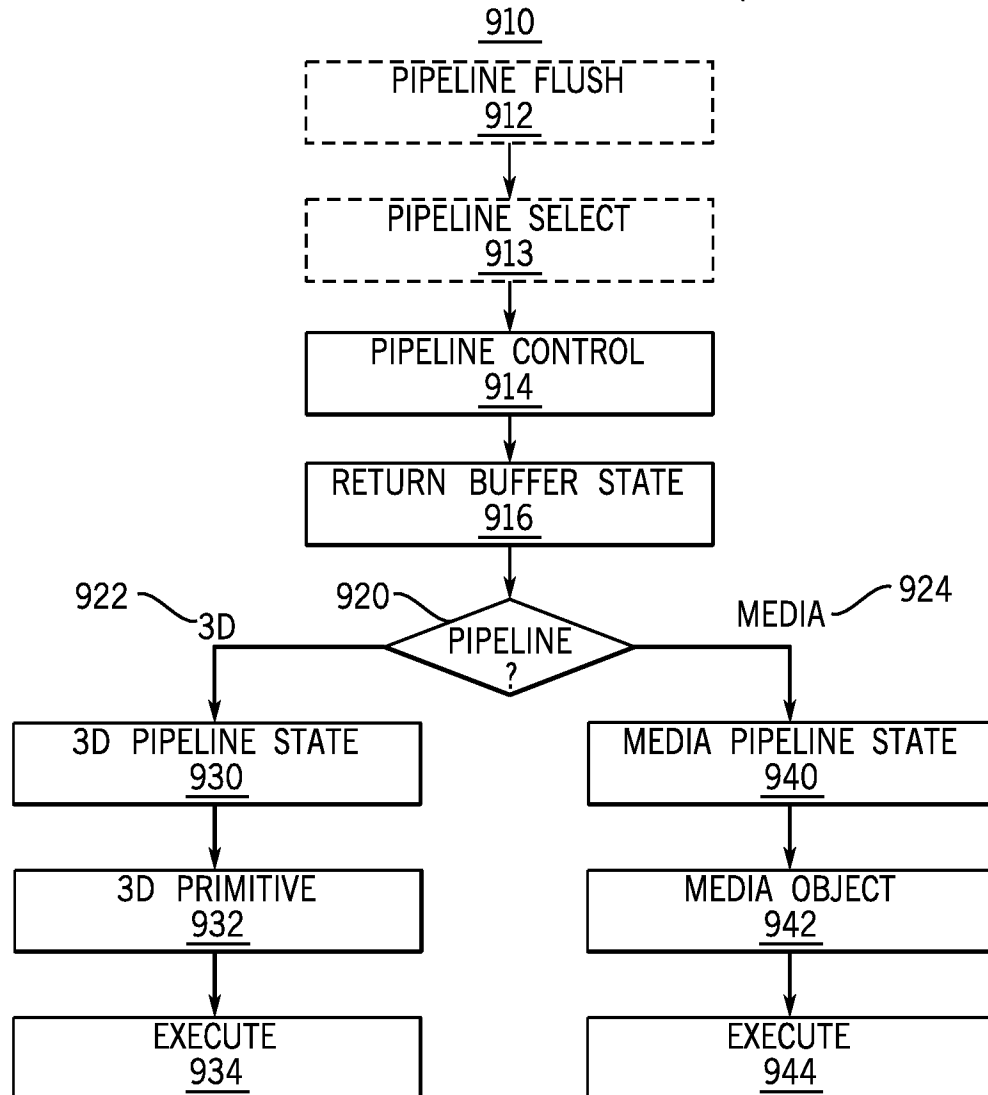


FIG. 16B

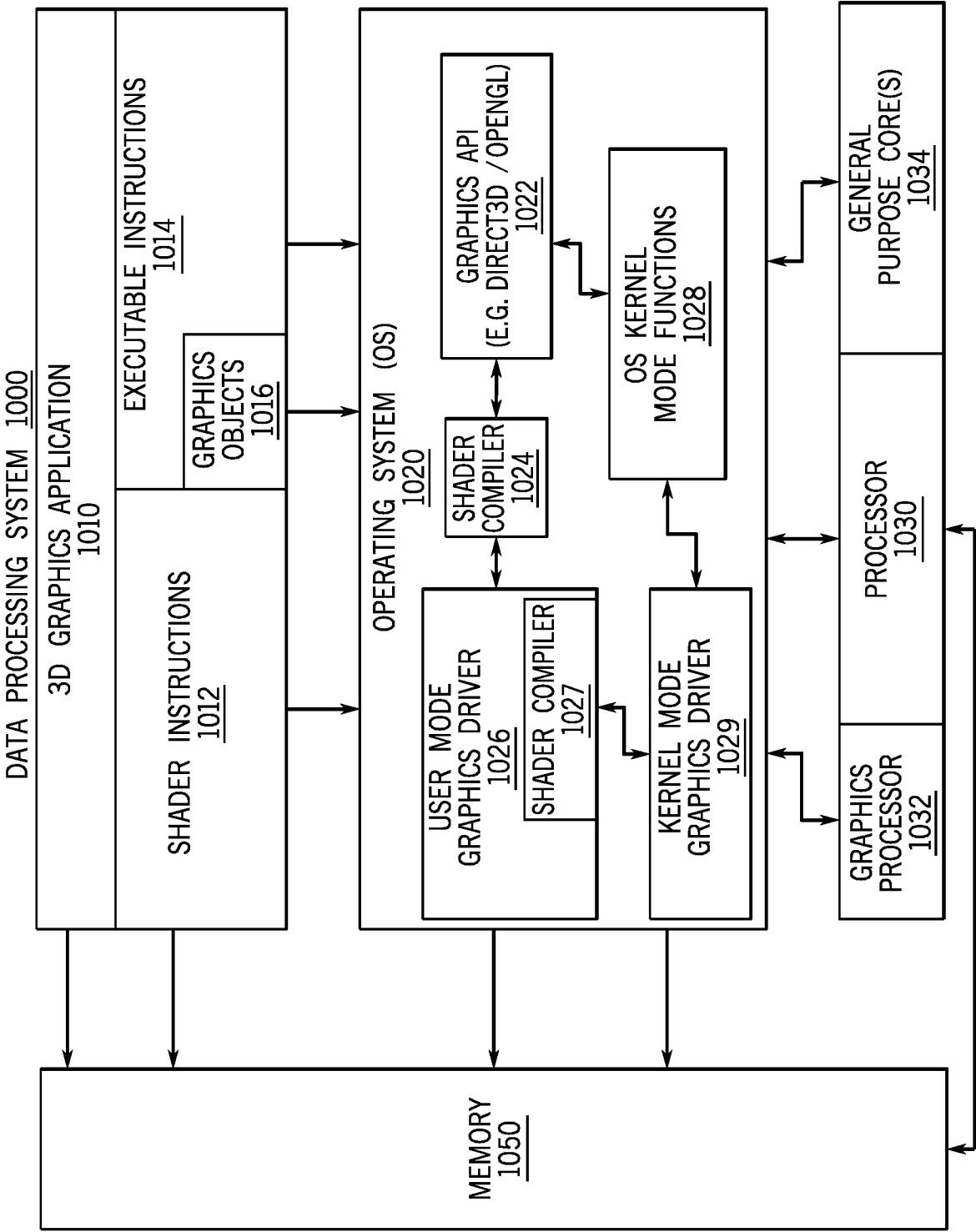


FIG. 17

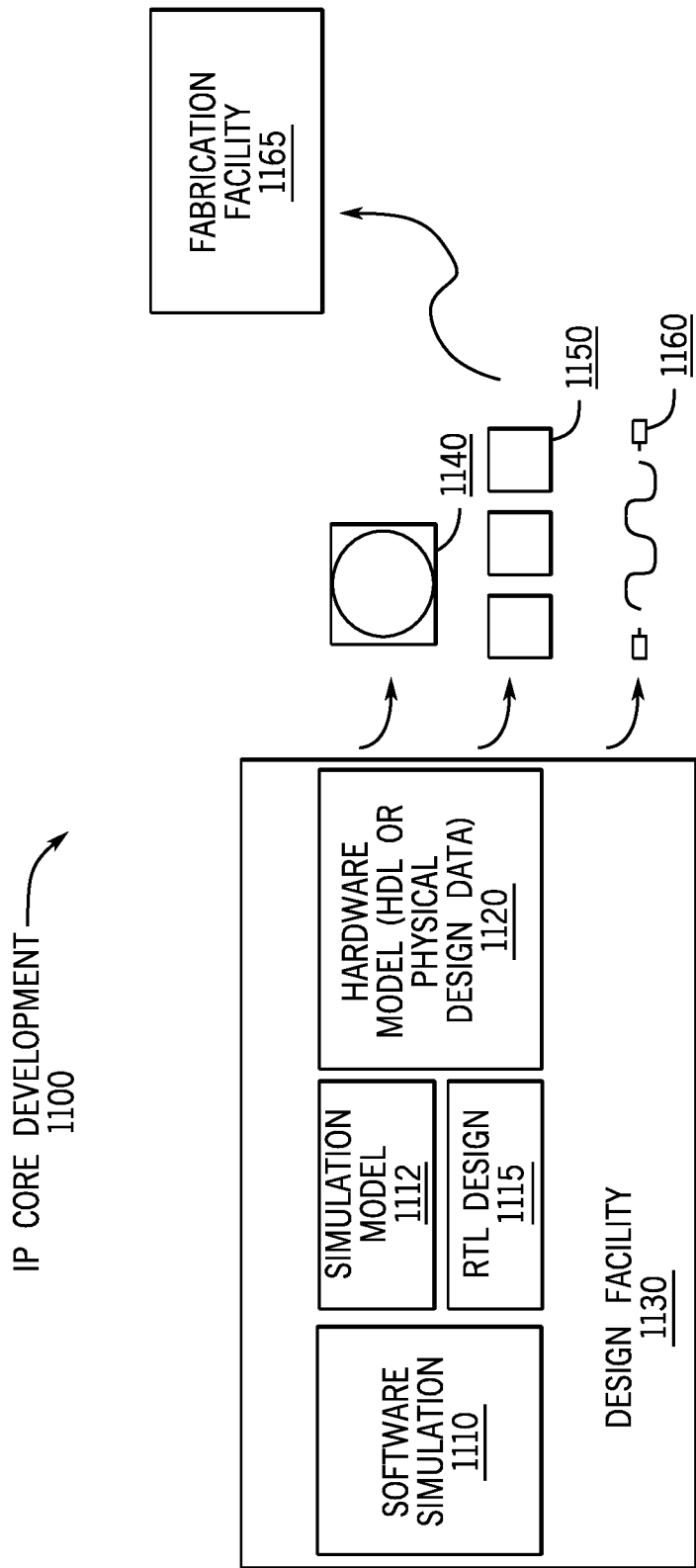


FIG. 18

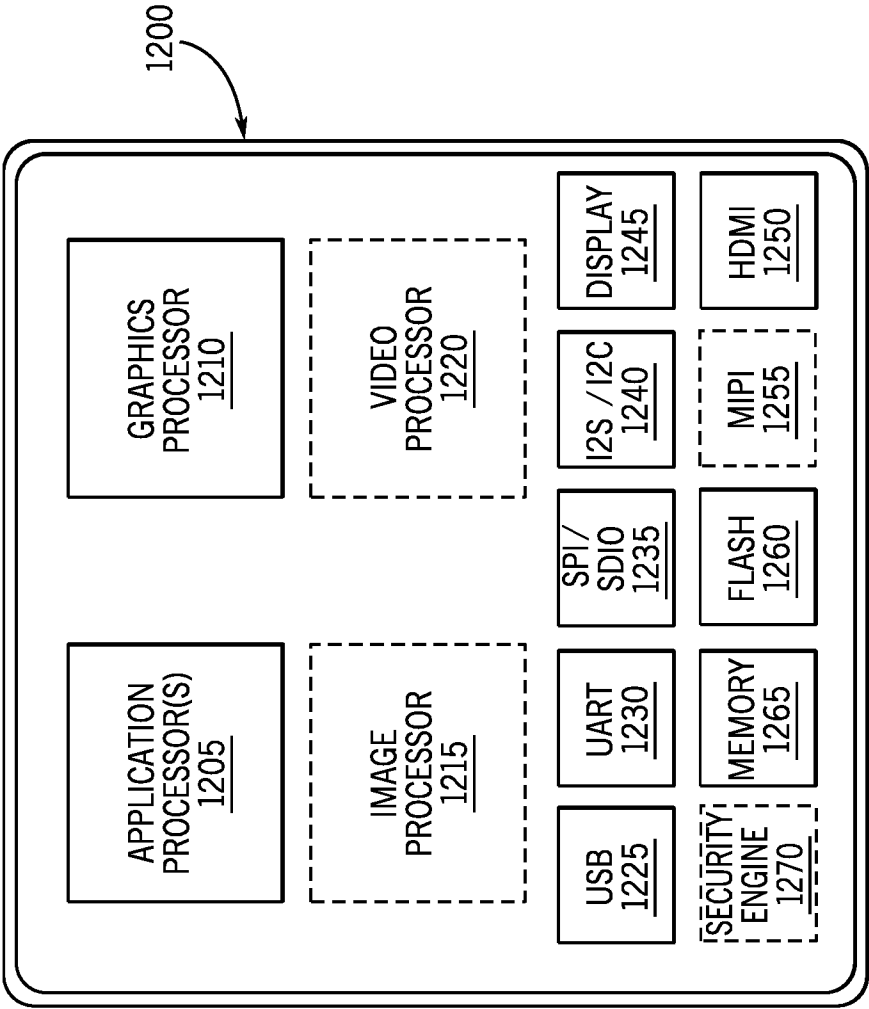


FIG. 19

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/048055**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06T 1/60(2006.01)i**

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

B. FIELDS SEARCHEDMinimum documentation searched (classification system followed by classification symbols)
G06F 9/30; G06F 12/02; G06F 12/10; G09G 5/39; G06T 1/60; G06F 12/08; G09G 5/00Documentation 