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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 January 2017 (12.01.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/007557 A1

(51) International Patent Classification:
*G06F 9/50* (2006.01) *G06F 9/38* (2006.01)
*G06F 9/48* (2006.01) *G06T 1/20* (2006.01)

(21) International Application Number:
PCT/US2016/035849

(22) International Filing Date:
3 June 2016 (03.06.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/794,521 8 July 2015 (08.07.2015) US

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

(72) Inventors; and
(71) Applicants *(for US only)*: **NALLURI, Hema, C.** [IN/IN]; 8-76 Sainagar Colony, Uppal Kalan, Hyderabad, AP 500039 (IN). **PAL, Supratim** [IN/IN]; 15A, SLP Nest, Owners Court, Opp Amrita Univeristy, Kasavanahalli, Bangalore, KA 560035 (IN). **MAIYURAN, Subramaniam** [US/US]; 1915 Black Slate Court, Gold River, CA 95670 (US). **CHANDRA, Joy** [IN/IN]; 136 Airport Road, Bangalore, KA 560017 (IN).

(74) Agent: **JORDAN, B., Delano**; Jordan IP Law, LLC, c/o CPA Global, 900 Second Avenue South, Suite 600, Minneapolis, MN 55402 (US).

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

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

**Published**:

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

(54) **Title**: DYNAMIC THREAD SPLITTING

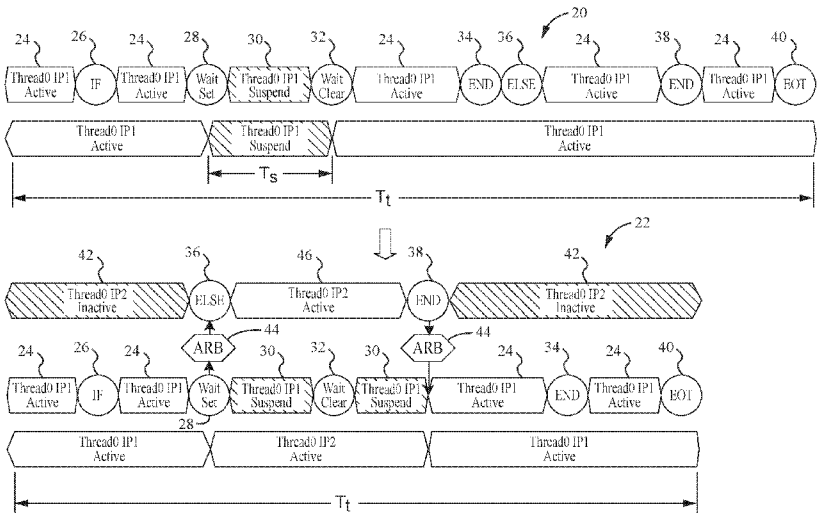



FIG. 1

(57) **Abstract**: Systems, apparatuses and methods may provide for associating a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread and activating a second instruction pointer in response to a dependency associated with the IF block. Additionally, the second instruction pointer may be associated with an ELSE block of the primary IF-ELSE conditional construct. In one example, the IF block and the ELSE block are executed, via the first instruction pointer and the second instruction pointer, one or more of independently from or parallel to one another.

WO 2017/007557 A1

# DYNAMIC THREAD SPLITTING

## CROSS-REFERENCE TO RELATED APPLICATIONS

The present application claims the benefit of priority to U.S. Non-Provisional Patent Application No. 14/794,521 filed on July 8, 2015.

## BACKGROUND

Visually intensive computer graphics applications may be executed on systems having graphics cores with arrays of execution units (EUs) that support multiple threads. When an active thread being executed on an EU encounters a dependency (e.g., wait) condition, the resources of the EU may be used more efficiently by rotating execution to another thread that does not currently have a dependency condition. While increasing the number of threads supported may reduce latency, such an approach may also increase the semiconductor gate count and/or register file size, and lead to higher manufacturing costs. Moreover, because dependencies encountered and the frequency at which they are encountered may be workload dependent, the optimal number of threads may also vary from workload-to-workload. Therefore, increasing the number of threads to accommodate a relatively low percentage of workloads encountered may unnecessarily increase costs.

## 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 an illustration of an example of a comparison between a conventional thread execution sequence and a thread execution sequence according to an embodiment;

FIG. 2 is a flowchart of an example of a method of operating a thread execution apparatus according to an embodiment;

FIG. 3 is a flowchart of an example of a first portion of a method of using status information to operate a thread execution apparatus according to an embodiment;

FIG. 4 is a flowchart of an example of a second portion of a method of using status information to operate a thread execution apparatus according to an embodiment;

FIG. 5 is a block diagram of an example of a thread execution apparatus 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;

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

FIG. 16 is a block diagram of an example of an intellectual property (IP) core development system according to an embodiment; and

FIG. 17 is a block diagram of an example of a system on a chip integrated circuit according to an embodiment.

## DETAILED DESCRIPTION

FIG. 1 shows a conventional thread execution sequence 20 in which a particular instruction pointer (IP) such as, for example, "IP1" assigned to a given thread (e.g., "Thread0") is initially in an active state 24. As will be discussed in greater detail, the thread may be generated in a multi-threaded single instruction multiple data (SIMD) environment based on a graphics-related workload such as, for example, 3D (three-dimensional) computer games, flight simulators and other 3D imaging applications involving user interaction, scene management and rendering, physics modeling, artificial intelligence and other relatively complex functions. The thread may include an IF-ELSE conditional construct, wherein some SIMD channels may satisfy an IF block of the IF-ELSE conditional construct and other SIMD channels may satisfy an ELSE block of the IF-ELSE conditional construct.

The conventional thread execution sequence 20 may generally execute the code of the thread serially, with the IF block execution being followed by the ELSE block execution (e.g., serially starting with channels satisfying an IF condition, followed by channels satisfying an ELSE condition). Thus, upon encountering an IF block 26, the instruction pointer remains in the active state 24 in the illustrated conventional thread execution sequence 20. When a dependency 28 (e.g., "Wait Set" for memory operation, texture sample, etc.) is detected, the instruction pointer may be placed in a suspended state 30. During the suspended state 30, the execution unit (EU) may switch to the next available thread. In cases where all threads have encountered a dependency, the EU may enter an idle state. When an end 32 (e.g., "Wait Clear") is detected for the dependency 28, the instruction pointer may resume the active state 24 until an end 34 of the IF block 26 is detected (e.g., all channels satisfying the IF condition have been executed). The conventional thread execution sequence 20 may then use the instruction pointer to begin execution of an ELSE block 36 in the IF-ELSE conditional construct. The instruction pointer may remain in the active state 24 until the instruction pointer encounters an end 38 of the ELSE block 36. The instruction pointer may also remain in the active state 24 until and end of thread (EOT) condition 40 is encountered. Thus, the overall amount of time used to execute the thread in question ($T_t$) is increased by the amount of time that the thread is suspended ($T_s$) in the illustrated conventional thread execution sequence 20.

By contrast, an enhanced thread execution sequence 22 may generally execute, via multiple instruction pointers, IF and ELSE blocks independently from one another (e.g., ELSE block execution may begin before IF block execution ends for a given IF-ELSE conditional construct). For example, a first instruction pointer (e.g., "IP1") assigned to a given thread (e.g., generated in a multi-threaded SIMD environment based on a graphics-related workload) might initially be in the active state 24 and a second instruction pointer (e.g., "IP2") may initially be in an inactive state 42. Indeed, parallel execution of the IF and ELSE blocks may be conducted if sufficient resources are available to execute IP1 and IP2 in parallel and/or there is no contention between the resources used to execute IP1 and IP2 in parallel.

Upon encountering the IF block 26, the first instruction pointer may remain in the active state 24 and the second instruction pointer may remain in the inactive state 42 in the enhanced thread execution sequence 22. When the dependency 28 (e.g., "Wait Set") is detected, a thread execution apparatus 44 (e.g., arbitrator/"ARB" with

multiplexing and/or dynamic thread splitting functionality) may activate the second instruction pointer and associate the second instruction pointer with the ELSE block 36 of the IF-ELSE conditional construct. Thus, the second instruction pointer may enter an active state 46 while the first instruction pointer enters the suspended state 30. Of particular note is that the second instruction pointer enables dynamic execution of the same thread being handled by the first instruction pointer, in the illustrated example. As a result, if there are no other available threads, idleness of the EU may be avoided. Moreover, the number of threads may be effectively doubled without incurring the costs associated with adding thread supporting registers, semiconductor gates, and so forth.

When the end 32 (e.g., "Wait Clear") of the dependency condition occurs, the first instruction pointer may remain in the suspended state 30. The end 38 of the ELSE block 36 may cause the thread execution apparatus 44 to place the second instruction pointer back in the inactive state 42, activate the first instruction pointer, and use the first instruction pointer to finish executing the IF block 26. Thus, the first instruction pointer may remain in the active state 24 until the end 34 of the IF block 26 is detected. The end 34 of the IF block 26 may cause the thread execution apparatus 44 to jump the first instruction pointer from the end 34 of the IF block 26 to the end 38 of the ELSE block 36, wherein the first instruction pointer remains in the active state 24 until the end of thread condition 40 is encountered. Thus, the overall amount of time used to execute the thread in question ($T_t$) is reduced for the illustrated enhanced thread execution sequence 22.

FIG. 2 shows a method 48 of operating a thread execution apparatus. The method 48 may be implemented in one or more modules as 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), flash memory, etc., as configurable logic such as, for example, programmable logic arrays (PLAs), field programmable gate arrays (FPGAs), complex programmable logic devices (CPLDs), as fixed-functionality logic hardware 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. For example, computer program code to carry out operations shown in the method 48 may be written in any combination of one or more programming languages, including an object oriented programming language such as

C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. Moreover, the method 48 may be implemented using any of the herein mentioned circuit technologies.

Illustrated processing block 50 provides for associating a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread. As will be discussed in greater detail, the primary IF-ELSE conditional construct may be a lowest level IF-ELSE conditional construct that may include other, nested IF-ELSE conditional constructs. A second instruction pointer may be activated at processing block 52, wherein illustrated processing block 54 associates the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct. Thus, the illustrated solution dynamically splits a single thread across multiple instruction pointers in order to reduce latency and maximize efficiency/performance.

Accordingly, the method 48 may substantially enhance execution of a simplified thread such as:

```
//Thread0 Instructions
Inst1
Inst2
…
(Pred)IF{ //Pred indicates the channels active for “If”
        Inst11
        Inst12
        …
        Send11 //Wait
        …
        }
(!Pred)ELSE{ // !Pred indicates the channels active for “Else”
        Inst21
        Inst22
        …
        …
        }
InstN
InstN+1
…
```

EOT

In the above Thread0 instruction sequence, instructions executed via the first IP of Thread0 are highlighted with a single underline and the instructions executed via second IP of Thread0 are highlighted with a double underline. In the above instruction flow, when the first IP encounters a dependency on "Send11" inside the IF block a second IP will be activated to execute instructions inside the ELSE block. The second IP may execute all instructions inside the ELSE block and then become deactivated. For simplicity, no Send instructions are shown in the ELSE block. While the second IP is in progress, the Send11 dependency might get cleared and the first IP may wait until the second IP completes. Once the second IP is complete, the first IP may resume (provided the Send11 dependency is cleared) and execute all instructions inside the IF block. On reaching the last instruction inside the IF block, the first IP may jump to InstN.

FIGs. 3 and 4 show first and second portions, respectively, of a method 58 of using status information to operate a thread execution apparatus. The method 58 may also be implemented in one or more modules as a set of logic instructions stored in a machine- or computer-readable storage medium such as RAM, ROM, PROM, flash memory, etc., as configurable logic such as, for example, PLAs, FPGAs, CPLDs, as fixed-functionality logic hardware using circuit technology such as, for example, ASIC, CMOS or TTL technology, or any combination thereof.

Illustrated processing block 59 starts a particular thread (e.g., "Thread0"), wherein processing block 60 may set a first IP status to active (e.g., Active=1) and a second IP status to inactive (e.g., Active=0). A determination may be made at processing block 61 as to whether the first IP is active (e.g., Active=1) and the first IP has not encountered a dependency (e.g. Wait=0). If so, a determination may be made at processing block 62 as to whether the first IP is not waiting for the second IP to complete. If it is determined at processing block 62 that the first IP is not waiting for the second IP to complete, an instruction may be fetched from the first IP at processing block 63. Illustrated processing block 64 decodes the fetched instruction, wherein a determination may be made at processing block 64 as to whether the decoded instruction gives rise to a dependency (e.g., send, wait, memory operation, texture sample).

If it is determined at processing block 65 that the decoded instruction does not give rise to a dependency, illustrated processing block 66 (FIG. 4) determines whether

the decoded instruction is an IF-ELSE conditional construct. If so, processing block 67 may determine whether there are any valid channels (e.g., SIMD channels) for the IF condition. If it is determined at processing block 67 that there are valid channels for the IF condition, illustrated processing block 68 sets the status of the IF block to active (e.g., Active=1). A determination may also be made at processing block 69 as to whether there are any valid channels (e.g., SIMD channels) for the ELSE condition. If so, block 70 may load the start of the ELSE block to the second IP and store the end of the ELSE block to temporary memory, wherein retrieval of the next instruction may be initiated at processing block 71. If, on the other hand, it is determined at processing block 69 that there are no valid channels for the ELSE condition, processing block 70 may be bypassed.

If it is determined at processing block 67 that there are no valid channels for the IF condition, illustrated processing block 72 jumps to the start of the ELSE block. A determination may be made at processing block 73 as to whether there are any valid channels for the ELSE condition. If it is determined at processing block 73 that there are no valid channels for the ELSE condition, processing block 74 may jump to the end of the ELSE block. If it is determined at processing block 73 that there are valid channels for the ELSE condition, processing block 74 may be bypassed.

If it is determined at processing block 66 that the decoded instruction is not an IF-ELSE conditional construct, a determination may be made at processing block 75 as to whether the decoded instruction is the start of an ELSE block. If so, a determination may be made at processing block 76 as to whether the ELSE block is being executed via the second IP. If it is determined at processing block 76 that the ELSE block is being executed via the second IP, a determination may be made at processing block 77 as to whether the second IP is complete. If it is determined at processing block 77 that the second IP is complete, illustrated processing block 78 clears the ELSE assignment status of the second IP and clears the activation status of the IF block. Additionally, processing block 79 may jump the second IP to the end of the ELSE block.

