

(51) International Patent Classification:
G06F 12/00 (2006.01)

(21) International Application Number:

PCT/US2012/042274

(22) International Filing Date:

13 June 2012 (13.06.2012)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

61/496,076	13 June 2011 (13.06.2011)	US
61/496,084	13 June 2011 (13.06.2011)	US
61/496,075	13 June 2011 (13.06.2011)	US
61/496,081	13 June 2011 (13.06.2011)	US
61/496,088	13 June 2011 (13.06.2011)	US
61/496,079	13 June 2011 (13.06.2011)	US
61/496,080	13 June 2011 (13.06.2011)	US
61/496,074	13 June 2011 (13.06.2011)	US
61/496,073	13 June 2011 (13.06.2011)	US

(74) Agents: POWSNER, David J. et al.; Nutter Mcclennen & Fish LLP, Seaport West, 155 Seaport Boulevard, Boston, MA 02210-2604 (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, 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, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, 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, 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, ML, MR, NE, SN, TD, TG).

(71) Applicant (for all designated States except US): PAN-EVE, LLC [US/US]; 100 Venture Way, Floor 3, Hadley, MA 01035 (US).

(72) Inventors; and

(75) Inventors/Applicants (for US only): FRANK, Steven, J. [US/US]; 680 North Farms Road, Florence, MA 01062 (US). LIN, Hai, China [CN/US]; 72 West Street, Apt. 1, Northampton, MA 01060 (US).

Published:

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

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

(54) Title: GENERAL PURPOSE DIGITAL DATA PROCESSOR, SYSTEMS AND METHODS

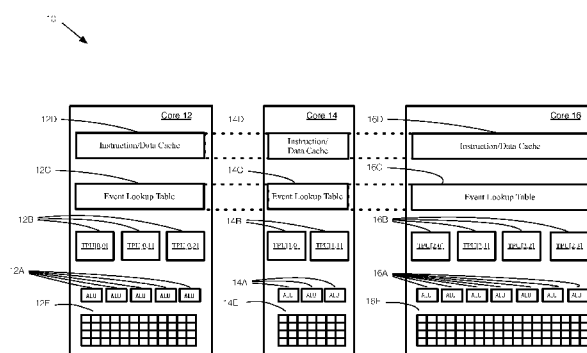


Figure 1

(57) Abstract: The invention provides improved data processing apparatus, systems and methods that include one or more nodes, e.g., processor modules or otherwise, that include or are otherwise coupled to cache, physical or other memory (e.g., attached flash drives or other mounted storage devices) collectively, "system memory." At least one of the nodes includes a cache memory system that stores data (and/or instructions) recently accessed (and/or expected to be accessed) by the respective node, along with tags specifying addresses and statuses (e.g., modified, reference count, etc.) for the respective data (and/or instructions). The tags facilitate translating system addresses to physical addresses, e.g., for purposes of moving data (and/or instructions) between system memory (and, specifically, for example, physical memory-such as attached drives or other mounted storage) and the cache memory system.

GENERAL PURPOSE DIGITAL DATA PROCESSOR, SYSTEMS AND METHODS**REFERENCE TO RELATED APPLICATIONS**

This application claims the benefit of filing of all of the following applications, the teachings of all of which are incorporated herein by reference:

- General Purpose Embedded Processor and Digital Data Processing System Executing a Pipeline of Software Components that Replace a Like Pipeline of Hardware Components, Application #61/496,080, Filed June 13, 2011 - Atty Docket 109451-20
- General Purpose Embedded Processor with Provision of Quality of Service Through Thread Installation, Maintenance and Optimization, Application No. 61/496,088, Filed June 13, 2011 - Atty Docket 109451-21
- General Purpose Embedded Processor with Location-Independent Shared Execution Environment, Application No. 61/496,084, Filed June 13, 2011 - Atty Docket 109451-22
- General Purpose Embedded Processor with Dynamic Assignment of Events to Threads, Application No. 61/496,081, Filed June 13, 2011 - Atty Docket 109451-23
- Digital Data Processor with JPEG2000 BIT Plane Stripe Column Encoding, Application No. 61/496,079, Filed June 13, 2011 - Atty Docket 109451-24
- Digital Data Processor with JPEG2000 Binary Arithmetic Coder Lookup, Application No. 61/496,076, Filed June 13, 2011 - Atty Docket 109451-25
- Digital Data Processor with Cache-Managed System Memory, Application No. 61/496,075, Filed June 13, 2011 - Atty Docket 109451-26
- Digital Data Processor With Cache Control Instruction Set and Cache-Initiated Optimization, Application No. 61/496,074, Filed June 13, 2011 - Atty Docket 109451-27

