US 20060026394A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2006/0026394 A1

Chauvel (43) Pub. Date: Feb. 2, 2006

(54) OPTIMIZING DATA MANIPULATION IN MEDIA PROCESSING APPLICATIONS

(75) Inventor: Gerard Chauvel, Antibes (FR)

Correspondence Address:
TEXAS INSTRUMENTS INCORPORATED
P O BOX 655474, M/S 3999
DALLAS, TX 75265

(73) Assignee: Texas Instruments Incorporated, Dallas, TX

(21) Appl. No.: 11/186,330

(22) Filed: Jul. 21, 2005

(30) Foreign Application Priority Data

Jul. 27, 2004 (EP) 04291918.3

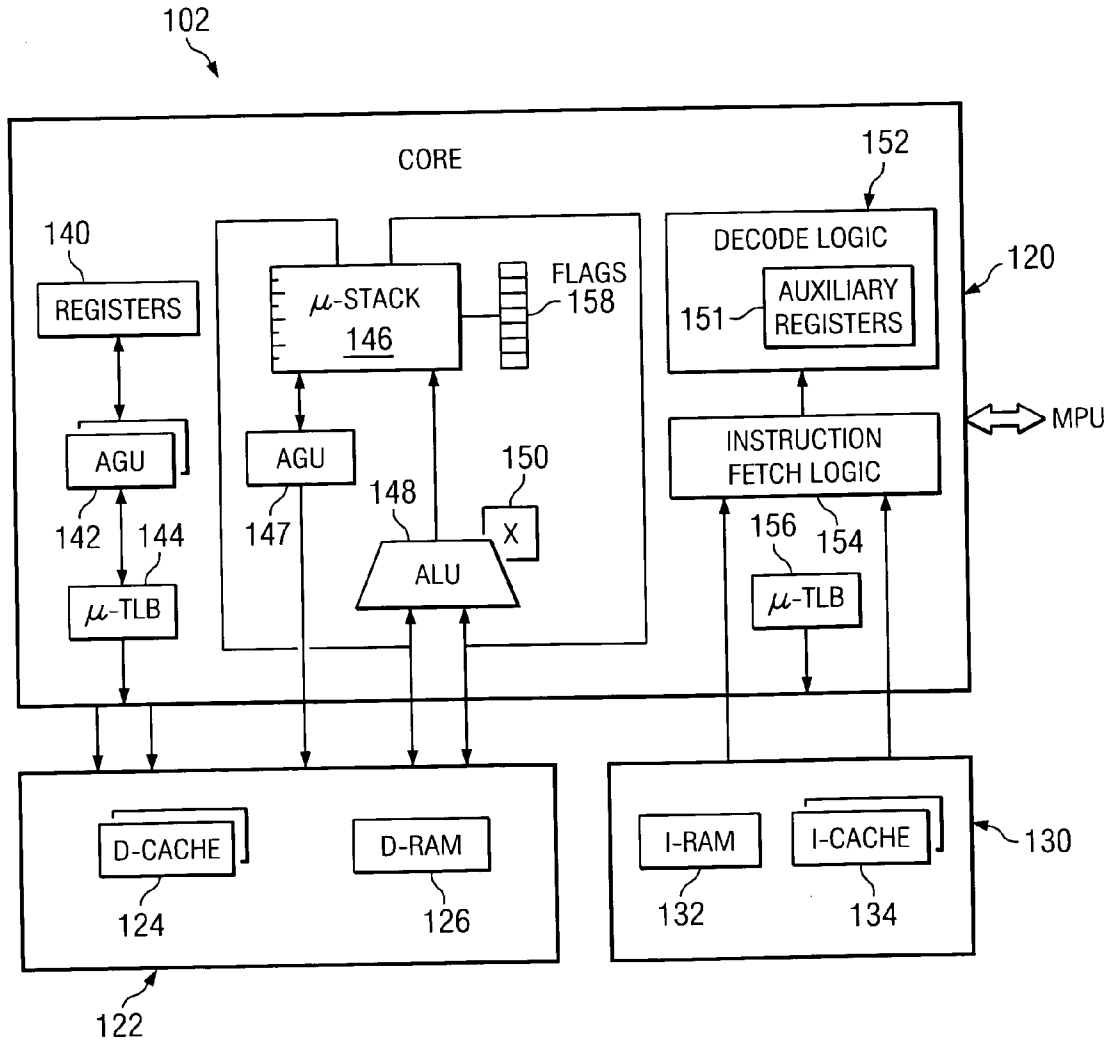
Publication Classification

(51) Int. Cl.
G06F 9/00 (2006.01)

(52) U.S. Cl. 712/221

(57) ABSTRACT

A system comprising a processor containing a first stack internal to a core of the processor, at least some data values in the first stack corresponding to values in a second stack external to the core. The system also comprises a memory coupled to the processor. In an iterative process, the processor pops a data value off of the first stack and begins to store the data value to the memory while the processor begins to use an existing data value from the first stack to produce a new data value to be stored on the first stack.



