

REVERSED POLISH NOTATION PROCESSING ELEMENTS

TECHNICAL FIELD

Embodiments generally relate to computer processing. More particularly, embodiments relate to the use of Reversed Polish Notation (RPN) processing elements.

BACKGROUND

10 Conventional graphics processing unit (GPU) architectures may include graphics processing elements having relatively large register files with, for example, hundreds of registers, wherein the registers may be used to store different types of data. Additionally, each graphics processing element may support hardware multithreading in which each thread maps to a separate copy of the register file. Current solutions may apply power to all registers in the register files during processing, which may have a negative impact on power efficiency, battery life and/or performance. Moreover, the operating system (OS) of the computing system may use a shader compiler to compile shader instructions into a lower-level shader language for execution on the graphics processing elements, wherein the shader compiler may
20 conduct a time consuming register allocation process (e.g., nonlinear graph coloring) that may further reduce power efficiency, battery life and/or performance.

BRIEF DESCRIPTION OF THE DRAWINGS

The various advantages of the embodiments will become apparent to one skilled in the art by reading the following specification and appended claims, and by referencing the following drawings, in which:

FIG. 1 is a block diagram of an example of a Reverse Polish Notation (RPN) program stream according to an embodiment;

30 FIG. 2 is a block diagram of an example of a method of operating a shader compiler according to an embodiment;

FIG. 3 is a flowchart of an example of a method of processing RPN program streams according to an embodiment;

FIG. 4 is a block diagram of an example of a processing element according to an embodiment;

FIG. 5 is a block diagram of an example of a multithreaded processing element according to an embodiment;

FIGs. 6-8 are block diagrams of an example of an overview of a data processing system according to an embodiment;

FIG. 9 is a block diagram of an example of a graphics processing engine according to an embodiment;

FIGs. 10-12 are block diagrams of examples of execution units according to an embodiment;

FIG. 13 is a block diagram of an example of a graphics pipeline according to an embodiment;

FIGs. 14A-14B are block diagrams of examples of graphics pipeline programming according to an embodiment; and

FIG. 15 is a block diagram of an example of a graphics software architecture according to an embodiment.

DESCRIPTION OF EMBODIMENTS

Turning now to FIG. 1, a computing architecture is shown in which a data interface 10 such as, for example, a network controller, memory controller, bus, etc., obtains one or more shader instructions 12, which may be provided to a shader compiler 14 on a host processor 16 (e.g., central processing unit/CPU). The shader instructions 12 may generally facilitate the rendering of graphical content such as, for example, images, video, animations, etc., as part of a three-dimensional (3D) graphics application or other application. The shader instructions 12 may be in a high-level shader language, wherein the illustrated shader compiler 14 compiles the shader instructions 12 into a Reverse Polish Notation (RPN) program stream 18. As will be discussed in greater detail, the RPN program stream 18 may include lower-level operands and operations, wherein each operation follows its corresponding operands in the program stream 18.

The RPN program stream 18 may be executed by a graphics processor 20 associated with a register file 22 with one or more register stacks that are used to store the operands and other arguments during execution of the operations in the RPN program stream 18. The illustrated register file 22 is power gated so that only "live" registers (e.g., registers within the live range of a given variable) are powered on during operation. Accordingly, the illustrated computing architecture may achieve

greater power efficiency, battery life and/or performance. Moreover, the generation of the RPN program stream 18 rather than a traditional instruction set may enable the shader compiler 14 to bypass time consuming register allocation processes such as, for example, nonlinear graph coloring.

FIG. 2 shows a method 24 of operating a shader compiler. The method 24 may generally be implemented in a shader compiler such as, for example, the shader compiler 14 (FIG. 1), already discussed. More particularly, the method 24 may be implemented as one or more modules in a set of logic instructions stored in a machine- or computer-readable storage medium such as random access memory (RAM), read only memory (ROM), programmable ROM (PROM), firmware, flash memory, etc., in configurable logic such as, for example, programmable logic arrays (PLAs), field programmable gate arrays (FPGAs), complex programmable logic devices (CPLDs), in fixed-functionality hardware logic using circuit technology such as, for example, application specific integrated circuit (ASIC), complementary metal oxide semiconductor (CMOS) or transistor-transistor logic (TTL) technology, or any combination thereof.

Illustrated processing block 26 provides for receiving one or more shader instructions, wherein the one or more shader instructions may be compiled into an RPN program stream at block 28. As already noted, the RPN program stream may include a set of operands and a set of operations. For example, the following RPN program stream might be generated, wherein the stack may be preloaded so that the symbol of “\$Stack[i]” indexes the i-th location on the stack, and the stack may be read back i-th positions for the results of operations previously completed and already used by other operations (e.g., referred to as “\$StackCurrent[-i]” in the example shown).

```
$Stack[2] 64 UMUL[pop2,push1](1) $Stack[1] UADD[pop2,push1](2)
READ[pop0,push1](3) $Stack[0] 3.14 FMUL[pop1,push1](4) FADD[pop2,push1] (5)
$StackCurrent[-1] WRITE[pop3](6)
```

Where:

UMUL is an unsigned integer multiplication operation (e.g., using the argument stored in register location “\$Stack[2]” and the value “64”), additionally for that operation options such as *pop2* specifies that the register stack is to be popped back 2 positions, and option *push1* specifies that the operation pushes 1 result to the register stack. Before the UMUL operation (1), the register stack has depth 4, locations \$Stack[0-2] are already storing some values (the register stack is preloaded

when threads starts – i.e., pushed with, for example, shader input values), and \$Stack[3] is storing value of 64. After the UMUL operation (1), the register stack has depth 3, locations \$Stack[0-1] are not modified, and location \$Stack[2] stores result of the operation (1).

UADD is an unsigned integer addition operation (2) (e.g., using the argument resulting from the UMUL operation (1) and the value stored in register location “\$Stack[1]”. Additionally for that operation, options such as *pop2* specifies that the register stack shall be pop back 2 positions, and option *push1* specifies that the operation pushes 1 result to the register stack. After the UADD operation (2), the registers stack has depth 2, location \$Stack[0] is unmodified, \$Stack[1] stores result of operation (2).

READ is a read operation (3) (to external memory, for example) using a single operand as an address. Additionally, for that operation options such as *pop0* specifies that the register stack shall be not pop back any positions, and option *push1* specifies that the operation pushes 1 result to the register stack. After the operation (3), the registers stack has depth 3, locations \$Stack[0-1] are unmodified, \$Stack[2] stores result of operation (3).

FMUL is a floating point multiplication operation (4) (e.g., using the argument stored in register location “\$Stack[0]” and the value “3.14”). Additionally, for that operation options such as *pop1* specifies that the register stack is to be popped back 1 position, and option *push1* specifies that the operation pushes 1 result to the register stack. Before the FMUL operation (4), the register stack has depth 4, with the following locations: \$Stack[0] is unmodified since the start of the program; \$Stack[1] stores the result of operation (2); \$Stack[2] stores the result of READ operation (3); and \$Stack[4] has been pushed with the value of “3.14”. After the operation (4), the register stack has depth 4, wherein locations \$Stack[0-2] are unmodified, and \$Stack[3] stores result of operation (4).

FADD is a floating point addition operation (5) (e.g., using two input operands, the result of the FMUL operation from location \$Stack[4] and the result of READ operation (3)). Additionally, for that operation options such as *pop2* specifies that the register stack shall be pop back 2 positions, and operation *push1* specifies that the operation pushes 1 result to the register stack. After the operation (5), the register stack has depth 3, locations \$Stack[0-1] are unmodified, and \$Stack[2] stores result of operation (5).

WRITE is a write operation (6) (to an external memory, for example) using two operands (e.g., second operand is the address obtained from $\$StackCurrent[-1]$). The stack depth is 3, so $\$StackCurrent$ is equal to $\$Stack[2]$, $\$StackCurrent[-1]$ is equal to $\$Stack[1]$ and the first operand is data to store from $\$Stack[2]$). In the illustrated example, without using $\$StackCurrent[-1]$, the same stack element may be fetched for the operation. Accordingly, this example is only to illustrate the RPN language flexibility to address indirectly the register stack locations. Additionally, for the operation (6) options such as *pop3* specifies that the register stack is to be popped back 3 positions. After the operation (6), the register stack has depth 0, so none of its registers are powered on.

Thus, the example RPN program stream shown above may correspond to the following pseudo code:

```
A = $Stack[2]; B = $Stack[1]; C = $Stack[0];
Variable = Read at address[ A * 64 + B ];
Variable = Variable + 3.14 * C;
Write at address[A * 64 + B] of value ( Variable )
```

Other operations such as, for example, “UDIV” to represent unsigned divide operations, “FDIV” to represent floating point divide operations, and so forth, may be defined for the RPN program stream architecture. In this example, the peak stack use is four registers after the time of the FMUL operation (4). Accordingly, all registers other than the four live registers may be powered off in such a case for the whole program lifetime. Also, during each operation, there no need to power on more registers than the current value R of the stack depth. Moreover, block 28 may include bypassing a register allocation stage during compilation of the one or more shader instructions because the RPN program stream may be generated without considering register allocation.

FIG. 3 shows a method 30 of processing RPN program streams. The method 30 may generally be implemented in a graphics processor such as, for example, the graphics processor 20 (FIG. 1), already discussed. More particularly, the method 24 may be implemented as one or more modules in a set of logic instructions stored in a machine- or computer-readable storage medium such as RAM, ROM, PROM, firmware, flash memory, etc., in configurable logic such as, for example, PLAs,

FPGAs, CPLDs, in fixed-functionality hardware logic using circuit technology such as, for example, ASIC, CMOS or TTL technology, or any combination thereof.

Illustrated processing block 32 provides for receiving an RPN program stream including a set of operands and a set of operations, wherein a register stack may be populated at block 34 with one or more operands in the set of operands. As will be discussed in greater detail, block 34 may include determining whether a thread switch condition exists in a multithreaded architecture. The thread switch condition may trigger a corresponding switch of register stacks. Block 36 may selectively power off one or more registers in the register stack based on a stack depth of the register stack.

10 The stack depth may generally correspond to the number of live registers in the register stack.

Additionally, one or more arguments may be read from the register stack at block 38. Illustrated block 40 conducts an execution of one or more operations on the argument(s). Block 42 may determine whether one or more results of the execution is to be output from the processing element. If so, illustrated block 44 sends the result(s) of the execution to an input output (IO) transmission pipeline. Otherwise, the register stack may be populated with the result(s) of the execution at block 46.

FIG. 4 shows a processing element 50 that may be used to process RPN
20 program streams. The processing element 50 may therefore implement a method such as, for example, the method 30 (FIG. 3), in a graphics processor such as, for example, the graphics processor 20 (FIG. 1), already discussed. The RPN program stream may, for example, be compiled from one or more shader instructions at the graphics processor 20 (FIG. 1), already discussed, at the shader compiler 14 (FIG. 1), already discussed, and so on, or any combination thereof. In the illustrated example, a program streamer interface 52 receives the RPN program stream and submits the RPN program stream to an operation decoder 54. The operation decoder 54 may parse the RPN program stream to identify stack immediate arguments such as values “64” and “3.14” in the sample RPN program discussed above. The stack immediate
30 arguments may be placed into a push pipeline 56, which may in turn feed the stack immediate arguments to a stack allocator 58a in a stack unit 58 (58a-58c). The stack unit 58 may be included in a register file such as, for example, the register file 22 (FIG. 1), already discussed.

The illustrated operation decoder 54 also places operand read commands into an argument fetcher 60. In response to the read commands, the argument fetcher 60 may use a set of operand retrievers 62 (62a-62b) to read arguments from a register stack 58b in the stack unit 58. The argument fetcher 60 may send the arguments read from the register stack 58b to either an executor 64 (64a-64c) or an input output (IO) transmission pipeline 66. The IO transmission pipeline 66 may issue read and write requests 68 to samplers, memory devices and/or other system components and receive read responses 70 to the requests 68, wherein data in the read responses 70 may also be sent to the stack allocator 58a for storage in the register stack 58b.

10 The illustrated executor 64 includes an arithmetic logic unit (ALU)/floating point unit (FPU) pipeline 64a that feeds the arguments received from the argument fetcher 60 into an ALU/FPU stage 64b. The ALU/FPU stage 64b may conduct executions of the operations in the RPN program stream on the arguments, wherein an accumulator 64c may send the results of the executions either to the stack allocator 58a for storage in the register stack 58b or to the IO transmission pipeline 66 for transmission to other system components.

Of particular note is that the illustrated stack unit 58 includes a power gating unit 58c that selectively powers off registers in the register stack 58b based on the stack depth of the register stack 58b. For example, the counter 22 may subtract the
20 number of “pops” from the register stack 58b from the number of “pushes” to the register stack 58b to determine the stack depth. A counter 72 may be used to track the stack depth, which may generally correspond to the number of live registers (e.g., in a register range “R”) in the register stack 58b. As already noted, the selectively powering off registers in the register stack 58b may increase power efficiency, battery life and/or performance.