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 modelsElectronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & keywords: GPU, CPU, swizzling, memory resource, tiled rendering, translation, cache, tag, TLB, page table**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 7932912 B1 (JAMES M. VAN DYKE) 26 April 2011 See column 3, line 59 - column 7, line 7; column 8, lines 56-61; column 10, line 18 - column 15, line 22; and figures 2-5, 14-15.	1-2, 4, 7, 10-12 , 15-17, 20-22, 25
A		3, 5-6, 8-9, 13-14 , 18-19, 23-24
A	US 2014-0164704 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 12 June 2014 See paragraphs [0076]-[0086]; and figures 6-7.	1-25
A	US 2009-0244074 A1 (JOHN S. MONTRYM et al.) 01 October 2009 See paragraphs [0023]-[0035]; claim 1; and figures 3A-6.	1-25
A	US 2014-0201495 A1 (NEVIN HYUSEINOVA et al.) 17 July 2014 See paragraphs [0021]-[0033]; and figures 1-2.	1-25
A	US 2007-0126756 A1 (DAVID B. GLASCO et al.) 07 June 2007 See paragraphs [0033]-[0049]; and figures 6-9.	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

28 November 2016 (28.11.2016)

Date of mailing of the international search report

29 November 2016 (29.11.2016)

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

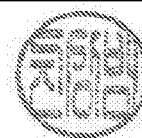


Facsimile No. +82-42-481-8578

Authorized officer

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2016/048055

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 7932912 B1	26/04/2011	US 7884829 B1	08/02/2011
US 2014-0164704 A1	12/06/2014	CN 103870397 A	18/06/2014
		US 2014-0164703 A1	12/06/2014
		US 9239791 B2	19/01/2016
		US 9244840 B2	26/01/2016
US 2009-0244074 A1	01/10/2009	US 7545382 B1	09/06/2009
		US 7859541 B2	28/12/2010
US 2014-0201495 A1	17/07/2014	US 9262336 B2	16/02/2016
		WO 2013-095644 A1	27/06/2013
US 2007-0126756 A1	07/06/2007	US 7616218 B1	10/11/2009
		US 8359454 B2	22/01/2013