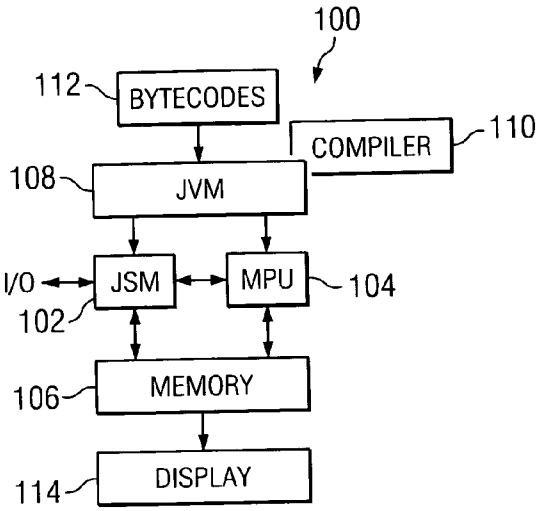


FIG. 1

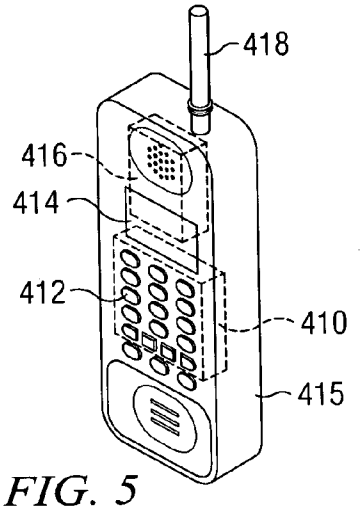


FIG. 5

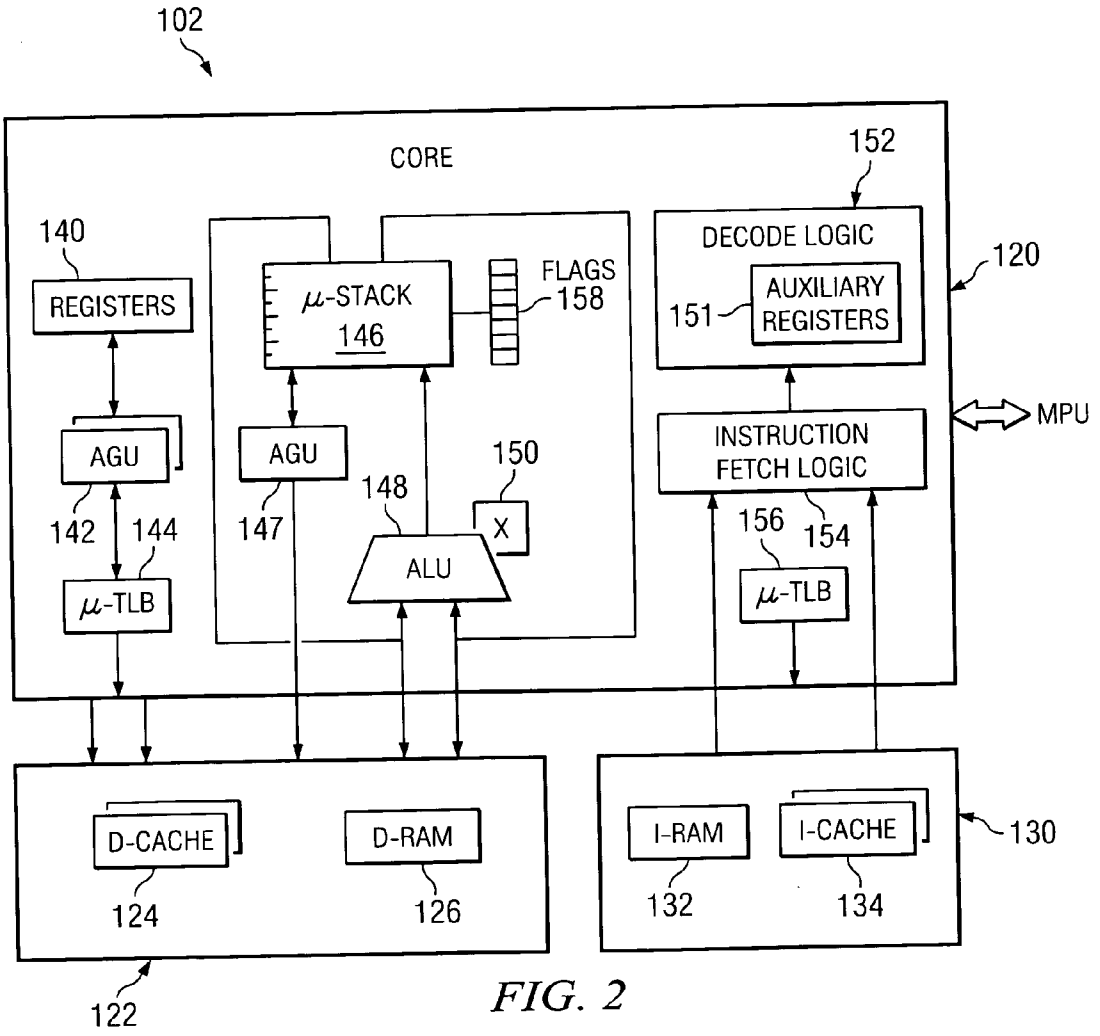


FIG. 2

R0	GENERAL PURPOSE (GP)
R1	GENERAL PURPOSE (GP)
R2	GENERAL PURPOSE (GP)
R3	GENERAL PURPOSE (GP)
R4	GENERAL PURPOSE (GP)
R5	GENERAL PURPOSE/LOCAL VARIABLE POINTER (LV)
R6	STACK POINTER (SP)
R7	TOP OF STACK (ToS)
R8	GENERAL PURPOSE (GP)
R9	GENERAL PURPOSE (GP)
R10	GENERAL PURPOSE (GP)
R11	GENERAL PURPOSE (GP)
R12	GENERAL PURPOSE (GP)
R13	GENERAL PURPOSE (GP)
R14	GENERAL PURPOSE (GP)
R15	STATUS AND CONTROL (ST)