Turning now to FIG. 5, a multithreaded processing element 70 is shown. In the illustrated example, a program streamer interface 76 receives the RPN program stream and submits the RPN program stream to the operation decoder 54. The operation decoder 54 may parse the RPN program stream to identify stack immediate
30 arguments, as already discussed. The stack immediate arguments may be placed into the push pipeline 56, which may in turn feed the stack immediate arguments to an interface 78 to a plurality of stack units 80 (80a-80n). More particularly, the stack immediate arguments may be directed to a stack unit 80 that corresponds to the current thread being processed. Each stack unit 80 may include a stack allocator, a

register stack and a power gating unit, as already discussed. Thus, for example, a first stack unit 80a might receive the stack immediate arguments associated with a first thread, a second stack unit 80b may receive the stack immediate arguments associated with a second thread, and so forth. The stack units 80 may be included in a register file such as, for example, the register file 22 (FIG. 1), already discussed.

The illustrated processing element 70 also includes an argument fetcher 82 that uses the set of operand retrievers 62 to read arguments from the register stacks in the stack units 80. The argument fetcher 82 may also include a thread controller 84 to detect a thread switch condition and issue a thread switch command 86 to the program streamer interface 76. The thread condition might include, for example, an inability to read one or more arguments because they are not ready (e.g., arguments have not arrived from an external unit such as a sampler). Upon receiving the thread switch command 86, the illustrated program streamer interface 76 passes a different RPN program stream to the operation decoder 54. As a result, a register stack in a different stack unit 80 may be populated with operands when the thread condition is detected. The illustrated approach may therefore further enhance the power efficiency, battery life and/or performance advantages already provided by register power gating and RPN program stream generation.

Overview – FIGs. 6-8

FIG. 6 is a block diagram of a data processing system 100, according to an embodiment. The data processing system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In one embodiment, the data processing system 100 is a system on a chip (SOC) integrated circuit for use in mobile, handheld, or embedded devices.

An embodiment of the data processing system 100 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In one embodiment, the data processing system 100 is a mobile phone, smart phone, tablet computing device or mobile Internet device. The 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 one embodiment, the 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.

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 one embodiment, each of the one or more processor cores 107 is configured to process a specific instruction set 109. The 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
10 processor cores 107 may each process a different instruction set 109 which may include instructions to facilitate the emulation of other instruction sets. A processor core 107 may also include other processing devices, such a digital signal processor (DSP).

In one embodiment, 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 one embodiment, the cache memory is shared among various components of the processor 102. In one embodiment, 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 the processor cores 107 using known cache
20 coherency techniques. A register file 106 is additionally included in the 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. In one example, the register file 106 is the same as the register file 22 (FIG. 1) and the instruction set 109 is an RPN program stream architecture, already discussed.

The processor 102 is coupled to a processor bus 110 to transmit data signals between the processor 102 and other components in the system 100. The system 100 uses an exemplary 'hub' system architecture, including a memory controller hub 116
30 and an input output (I/O) controller hub 130. The memory controller hub 116 facilitates communication between a memory device and other components of the system 100, while the I/O controller hub (ICH) 130 provides connections to I/O devices via a local I/O bus.

The memory device 120, can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or some other memory device having suitable performance to serve as process memory. The memory 120 can store data 122 and instructions 121 for use when the processor 102 executes a process. The 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 the processors 102 to perform graphics and media operations.

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

FIG. 7 is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-N, an integrated memory controller 214, and an integrated
20 graphics processor 208. The processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of the cores 202A-N includes one or more internal cache units 204A-N. In one embodiment each core also has access to one or more shared cached units 206.

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

The processor 200 may also include a set of one or more bus controller units 216 and a system agent 210. The one or more bus controller units 216 manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses

(e.g., PCI, PCI Express). The system agent 210 provides management functionality for the various processor components. In one embodiment, the system agent 210 includes one or more integrated memory controllers 214 to manage access to various external memory devices (not shown).

In one embodiment, one or more of the cores 202A-N include support for simultaneous multi-threading. In such embodiment, the system agent 210 includes components for coordinating and operating cores 202A-N during multi-threaded processing. The system agent 210 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of the cores
10 202A-N and the graphics processor 208.

The processor 200 additionally includes a graphics processor 208 to execute graphics processing operations. In one embodiment, the graphics processor 208 couples with the set of shared cache units 206, and the system agent unit 210, including the one or more integrated memory controllers 214. In one embodiment, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. The 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 210.

In one embodiment a ring based interconnect unit 212 is used to couple the
20 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 one embodiment, the graphics processor 208 couples with the ring interconnect 212 via an I/O link 213.

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 one embodiment each of the cores 202-N and the graphics processor 208 use the embedded memory modules 218 as shared last level cache.

30 In one embodiment cores 202A-N are homogenous cores executing the same instruction set architecture. In another embodiment, the cores 202A-N are heterogeneous in terms of instruction set architecture (ISA), where one or more of the 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.

The processor 200 can be a part of or implemented on one or more substrates using any of a number of process technologies, for example, Complementary metal-oxide-semiconductor (CMOS), Bipolar Junction/Complementary metal-oxide-semiconductor (BiCMOS) or N-type metal-oxide-semiconductor logic (NMOS). Additionally, the processor 200 can be implemented on one or more chips or as a system on a chip (SOC) integrated circuit having the illustrated components, in addition to other components.

FIG. 8 is a block diagram of one embodiment of a graphics processor 300 which may be a discrete graphics processing unit, or may be graphics processor
10 integrated with a plurality of processing cores. In one embodiment, the graphics processor is communicated with via a memory mapped I/O interface to registers on the graphics processor and via commands placed into the processor memory. The graphics processor 300 includes a memory interface 314 to access memory. The 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.

The graphics processor 300 also includes a display controller 302 to drive display output data to a display device 320. The 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 one embodiment the graphics processor
20 300 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.

In one embodiment, the 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,
30 2D graphics operations are performed using one or more components of the graphics-processing engine (GPE) 310. The graphics-processing engine 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

The 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 the 3D pipeline 312 can be used to perform media operations, an embodiment of the GPE 310 also includes a media pipeline 316 that is specifically used to perform media operations, such as video post processing and image enhancement.

10 In one embodiment, the 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 the video codec engine 306. In one embodiment, the media pipeline 316 additionally includes a thread spawning unit to spawn threads for execution on the 3D/Media sub-system 315. The spawned threads perform computations for the media operations on one or more graphics execution units included in the 3D/Media sub-system.

The 3D/Media subsystem 315 includes logic for executing threads spawned by the 3D pipeline 312 and media pipeline 316. In one embodiment, the pipelines send thread execution requests to the 3D/Media subsystem 315, which includes thread
20 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 one embodiment, the 3D/Media subsystem 315 includes one or more internal caches for thread instructions and data. In one embodiment, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

3D/Media Processing - FIG. 9

FIG. 9 is a block diagram of an embodiment of a graphics processing engine 410 for a graphics processor. In one embodiment, the graphics processing engine (GPE) 410 is a version of the GPE 310 shown in FIG. 8. The GPE 410 includes a 3D
30 pipeline 412 and a media pipeline 416, each of which can be either different from or similar to the implementations of the 3D pipeline 312 and the media pipeline 316 of FIG. 8.

In one embodiment, the GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. The

command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer 403 receives commands from the memory and sends the commands to the 3D pipeline 412 and/or media pipeline 416. The 3D and media pipelines process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the execution unit array 414. In one embodiment, the 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 the GPE 410.

10 A sampling engine 430 couples with memory (e.g., cache memory or system memory) and the execution unit array 414. In one embodiment, the sampling engine 430 provides a memory access mechanism for the scalable execution unit array 414 that allows the execution unit array 414 to read graphics and media data from memory. In one embodiment, the sampling engine 430 includes logic to perform specialized image sampling operations for media.

The specialized media sampling logic in the sampling engine 430 includes a de-noise / de-interlace module 432, a motion estimation module 434, and an image scaling and filtering module 436. The 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
20 content into a single frame of video. The de-noise logic reduces or removes data noise from video and image data. In one embodiment, 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 one embodiment, the de-noise / de-interlace module 432 includes dedicated motion detection logic (e.g., within the motion estimation engine 434).

The 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
30 video frames. In one embodiment, a graphics processor media codec uses the video motion estimation engine 434 to perform operations on video at the macro-block level that may otherwise be computationally intensive to perform using a general-purpose processor. In one embodiment, the 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.

The image scaling and filtering module 436 performs image-processing operations to enhance the visual quality of generated images and video. In one embodiment, the scaling and filtering module 436 processes image and video data during the sampling operation before providing the data to the execution unit array 414.

10 In one embodiment, the graphics processing engine 410 includes a data port 444, which provides an additional mechanism for graphics subsystems to access memory. The 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 one embodiment, the 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 one embodiment, threads executing on an execution unit in the 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 the graphics processing engine 410.

20 Execution Units – FIGs. 10-12

FIG. 10 is a block diagram of another embodiment of a graphics processor. In one embodiment, the graphics processor includes a ring interconnect 502, a pipeline front-end 504, a media engine 537, and graphics cores 580A-N. The ring interconnect 502 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In one embodiment, the graphics processor is one of many processors integrated within a multi-core processing system.

30 The graphics processor receives batches of commands via the ring interconnect 502. The incoming commands are interpreted by a command streamer 503 in the pipeline front-end 504. The graphics processor includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 580A-N. For 3D geometry processing commands, the command streamer 503 supplies the commands to the geometry pipeline 536. For at least some media processing commands, the command streamer 503 supplies the commands to a video

front end 534, which couples with the media engine 537. The 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. The geometry pipeline 536 and media engine 537 each generate execution threads for the thread execution resources provided by at least one graphics core 580A.

The graphics processor includes scalable thread execution resources featuring modular cores 580A-N (sometime referred to as core slices), each having multiple sub-cores 550A-N, 560A-N (sometimes referred to as core sub-slices). The graphics processor can have any number of graphics cores 580A through 580N. In one embodiment, the graphics processor includes a graphics core 580A having at least a first sub-core 550A and a second core sub-core 560A. In another embodiment, the graphics processor is a low power processor with a single sub-core (e.g., 550A). In one embodiment, the graphics processor includes multiple graphics cores 580A-N, each including a set of first sub-cores 550A-N and a set of second sub-cores 560A-N. Each sub-core in the set of first sub-cores 550A-N includes at least a first set of execution units 552A-N and media/texture samplers 554A-N. Each sub-core in the set of second sub-cores 560A-N includes at least a second set of execution units 562A-N and samplers 564A-N. In one embodiment, each sub-core 550A-N, 560A-N shares a set of shared resources 570A-N. In one embodiment, 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.

FIG. 11 illustrates thread execution logic 600 including an array of processing elements employed in one embodiment of a graphics processing engine. In one embodiment, the thread execution logic 600 includes a pixel shader 602, a thread dispatcher 604, instruction cache 606, a scalable execution unit array including a plurality of execution units 608A-N, a sampler 610, a data cache 612, and a data port 614. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. The thread execution logic 600 includes one or more connections to memory, such as system memory or cache memory, through one or more of the instruction cache 606, the data port 614, the sampler 610, and the execution unit array 608A-N. In one embodiment, 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. The execution unit array 608A-N includes any number individual execution units.

In one embodiment, the execution unit array 608A-N is primarily used to execute “shader” programs. In one embodiment, the execution units in the array 608A-N execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders).

Each execution unit in the execution unit array 608A-N operates on arrays of data elements. The number of data elements is the “execution size,” or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical ALUs or FPUs for a particular graphics processor. The execution units 608A-N support integer and floating-point data types.

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.

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 one embodiment, one or more data caches (e.g., 612) are included to cache thread data during thread execution. A sampler 610 is included to provide texture sampling for 3D operations and media sampling for media operations. In one embodiment, the 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.

During execution, the graphics and media pipelines send thread initiation requests to the thread execution logic 600 via thread spawning and dispatch logic. The thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-N. For example, the geometry pipeline (e.g., 536 of FIG. 10) dispatches vertex processing, tessellation, or geometry processing threads to the thread execution logic 600. The thread dispatcher
10 604 can also process runtime thread spawning requests from the executing shader programs.

Once a group of geometric objects have been processed and rasterized into pixel data, the 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 one embodiment, the pixel shader 602 calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel shader 602 then executes an API-supplied pixel shader program. To execute the pixel shader program, the pixel shader 602 dispatches threads to an execution unit (e.g., 608A) via the thread dispatcher 604. The pixel shader 602 uses texture
20 sampling logic in the 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.

In one embodiment, 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 one embodiment, 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.

FIG. 12 is a block diagram illustrating a graphics processor execution unit
30 instruction format according to an embodiment. In one 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. The instruction

format 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.

In one embodiment, 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 varies by
10 embodiment. In one embodiment, 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.