If it is determined at processing block 77 that the second IP is not complete, a wait status of the second IP may be set so that the second IP enters a wait state 80 (FIG. 3) and the determination at processing block 61 may be repeated as to whether the first IP is active and the first IP has not encountered a dependency. If it is

determined at processing block 76 that the ELSE block is not being executed via the second IP, processing blocks 77-79 may be bypassed.

If it is determined at processing block 75 that the decoded instruction is not the start of an ELSE block, a determination may be made at processing block 81 as to whether the decoded instruction is an end of thread (EOT) instruction. A positive determination at processing block 81 may terminate the thread at processing block 82. Otherwise, the instruction may be executed at processing block 83.

If it is determined at processing block 65 (FIG. 3) that the decoded instruction gives rise to a dependency, illustrated processing block 84 sets a wait status of the first IP and the first IP enters a wait state 85. A determination may be made at processing block 86 as to whether the IF block is active. If so, illustrated processing block 87 determines whether there are any valid channels for the ELSE condition. If there ae valid channels for the ELSE condition, processing block 88 may activate the second IP, set the activation status of the second IP, clear the completion status of the second IP, and set the ELSE assignment status of the second IP.

Illustrated processing block 89 fetches an instruction from the second IP, wherein the fetched instruction may be decoded at processing block 90. Additionally, a determination may be made at processing block 91 as to whether the decoded instruction gives rise to a dependency. If so, illustrated processing block 92 sets the wait status of the second IP and the second IP enters a wait state 93. The determination at processing block 61 may then be repeated as to whether the first IP is active and the first IP has not encountered a dependency.

If it is determined at processing block 91 that the decoded instruction does not give rise to a dependency, the instruction may be executed at processing block 94. Additionally, a determination may be made at processing block 95 as to whether the second IP is at the end of the ELSE block. If so, illustrated processing block 96 clears the activation status of the second IP and sets the completion status of the second IP so that the second IP enters the wait state 85. If, on the other hand, it is determined at processing block 95 that the second IP is not at the end of the ELSE block, retrieval of the next instruction may be initiated at processing block 97.

If it is determined at processing block 61 that either the first IP is not active or the first IP has encountered a dependency, illustrated processing block 98 determines whether the second IP is active and the second IP is not in a wait state. If the second IP is active and the second IP is not in a wait state, an instruction may be fetched from

the second IP at processing block 89. Otherwise, the determination at processing block 61 may be repeated as to whether the first IP is active and the first IP has not encountered a dependency.

Although the above discussion describes the first IP as executing all of the code inside the IF block and the second IP as executing all of the code inside ELSE block, these techniques may be extended to nested IF-ELSE conditional constructs. In this regard, the IF and ELSE blocks may not always have equal amount of instructions to execute. In those cases, either the first IP or the second may get freed up relatively soon based on the amount of instructions in the IF or ELSE block. Freed up instruction pointers may be further used/recycled to conduct split threading functionality in the ELSE or IF block, respectively, on encountering a nested IF/ELSE inside parent/primary IF or ELSE blocks.

One of the possible ways to implement such an approach is based on the following principles.

1. In the Primary (Level0) IF block, IP1 is master and IP2 is slave.

2. In the Primary (Level0) ELSE block, IP2 is master and IP1 is slave.

3. The slave IP may always be assigned an ELSE block to execute on encountering a dependency inside an IF block.

4. Once a slave IP is assigned an ELSE block, all the instructions inside the ELSE block may only be executed by the slave IP (including any nested IF/ELSE).

5. A status table may be maintained to track the assignment of the ELSE block to the slave IP for execution and the corresponding execution status. Two status tables may be maintained – one for the primary IF block and one for primary ELSE block. Optimization is possible with a single table, which is not shown here to simplify the discussion.

6. The number of entries in the status table may be a function of how many nested IF/ELSE blocks can be supported. The table may be memory-backed to simplify the implementation and optimize the size of the table, wherein most recent entries (e.g., eight entries) may be stored on-die and the rest of the table may reside in off-die memory. Entries may be pushed and popped to and from memory as appropriate.

7. Two counters may be maintained – one for each of the first and second instruction pointers. Counters may increment when the corresponding IP executes the start of an IF block and decrement on encountering an ELSE block. When a slave IP

is assigned to execute an ELSE block, the master IP's counter may be used to populate the appropriate nested level entry in the status table as assigned. The most recent nested entry level populated may also be also recorded.

8. When the slave IP executes the complete ELSE block assigned, it may set the execution status completed to the most recent nested level entry.

9. When the master IP's counter encounters the start of an ELSE block, it may read the status table with its counter value as the nested level entry to determine whether if a slave IP has been assigned to execute before making progress. If a slave IP has been assigned, the master IP may wait for the slave IP to complete, reset the assignment status and reset completion status before jumping to end of the ELSE block. If a slave IP has not been assigned, the master IP may make forward progress without a jump. Tables I and II below illustrate examples of IF and ELSE status tables, respectively.

| Primary (Level0) IF Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | | |
| Level1 | | |
| Level2 | | |
| … | … | … |
| Level-n | | |

Table I

| Primary (Level0) ELSE Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | | |
| Level1 | | |
| Level2 | | |
| … | … | … |
| Level-n | | |

Table II

Thus, execution of a thread with nested IF-ELSE blocks as shown below may be enhanced using these techniques. For simplicity, nested IF-ELSE conditional constructs are not considered in the Level0 ELSE block.

//Thread0 Instructions

Inst1

Inst2

```
…
(Pred1)IF{ //"IF", Level0
        Inst11
        Inst12
        …
        Send11 //Wait
        …
        (Pred2)IF{ //Nested "IF", Level1
                Inst111
                Inst112
                …
                Send111 //Wait
                …
        }
        (!Pred2)ELSE{
                Inst211
                Inst212
                …
                …
        }
        Inst13
        Inst14
        …
        }
(!Pred1)Else{ // !Pred indicates the channels active for "Else"
        Inst21
        Inst22
        …
        …
}
InstN
InstN+1
…
EOT
```

In the above Thread0 instruction sequence, instructions executed by the first IP are highlighted in single underline and the instructions executed by second IP are highlighted in double underline. In the above instruction flow, when the first IP encounters a dependency on "Send11" inside the Level0 IF block, the set Level0 "ELSE assignment status" may be set to valid and the "ELSE completion status" may be set as incomplete in the primary IF status table. Additionally, the second IP may be activated to execute instructions inside the Level0 ELSE block.

| Primary (Level0) IF Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | Valid | Incomplete |
| Level1 | Invalid | Incomplete |
| Level2 | Invalid | Incomplete |

The second IP may execute all instructions inside ELSE block, set the Level0 "ELSE completion status" to complete and get deactivated.

| Primary (Level0) IF Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | Valid | Complete |
| Level1 | Invalid | Incomplete |
| Level2 | Invalid | Incomplete |

While the second IP is in progress, the Send11 dependency might get cleared and the first IP may wait until the second IP completes. Once the second IP is complete, the first IP resume (provided the Send11 dependency is cleared). When the first IP encounters the Level1 IF condition, the counter associated with the first IP may be incremented to "1". When the first IP encounters the Send111 instruction inside the Level1 IF block, the Level1 ELSE assignment status may be set to valid and the ELSE completion status may be set as incomplete in Primary IF Status Table. Additionally, the second IP may be activated to execute instructions inside the Level1 ELSE block.

| Primary (Level0) IF Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | Valid | Complete |
| Level1 | Valid | Incomplete |
| Level2 | Invalid | Incomplete |

The second IP may execute all the instructions inside the Level1 ELSE block, set the Level1 "ELSE Completion Status" to complete and become deactivated.

| Primary (Level0) IF Status Table | | |
|---|---|---|
| | ELSE assignment status | ELSE completion status |
| Level0 | Valid | Complete |
| Level1 | Valid | Complete |
| Level2 | Invalid | Incomplete |

While the second IP is in progress, the Send111 dependency might get cleared and the first IP may wait until second IP completes. Once the second IP is complete, the first IP may resume (provided Send11 dependency is cleared). When the first IP encounters the Level1 ELSE condition, the first IP may use its counter value of "1" to read the Level1 entry from the primary IF status table and determine that it has been assigned to the second IP and has been completed. The first IP may then set the Level1 ELSE assignment status to invalid and the ELSE completion status as incomplete in the primary IF status table. The value of the counter associated with the first IP may also be decremented to "0", wherein the first IP may jump to instruction Inst13 and start executing.

When the first IP encounters the Level0 ELSE condition, its counter value of "0" may be used to read the Level0 entry from the Primary IF Status Table and determine that it has been assigned to the second IP and has been completed. The first IP may set the Level0 ELSE assignment status to invalid and the ELSE completion status as incomplete in the primary IF status table. The first IP may also jump to instruction InstN and start executing.

Although the above example is with two instruction pointers for a single thread, the number of instruction pointers may be increased to exploit the parallelism for heavily nested IF-ELSE conditional constructs. Handling multiple instruction pointers, however, may add more complexity. Simply put, two instruction pointers may sufficiently justify exploitation the IF-ELSE conditional constructs.

FIG. 5 shows a multi-threaded computing system having a thread execution apparatus 99 (99a-99h). The thread execution apparatus 99 may generally implement one or more aspects of the method 48 (FIG. 2) and/or the method 58 (FIGs. 3 and 4), already discussed. In the illustrated example, a workload source 101 such as, for

example, a network controller, memory controller, user interface (UI) device, etc., generates a graphics-related workload 103 and a thread generator 105 generates a thread 111 based on the graphics-related workload. The thread 111 may include a primary IF-ELSE conditional construct, as already noted.

The thread execution apparatus 99 may include a condition handler 99a to associate a first IP with an IF block of the primary IF-ELSE conditional construct. Additionally, a thread splitter 99b (e.g., hardware multiplexer) may activate a second IP in response to a dependency associated with the IF block and associate the second IP with an ELSE block of the primary IF-ELSE conditional construct. The thread execution apparatus 99 may also include one or more processor cores 99c to execute, via the first IP and the second IP, the IF block and the ELSE block independently from and/or parallel to one another. In one example, a boundary detector 99d detects an end of the IF block and a jump component 99e jumps the first IP from the end of the IF block to an end of the ELSE block, wherein the apparatus 99 is to process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block by the second IP has completed.

Moreover, the thread execution apparatus 99 may further include a nest detector 99f to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct. In such a case, the one or more processor cores 99c may independently execute, via the first IP and the second IP, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct. Additionally, a status monitor 99g may track an ELSE assignment status and an ELSE completion status of the first and second IPs in one or more status tables on a per construct level basis. In one example, the thread execution apparatus 99 further includes one or more counters 99h to track construct levels. The condition handler 99a may also confirm that the IF-ELSE conditional construct has one or more valid SIMD channels to execute from the ELSE block.

System Overview

FIG. 6 is a block diagram of a processing system 100, according to an embodiment. In various embodiments the system 100 includes one or more processors 102 and one or more graphics processors 108, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 102 or processor cores 107. In on embodiment,

the system 100 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

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

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

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

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

Memory device 120 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 120 can operate as system memory for the system 100, to store data 122 and instructions 121 for use when the one or more processors 102 executes an application or process. Memory controller hub 116 also couples with an optional external graphics processor 112, which may communicate with the one or more graphics processors 108 in processors 102 to perform graphics and media operations.

In some embodiments, ICH 130 enables peripherals to connect to memory device 120 and processor 102 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 146, a firmware interface 128, a wireless transceiver 126 (e.g., Wi-Fi, Bluetooth), a data storage device 124 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller 140 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 142 connect input devices, such as keyboard and mouse 144 combinations. A network controller 134 may also couple to ICH 130. In some embodiments, a high-performance network controller (not shown) couples to processor bus 110. It will be appreciated that the system 100 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub 130 may be integrated within the one or more processor 102, or the memory controller hub 116 and I/O controller hub 130 may be integrated into a discreet external graphics processor, such as the external graphics processor 112.

FIG. 7 is a block diagram of an embodiment of a processor 200 having one or more processor cores 202A-202N, an integrated memory controller 214, and an integrated graphics processor 208. Those elements of FIG. 7 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor 200 can include additional cores up to and including additional core 202N represented by the dashed lined boxes. Each of processor cores 202A-202N includes one or more internal cache units 204A-204N. In some embodiments each processor core also has access to one or more shared cached units 206.

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

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

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

In some embodiments, processor 200 additionally includes graphics processor 208 to execute graphics processing operations. In some embodiments, the graphics processor 208 couples with the set of shared cache units 206, and the system agent core 210, including the one or more integrated memory controllers 214. In some

embodiments, a display controller 211 is coupled with the graphics processor 208 to drive graphics processor output to one or more coupled displays. In some embodiments, display controller 211 may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 208 or system agent core 210.

In some embodiments, a ring based interconnect unit 212 is used to couple the internal components of the processor 200. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some embodiments, graphics processor 208 couples with the ring interconnect 212 via an I/O link 213. The exemplary I/O link 213 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 218, such as an eDRAM module. In some embodiments, each of the processor cores 202-202N and graphics processor 208 use embedded memory modules 218 as a shared Last Level Cache.

In some embodiments, processor cores 202A-202N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 202A-202N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 202A-N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores 202A-202N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor 200 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

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

memory. Memory interface 314 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

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

In some embodiments, graphics processor 300 includes a block image transfer (BLIT) engine 304 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) 310. In some embodiments, graphics processing engine 310 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

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

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

for the media operations on one or more graphics execution units included in 3D/Media sub-system 315.

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

3D/Media Processing

FIG. 9 is a block diagram of a graphics processing engine 410 of a graphics processor in accordance with some embodiments. In one embodiment, the GPE 410 is a version of the GPE 310 shown in FIG. 8. Elements of FIG. 9 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, GPE 410 couples with a command streamer 403, which provides a command stream to the GPE 3D and media pipelines 412, 416. In some embodiments, command streamer 403 is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 403 receives commands from the memory and sends the commands to 3D pipeline 412 and/or media pipeline 416. The commands are directives fetched from a ring buffer, which stores commands for the 3D and media pipelines 412, 416. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The 3D and media pipelines 412, 416 process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to an execution unit array 414. In some embodiments, execution unit array 414 is scalable, such that the array includes a variable number of execution units based on the target power and performance level of GPE 410.

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

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

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

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

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

Execution Units

FIG. 10 is a block diagram of another embodiment of a graphics processor 500. Elements of FIG. 10 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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

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

threads for the thread execution resources provided by at least one graphics core 580A.

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

FIG. 11 illustrates thread execution logic 600 including an array of processing elements employed in some embodiments of a GPE. Elements of FIG. 11 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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

processor capable of executing multiple simultaneous threads and processing multiple data elements in parallel for each thread. In some embodiments, execution unit array 608A-608N includes any number individual execution units.