FIG. 3

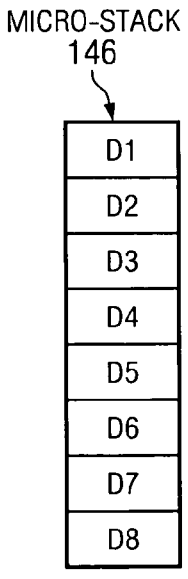


FIG. 4A

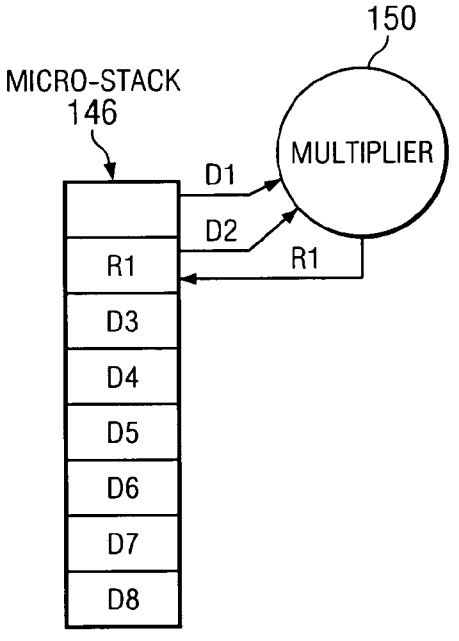


FIG. 4B

FIG. 4C

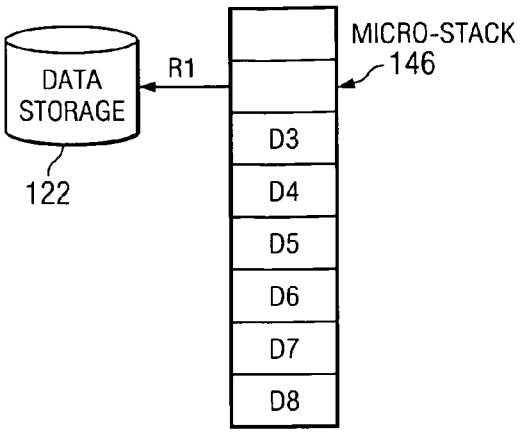


FIG. 4D

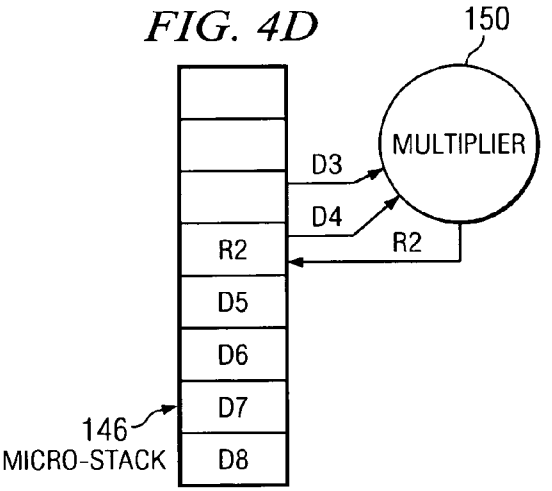


FIG. 4E

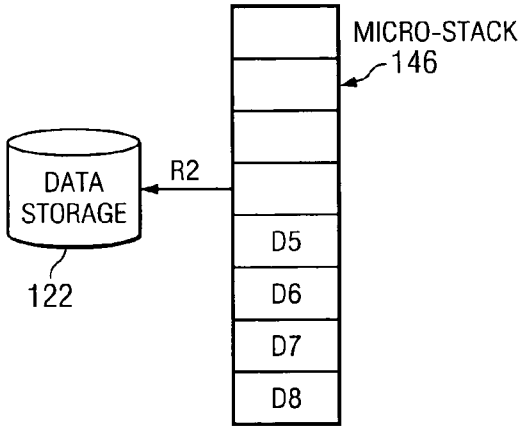


FIG. 4F

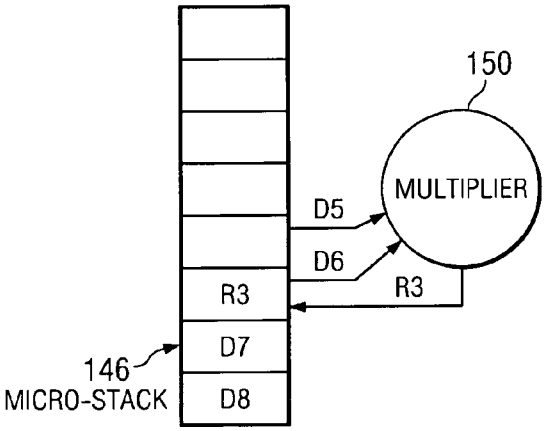


FIG. 4G

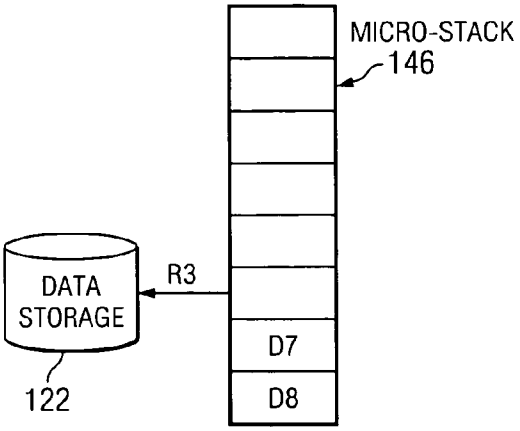
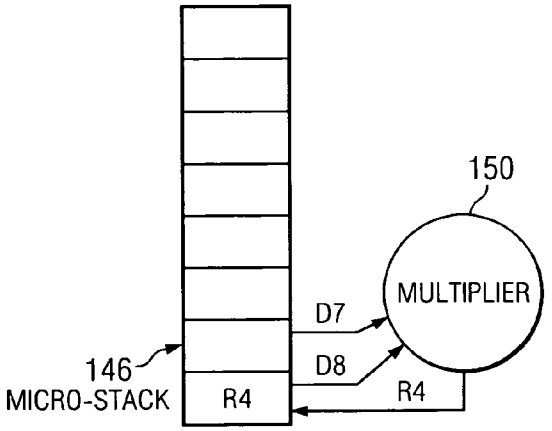


FIG. 4H



OPTIMIZING DATA MANIPULATION IN MEDIA PROCESSING APPLICATIONS

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to European Patent Application No. 04291918.3, filed on Jul. 27, 2004 and incorporated herein by reference.

BACKGROUND

[0002] Many types of electronic devices are battery operated and thus preferably consume as little power as possible. An example is a cellular telephone. Further, it may be desirable to implement various types of multimedia functionality in an electronic device such as a cell phone. Examples of multimedia functionality may include, without limitation, games, audio decoders, digital cameras, etc. It is thus desirable to implement such functionality in an electronic device in a way that, all else being equal, is fast, consumes as little power as possible and is as efficient as possible. Improvements in this area are desirable.

BRIEF SUMMARY

[0003] Described herein is a mechanism for synchronizing multiple processor stacks and a technique for improving processor efficiency using at least one of the stacks. One illustrative embodiment may comprise a system comprising a processor containing a first stack internal to a core of the processor, at least some data values in the first stack corresponding to values in a second stack external to the core. The system also comprises a memory coupled to the processor. In an iterative process, the processor pops a data value off of the first stack and begins to store the data value to the memory while the processor begins to use an existing data value from the first stack to produce a new data value to be stored on the first stack.

[0004] Another illustrative embodiment comprises a processor including a data stack located in the processor's core and comprising a plurality of data values, at least some of the data values corresponding to values in a main stack located outside the processor's core. The processor also includes a storage unit coupled to the data stack. In an iterative process, the processor pops a first data value off of the data stack and begins to store the first data value to the storage unit while the processor begins to use a second data value to produce a result to be stored on the data stack.