For each format, an 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. An
20 instruction control field 714 enables control over certain execution options, such as channel 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. The exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

Some execution unit instructions have up to three operands including two source operands, src0 720, src1 722, and one destination 718. In one embodiment, 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.
30 An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In one embodiment instructions are grouped based on opcode bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is

exemplary. In one embodiment, a move and logic opcode group 742 includes data movement and logic instructions (e.g., mov, cmp). The move and logic group 742 shares the five most significant bits (MSB), where move instructions are in the form of 0000xxxxb (e.g., 0x0x) and logic instructions are in the form of 0001xxxxb (e.g., 0x01). A flow control instruction group 744 (e.g., call, 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, 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. In one example, the illustrated instruction set architecture is replaced with RPN program streams as discussed herein.

Graphics Pipeline – FIG. 13

FIG. 13 is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline 820, a media pipeline 830, a display engine 840, thread execution logic 850, and a render output pipeline 870. In one embodiment, the graphics processor 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 the graphics processor via a ring interconnect 802. The ring interconnect 802 couples the graphics processor to other processing components, such as other graphics processors or general-purpose processors. Commands from the ring interconnect are interpreted by a command streamer 803 which supplies instructions to individual components of the graphics pipeline 820 or media pipeline 830.

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

In one embodiment, the execution units 852A, 852B are an array of vector processors having an instruction set for performing graphics and media operations. The 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.

In one embodiment, the graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. A programmable hull shader 811 configures the tessellation operations. A
10 programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of the 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 the graphics pipeline 820. If tessellation is not used, the tessellation components 811, 813, 817 can be bypassed.

The complete geometric objects can be processed by a geometry shader 819 via one or more threads dispatched to the execution units 852A, 852B, or can proceed directly to the clipper 829. 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
20 the vertex shader 807. The geometry shader 819 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

Prior to rasterization, vertex data is processed by a clipper 829, which is either a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In one embodiment, a rasterizer 873 in the render output pipeline 870 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In one embodiment, pixel shader logic is included in the thread execution logic 850.

The graphics engine has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the
30 major components of the graphics engine. In one embodiment the 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 graphics engine. In

one embodiment, the sampler 854, caches 851, 858 and execution units 852A, 852B each have separate memory access paths.

In one embodiment, the render output pipeline 870 contains a rasterizer and depth test component 873 that converts vertex-based objects into their associated pixel-based representation. In one embodiment, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render and depth buffer caches 878, 879 are also available in one embodiment. A pixel operations component 877 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations
10 (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 one embodiment a shared L3 cache 875 is available to all graphics components, allowing the sharing of data without the use of main system memory.

The graphics processor media pipeline 830 includes a media engine 837 and a video front end 834. In one embodiment, the video front end 834 receives pipeline commands from the command streamer 803. However, in one embodiment the media pipeline 830 includes a separate command streamer. The video front-end 834 processes media commands before sending the command to the media engine 837. In one embodiment, the media engine 837 includes thread spawning functionality to
20 spawn threads for dispatch to the thread execution logic 850 via the thread dispatcher 831.

In one embodiment, the graphics engine includes a display engine 840. In one embodiment, the display engine 840 is external to the graphics processor and couples with the graphics processor via the ring interconnect 802, or some other interconnect bus or fabric. The display engine 840 includes a 2D engine 841 and a display controller 843. The display engine 840 contains special purpose logic capable of operating independently of the 3D pipeline. The 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 an display device
30 connector.

The 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 one embodiment, 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 various embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) supported by the Khronos Group, the Direct3D library from the Microsoft Corporation, or, in one embodiment, 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.

Graphics Pipeline Programming – FIGs. 14A-B

10 FIG. 14A is a block diagram illustrating a graphics processor command format according to an embodiment and FIG. 14B is a block diagram illustrating a graphics processor command sequence according to an embodiment. The solid lined boxes in FIG. 14A 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 FIG. 14A 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.

20 The client 902 specifies the client unit of the graphics device that processes the command data. In one embodiment, 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 one embodiment, 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 the data 906 field of the command. For some
30 commands an explicit command size 908 is expected to specify the size of the command. In one embodiment, the command parser automatically determines the size of at least some of the commands based on the command opcode. In one embodiment commands are aligned via multiples of a double word.

The flow chart in FIG. 14B shows a sample command sequence 910. In one embodiment, software or firmware of a data processing system that features an embodiment of the 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 exemplary purposes, however embodiments are not limited to these 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 an at least partially concurrent manner.

10 The sample command sequence 910 may begin with a pipeline flush command 912 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In one embodiment, 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. A pipeline flush command 912 can be used for pipeline synchronization or before placing the graphics processor into a low power
20 state.

A pipeline select command 913 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. 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 one embodiment, a pipeline flush command 912 is required immediately before a pipeline switch via the pipeline select command 913.

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. The pipeline control command 914 configures the pipeline state for the active pipeline. In one
30 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.

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. The graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. The return buffer state commands 916 include selecting the size and number of return buffers to use for a set of pipeline operations.

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.

10 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. 3D pipeline state 930 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

20 The 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. The 3D primitive 932 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, the 3D pipeline 922 dispatches shader execution threads to graphics processor execution units.

30 The 3D pipeline 922 is triggered via an execute 934 command or event. In one embodiment a register write triggers command execution. In one embodiment 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.

The sample command sequence 910 follows the media pipeline 924 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 924 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. 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.

The 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. The 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. The media pipeline state commands 940 also support the use of one or more pointers to “indirect” state elements that contain a batch of state settings.

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 one embodiment, all media pipeline state 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 944 command or an equivalent execute event (e.g., register write). Output from the media pipeline 924 may then be post processed by operations provided by the 3D pipeline 922 or the media pipeline 924. In one embodiment, GPGPU operations are configured and executed in a similar manner as media operations.

Graphics Software Architecture – FIG. 15

FIG. 15 illustrates exemplary graphics software architecture for a data processing system according to an embodiment. The software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. The 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.

In one embodiment, the 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.

10 The operating system 1020 may be 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. In one example, the shader compiler 1024 is the same as the shader compiler 14 (FIG. 1), already discussed. The compilation may be a just-in-time compilation or the application can perform share pre-compilation. In one embodiment, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 1010.

20 The user mode graphics driver 1026 may contain 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. The user mode graphics driver uses operating system kernel mode functions 1028 to communicate with a kernel mode graphics driver 1029. The kernel mode graphics driver 1029 communicates with the graphics processor 1032 to dispatch commands and instructions.

30 To the extent various operations or functions are described herein, they can be described or defined as hardware circuitry, software code, instructions, configuration, and/or data. The content can be embodied in hardware logic, or as directly executable software ("object" or "executable" form), source code, high level shader code designed for execution on a graphics engine, or low level assembly language code in an instruction set for a specific processor or graphics core. The software content of the embodiments described herein can be provided via an article of manufacture with

the content stored thereon, or via a method of operating a communication interface to send data via the communication interface.

A non-transitory machine readable storage medium can cause a machine to perform the functions or operations described, and includes any mechanism that stores information in a form accessible by a machine (e.g., computing device, electronic system, etc.), such as recordable/non-recordable media (e.g., read only memory (ROM), random access memory (RAM), magnetic disk storage media, optical storage media, flash memory devices, etc.). A communication interface includes any mechanism that interfaces to any of a hardwired, wireless, optical, etc., medium to
10 communicate to another device, such as a memory bus interface, a processor bus interface, an Internet connection, a disk controller, etc. The communication interface is configured by providing configuration parameters or sending signals to prepare the communication interface to provide a data signal describing the software content. The communication interface can be accessed via one or more commands or signals sent to the communication interface.

Various components described can be a means for performing the operations or functions described. Each component described herein includes software, hardware, or a combination of these. The components can be implemented as software modules, hardware modules, special-purpose hardware (e.g., application
20 specific hardware, application specific integrated circuits (ASICs), digital signal processors (DSPs), etc.), embedded controllers, hardwired circuitry, etc. Besides what is described herein, various modifications can be made to the disclosed embodiments and implementations of the invention without departing from their scope. Therefore, the illustrations and examples herein should be construed in an illustrative, and not a restrictive sense. The scope of the invention should be measured solely by reference to the claims that follow.

Additional Notes and Examples:

Example 1 may include a graphics computing system to process shader instructions, comprising a data interface including one or more of a network
30 controller, a memory controller or a bus, the data interface to obtain one or more shader instructions, a graphics processor including a program streamer interface to receive a Reverse Polish Notation (RPN) program stream including a first set of operands and a first set of operations, wherein the RPN program stream is to be compiled from the one or more shader instructions, a first register stack, a first stack

allocator to populate the first register stack with one or more operands in the first set of operands, and a first power gating unit to selectively power off one or more registers in the first register stack based on a stack depth of the first register stack.

Example 2 may include the computing system of Example 1, further including an argument fetcher to read one or more arguments from the first register stack, and an executor to conduct an execution of one or more operations in the first set of operations on the one or more arguments.

Example 3 may include the computing system of any one of Examples 1 to 2, wherein the first stack allocator is to populate the first register stack with one or more results of the execution.

Example 4 may include the computing system of any one of Examples 1 to 3, further including an input output (IO) transmission pipeline, wherein the executor is to send one or more results of the execution to the IO transmission pipeline.

Example 5 may include the computing system of any one of Examples 1 to 4, wherein the RPN program stream is to include a second set of operands and a second set of operations, the computing system further including a second register stack, a thread controller to detect a thread switch condition, and a second stack allocator to populate the second register stack with one or more operands in the second set of operands in response to the thread switch condition.

Example 6 may include the computing system of any one of Examples 1 to 5, further including a second power gating unit to selectively power off one or more registers in the second register stack based on a stack depth of the second register stack.

Example 7 may include the computing system of any one of Examples 1 to 6, wherein the stack depth is to correspond to a number of live registers in the first register stack.

Example 8 may include an apparatus to process RPN program streams, comprising a first register stack, a program streamer interface to receive an RPN program stream including a first set of operands and a first set of operations, a first stack allocator to populate the first register stack with one or more operands in the first set of operands, and a first power gating unit to selectively power off one or more registers in the first register stack based on a stack depth of the first register stack.

Example 9 may include the apparatus of Example 8, further including an argument fetcher to read one or more arguments from the first register stack, and an

executor to conduct an execution of one or more operations in the first set of operations on the one or more arguments.

Example 10 may include the apparatus of any one of Examples 8 to 9, wherein the first stack allocator is to populate the first register stack with one or more results of the execution.

Example 11 may include the apparatus of any one of Examples 8 to 10, further including an input output (IO) transmission pipeline, wherein the executor is to send one or more results of the execution to the IO transmission pipeline.

10 Example 12 may include the apparatus of any one of Examples 8 to 11, wherein the RPN program stream is to include a second set of operands and a second set of operations, the apparatus further including a second register stack, a thread controller to detect a thread switch condition, and a second stack allocator to populate the second register stack with one or more operands in the second set of operands in response to the thread switch condition.

Example 13 may include the apparatus of any one of Examples 8 to 12, further including a second power gating unit to selectively power off one or more registers in the second register stack based on a stack depth of the second register stack.

20 Example 14 may include the apparatus of any one of Examples 8 to 13, wherein the stack depth is to correspond to a number of live registers in the first register stack.

Example 15 may include a method of processing RPN program streams, comprising receiving an RPN program stream including a first set of operands and a first set of operations, populating a first register stack with one or more operands in the first set of operands, and selectively powering off one or more registers in the first register stack based on a stack depth of the first register stack.

Example 16 may include the method of Example 15, further comprising reading one or more arguments from the first register stack, and conducting an execution of one or more operations in the first set of operations on the one or more arguments.

30 Example 17 may include the method of any one of Examples 15 to 16, further comprising populating the first register stack with one or more results of the execution.

Example 18 may include the method of any one of Examples 15 to 17, further comprising sending one or more results of the execution to an input output (IO) transmission pipeline.

Example 19 may include the method of any one of Examples 15 to 18, wherein the RPN program stream includes a second set of operands and a second set of operations, the method further including detecting a thread switch condition, and populating a second register stack with one or more operands in the second set of operands in response to the thread switch condition.

10 Example 20 may include the method of any one of Examples 15 to 19, further including selectively powering off one or more registers in the second register stack based on a stack depth of the first register stack.

Example 21 may include the method of any one of Examples 15 to 20, wherein the stack depth corresponds to a number of live registers in the first register stack.

Example 22 may include at least one computer readable storage medium comprising a set of compiler instructions which, when executed by a computing system, cause the computing system to receive one or more shader instructions and compile the one or more shader instructions into an RPN program stream.

20 Example 23 may include the at least one computer readable storage medium of Example 22, wherein the RPN program stream is to include a set of operands and a set of operations.

Example 24 may include the at least one computer readable storage medium of any one of Examples 22 to 23, wherein the compiler instructions, when executed, cause the computing system to bypass a register allocation stage during compilation of the one or more shader instructions.