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

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

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

some embodiments, sampler 610 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 600 via thread spawning and dispatch logic. In some embodiments, thread execution logic 600 includes a local thread dispatcher 604 that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units 608A-608N. For example, the geometry pipeline (e.g., 536 of FIG. 10) dispatches vertex processing, tessellation, or geometry processing threads to thread execution logic 600 (FIG. 11). In some embodiments, thread dispatcher 604 can also process runtime thread spawning requests from the executing shader programs.

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

In some embodiments, the data port 614 provides a memory access mechanism for the thread execution logic 600 output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data port 614 includes or couples to one or more cache memories (e.g., data cache 612) to cache data for memory access via the data port. Moreover, in some embodiments, the thread execution logic 600 performs dynamic thread splitting as described with regard to FIGs. 1-5.

FIG. 12 is a block diagram illustrating a graphics processor instruction formats 700 according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple

formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format 700 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

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

For each format, instruction opcode 712 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field 714 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions 710 an exec-size field 716 limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field 716 is not available for use in the 64-bit compact instruction format 730.

Some execution unit instructions have up to three operands including two source operands, src0 722, src1 722, and one destination 718. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 724), where the instruction opcode 712 determines the number of source

operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In some embodiments, the 128-bit instruction format 710 includes an access/address mode information 726 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction 710.

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

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

In some embodiments instructions are grouped based on opcode 712 bit-fields to simplify Opcode decode 740. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 742 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 742 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 744 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 746 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30).

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

Graphics Pipeline

FIG. 13 is a block diagram of another embodiment of a graphics processor 800. Elements of FIG. 13 having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

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

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

In some embodiments, execution units 852A, 852B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units 852A, 852B have an attached L1 cache 851 that is specific for each array or shared between the arrays. The cache can be configured as

a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

In some embodiments, graphics pipeline 820 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 813 operates at the direction of hull shader 811 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 820. In some embodiments, if tessellation is not used, tessellation components 811, 813, 817 can be bypassed.

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

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

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

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

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

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

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

In some embodiments, graphics pipeline 820 and media pipeline 830 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor

translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) from the Khronos Group, the Direct3D library from the Microsoft Corporation, or support may be provided to both OpenGL and D3D. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming

FIG. 14A is a block diagram illustrating a graphics processor command format 900 according to some embodiments. FIG. 14B is a block diagram illustrating a graphics processor command sequence 910 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.

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

The flow diagram in FIG. 14B shows an exemplary graphics processor command sequence 910. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

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

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

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

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

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.

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

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

In some embodiments, 3D pipeline 922 is triggered via an execute 934 command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a ‘go’ or ‘kick’ command in the command sequence. In one embodiment command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D

primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

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

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

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

Graphics Software Architecture

FIG. 15 illustrates exemplary graphics software architecture for a data processing system 1000 according to some embodiments. In some embodiments, software architecture includes a 3D graphics application 1010, an operating system 1020, and at least one processor 1030. In some embodiments, processor 1030 includes a graphics processor 1032 and one or more general-purpose processor core(s) 1034. The graphics application 1010 and operating system 1020 each execute in the system memory 1050 of the data processing system.

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

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

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

IP Core Implementations

One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

FIG. 16 is a block diagram illustrating an IP core development system 1100 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 1100 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 1130 can generate a software simulation 1110 of an IP core design in a high level programming language (e.g., C/C++). The software simulation 1110 can be used to design, test, and verify the behavior of the IP core using a simulation model 1112. The simulation model 1112 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design can then be created or synthesized from the simulation model 11001112. The RTL design 1115 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 1115, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

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

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

FIG. 17 is a block diagram illustrating an exemplary system on a chip integrated circuit 1200 that may be fabricated using one or more IP cores, according to an embodiment. The exemplary integrated circuit includes one or more application processors 1205 (e.g., CPUs), at least one graphics processor 1210, and may additionally include an image processor 1215 and/or a video processor 1220, any of which may be a modular IP core from the same or multiple different design facilities. The integrated circuit includes peripheral or bus logic including a USB controller 1225, UART controller 1230, an SPI/SDIO controller 1235, and an I2S/I2C controller 1240. Additionally, the integrated circuit can include a display device 1245 coupled to one or more of a high-definition multimedia interface (HDMI) controller 1250 and a mobile industry processor interface (MIPI) display interface 1255. Storage may be provided by a flash memory subsystem 1260 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 1265 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 1270.

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

Additional Notes and Examples:

Example 1 may include a multi-threaded computing system comprising a workload source to generate a graphics-related workload, a thread generator to generate a thread based on the graphics-related workload, the thread including a primary IF-ELSE conditional construct. The system may also comprise a thread execution apparatus including a condition handler to associate a first instruction pointer with an IF block of the primary IF-ELSE conditional construct and a thread splitter to activate a second instruction pointer in response to a dependency associated with the IF block and associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

Example 2 may include the system of Example 1, wherein the thread execution apparatus further includes one or more processor cores to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block independently from one another.

Example 3 may include the system of Example 1, wherein the thread execution apparatus further includes a boundary detector to detect an end of the IF block, and a jump component to jump the first instruction pointer from the end of the IF block to an end of the ELSE block, wherein the apparatus is to process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

Example 4 may include the system of Example 1, wherein the thread execution apparatus further includes a nest detector to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct, one or more processor cores to independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct, and a status monitor to track an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

Example 5 may include the system of Example 4, wherein the thread execution apparatus further includes one or more counters to track construct levels.

Example 6 may include the system of any one of Examples 1 to 5, wherein the condition handler is to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

Example 7 may include a thread execution apparatus comprising a condition handler to associate a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread and a thread splitter to activate a second instruction pointer in response to a dependency associated with the IF block and associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

Example 8 may include the apparatus of Example 7, further including one or more processor cores to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block independently from one another.

Example 9 may include the apparatus of Example 7, further including a boundary detector to detect an end of the IF block, and a jump component to jump the first instruction pointer from the end of the IF block to an end of the ELSE block, wherein the apparatus is to process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

Example 10 may include the apparatus of Example 7, further including a nest detector to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct, one or more processor cores to independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct, and a status monitor to track an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

Example 11 may include the apparatus of Example 10, further including one or more counters to track construct levels.

Example 12 may include the apparatus of any one of Examples 7 to 11, wherein the condition handler is to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

Example 13 may include a method of operating a thread execution apparatus comprising associating a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread, activating a second instruction pointer in response to a dependency associated with the IF block, and associating the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

Example 14 may include the method of Example 13, further including executing, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block independently from one another.

Example 15 may include the method of Example 13, further including detecting an end of the IF block, jumping the first instruction pointer from the end of the IF block to an end of the ELSE block, and processing one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

Example 16 may include the method of Example 13, further including detecting a nested IF-ELSE conditional construct within the primary IF-ELSE

conditional construct, independently executing, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct, and tracking an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

Example 17 may include the method of Example 16, further including tracking construct levels with one or more counters.

Example 18 may include the method of any one of Examples 13 to 17, further including confirming that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

Example 19 may include at least one computer readable storage medium comprising a set of instructions, which when executed by a computing device, cause the computing device to associate a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread, activate a second instruction pointer in response to a dependency associated with the IF block, and associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

Example 20 may include the at least one computer readable storage medium of Example 19, wherein the instructions, when executed, cause a computing device to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block independently from one another.

Example 21 may include the at least one computer readable storage medium of Example 19, wherein the instructions, when executed, cause a computing device to detect an end of the IF block, jump the first instruction pointer from the end of the IF block to an end of the ELSE block, and process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

Example 22 may include the at least one computer readable storage medium of Example 19, wherein the instructions, when executed, cause a computing device to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct, independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct, and track an ELSE assignment status and an ELSE completion

status of the first and second instruction pointers in one or more status tables on a per construct level basis.

Example 23 may include the at least one computer readable storage medium of Example 22, wherein the instructions, when executed, cause a computing device to track construct levels with one or more counters.

Example 24 may include the at least one computer readable storage medium of any one of Examples 19 to 23, wherein the instructions, when executed, cause a computing device to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

Example 25 may include a thread execution apparatus comprising means for associating a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread, means for activating a second instruction pointer in response to a dependency associated with the IF block, and means for associating the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

Example 26 may include the apparatus of Example 25, further including means for executing, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block independently from one another.

Example 27 may include the apparatus of Example 25, further including means for detecting an end of the IF block, means for jumping the first instruction pointer from the end of the IF block to an end of the ELSE block, and means for processing one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

Example 28 may include the apparatus of Example 25, further including means for detecting a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct, means for independently executing, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct, and means for tracking an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

Example 29 may include the apparatus of Example 28, further including means for tracking construct levels with one or more counters.

Example 30 may include the apparatus of any one of Examples 25 to 29, further including means for confirming that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

Techniques described herein may therefore provide for an execution unit supporting, for example, seven threads and thread splitting capability to be equivalent to supporting fourteen threads. The split threading capability may substantially increase instructions per cycle (IPC) rates for workloads having IF-ELSE conditional constructs. Indeed, many GPGPU and 3D workload kernels may have IF-ELSE conditional constructs with dependencies that may be leveraged using the techniques described herein.

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. Additionally, it is understood that the indefinite articles "a" or "an" carries the meaning of "one or more" or "at least one".

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

We claim:

1. A multi-threaded computing system comprising:

a workload source to generate a graphics-related workload;

a thread generator to generate a thread based on the graphics-related workload, the thread including a primary IF-ELSE conditional construct; and

a thread execution apparatus including:

a condition handler to associate a first instruction pointer with an IF block of the primary IF-ELSE conditional construct, and

a thread splitter to activate a second instruction pointer in response to a dependency associated with the IF block and associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

2. The system of claim 1, wherein the thread execution apparatus further includes one or more processor cores to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block one or more of independently from or parallel to one another.

3. The system of claim 1, wherein the thread execution apparatus further includes:

a boundary detector to detect an end of the IF block; and

a jump component to jump the first instruction pointer from the end of the IF block to an end of the ELSE block, wherein the apparatus is to process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

4. The system of claim 1, wherein the thread execution apparatus further includes:

a nest detector to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct;

one or more processor cores to independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct; and

a status monitor to track an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

5. The system of claim 4, wherein the thread execution apparatus further includes one or more counters to track construct levels.

6. The system of any one of claims 1 to 5, wherein the condition handler is to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

7. A thread execution apparatus comprising:

a condition handler to associate a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread; and

a thread splitter to activate a second instruction pointer in response to a dependency associated with the IF block and associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

8. The apparatus of claim 7, further including one or more processor cores to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block one or more of independently from or parallel to one another.

9. The apparatus of claim 7, further including:

a boundary detector to detect an end of the IF block; and

a jump component to jump the first instruction pointer from the end of the IF block to an end of the ELSE block, wherein the apparatus is to process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

10. The apparatus of claim 7, further including:

a nest detector to detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct;

one or more processor cores to independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct; and

a status monitor to track an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

11. The apparatus of claim 10, further including one or more counters to track construct levels.

12. The apparatus of any one of claims 7 to 11, wherein the condition handler is to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

13. A method of operating a thread execution apparatus, comprising:

associating a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread;

activating a second instruction pointer in response to a dependency associated with the IF block; and

associating the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

14. The method of claim 13, further including executing, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block one or more of independently from or parallel to one another.

15. The method of claim 13, further including:

detecting an end of the IF block;

jumping the first instruction pointer from the end of the IF block to an end of the ELSE block; and

processing one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

16. The method of claim 13, further including:

detecting a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct;

independently executing, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct; and

tracking an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

17. The method of claim 16, further including tracking construct levels with one or more counters.

18. The method of any one of claims 13 to 17, further including confirming that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

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

associate a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread;

activate a second instruction pointer in response to a dependency associated with the IF block; and

associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct.

20. The at least one computer readable storage medium of claim 19, wherein the instructions, when executed, cause a computing device to execute, via the first instruction pointer and the second instruction pointer, the IF block and the ELSE block one or more of independently from or parallel to one another.

21. The at least one computer readable storage medium of claim 19, wherein the instructions, when executed, cause a computing device to:

detect an end of the IF block;

jump the first instruction pointer from the end of the IF block to an end of the ELSE block; and

process one or more instructions outside the primary IF-ELSE conditional construct if execution of the ELSE block has completed.

22. The at least one computer readable storage medium of claim 19, wherein the instructions, when executed, cause a computing device to:

detect a nested IF-ELSE conditional construct within the primary IF-ELSE conditional construct;

independently execute, via the first instruction pointer and the second instruction pointer, instructions of IF and ELSE blocks in the nested IF-ELSE conditional construct; and

track an ELSE assignment status and an ELSE completion status of the first and second instruction pointers in one or more status tables on a per construct level basis.

23. The at least one computer readable storage medium of claim 22, wherein the instructions, when executed, cause a computing device to track construct levels with one or more counters.

24. The at least one computer readable storage medium of any one of claims 19 to 23, wherein the instructions, when executed, cause a computing device to confirm that the IF-ELSE conditional construct has one or more valid single instruction multiple data (SIMD) channels to be executed from the ELSE block.