[0005] Yet another illustrative embodiment comprises a iterative process that includes popping a first data value off of a data stack internal to a processor's core, at least some data values in the data stack corresponding to values in a main stack external to the processor's core. The iterative process also comprises, while beginning to store the first data value in a memory, popping a second data value off of the data stack and using the second data value to produce a result to be stored on the data stack.

NOTATION AND NOMENCLATURE

[0006] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, various companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to . . . . " Also, the term "couple" or "couples" is intended to mean either an indirect or direct connection. Thus, if a first device couples to a second device, that connection may be through a direct connection, or through an indirect connection via other devices and connections. The term "system" is used to refer to a collection of components. For example, a system may comprise a processor and memory and other components. A system also may comprise a collection of components internal to a single processor and, as such, a processor may be referred to as a system.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] For a more detailed description of the preferred embodiments of the present invention, reference will now be made to the accompanying drawings, wherein:

[0008] **FIG. 1** shows a diagram of a system in accordance with preferred embodiments of the invention and including a Java Stack Machine ("JSM") and a Main Processor Unit ("MPU");

[0009] **FIG. 2** shows a block diagram of the JSM of **FIG. 1** in accordance with preferred embodiments of the invention;

[0010] **FIG. 3** shows various registers used in the JSM of **FIGS. 1 and 2**, in accordance with embodiments of the invention;

[0011] **FIGS. 4A-4H** show the operation of instructions that pop data off of the micro-stack shown in **FIG. 2**, manipulate the data, push results onto the data stack, and store results in a memory, in accordance with preferred embodiments of the invention; and