30 Example 25 may include an apparatus to process Reverse Polish Notation (RPN) program streams, comprising means for receiving an RPN program stream including a first set of operands and a first set of operations, means for populating a first register stack with one or more operands in the first set of operands, and means for selectively powering off one or more registers in the first register stack based on a stack depth of the first register stack.

Example 26 may include the apparatus of Examples 25, further comprising: means for reading one or more arguments from the first register stack, and means for

conducting an execution of one or more operations in the first set of operations on the one or more arguments.

Example 27 may include the apparatus of any one of Examples 25 to 26, further comprising means for populating the first register stack with one or more results of the execution.

Example 28 may include the apparatus of any one of Examples 25 to 27, further comprising means for sending one or more results of the execution to an input output (IO) transmission pipeline.

10 Example 29 may include the apparatus of any one of Examples 25 to 28, wherein the RPN program stream is to include a second set of operands and a second set of operations, the apparatus further including means for detecting a thread switch condition, and means for populating a second register stack with one or more operands in the second set of operands in response to the thread switch condition.

Example 30 may include the apparatus of any one of Examples 25 to 29, further including means for selectively powering off one or more registers in the second register stack based on a stack depth of the first register stack.

Example 31 may include the apparatus of any one of Examples 25 to 30, wherein the stack depth is to correspond to a number of live registers in the first register stack.

20 Example 32 may include a method of processing shader instructions, comprising receiving one or more shader instructions, and compiling the one or more shader instructions into a Reverse Polish Notation (RPN) program stream.

Example 33 may include the method of Example 32, wherein the RPN program stream includes a set of operands and a set of operations.

Example 34 may include the method of Examples 32 to 33, further including bypassing a register allocation stage during compilation of the one or more shader instructions.

30 Example 35 may include an apparatus to process shader instructions comprising a shader compiler to receive one or more shader instructions and compile the one or more shader instructions into a Reverse Polish Notation (RPN) program stream.

Example 36 may include the apparatus of Example 35, wherein the RPN program stream is to include a set of operands and a set of operations.

Example 37 may include the apparatus of any one of Examples 35 to 36, wherein the shader compiler is to bypass a register allocation stage during compilation of the one or more shader instructions.

Thus, techniques may provide a simplified and power efficient graphic processing architecture based on Reverse Polish Notation equation processing (e.g., each shader kernel may be in the form of an RPN program stream). The processing elements may be optimized to process RPN equations. Moreover, gating off unused portions of the register stacks may result in significantly better performance, power efficiency and battery life. For example, the same amount of power may be sufficient
10 to supply an order of magnitude more registers relative to conventional ISA solutions. Accordingly, each processing element may implement substantially more hardware threads (e.g., seventy RPN-based hardware threads versus seven conventional hardware threads) for the same power budget.

Embodiments are applicable for use with all types of semiconductor integrated circuit ("IC") chips. Examples of these IC chips include but are not limited to processors, controllers, chipset components, programmable logic arrays (PLAs), memory chips, network chips, systems on chip (SoCs), SSD/NAND controller ASICs, and the like. In addition, in some of the drawings, signal conductor lines are represented with lines. Some may be different, to indicate more constituent signal
20 paths, have a number label, to indicate a number of constituent signal paths, and/or have arrows at one or more ends, to indicate primary information flow direction. This, however, should not be construed in a limiting manner. Rather, such added detail may be used in connection with one or more exemplary embodiments to facilitate easier understanding of a circuit. Any represented signal lines, whether or not having additional information, may actually comprise one or more signals that may travel in multiple directions and may be implemented with any suitable type of signal scheme, e.g., digital or analog lines implemented with differential pairs, optical fiber lines, and/or single-ended lines.

Example sizes/models/values/ranges may have been given, although
30 embodiments are not limited to the same. As manufacturing techniques (e.g., photolithography) mature over time, it is expected that devices of smaller size could be manufactured. In addition, well known power/ground connections to IC chips and other components may or may not be shown within the figures, for simplicity of illustration and discussion, and so as not to obscure certain aspects of the

embodiments. Further, arrangements may be shown in block diagram form in order to avoid obscuring embodiments, and also in view of the fact that specifics with respect to implementation of such block diagram arrangements are highly dependent upon the platform within which the embodiment is to be implemented, i.e., such specifics should be well within purview of one skilled in the art. Where specific details (e.g., circuits) are set forth in order to describe example embodiments, it should be apparent to one skilled in the art that embodiments can be practiced without, or with variation of, these specific details. The description is thus to be regarded as illustrative instead of limiting.

10 The term “coupled” may be used herein to refer to any type of relationship, direct or indirect, between the components in question, and may apply to electrical, mechanical, fluid, optical, electromagnetic, electromechanical or other connections. In addition, the terms “first”, “second”, etc. may be used herein only to facilitate discussion, and carry no particular temporal or chronological significance unless otherwise indicated.

As used in this application and in the claims, a list of items joined by the term “one or more of” may mean any combination of the listed terms. For example, the phrases “one or more of A, B or C” may mean A, B, C; A and B; A and C; B and C; or A, B and C.

20 Those skilled in the art will appreciate from the foregoing description that the broad techniques of the embodiments can be implemented in a variety of forms. Therefore, while the embodiments have been described in connection with particular examples thereof, the true scope of the embodiments should not be so limited since other modifications will become apparent to the skilled practitioner upon a study of the drawings, specification, and following claims.

CLAIMS

1. A graphics computing system comprising:
 - a data interface including one or more of a network controller, a memory controller or a bus, the data interface to obtain one or more shader instructions;
 - a graphics processor including a program streamer interface to receive a Reverse Polish Notation (RPN) program stream including a first set of operands and a first set of operations, wherein the RPN program stream is to be compiled from the
10 one or more shader instructions;
 - a first register stack;
 - a first stack allocator to populate the first register stack with one or more operands in the first set of operands; and
 - a first power gating unit to selectively power off one or more registers in the first register stack based on a stack depth of the first register stack.
2. The computing system of claim 1, further including:
 - an argument fetcher to read one or more arguments from the first register
stack; and
 - 20 an executor to conduct an execution of one or more operations in the first set of operations on the one or more arguments.
3. The computing system of claim 2, wherein the first stack allocator is to populate the first register stack with one or more results of the execution.
4. The computing system of claim 2, further including an input output (IO) transmission pipeline, wherein the executor is to send one or more results of the execution to the IO transmission pipeline.
- 30 5. The computing system of claim 1, wherein the RPN program stream is to include a second set of operands and a second set of operations, the computing system further including:
 - a second register stack;
 - a thread controller to detect a thread switch condition; and

a second stack allocator to populate the second register stack with one or more operands in the second set of operands in response to the thread switch condition.

6. The computing system of claim 5, further including a second power gating unit to selectively power off one or more registers in the second register stack based on a stack depth of the second register stack.

7. The computing system of any one of claims 1 to 6, wherein the stack depth is to correspond to a number of live registers in the first register stack.

10

8. An apparatus to process Reverse Polish Notation (RPN) program streams, comprising:

a first register stack;

a program streamer interface to receive an RPN program stream including a first set of operands and a first set of operations;

a first stack allocator to populate the first register stack with one or more operands in the first set of operands; and

a first power gating unit to selectively power off one or more registers in the first register stack based on a stack depth of the first register stack.

20

9. The apparatus of claim 8, further including:

an argument fetcher to read one or more arguments from the first register stack; and

an executor to conduct an execution of one or more operations in the first set of operations on the one or more arguments.

10. The apparatus of claim 9, wherein the first stack allocator is to populate the first register stack with one or more results of the execution.

30

11. The apparatus of claim 9, further including an input output (IO) transmission pipeline, wherein the executor is to send one or more results of the execution to the IO transmission pipeline.

12. The apparatus of claim 9, wherein the RPN program stream is to include a second set of operands and a second set of operations, the apparatus further including:

- a second register stack;
- a thread controller to detect a thread switch condition; and
- a second stack allocator to populate the second register stack with one or more operands in the second set of operands in response to the thread switch condition.

13. The apparatus of claim 12, further including a second power gating
10 unit to selectively power off one or more registers in the second register stack based on a stack depth of the second register stack.

14. The apparatus of any one of claims 9 to 13, wherein the stack depth is to correspond to a number of live registers in the first register stack.

15. A method of processing Reverse Polish Notation (RPN) program streams, comprising:

- receiving an RPN program stream including a first set of operands and a first set of operations;
- 20 populating a first register stack with one or more operands in the first set of operands; and
- selectively powering off one or more registers in the first register stack based on a stack depth of the first register stack.

16. The method of claim 15 further comprising:
reading one or more arguments from the first register stack; and
conducting an execution of one or more operations in the first set of operations on the one or more arguments.

17. The method of claim 16, further comprising populating the first
30 register stack with one or more results of the execution.

18. The method of claim 16, further comprising sending one or more results of the execution to an input output (IO) transmission pipeline.

19. The method of claim 15, wherein the RPN program stream includes a second set of operands and a second set of operations, the method further including:
detecting a thread switch condition; and
populating a second register stack with one or more operands in the second set of operands in response to the thread switch condition.

20. The method of claim 19, further including selectively powering off one or more registers in the second register stack based on a stack depth of the first register stack.

10

21. The method of any one of claims 15 to 20, wherein the stack depth corresponds to a number of live registers in the first register stack.

22. At least one computer readable storage medium comprising a set of compiler instructions which, when executed by a computing system, cause the computing system to:

receive one or more shader instructions; and

compile the one or more shader instructions into a Reverse Polish Notation (RPN) program stream.

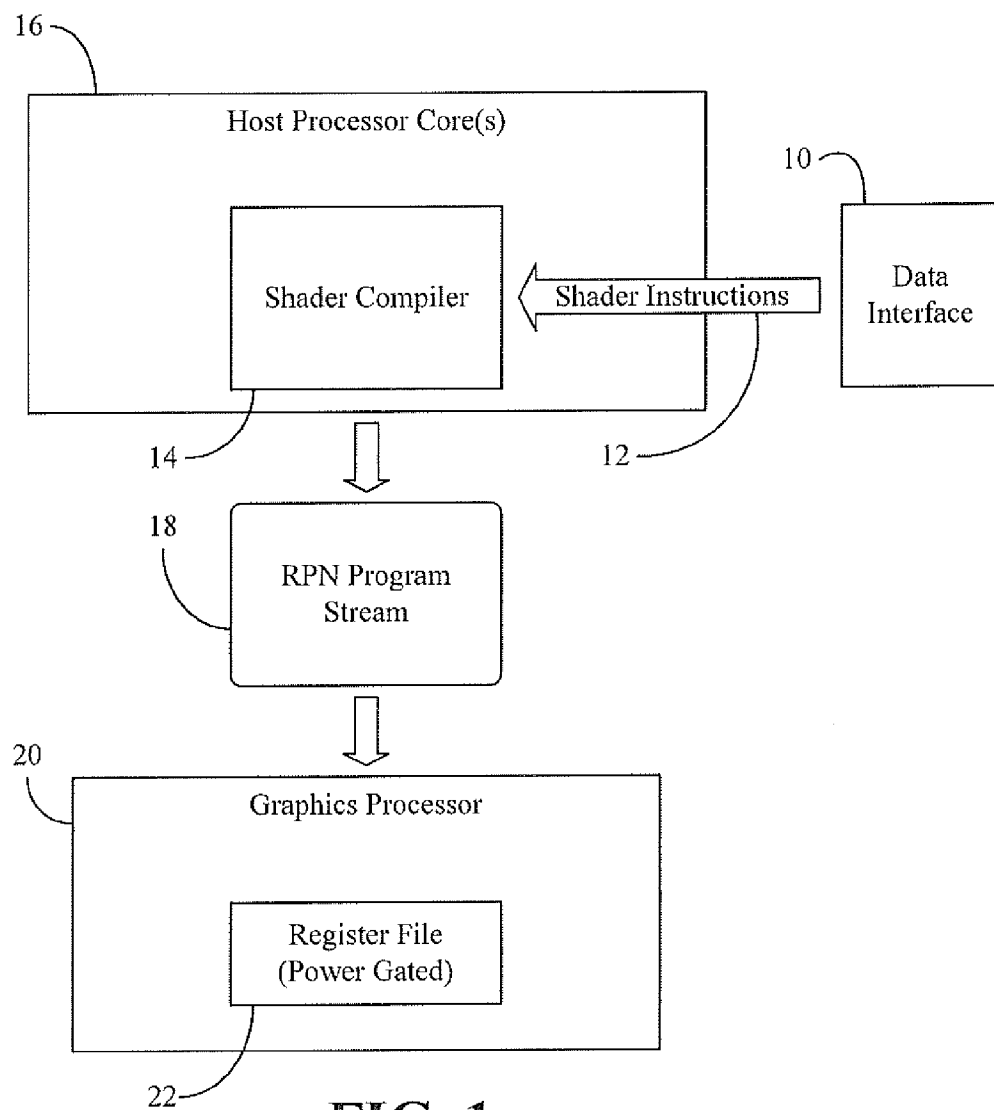
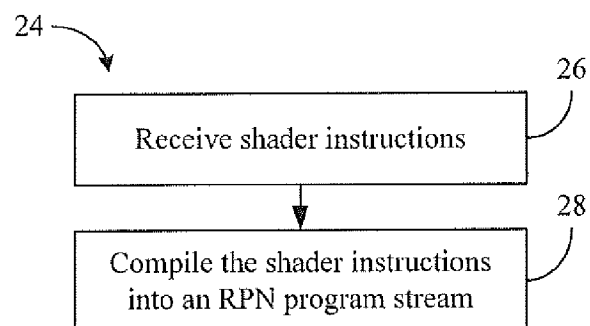
20

23. The at least one computer readable storage medium of claim 22, wherein the RPN program stream is to include a set of operands and a set of operations.