25. A thread execution apparatus comprising means for performing the method of any one of claims 13 to 17.

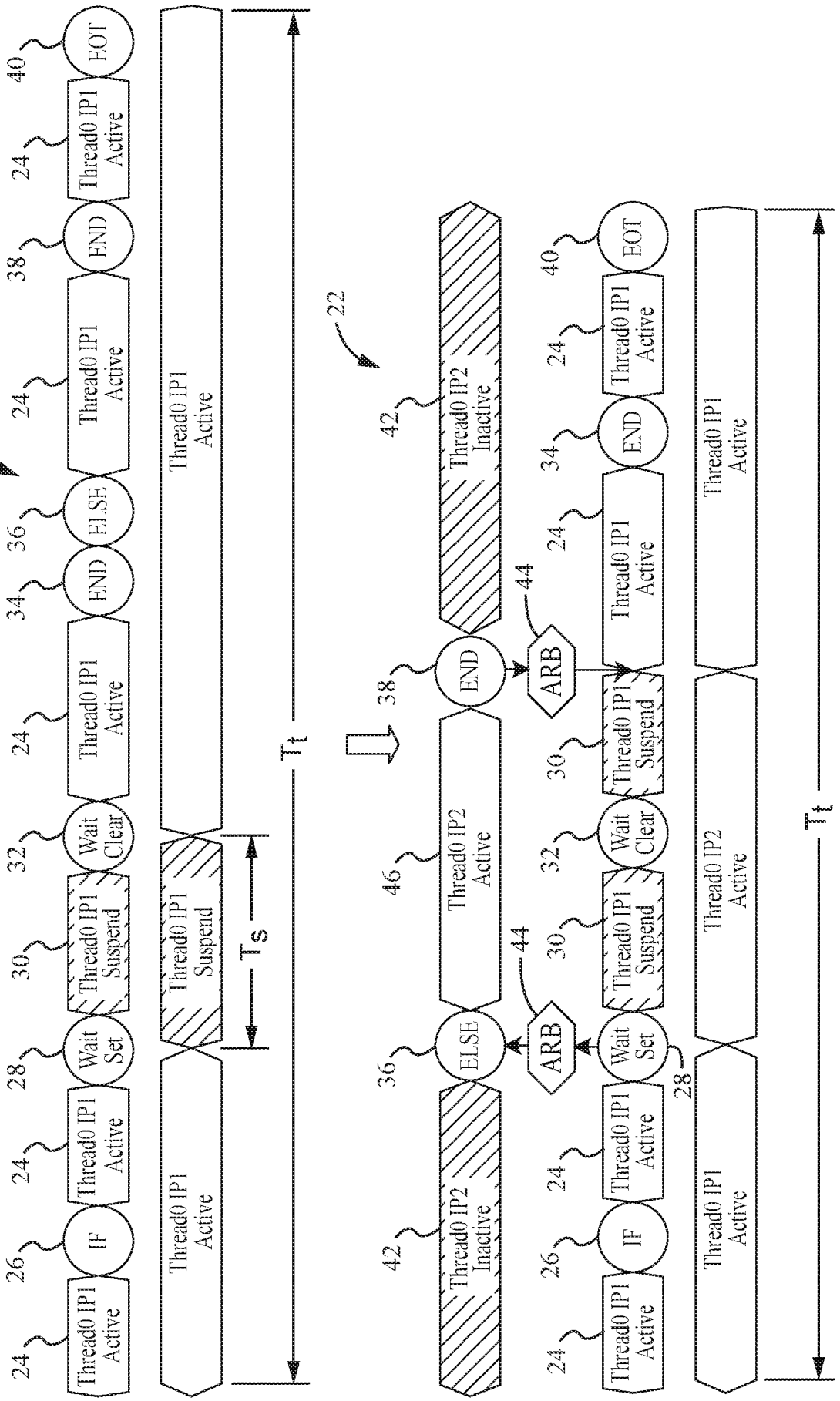

20
22
24
26
28
30
32
34
36
38
40
42
44
46
Thread0 IP1 Active
IF
Wait Set
Thread0 IP1 Suspend
Wait Clear
ELSE
END
EOT
Thread0 IP2 Inactive
Thread0 IP2 Active
ARB
Ts
Tt

FIG. 1

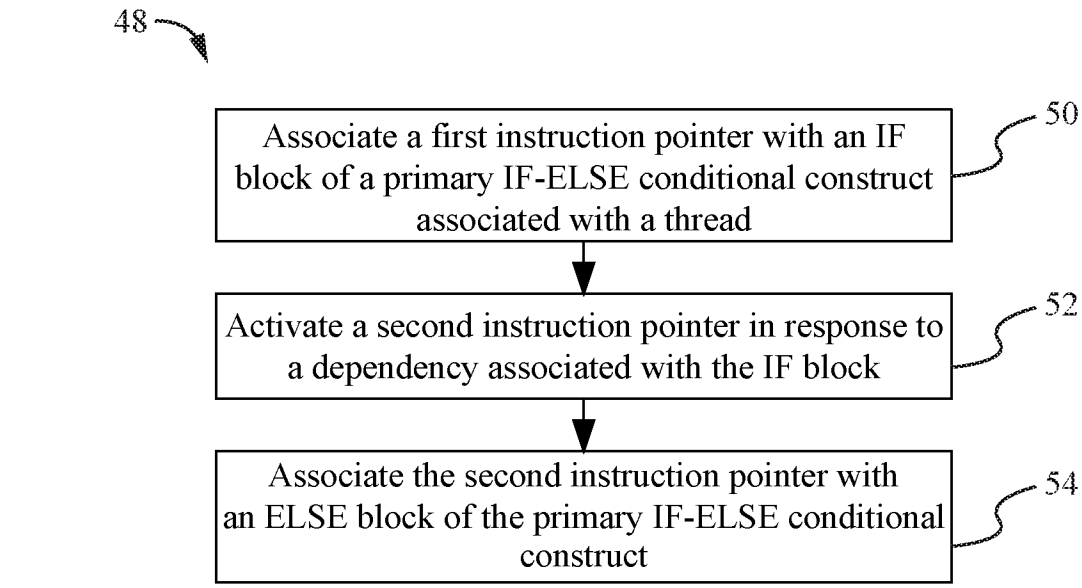

48
Associate a first instruction pointer with an IF block of a primary IF-ELSE conditional construct associated with a thread
50
Activate a second instruction pointer in response to a dependency associated with the IF block
52
Associate the second instruction pointer with an ELSE block of the primary IF-ELSE conditional construct
54

FIG. 2

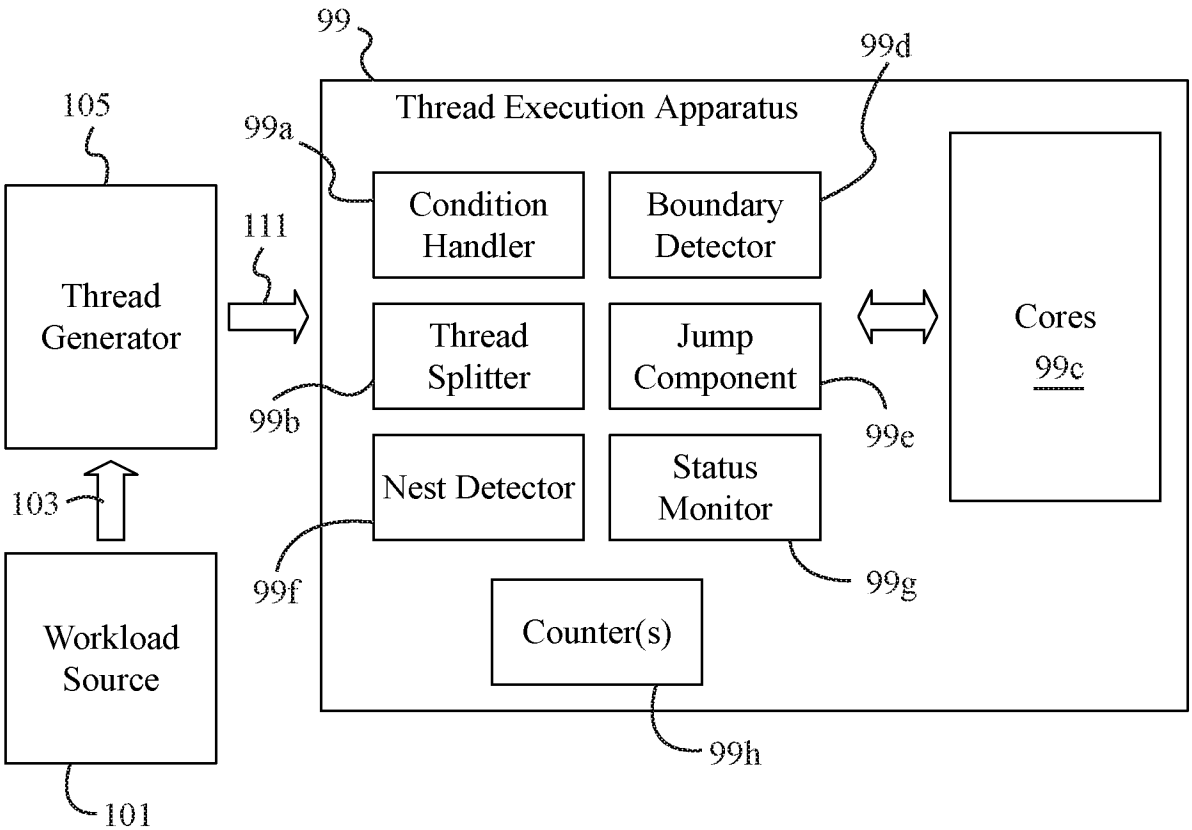

99
Thread Execution Apparatus
99d
105
99a
Condition Handler
Boundary Detector
111
Thread Generator
Thread Splitter
Jump Component
Cores
99c
99b
99e
Nest Detector
Status Monitor
103
99f
99g
Workload Source
Counter(s)
99h
101

FIG. 5

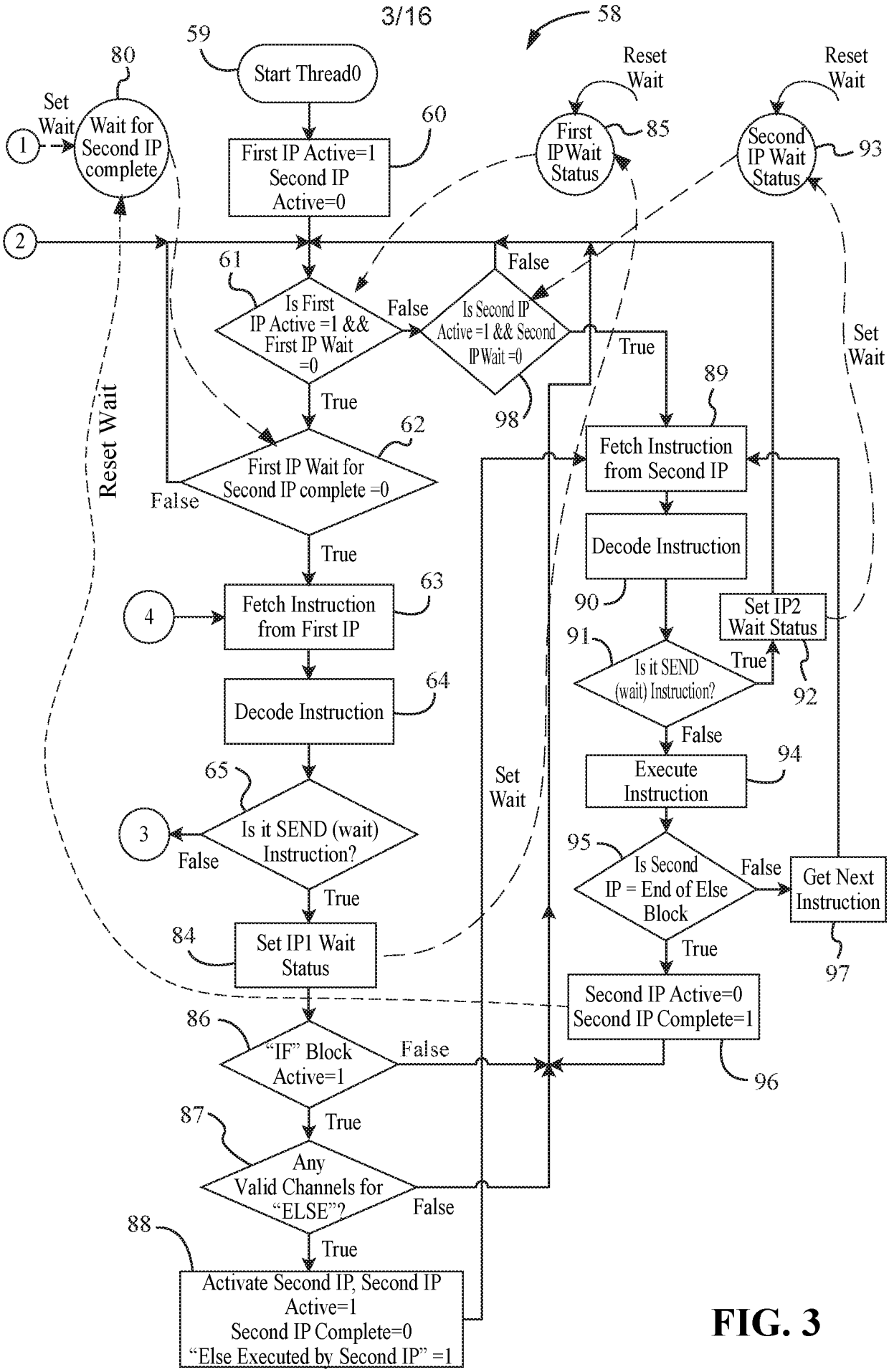

3/16
58
59
Start Thread0
80
Set Wait
Wait for Second IP complete
1
60
First IP Active=1 Second IP Active=0
Reset Wait
First IP Wait Status
85
Reset Wait
Second IP Wait Status
93
2
61
Is First IP Active =1 && First IP Wait =0
False
Is Second IP Active =1 && Second IP Wait =0
False
True
98
True
Set Wait
89
Reset Wait
62
First IP Wait for Second IP complete =0
False
Fetch Instruction from Second IP
Decode Instruction
90
True
63
4
Fetch Instruction from First IP
Set IP2 Wait Status
91
Is it SEND (wait) Instruction?
True
92
64
Decode Instruction
False
94
Execute Instruction
65
Set Wait
3
Is it SEND (wait) Instruction?
False
95
Is Second IP = End of Else Block
False
Get Next Instruction
True
97
84
Set IP1 Wait Status
Second IP Active=0 Second IP Complete=1
86
"IF" Block Active=1
False
96
True
87
Any Valid Channels for "ELSE"?
False
88
True
Activate Second IP, Second IP Active=1
Second IP Complete=0
"Else Executed by Second IP" =1