[0012] **FIG. 5** depicts an exemplary embodiment of the system described herein, in accordance with preferred embodiments of the invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0013] The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims, unless otherwise specified. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

[0014] The subject matter disclosed herein is directed to a programmable electronic device such as a processor. The processor described herein is particularly suited for executing Java™ bytecodes or comparable, code. As is well known, Java is particularly suited for embedded applications. Java is a stack-based language, meaning that a processor stack is heavily used when executing various instructions (e.g., Bytecodes), which instructions generally have a

size of 8 bits. Java is a relatively "dense" language meaning that on average each instruction may perform a large number of functions compared to various other instructions. The dense nature of Java is of particular benefit for portable, battery-operated devices that preferably include as little memory as possible to save space and power. The reason, however, for executing Java code is not material to this disclosure or the claims that follow. The processor described herein may be used in a wide variety of electronic systems. By way of example and without limitation, the Java-executing processor described herein may be used in a portable, battery-operated cell phone. Further, the processor advantageously includes one or more features that reduce the amount of power consumed by the Java-executing processor.

[0015] Referring now to **FIG. 1**, a system **100** is shown in accordance with a preferred embodiment of the invention. As shown, the system includes at least two processors **102** and **104**. Processor **102** is referred to for purposes of this disclosure as a Java Stack Machine ("JSM") and processor **104** may be referred to as a Main Processor Unit ("MPU"). System **100** may also include an external memory **106** coupled to both the JSM **102** and MPU **104** and thus accessible by both processors. The external memory **106** may exist on a separate chip than the JSM **102** and the MPU **104**. At least a portion of the external memory **106** may be shared by both processors meaning that both processors may access the same shared memory locations. Further, if desired, a portion of the external memory **106** may be designated as private to one processor or the other. System **100** also includes a Java Virtual Machine ("JVM") **108**, compiler **110**, and a display **114**. The JSM **102** preferably includes an interface to one or more input/output ("I/O") devices such as a keypad to permit a user to control various aspects of the system **100**. In addition, data streams may be received from the I/O space into the JSM **102** to be processed by the JSM **102**. Other components (not specifically shown) may include, without limitation, a battery and an analog transceiver to permit wireless communications with other devices. As noted above, while system **100** may be representative of, or adapted to, a wide variety of electronic systems, an exemplary electronic system may comprise a battery-operated, mobile cell phone.

[0016] As is generally well known, Java code comprises a plurality of "Bytecodes"**112**. Bytecodes **112** may be provided to the JVM **108**, compiled by compiler **110** and provided to the JSM **102** and/or MPU **104** for execution therein. In accordance with a preferred embodiment of the invention, the JSM **102** may execute at least some, and generally most, of the Java bytecodes. When appropriate, however, the JSM **102** may request the MPU **104** to execute one or more Java bytecodes not executed or executable by the JSM **102**. In addition to executing Java bytecodes, the MPU **104** also may execute non-Java instructions. The MPU **104** also hosts an operating system ("O/S") (not specifically shown), which performs various functions including system memory management, the system task management that schedules the JVM **108** and most or all other native tasks running on the system, management of the display **114**, receiving input from input devices, etc. Without limitation, Java code may be used to perform any one of a variety of applications including multimedia, games or web based applications in the system **100**, while non-Java code, which may comprise the O/S and other native applications, may still run on the system on the MPU **104**.

[0017] The JVM **108** generally comprises a combination of software and hardware. The software may include the compiler **110** and the hardware may include the JSM **102**. In accordance with preferred embodiments of the invention, the JSM **102** may execute at least two instruction sets. One instruction set may comprise standard Java bytecodes. As is well-known, Java bytecode is a stack-based intermediate language in which instructions generally target a stack. For example, an integer add ("IADD") Java instruction pops two integers off the top of the stack, adds them together, and pushes the sum back on the stack. As will be explained in more detail below, the JSM **102** comprises a stack-based architecture with various features that accelerate the execution of stack-based Java code, where the stack may include multiple portions that exist in different physical locations.

[0018] Another instruction set executed by the JSM **102** may include instructions other than standard Java instructions. In accordance with at least some embodiments of the invention, other instruction sets may include register-based and memory-based operations to be performed. This other instruction set generally complements the Java instruction set and, accordingly, may be referred to as a complementary instruction set architecture ("C-ISA"). By complementary, it is meant that the execution of more complex Java bytecodes may be substituted by a "micro-sequence" comprising one or more C-ISA instructions that permit address calculation to readily "walk through" the JVM data structures. A micro-sequence also may include one or more bytecode instructions. The execution of Java may be made more efficient and run faster by replacing some sequences of bytecodes by preferably shorter and more efficient sequences of C-ISA instructions. The two sets of instructions may be used in a complementary fashion to obtain satisfactory code density and efficiency. As such, the JSM **102** generally comprises a stack-based architecture for efficient and accelerated execution of Java bytecodes combined with a register-based architecture for executing register and memory based C-ISA instructions. Both architectures preferably are tightly combined and integrated through the C-ISA. Because various data structures described herein are generally JVM-dependent and thus may change from one JVM implementation to another, the software flexibility of the micro-sequence provides a mechanism for various JVM optimizations now known or later developed.