24. The at least one computer readable storage medium of any one of claims 22 or 23, wherein the compiler instructions, when executed, cause the computing system to bypass a register allocation stage during compilation of the one or more shader instructions.

30

25. An apparatus to process Reverse Polish Notation (RPN) program streams, comprising means for performing the method of any one of claims 15 to 20.

**FIG. 1****FIG. 2**

2/14

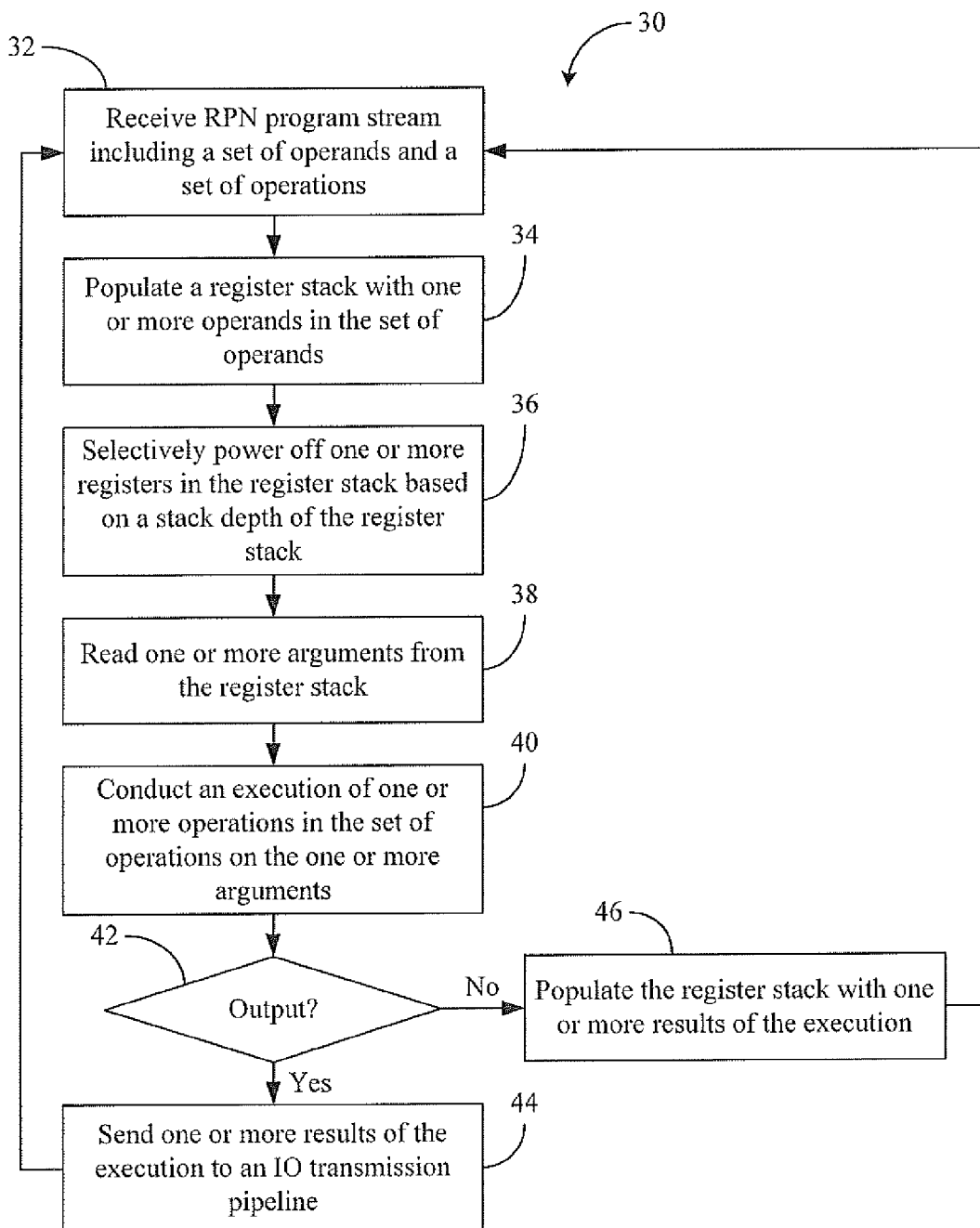


FIG. 3

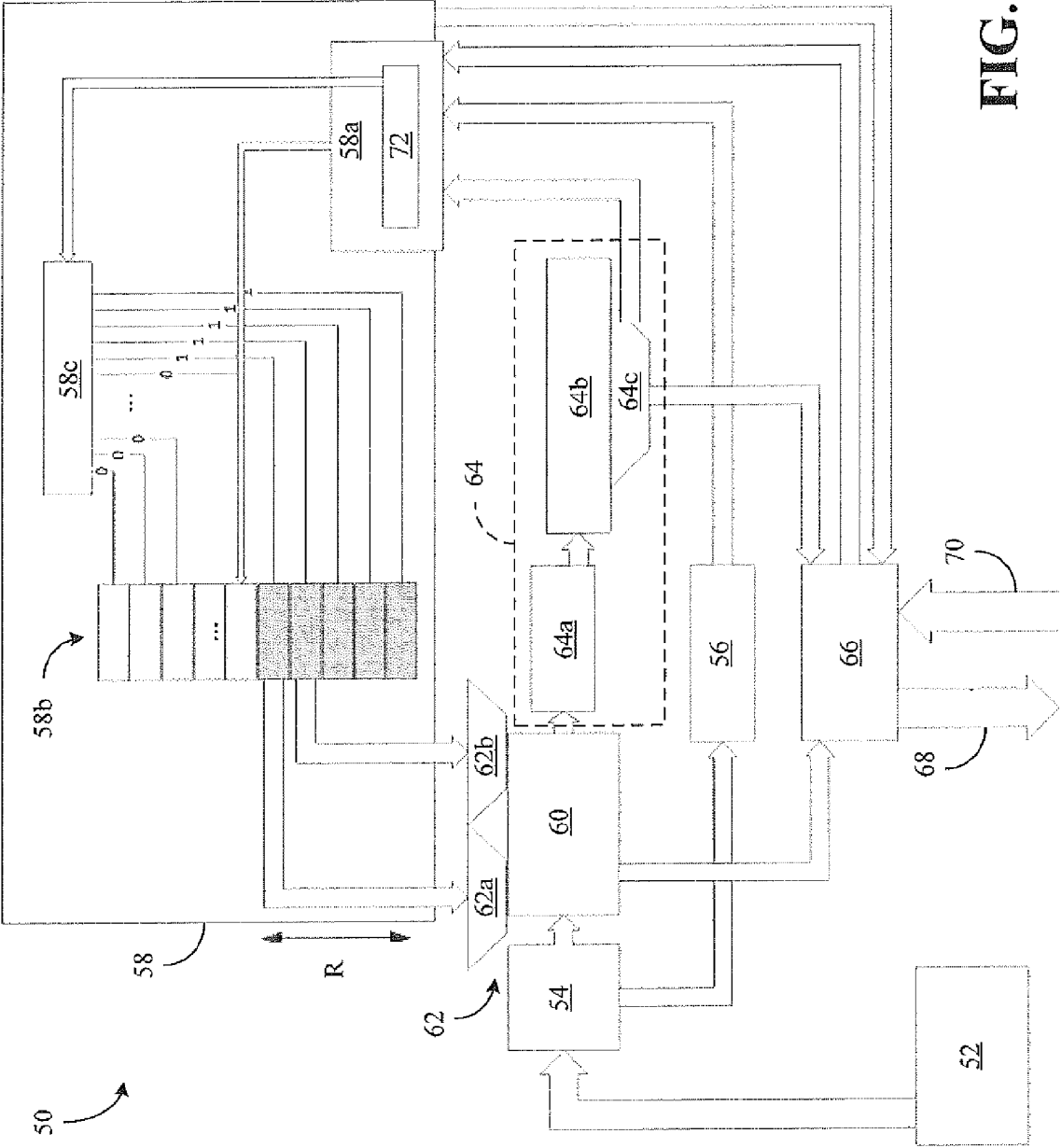


FIG. 4

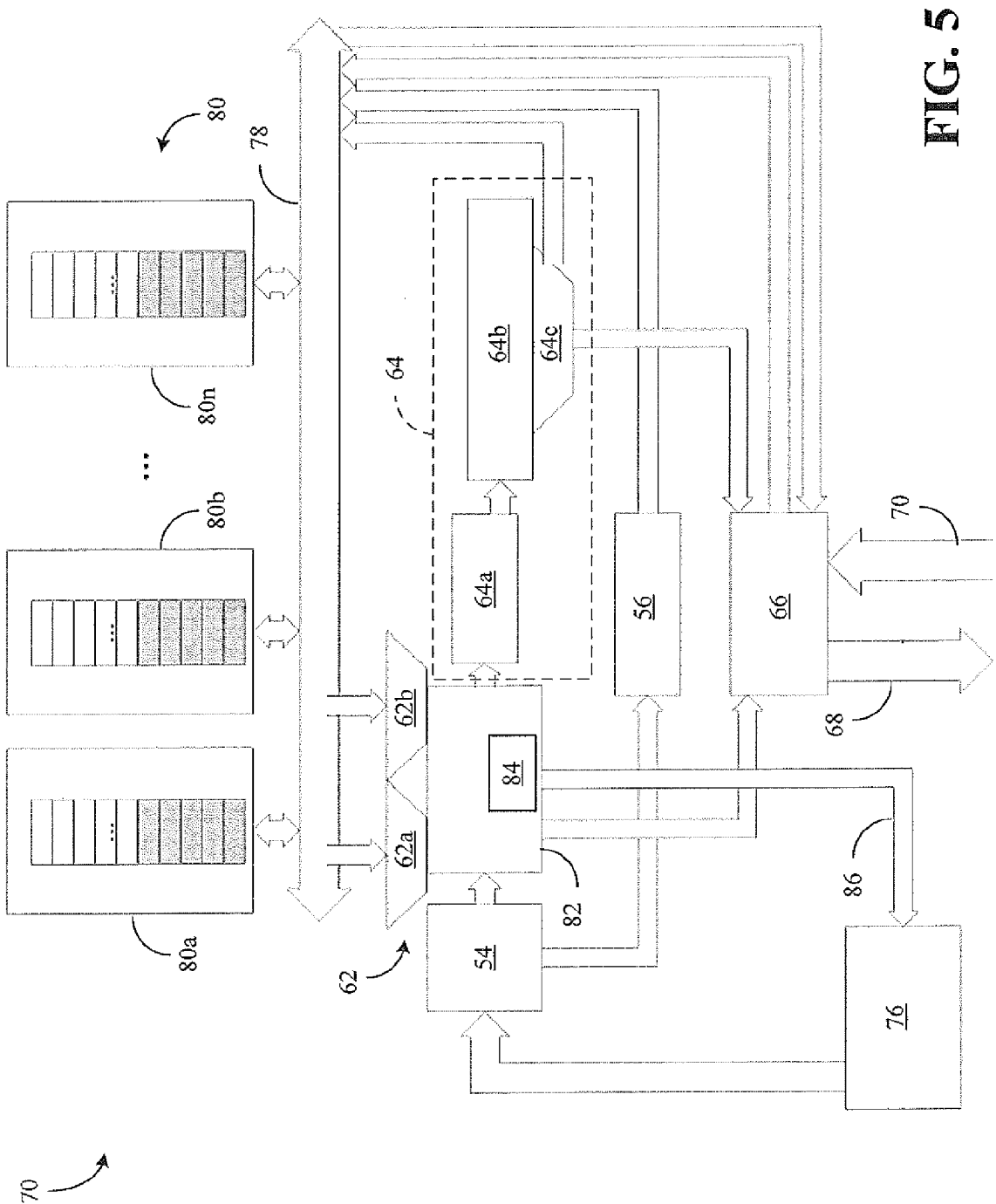


FIG. 5

5/14

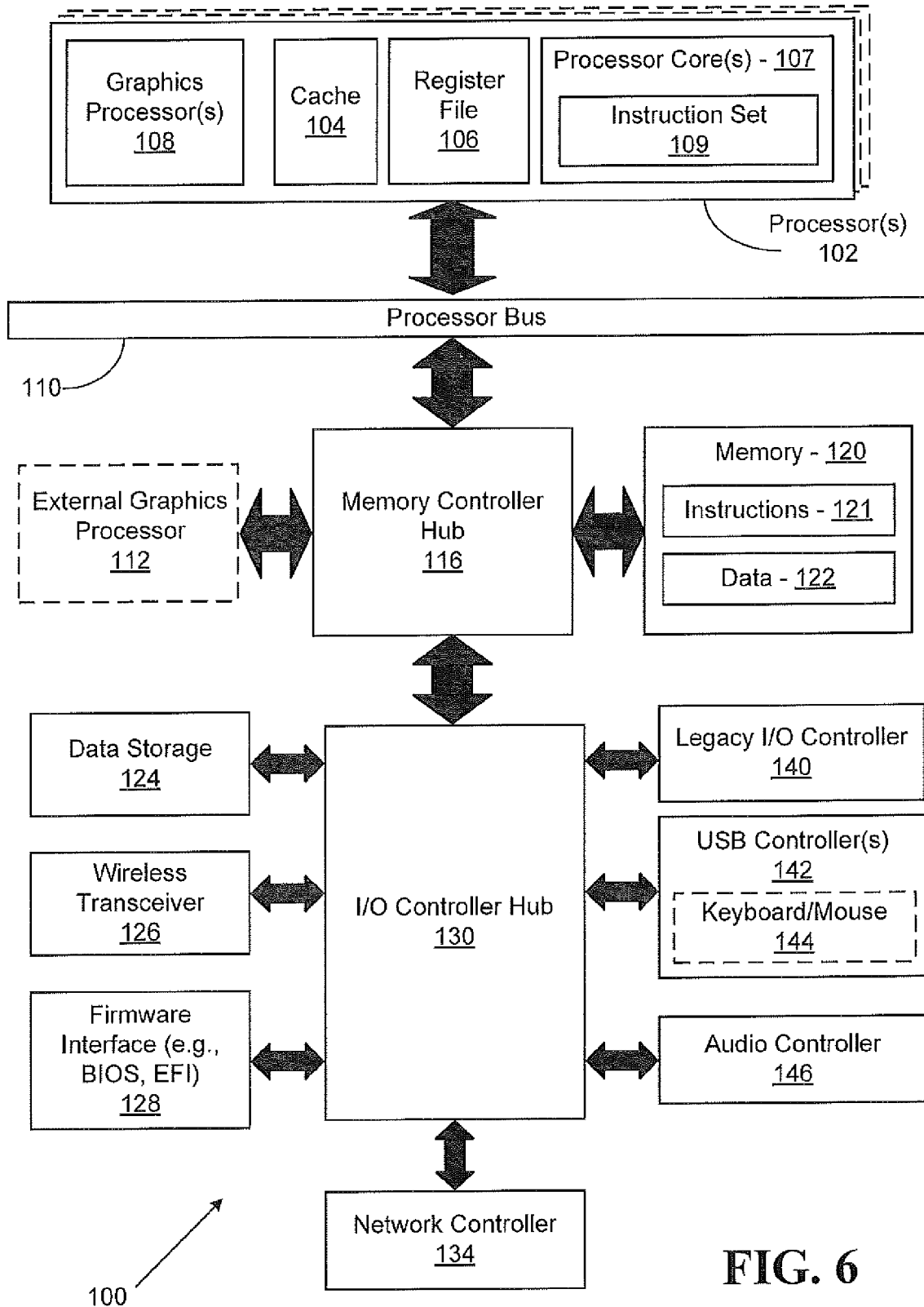