FIG. 3

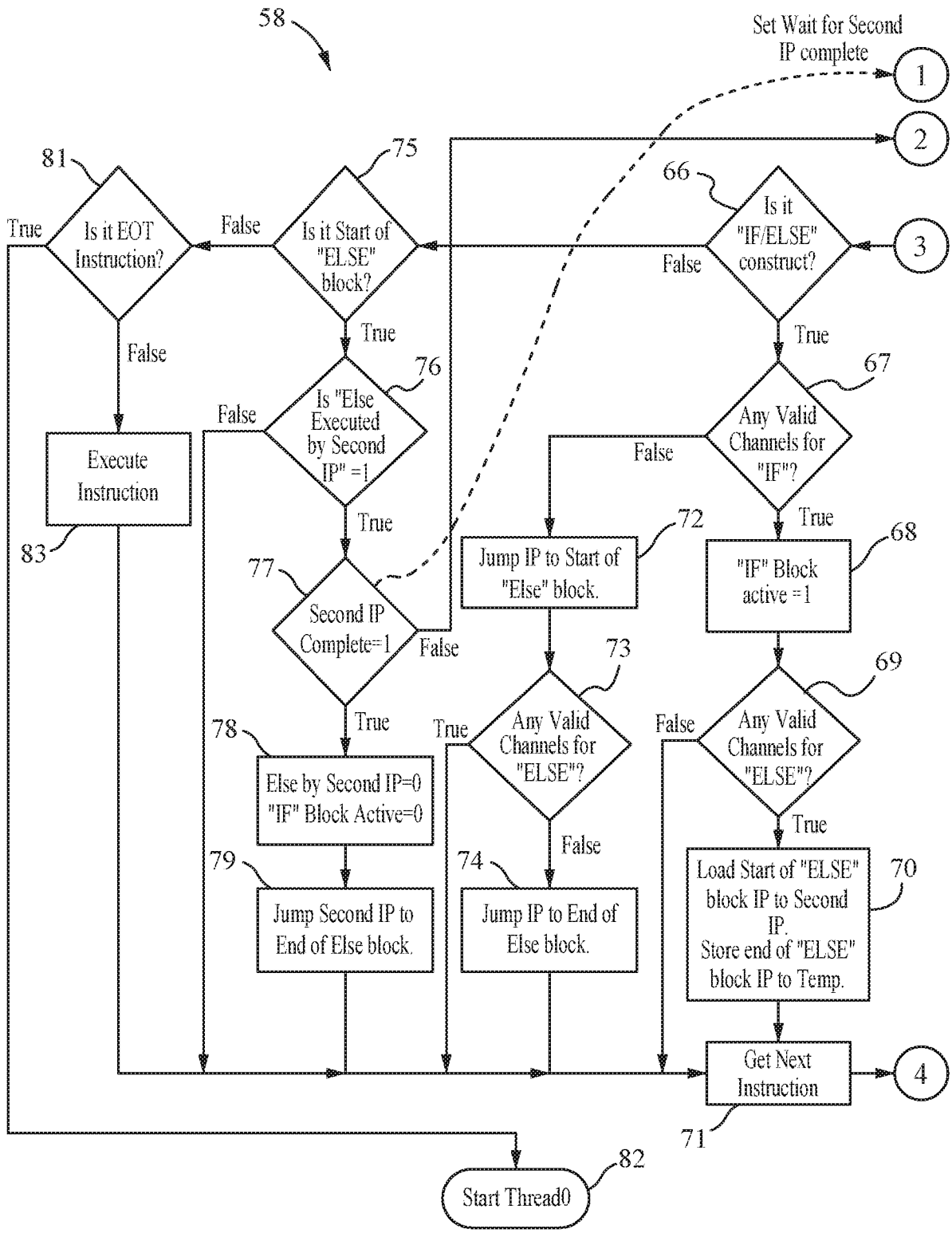

58
Set Wait for Second IP complete
1
2
3
4
81
Is it EOT Instruction?
True
False
75
Is it Start of "ELSE" block?
False
True
66
Is it "IF/ELSE" construct?
False
True
Execute Instruction
83
76
Is "Else Executed by Second IP" =1
False
True
67
Any Valid Channels for "IF"?
False
True
77
Second IP Complete=1
False
True
72
Jump IP to Start of "Else" block.
68
"IF" Block active =1
73
Any Valid Channels for "ELSE"?
True
False
69
Any Valid Channels for "ELSE"?
False
True
78
Else by Second IP=0 "IF" Block Active=0
79
Jump Second IP to End of Else block.
74
Jump IP to End of Else block.
70
Load Start of "ELSE" block IP to Second IP. Store end of "ELSE" block IP to Temp.
Get Next Instruction
71
82
Start Thread0

FIG. 4

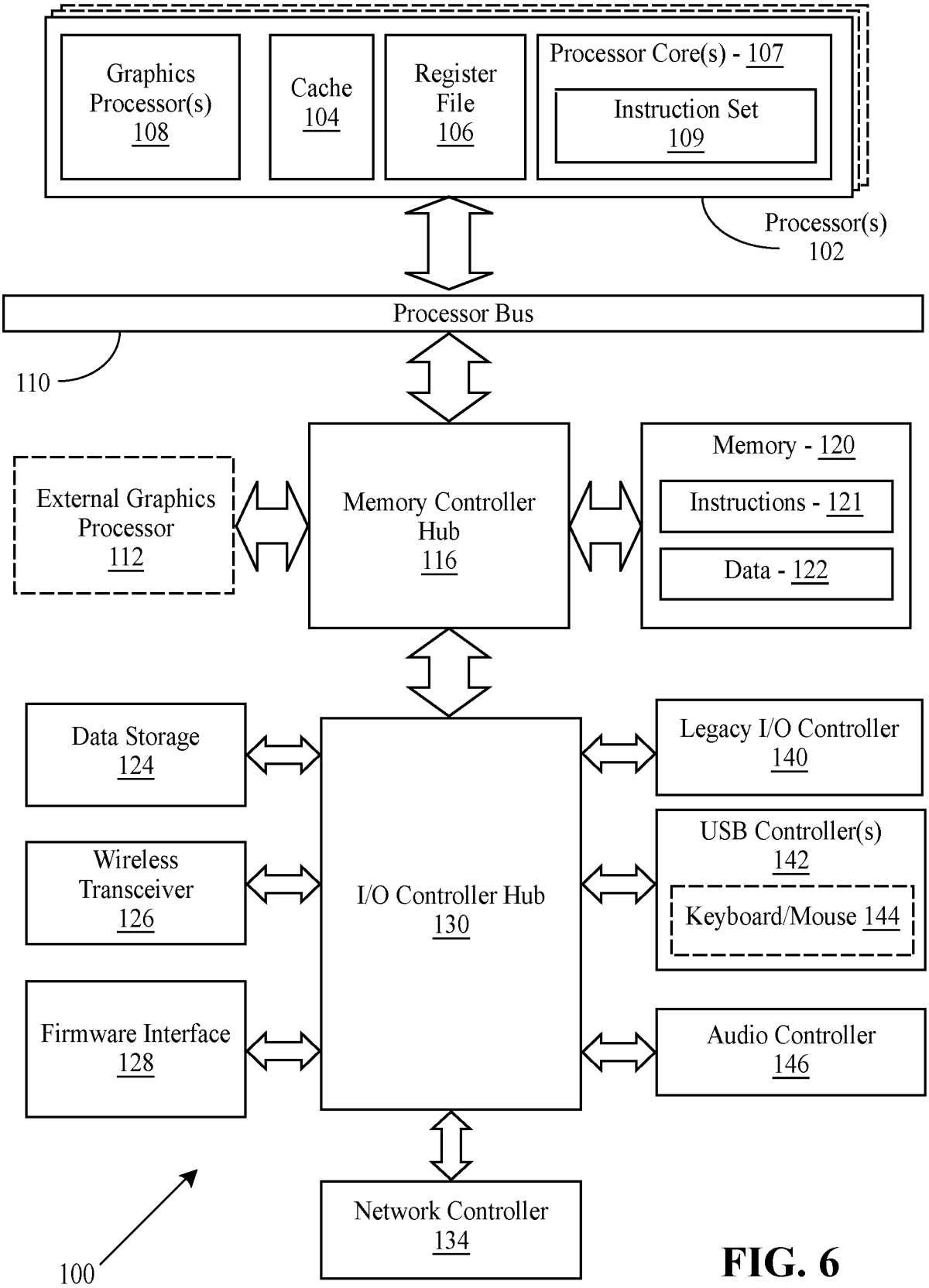

Graphics Processor(s) 108
Cache 104
Register File 106
Processor Core(s) - 107
Instruction Set 109
Processor(s) 102
Processor Bus
110
External Graphics Processor 112
Memory Controller Hub 116
Memory - 120
Instructions - 121
Data - 122
Data Storage 124
Wireless Transceiver 126
Firmware Interface 128
I/O Controller Hub 130
Legacy I/O Controller 140
USB Controller(s) 142
Keyboard/Mouse 144
Audio Controller 146
Network Controller 134
100

FIG. 6

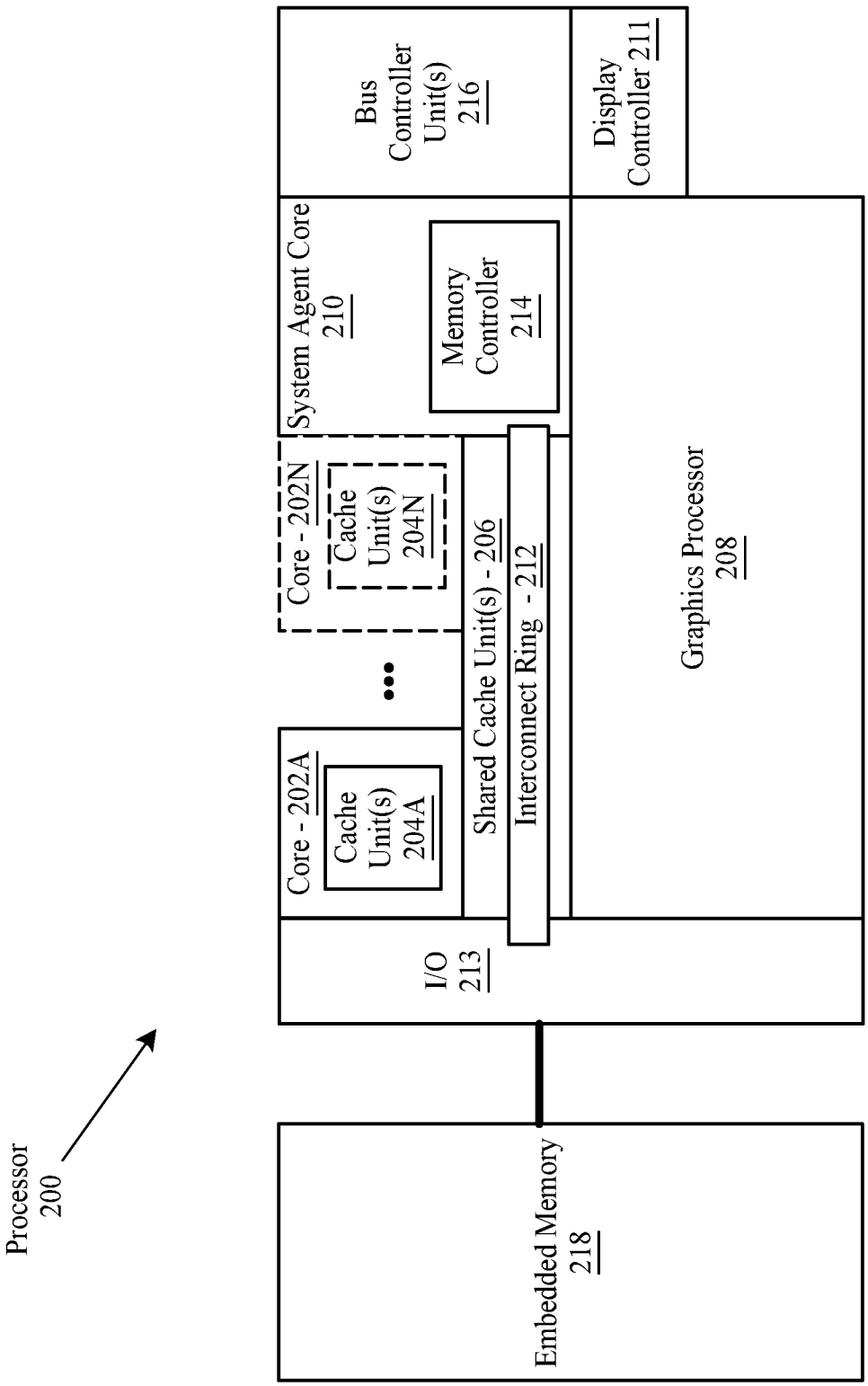

Processor
200
Embedded Memory
218
I/O
213
Core - 202A
Cache
Unit(s)
204A
Core - 202N
Cache
Unit(s)
204N
Shared Cache Unit(s) - 206
Interconnect Ring - 212
System Agent Core
210
Memory
Controller
214
Graphics Processor
208
Bus
Controller
Unit(s)
216
Display
Controller 211

FIG. 7

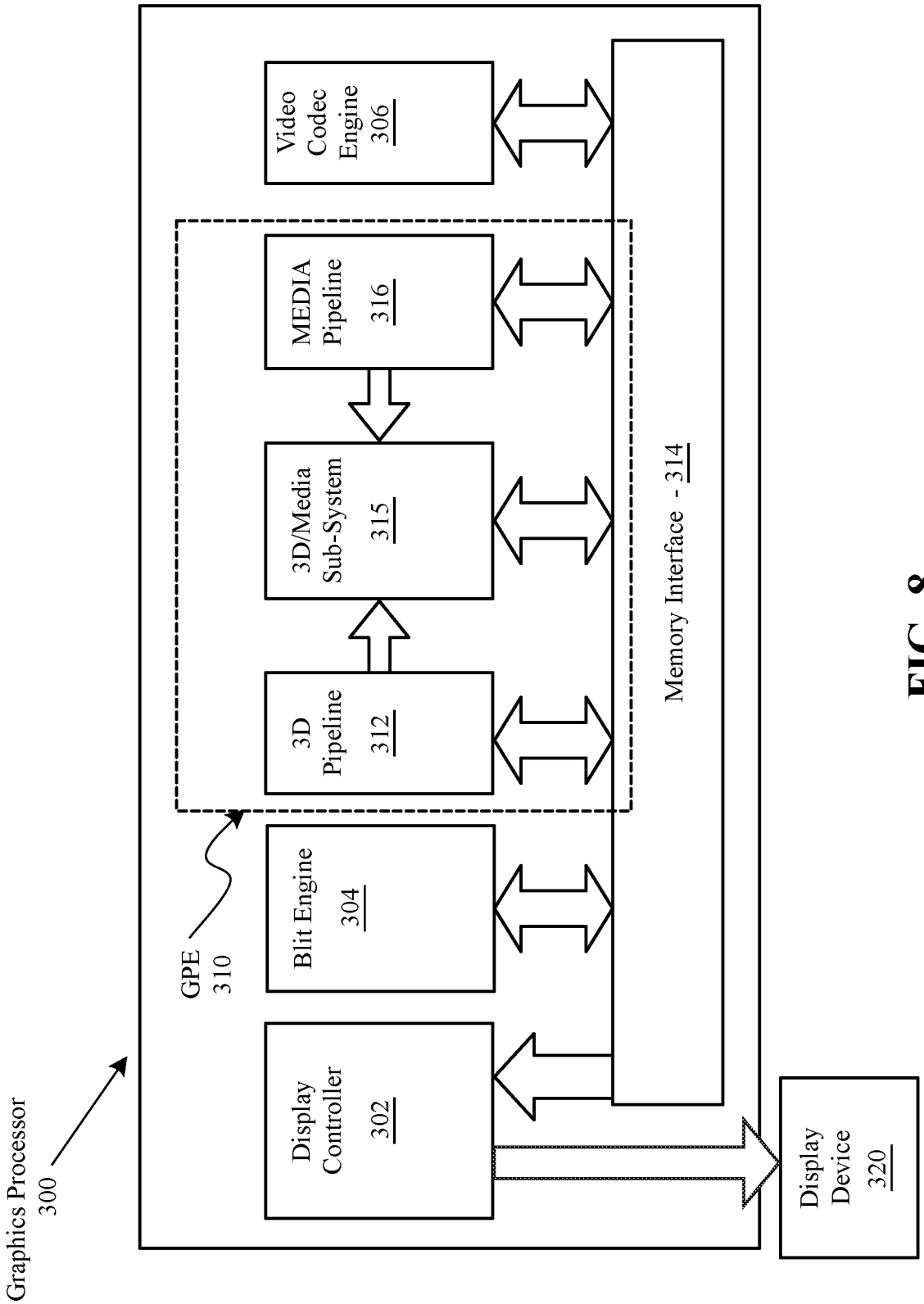

Graphics Processor
300
Display Controller
302
Blit Engine
304
GPE
310
3D Pipeline
312
3D/Media Sub-System
315
MEDIA Pipeline
316
Video Codec Engine
306
Memory Interface - 314
Display Device
320