[0019] **FIG. 2** shows an exemplary block diagram of the JSM **102**. As shown, the JSM includes a core **120** coupled to a data storage **122** and an instruction storage **130**. Storage **122** and **130** are preferably integrated, along with core **120**, on the same JSM chip. Integrating storage **122** and **130** on the same chip as the core **120** may reduce data transfer time from storage **122** and **130** to the core **120**. The core **120** may include one or more components as shown. Such components preferably include a plurality of registers **140**, several (e.g., three) address generation units ("AGUs") **142**, **147**, micro-translation lookaside buffers (micro-TLBs) **144**, **156**, a multi-entry micro-stack **146**, an arithmetic logic unit ("ALU") **148**, a multiplier **150**, decode logic **152**, and instruction fetch logic **154**. In general, operands may be retrieved from a main stack and processed by the ALU **148**, where the main stack may include multiple portions that exist in different physical locations. For example, the main

stack may reside in external memory **106** and/or data storage **122**. Selected entries from the main stack may exist on the micro-stack **146**. In this manner, selected entries on the micro-stack **146** may represent the most current version of the operands in the system **100**. Accordingly, operands in external memory **106** and data storage **122** may not be coherent with the versions contained on the micro-stack **146**. A plurality of flags **158** are associated with the micro-stack **146**. Each micro-stack entry preferably has an associated flag **158**. Each flag **158** indicates whether the data in the associated micro-stack entry is valid and whether the data has been modified. Also, stack coherency operations may be performed by examining the flags **158** and updating the main stack with valid operands from the micro-stack **146** as will be explained below.

[**0020**] The micro-stack **146** preferably comprises, at most, the top n entries of the main stack that is implemented in data storage **122** and/or external memory **106**. The micro-stack **146** preferably comprises a plurality of gates in the core **120** of the JSM **102**. By implementing the micro-stack **146** in gates (e.g., registers) in the core **120** of the JSM **102**, access to the data contained on the micro-stack **146** is generally quite fast. Therefore, data access time may be reduced by providing data from the micro-stack **146** instead of the main stack. General stack requests are provided by the micro-stack **146** unless the micro-stack **146** cannot fulfill the stack requests. For example, when the micro-stack **146** is in an overflow condition or when the micro-stack **146** is in an underflow condition (as will be described below), general stack requests may be fulfilled by the main stack. By analyzing trends of the main stack, the value of n, which represents the size of the micro-stack **146**, may be optimized such that a majority of general stack requests are fulfilled by the micro-stack **146**, and therefore may provide requested data in fewer cycles. As a result, power consumption of the system **102** may be reduced. Although the value of n may vary in different embodiments, in accordance with at least some embodiments, the value of n may be the top eight entries in the main stack. In this manner, about 98% of the general stack accesses may be provided by the micro-stack **146**, and the number of accesses to the main stack may be reduced. As will be seen below, the main stack may not always be coherent with the micro-stack and, there may be a need, at times, to synchronize the main stack to the micro-stack.

[**0021**] Instructions may be fetched from instruction storage **130** by fetch logic **154** and decoded by decode logic **152**. The address generation unit **142** may be used to calculate addresses based, at least in part on data contained in the registers **140**. The AGUs **142** may calculate addresses for C-ISA instructions. The AGUs **142** may support parallel data accesses for C-ISA instructions that perform array or other types of processing. AGU **147** couples to the micro-stack **146** and may manage overflow and underflow conditions on the micro-stack **146** preferably in parallel. The micro-TLBs **144**, **156** generally perform the function of a cache for the address translation and memory protection information bits that are preferably under the control of the operating system running on the MPU **104**.

[**0022**] Referring now to **FIG. 3**, the registers **140** may include 16 registers designated as R**0**-R**15**. Registers R**0**-R**5** and R**8**-R**14** may be used as general purpose ("GP") registers usable for any purpose by the programmer. Other registers, and some of the GP registers, may be used for specific functions. For example, in addition to use as a GP register, register R**5** may be used to store the base address of a portion of memory in which Java local variables may be stored when used by the current Java method. The top of the micro-stack **146** is reflected in registers R**6** and R**7**. The top of the micro-stack **146** has a matching address in external memory pointed to by register R**6**. The operands contained on the micro-stack **146** are the latest updated values, while their corresponding values in external memory may or may not be up to date. Register R**7** provides the data value stored at the top of the micro-stack **146**. Register R**15** may be used for status and control of the JSM **102**. As an example, one status/control bit (called the "Micro-Sequence-Active" bit) may indicate if the JSM **102** is executing a "simple" instruction or a "complex" instruction through a micro-sequence as explained above. This bit controls in particular, which program counter is used (PC or micro-PC) to fetch the next instruction, as will be explained below.

[**0023**] Referring again to **FIG. 2**, the ALU **148** adds, subtracts, and shifts data. The multiplier **150** may be used to multiply two values together in one or more cycles. The instruction fetch logic **154** generally fetches instructions from instruction storage **130**. The instructions may be decoded by decode logic **152**. Because the JSM **102** is adapted to process instructions from at least two instruction sets, the decode logic **152** generally comprises at least two modes of operation, one mode for each instruction set. As such, the decode logic unit **152** may include a Java mode in which Java instructions may be decoded and a C-ISA mode in which C-ISA instructions may be decoded.