FIG. 6

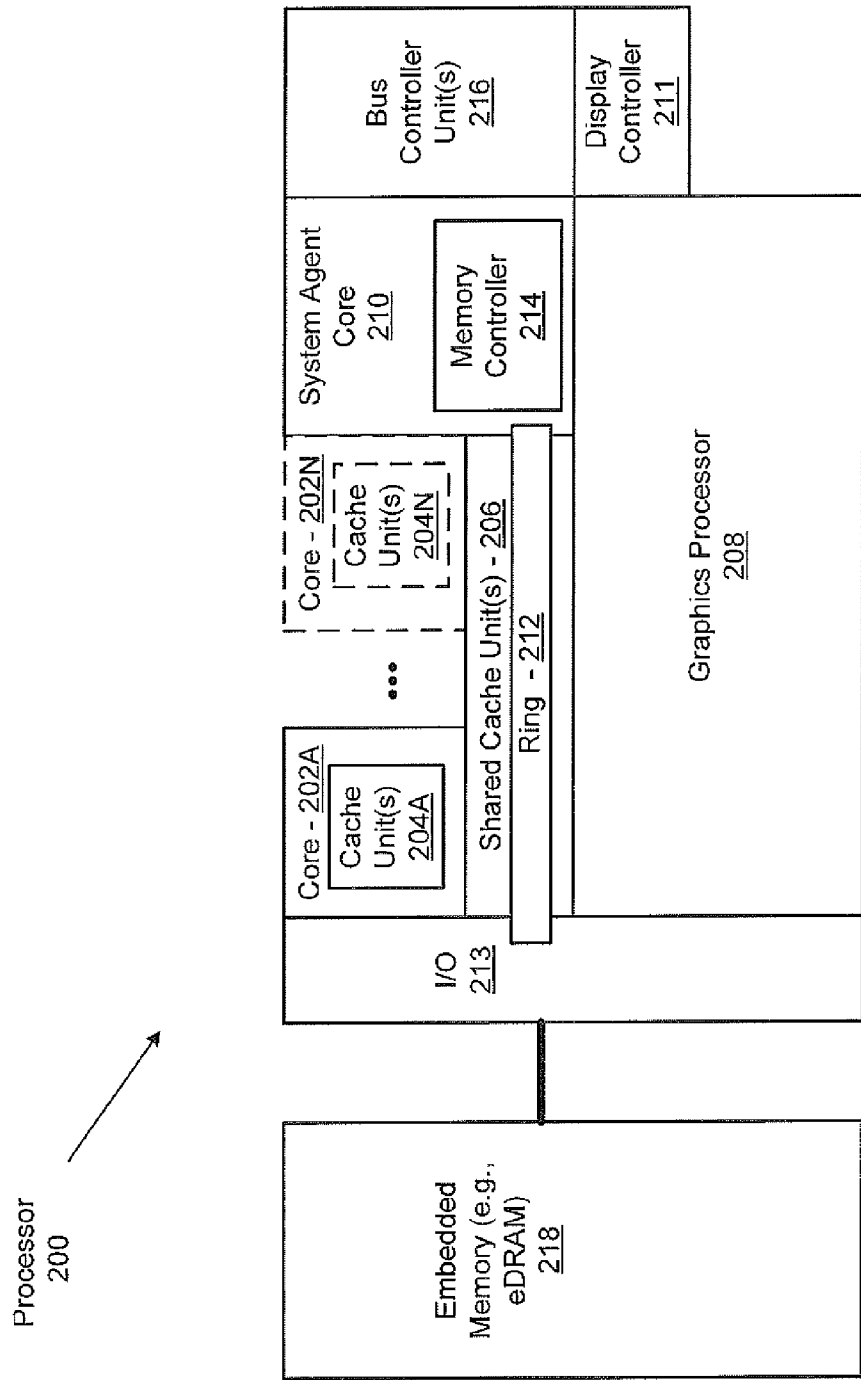


FIG. 7

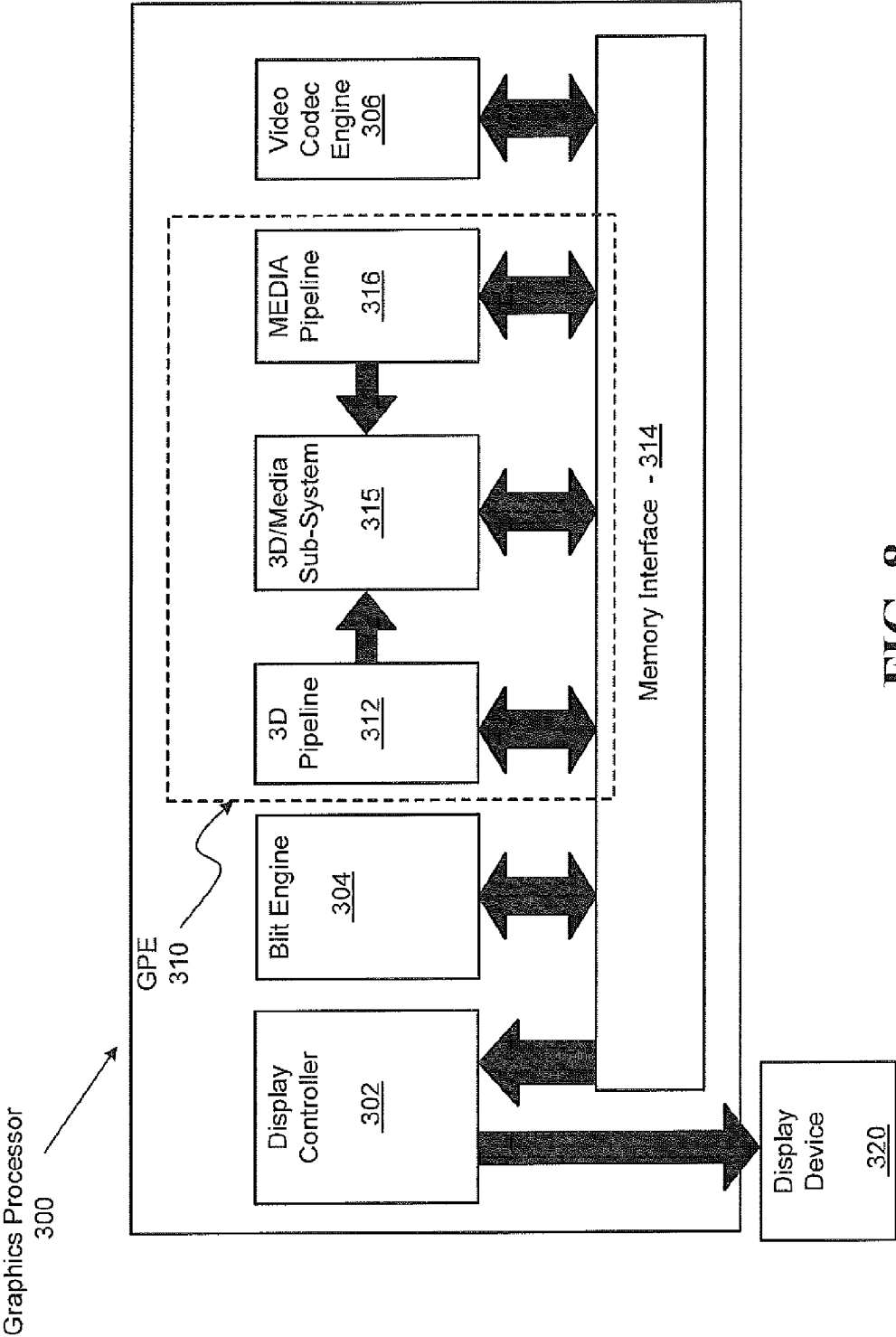


FIG. 8

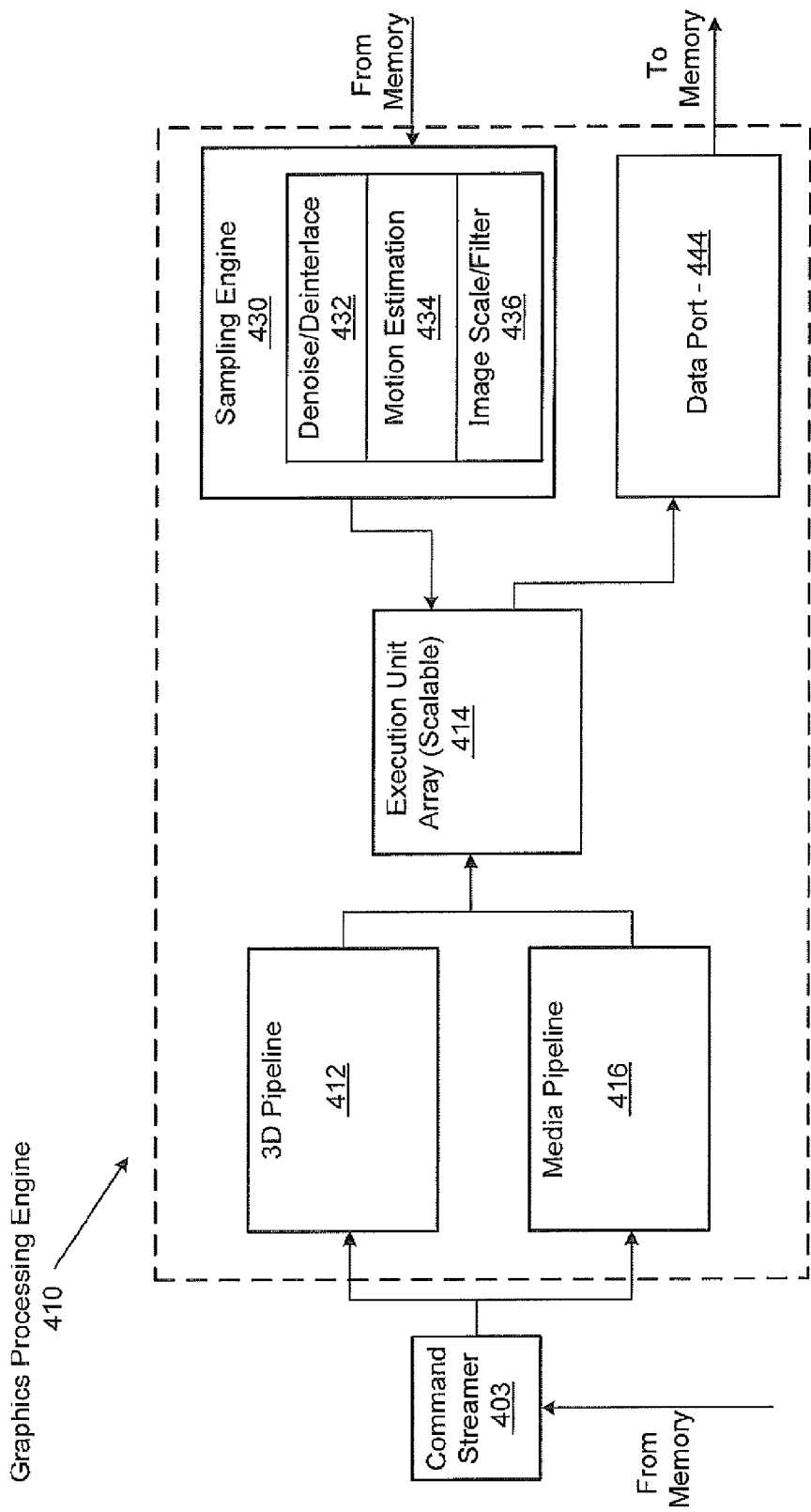


FIG. 9

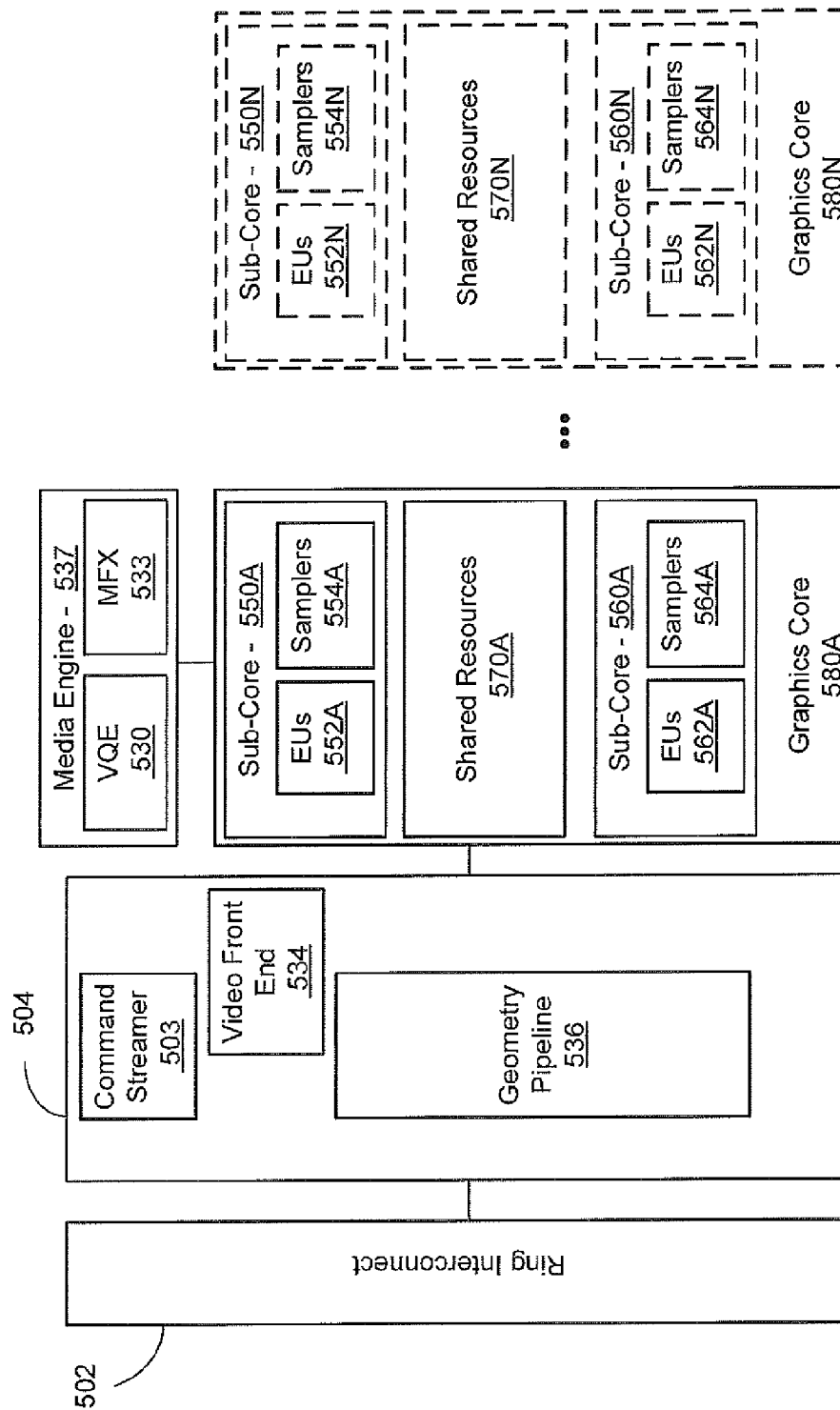


FIG. 10

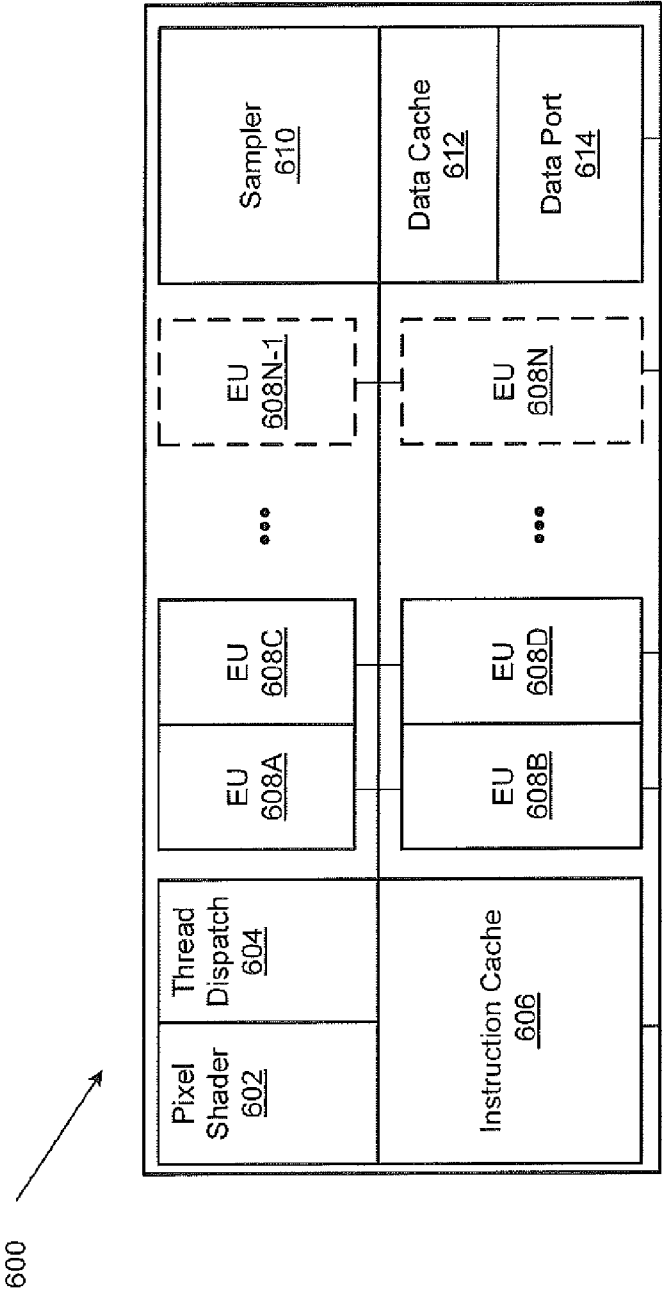
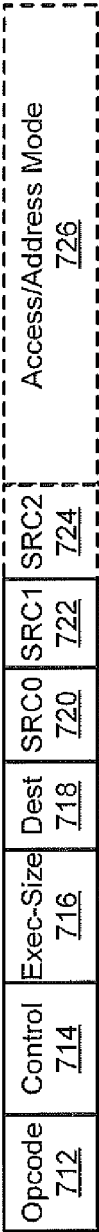