FIG. 8

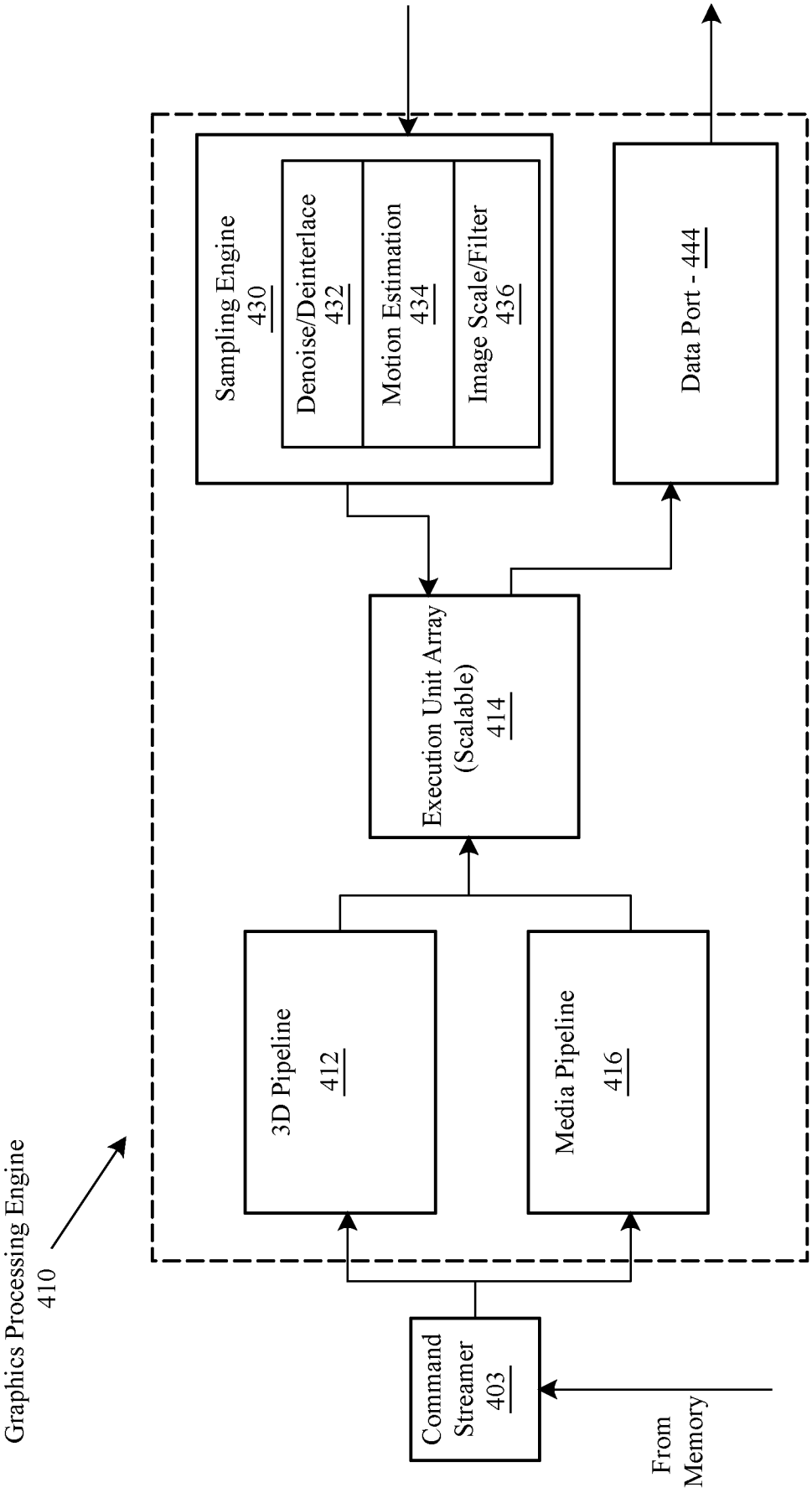

Graphics Processing Engine
410
Command Streamer
403
From Memory
3D Pipeline
412
Media Pipeline
416
Execution Unit Array
(Scalable)
414
Sampling Engine
430
Denoise/Deinterlace
432
Motion Estimation
434
Image Scale/Filter
436
Data Port - 444

FIG. 9

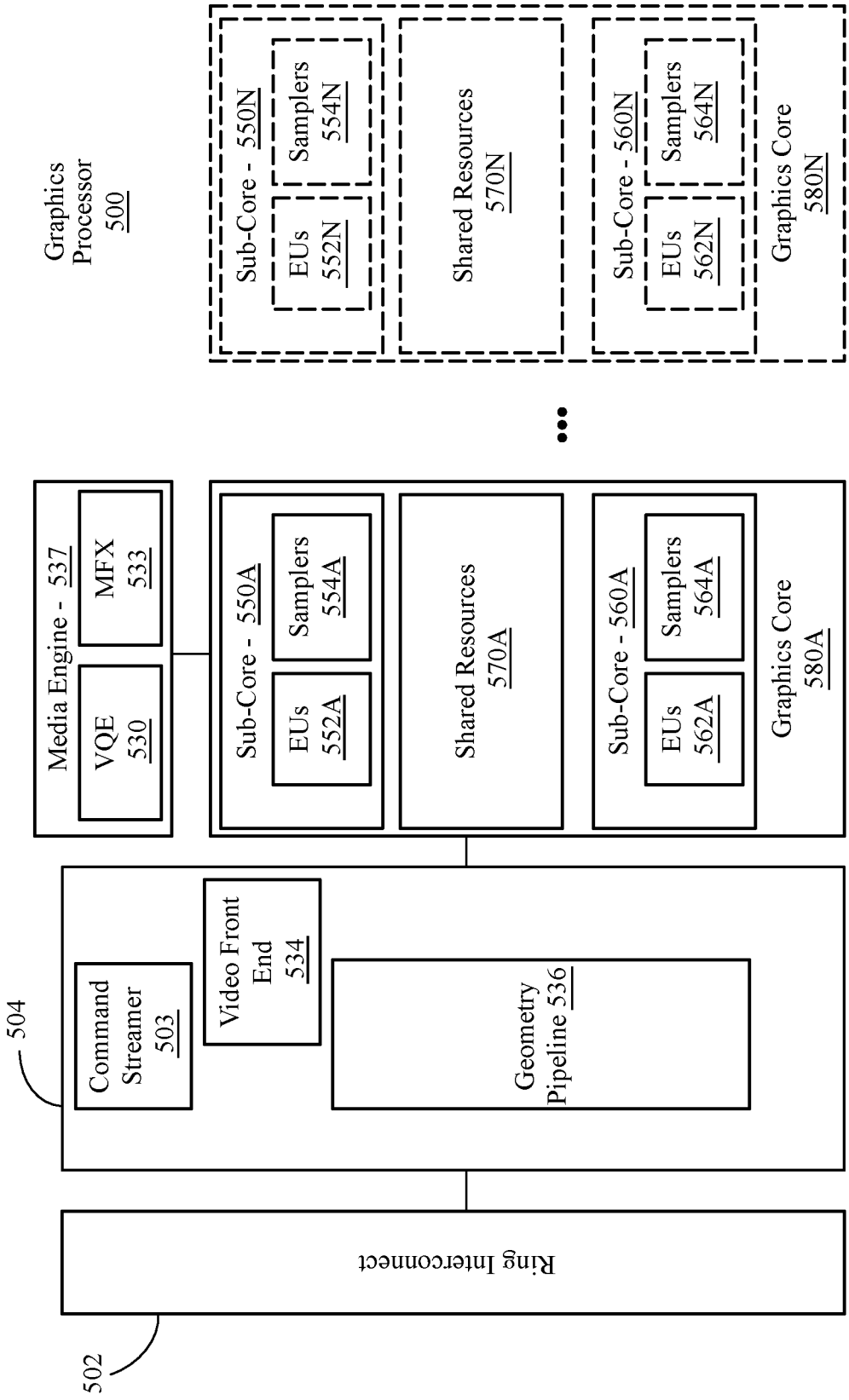

Graphics Processor 500
Sub-Core - 550N
EUs 552N
Samplers 554N
Shared Resources 570N
Sub-Core - 560N
EUs 562N
Samplers 564N
Graphics Core 580N
Media Engine - 537
VQE 530
MFX 533
Sub-Core - 550A
EUs 552A
Samplers 554A
Shared Resources 570A
Sub-Core - 560A
EUs 562A
Samplers 564A
Graphics Core 580A
504
Command Streamer 503
Video Front End 534
Geometry Pipeline 536
Ring Interconnect
502

FIG. 10

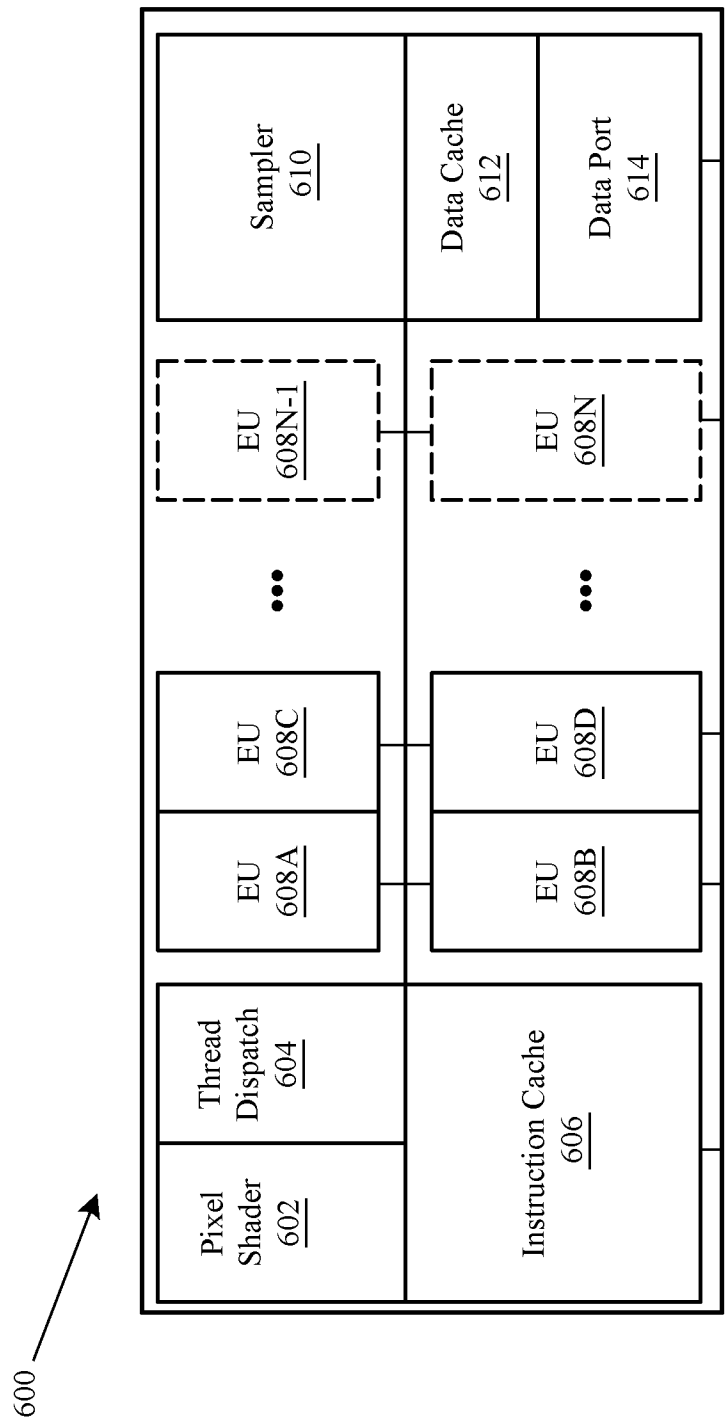

600
Pixel Shader 602
Thread Dispatch 604
Instruction Cache 606
EU 608A
EU 608C
EU 608B
EU 608D
EU 608N-1
EU 608N
Sampler 610
Data Cache 612
Data Port 614