[**0024**] The data storage **122** generally comprises data cache ("D-cache") **124** and data random access memory ("D-RAM") **126**. Reference may be made to U.S. Pat. No. 6,826,652, filed Jun. 9, 2000 and U.S. Pat. No. 6,792,508, filed Jun. 9, 2000 both of which are incorporated herein by reference. Reference also may be made to U.S. Ser. No. 09/932,794 (Publication No. 20020069332), filed Aug. 17, 2001 and incorporated herein by reference. The main stack, arrays and non-critical data may be stored in the D-cache **124**, while Java local variables, critical data and non-Java variables (e.g., C, C++) may be stored in D-RAM **126**. The instruction storage **130** may comprise instruction RAM ("I-RAM") **132** and instruction cache ("I-cache") **134**. The I-RAM **132** may be used for "complex" micro-sequenced bytecodes or micro-sequences or predetermined sequences of code, as will be described below. The I-cache **134** may be used to store other types of Java bytecode and mixed Java/C-ISA instructions.

[**0025**] As noted above, the C-ISA instructions generally complement the standard Java bytecodes. For example, the compiler **110** may scan a series of Java bytes codes **112** and replace one or more of such bytecodes with an optimized code segment mixing C-ISA and bytecodes and which is capable of more efficiently performing the function(s) performed by the initial group of Java bytecodes. In at least this way, Java execution may be accelerated by the JSM **102**.

[**0026**] The micro-stack mechanism described herein may be implemented in any of a variety of systems to optimize performance. For example, the micro-stack mechanism may be used in conjunction with media processing software (and other similar, "high-performance" software, such as video

compression software, video decoding software, audio software, sound rate conversion software) to optimize JSM **102** performance over that of processors that do not use the micro-stack mechanism.

[**0027**] Execution of media processing software and/or other such high-performance software causes the JSM **102** to use the micro-stack mechanism to manipulate streams of data as dictated by instructions (e.g., Bytecodes) in the software. In a preferred embodiment, the JSM **102** loads data into the micro-stack **146**, manipulates the data in micro-stack **146**, and subsequently stores the data to data storage **122** as described below in context of **FIGS. 4A-4H**. More specifically, as shown in **FIG. 4A**, operands D1-D8 are stored in the micro-stack **146**. The micro-stack **146** preferably comprises eight entries, although the scope of disclosure is not limited as such. The operands D1-D8 are pushed onto the micro-stack **146** by, for instance, a Bytecode from Bytecodes **112** that may be part of a media processing software program or other similar program. The operands D1-D8 may be obtained from any storage device in the JSM **102**, such as the I-cache **134** or the registers **140**.

[**0028**] As shown in **FIG. 4B**, the operands D1, D2 stored in the micro-stack **146** are manipulated as directed by Bytecodes from Bytecodes **112**. For example, the Bytecodes may cause the ALU **148** and/or the multiplier **150** to perform mathematical operations on the operands D1 and D2, such that the operands D1 and D2 are popped off of the micro-stack **146** and manipulated to form a result. The result then is pushed onto the micro-stack **146**. Although the scope of disclosure is not limited to performing any particular type of mathematical operation on any particular number of operands stored in the micro-stack **146**, in some embodiments, the operands D1, D2 may be multiplied together by the multiplier **150** to produce a product R1. The product R1 then is pushed onto the micro-stack **146**. Operands also may be added, subtracted, divided, etc. As shown in **FIG. 4B**, the operands D1, D2 are no longer present in the micro-stack **146** and have been replaced by the product R1.

[**0029**] A subsequent Bytecode then causes the product R1 to be popped off of the micro-stack **146** and stored into data storage **122**. Thus, as shown in **FIG. 4C**, the product R1 is no longer present in the micro-stack **146**, and the operand D3 now is the top entry in the micro-stack **146**. As shown in **FIG. 4D** and as with operands D1 and D2 above, the operands D3, D4 are popped off of the micro-stack **146** and multiplied by multiplier **150** to form a product R2, which product R2 is pushed onto the micro-stack **146**. In a preferred embodiment, the product R1 is popped off the micro-stack **146** and stored into data storage **122** at or about the same time (i.e., in parallel) the operands D3, D4 are popped off the micro-stack **146** and multiplied to form product R2, and at which R2 is pushed onto the micro-stack **146**. Such parallel (i.e., simultaneous) execution is possible due to the inherent capability of the pipeline of the JSM **102** (i.e., the fetch logic **154**, decode logic **152**, execution logic **148**, **150**, etc.) to complete a command that is not in a final stage of the pipeline at about the same time as a command that is in the final stage of the pipeline, provided the JSM **102** has access to any information needed to process the command that is not in the final stage of the pipeline. Parallel execution is desirable because a greater number of processor operations are completed within a finite amount of time, thus increasing processor efficiency.