FIG. 11

Graphics Core Instruction Formats

700

128-Bit Instruction 710



64-Bit Compact Instruction 730

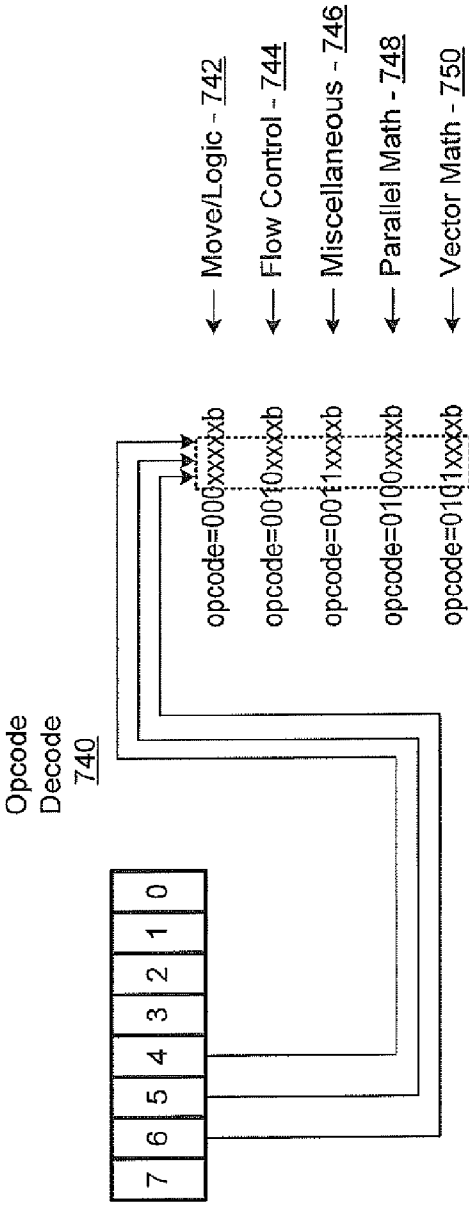


FIG. 12

12/14

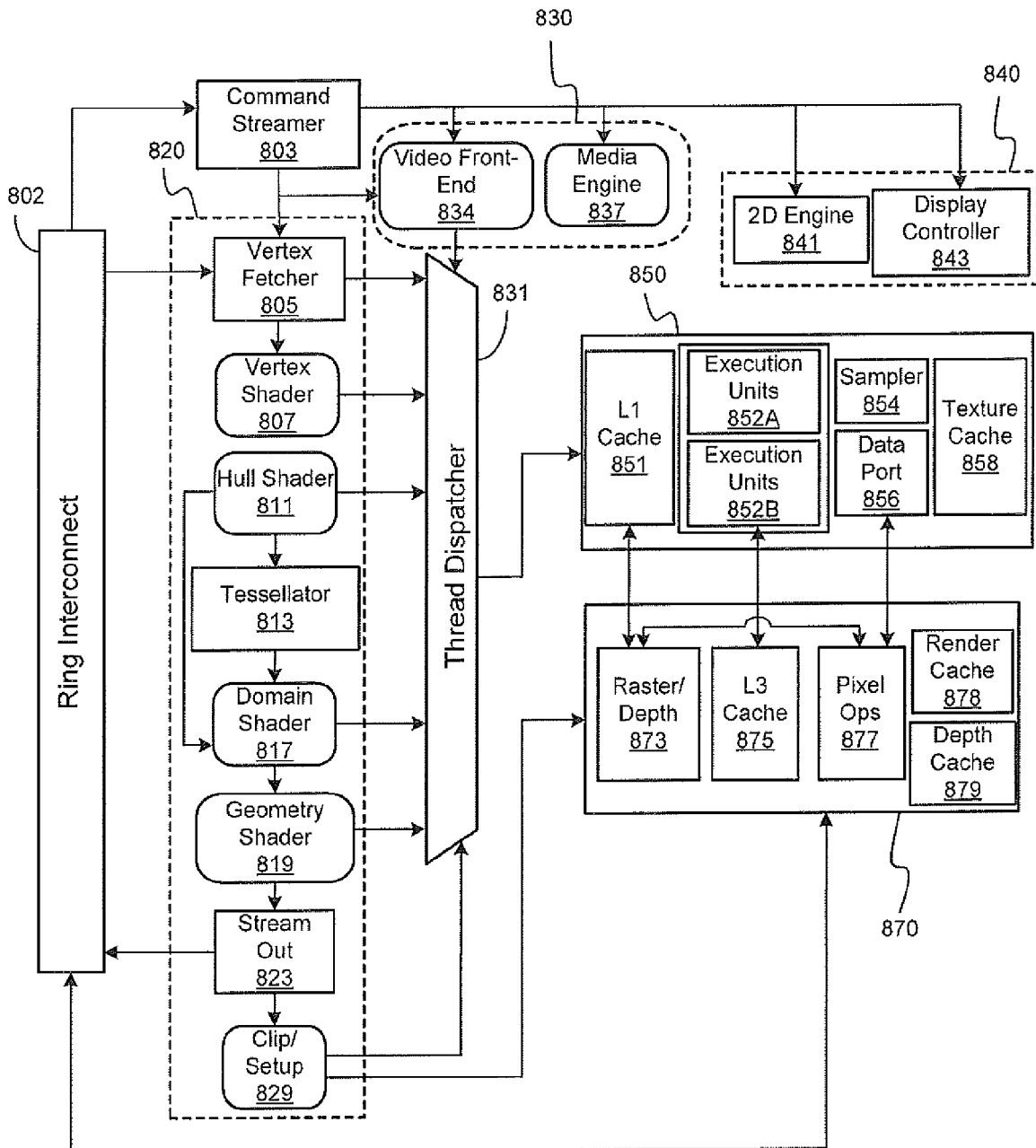
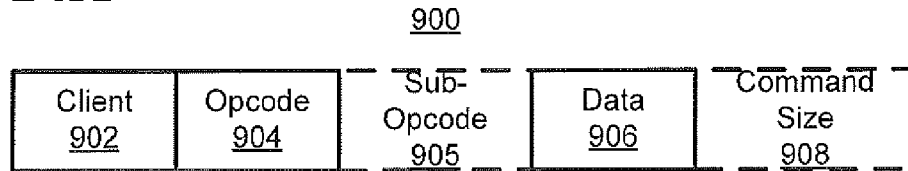


FIG. 13

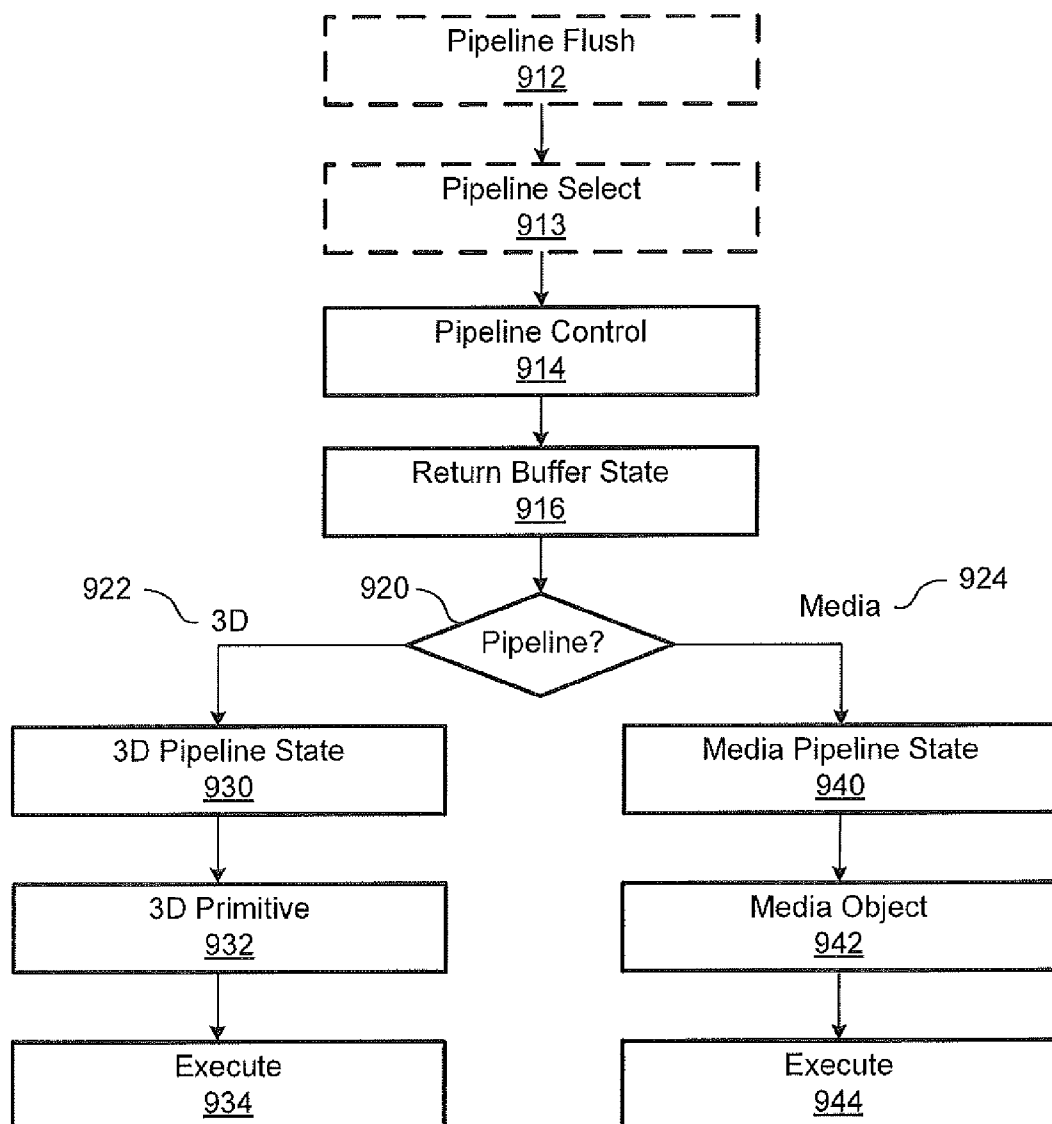
FIG. 14A

Sample Command Format

**FIG. 14B**

Sample Command

Sequence

910

14/14

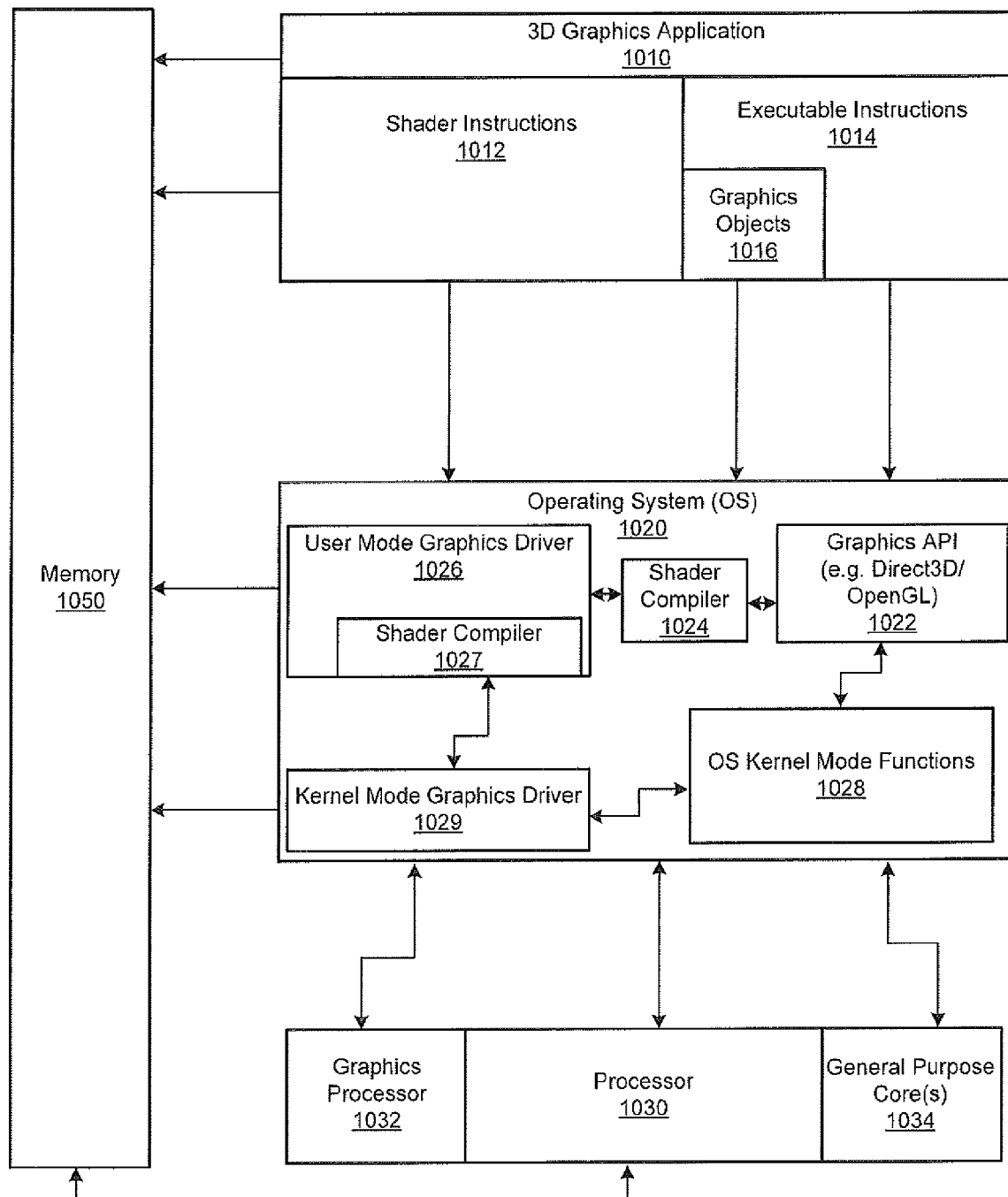


FIG. 15

INTERNATIONAL SEARCH REPORT

International application No

PCT/PL2014/050079

A. CLASSIFICATION OF SUBJECT MATTER

INV. G06F9/30 G06F1/32
ADD.

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F G11C

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

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

EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 2006/091846 A2 (QUALCOMM INC [US]; RYCHLIK BOHUSLAV [US]) 31 August 2006 (2006-08-31)	8-21,25
A	paragraph [0039] paragraph [0043] - paragraph [0045] paragraphs [0047], [0094] -----	1-7
A	EP 0 312 235 A2 (AMERICAN TELEPHONE & TELEGRAPH [US]) 19 April 1989 (1989-04-19) abstract; claim 1 column 6, line 18 - column 7, line 8 column 11, line 34 - column 12, line 35 -----	1



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

29 September 2015

Date of mailing of the international search report

26/01/2016

Name and mailing address of the ISA/

European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040,
Fax: (+31-70) 340-3016

Authorized officer

Daskalakis, T

INTERNATIONAL SEARCH REPORT

International application No.
PCT/PL2014/050079

Box No. II Observations where certain claims were found unsearchable (Continuation of item 2 of first sheet)

This international search report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:
2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:
3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box No. III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.
2. ☐ As all searchable claims could be searched without effort justifying an additional fees, this Authority did not invite payment of additional fees.
3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:
4. ☒ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

1-21, 25

Remark on Protest

- ☐ The additional search fees were accompanied by the applicant's protest and, where applicable, the payment of a protest fee.
- ☐ The additional search fees were accompanied by the applicant's protest but the applicable protest fee was not paid within the time limit specified in the invitation.
- ☐ No protest accompanied the payment of additional search fees.

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No

PCT/PL2014/050079

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2006091846 A2	31-08-2006	BR PI0608556 A2	12-01-2010
		CN 101160559 A	09-04-2008
		EP 1851618 A2	07-11-2007
		JP 4801100 B2	26-10-2011
		JP 2008539472 A	13-11-2008
		KR 20070107784 A	07-11-2007
		US 2006195707 A1	31-08-2006
		WO 2006091846 A2	31-08-2006

EP 0312235 A2	19-04-1989	CA 1309778 C	03-11-1992
		EP 0312235 A2	19-04-1989
		JP H01128166 A	19-05-1989

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. claims: 1-21, 25

Reverse polish notation processing elements with power saving

2. claims: 22-24

Compiling shader instructions into a Reverse Polish Notation program stream