FIG. 11

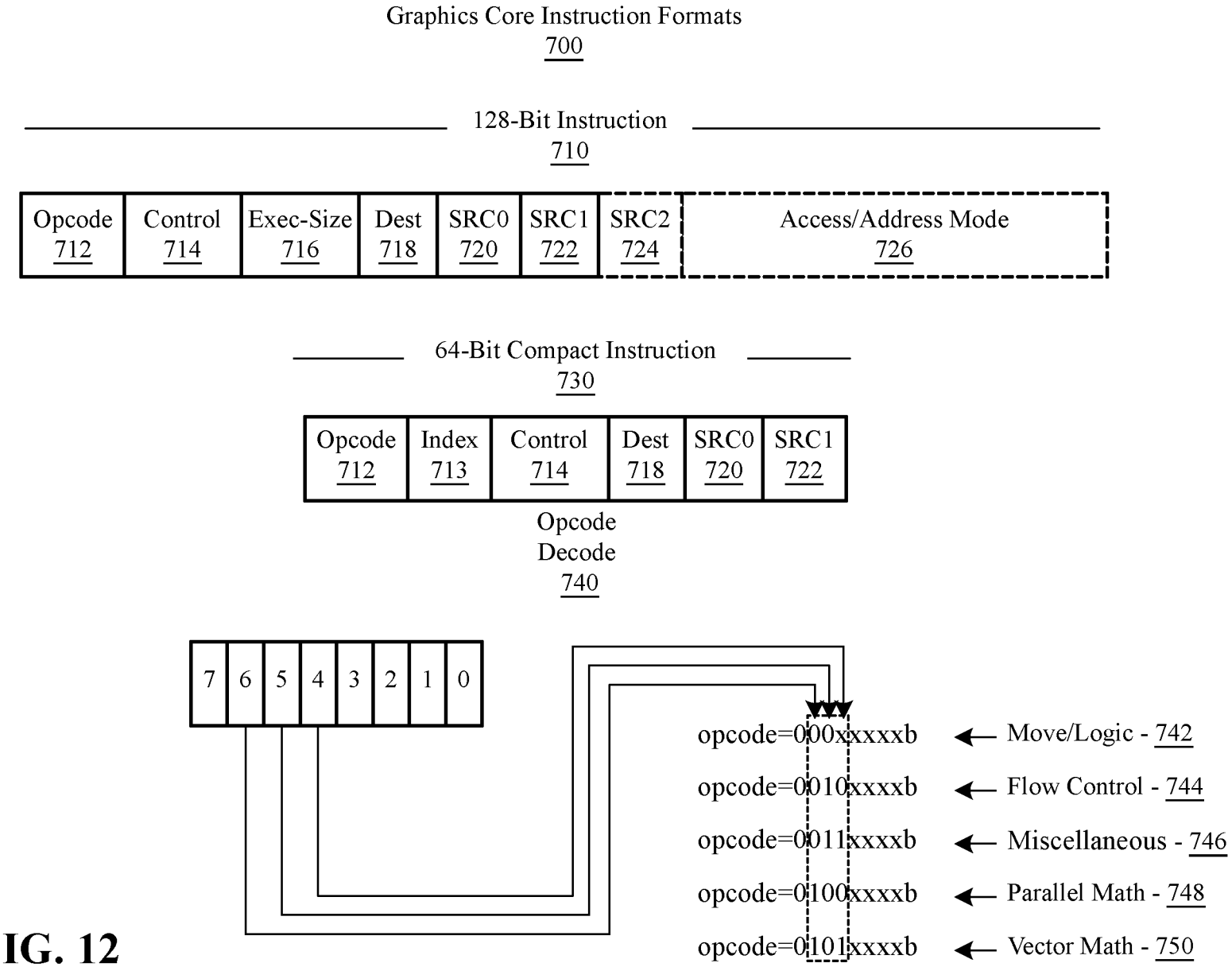

Graphics Core Instruction Formats
700
128-Bit Instruction
710
Opcode 712
Control 714
Exec-Size 716
Dest 718
SRC0 720
SRC1 722
SRC2 724
Access/Address Mode 726
64-Bit Compact Instruction
730
Opcode 712
Index 713
Control 714
Dest 718
SRC0 720
SRC1 722
Opcode Decode 740
7 6 5 4 3 2 1 0
opcode=000xxxxb ← Move/Logic - 742
opcode=0010xxxxb ← Flow Control - 744
opcode=0011xxxxb ← Miscellaneous - 746
opcode=0100xxxxb ← Parallel Math - 748
opcode=0101xxxxb ← Vector Math - 750

**FIG. 12**

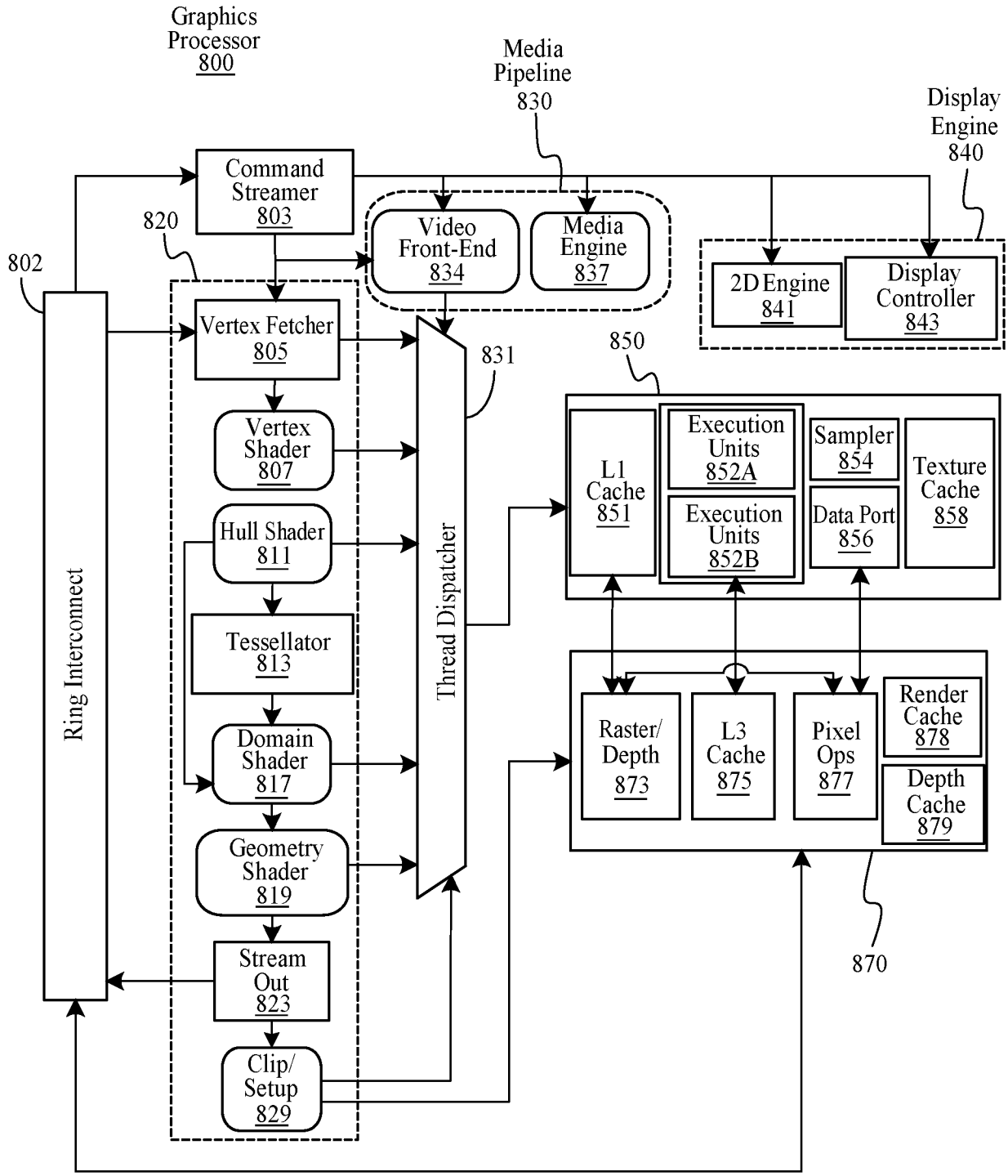

Graphics Processor 800
Media Pipeline 830
Display Engine 840
Command Streamer 803
820
802
Video Front-End 834
Media Engine 837
2D Engine 841
Display Controller 843
Vertex Fetcher 805
831
850
Vertex Shader 807
Hull Shader 811
Tessellator 813
Domain Shader 817
Geometry Shader 819
Stream Out 823
Clip/ Setup 829
Ring Interconnect
Thread Dispatcher
L1 Cache 851
Execution Units 852A
Execution Units 852B
Sampler 854
Data Port 856
Texture Cache 858
Raster/ Depth 873
L3 Cache 875
Pixel Ops 877
Render Cache 878
Depth Cache 879
870

FIG. 13

FIG. 14A

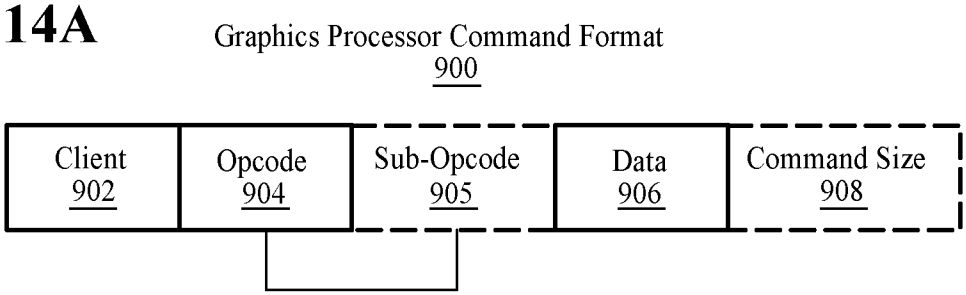
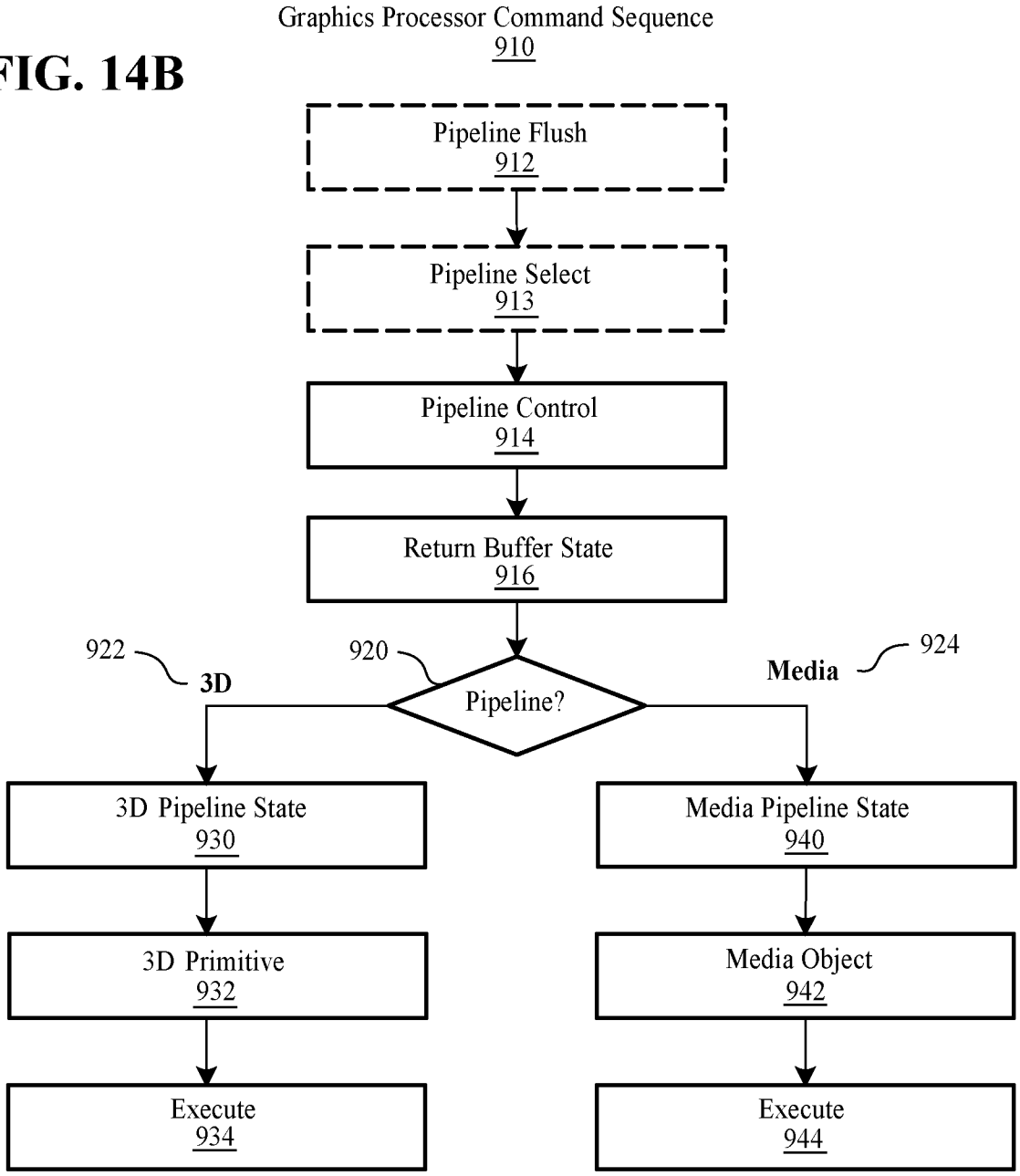