[**0030**] **FIGS. 4E and 4F** show simultaneous operations similar to the simultaneous operations described above. More specifically, as shown in **FIG. 4E**, the product R2 is popped off the micro-stack **146** and stored to data storage **122**. Thus, the top entry in the micro-stack **146** is the operand D5. As shown in **FIG. 4F**, at substantially the same time that R2 is popped off the micro-stack **146** and stored to data storage **122**, the operands D5 and D6 are popped off the micro-stack **146** and multiplied to form a product R3, which product R3 is pushed onto the micro-stack **146**. In some embodiments, at least a portion of the process reflected in **FIG. 4C** occurs simultaneously with at least a portion of the process reflected in **FIG. 4D**. Similarly, at least a portion of the process reflected in **FIG. 4E** occurs at the same time as a portion of the process reflected in **FIG. 4F**. The simultaneous storage and multiplication processes continue in this manner, as shown in **FIGS. 4G and 4H**, until the micro-stack **146** no longer contains any operands. In embodiments where the micro-stack **146** comprises more than eight operands (i.e., the micro-stack **146** comprises more than eight entries), the simultaneous storage and multiplication processes may continue as described above until the micro-stack **146** is empty. Because the performance penalty to access the micro-stack **146** is low compared to the performance penalty to access memory, executing media processing software (or other such high-performance software) in conjunction with the micro-stack mechanism, as described above, improves efficiency of the JSM **102** over processors that do not comprise the micro-stack mechanism.

[**0031**] As noted previously, system **100** may be implemented as a battery-operated mobile (i.e., wireless) communication device (e.g., a mobile phone) **415** such as that shown in **FIG. 5**. As shown, a mobile communication device includes an integrated keypad **412** and display **414**. The JSM **102** and MPU **104** and other components may be included in electronics package **410** connected to the keypad **412**, display **414**, and radio frequency ("RF") circuitry **416**. The RF circuitry **416** may be connected to an antenna **418**.

[**0032**] While the preferred embodiments of the present invention have been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit and teachings of the invention. The embodiments described herein are exemplary only, and are not intended to be limiting. Many variations and modifications of the invention disclosed herein are possible and are within the scope of the invention. Accordingly, the scope of protection is not limited by the description set out above. Each and every claim is incorporated into the specification as an embodiment of the present invention.

What is claimed is:

1. A system, comprising:

a processor containing a first stack internal to a core of the processor, at least some data values in the first stack corresponding to values in a second stack external to said core; and

a memory coupled to the processor;

wherein, in an iterative process, the processor pops a data value off of the first stack and begins to store the data value to said memory while the processor begins to use an existing data value from the first stack to produce a new data value to be stored on said first stack.

2. The system of claim 1, wherein the processor manipulates the existing data value to produce the new data value.

3. The system of claim 2, wherein the processor manipulates the existing data value by performing a mathematical operation.

4. The system of claim 1, wherein the processor pops the existing data value off of the first stack, manipulates said existing data value to produce the new data value, and pushes said new data value onto the first stack.

5. The system of claim 1, wherein the first stack is capable of storing a number of values, said number of values selected from the group consisting of 4 values, 8 values, 16 values, 32 values and 64 values.

6. The system of claim 1, wherein the first stack stores at least 4 values.

7. The system of claim 1, wherein the system comprises at least one of a battery-operated device or a wireless communication device.

8. The system of claim 1, wherein said data values are retrieved from a different memory before the data values are stored on the first stack.

9. A processor, comprising:

a data stack located in the processor's core and comprising a plurality of data values, at least some of said data values corresponding to values in a main stack located outside the processor's core; and

a storage unit coupled to the data stack;

wherein, in an iterative process, the processor pops a first data value off of the data stack and begins to store said first data value to the storage unit while the processor begins to use a second data value to produce a result to be stored on said data stack.

10. The processor of claim 9, wherein the processor uses the second data value to produce the result by manipulating the second data value.

11. The processor of claim 10, wherein the processor manipulates said second data value by performing a mathematical operation selected from the group consisting of addition, multiplication, division and subtraction.

12. The processor of claim 9, wherein the first data value is obtained from a memory prior to being stored on the data stack.

13. The processor of claim 9, wherein the main stack is capable of storing more values than the data stack.

14. The processor of claim 9, wherein the data stack is capable of storing a number of values, said number selected from the group consisting of 4 values, 8 values, 16 values, 32 values, 64 values.

15. The processor of claim 9, wherein the data stack stores at least 4 values.

16. An iterative process, comprising:

popping a first data value off of a data stack internal to a processor's core, at least some data values in said data stack corresponding to values in a main stack external to the processor's core; and

while beginning to store said first data value in a memory, popping a second data value off of the data stack and using said second data value to produce a result to be stored on the data stack.

17. The iterative process of claim 16, wherein using said second data value to produce the result comprises manipulating said second data value.

18. The iterative process of claim 17, wherein manipulating comprises performing one of a multiplication, addition, subtraction or division operation.

19. The iterative process of claim 16, wherein popping the first data value off of the data stack comprises using a data stack capable of storing a number of values, said number of values selected from the group consisting of 4 values, 8 values, 16 values, 32 values, and 64 values.

20. The iterative process of claim 16 further comprising obtaining the first data value from a storage unit and pushing the first data value onto the data stack.

21. The iterative process of claim 16, wherein popping the first data value off of the data stack comprises using a data stack having less storage space than the main stack.

* * * * *