FIG. 14B



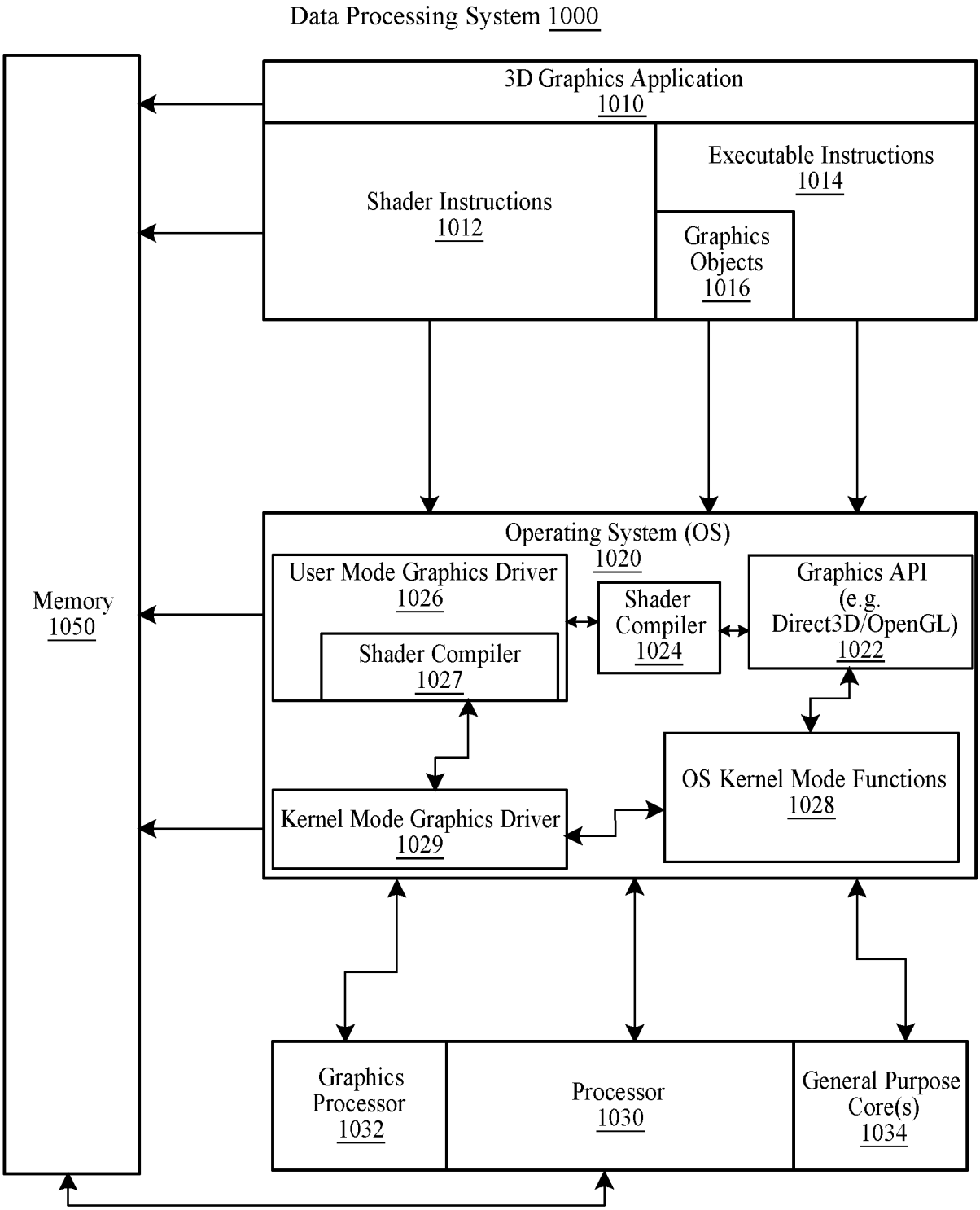

Data Processing System 1000
3D Graphics Application 1010
Shader Instructions 1012
Executable Instructions 1014
Graphics Objects 1016
Memory 1050
Operating System (OS) 1020
User Mode Graphics Driver 1026
Shader Compiler 1027
Shader Compiler 1024
Graphics API (e.g. Direct3D/OpenGL) 1022
OS Kernel Mode Functions 1028
Kernel Mode Graphics Driver 1029
Graphics Processor 1032
Processor 1030
General Purpose Core(s) 1034

FIG. 15

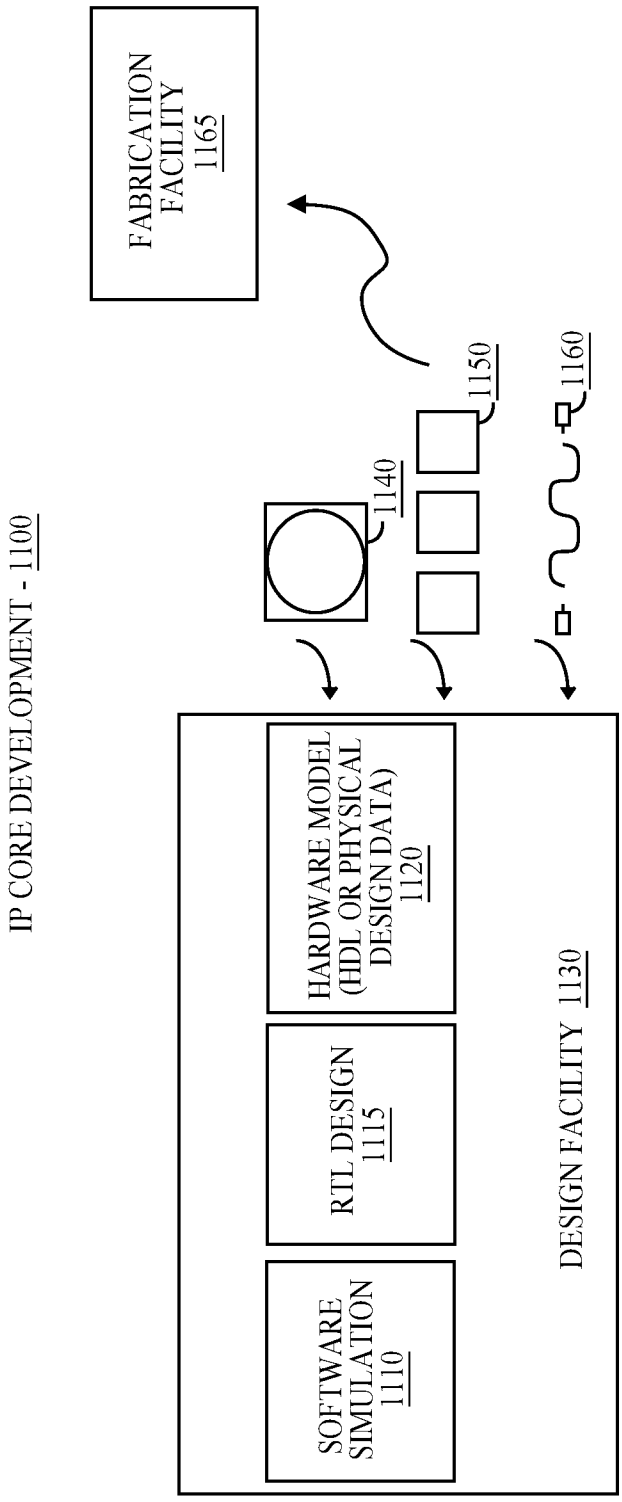

IP CORE DEVELOPMENT - 1100
SOFTWARE SIMULATION 1110
RTL DESIGN 1115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 1120
DESIGN FACILITY 1130
1140
1150
1160
FABRICATION FACILITY 1165

FIG. 16

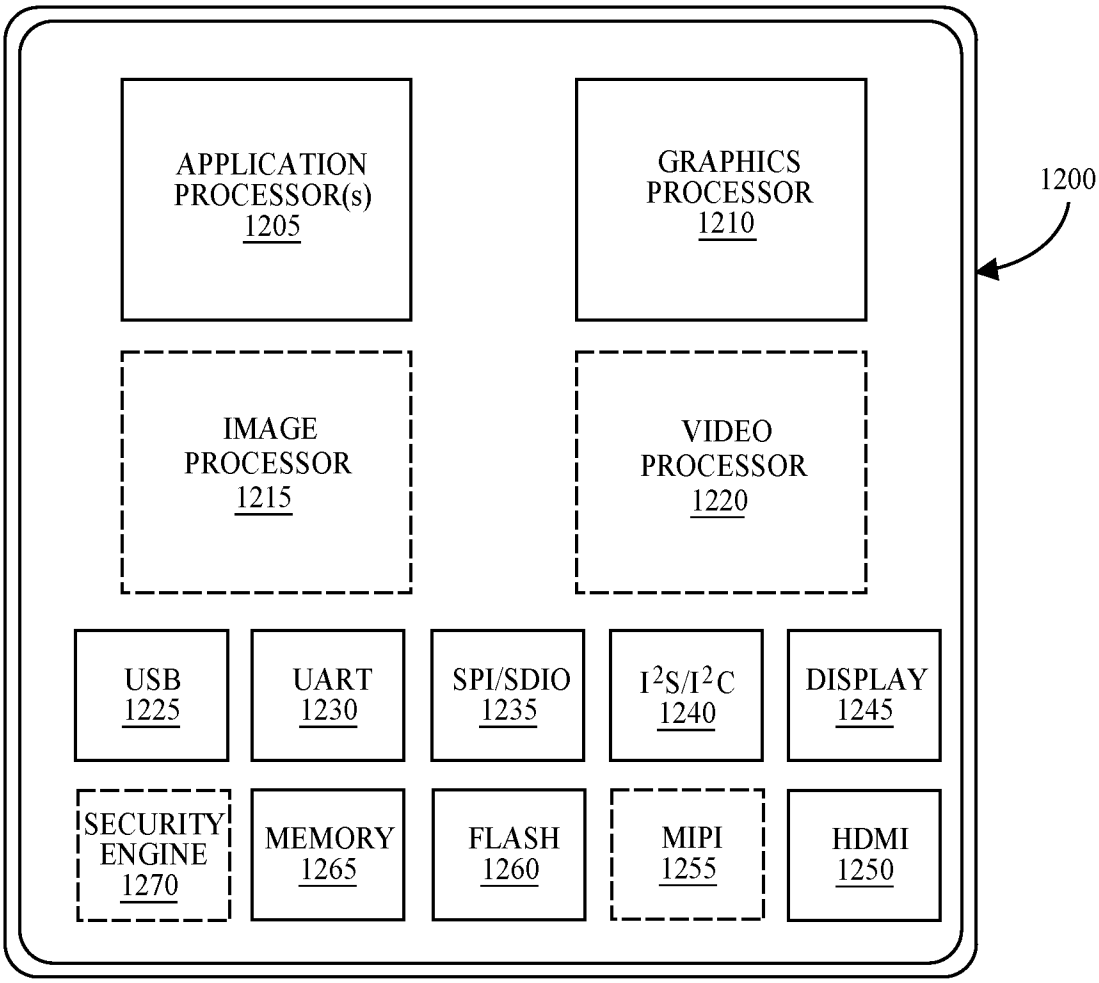

APPLICATION PROCESSOR(s) 1205
GRAPHICS PROCESSOR 1210
1200
IMAGE PROCESSOR 1215
VIDEO PROCESSOR 1220
USB 1225
UART 1230
SPI/SDIO 1235
I2S/I2C 1240
DISPLAY 1245
SECURITY ENGINE 1270
MEMORY 1265
FLASH 1260
MIPI 1255
HDMI 1250

FIG. 17

**INTERNATIONAL SEARCH REPORT**

International application No.
**PCT/US2016/035849**

**A. CLASSIFICATION OF SUBJECT MATTER**

**G06F 9/50(2006.01)I, G06F 9/48(2006.01)I, G06F 9/38(2006.01)I, G06T 1/20(2006.01)I**

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

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
G06F 9/50; G06F 15/16; G06F 15/347; G06F 17/16; G06F 9/46; G06T 1/20; G06F 9/38; G06T 1/00; G06F 9/48

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

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & keywords: multi-thread, workload, graphic, IF-ELSE, instruction, block, pointer, splitter

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| Y | US 2013-0332939 A1 (ARM LIMITED) 12 December 2013<br>See paragraphs [0037]-[0041]; claim 1; and figure 1. | 1-25 |
| Y | US 5604913 A (YOICHI KOYANAGI et al.) 18 February 1997<br>See column 3, line 49 - column 5, line 44; claims 1-4; and figures 4-5, 15. | 1-25 |
| Y | US 5790854 A (JASON SPIELMAN et al.) 04 August 1998<br>See column 3, lines 19-22; column 11, line 31 - column 13, line 53; and figures 10-15. | 4-6,10-12,16-18,22-24 |
| A | US 2015-0187040 A1 (JAYANTH N. RAO et al.) 02 July 2015<br>See paragraphs [0027]-[0046]; claim 1; and figures 2-4. | 1-25 |
| A | US 2008-0074433 A1 (GUOFANG JIAO et al.) 27 March 2008<br>See paragraphs [0032]-[0047]; claims 1, 4, 6-7; and figure 3. | 1-25 |
| A | US 2013-0138926 A1 (NVIDIA CORPORATION) 30 May 2013<br>See paragraphs [0039]-[0072]; claim 1; and figures 3-4C. | 1-25 |

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

☒ See patent family annex.

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

| Date of the actual completion of the international search | Date of mailing of the international search report |
|---|---|
| 05 October 2016 (05.10.2016) | **05 October 2016 (05.10.2016)** |

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

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

**INTERNATIONAL SEARCH REPORT**

Information on patent family members

International application No.
**PCT/US2016/035849**

| Patent document cited in search report | Publication date | Patent family member(s) | Publication date |
|---|---|---|---|
| US 2013-0332939 A1 | 12/12/2013 | US 2012-0304194 A1 | 29/11/2012 |
| | | US 8601485 B2 | 03/12/2013 |
| | | US 8978038 B2 | 10/03/2015 |
| US 5604913 A | 18/02/1997 | JP 07-056892 A | 03/03/1995 |
| US 5790854 A | 04/08/1998 | CN 1080906 C | 13/03/2002 |
| | | EP 0619557 A2 | 12/10/1994 |
| | | JP 07-073149 A | 17/03/1995 |
| | | JP 2006-012182 A | 12/01/2006 |
| | | KR 10-1994-0022257 A | 20/10/1994 |
| | | US 5537562 A | 16/07/1996 |
| | | US 5548768 A | 20/08/1996 |
| | | US 5559973 A | 24/09/1996 |
| | | US 5572689 A | 05/11/1996 |
| | | US 5598571 A | 28/01/1997 |
| | | US 5600811 A | 04/02/1997 |
| | | US 5600846 A | 04/02/1997 |
| | | US 5664134 A | 02/09/1997 |
| | | US 5706488 A | 06/01/1998 |
| | | US 5717947 A | 10/02/1998 |
| | | US 5734879 A | 31/03/1998 |
| | | US 5737586 A | 07/04/1998 |
| | | US 5742786 A | 21/04/1998 |
| | | US 5752074 A | 12/05/1998 |
| | | US 5754805 A | 19/05/1998 |
| | | US 5805874 A | 08/09/1998 |
| | | US 6085275 A | 04/07/2000 |
| US 2015-0187040 A1 | 02/07/2015 | None | |
| US 2008-0074433 A1 | 27/03/2008 | US 8345053 B2 | 01/01/2013 |
| | | WO 2008-036852 A1 | 27/03/2008 |
| US 2013-0138926 A1 | 30/05/2013 | GB 0904014 D0 | 22/04/2009 |
| | | GB 2458556 A | 30/09/2009 |
| | | GB 2458556 B | 07/07/2010 |
| | | JP 2009-230756 A | 08/10/2009 |
| | | JP 2013-016192 A | 24/01/2013 |
| | | JP 5701487 B2 | 15/04/2015 |
| | | US 2009-0240931 A1 | 24/09/2009 |
| | | US 8312254 B2 | 13/11/2012 |