- Digital Data Processor with Arithmetic Operation Transpose Parameter, Application No. 61/496,073, Filed June 13, 2011 - Atty Docket 109451-28

BACKGROUND OF THE INVENTION

The invention pertains to digital data processing and, more particularly, to digital data processing modules, systems and methods with improved software execution. The invention has application, by way of example, to embedded processor architectures and operation. The invention has application in high-definition digital television, game systems, digital video recorders, video and/or audio players, personal digital assistants, personal knowledge navigators, mobile phones, and other multimedia and non-multimedia devices. It also has application in desktop, laptop, mini computer, mainframe computer and other computing devices.

Prior art embedded processor-based or application systems typically combine: (1) one or more general purpose processors, e.g., of the ARM, MIPS or x86 variety, for handling user interface processing, high level application processing, and operating system tasks, with (2) one or more digital signal processors (DSPs), including media processors, dedicated to handling specific types of arithmetic computations at specific interfaces or within specific applications, on real-time/low latency bases. Instead of, or in addition to, the DSPs, special-purpose hardware is often provided to handle dedicated needs that a DSP is unable to handle on a programmable basis, e.g., because the DSP cannot handle multiple activities at once or because the DSP cannot meet needs for a very specialized computational element.

The prior art also includes personal computers, workstations, laptop computers and other such computing devices which typically combine a main processor with a separate graphics processor and a separate sound processor; game systems, which typically combine a main processor and separately programmed graphics processor; digital video recorders, which typically combine a general purpose processor, mpeg2 decoder and encoder chips, and special-purpose digital signal processors; digital televisions, which typically combine a general purpose processor, mpeg2 decoder and encoder chips, and special-purpose DSPs or media processors; mobile phones, which typically combine a processor for user interface and applications processing and special-purpose DSPs for mobile phone GSM, CDMA or other protocol processing.

Earlier prior art patents include United States Patent 6,408,381, disclosing a pipeline processor utilizing snapshot files with entries indicating the state of instructions in the various pipeline stages, and United States Patent 6,219,780, which concerns improving the throughput of computers with multiple execution units grouped in clusters. One problem with the earlier prior art approaches was hardware design complexity, combined with software complexity in programming and interfacing heterogeneous types of computing elements. Another problem was that both hardware and software must be re-engineered for every application. Moreover, early prior art systems do not load balance: capacity cannot be transferred from one hardware element to another.

Among other trends, the world is going video—that is, the consumer, commercial, educational, governmental and other markets are increasingly demanding video creation and/or playback to meet user needs. Video and image processing is, thus, one dominant usage for embedded devices and is pervasive in devices, throughout the consumer and business devices, among others. However, many of the processors still in use today rely on decades-old Intel and ARM architectures that were optimized for text processing in eras gone by.

An object of this invention is to provide improved modules, systems and methods for digital data processing.

A further object of the invention is to provide such modules, systems and methods with improved software execution.

A related object is to provide such modules, systems and methods as are suitable for an embedded environment or application.

A further related object is to provide such modules, systems and methods as are suitable for video and image processing.

Another related object is to provide such modules, systems and methods as facilitate design, manufacture, time-to-market, cost and/or maintenance.

A further object of the invention is to provide improved modules, systems and methods for embedded (or other) processing that meet the computational, size, power and cost requirements of today's and future appliances, including by way of non-limiting example, digital televisions, digital video recorders, video and/or audio players, personal digital assistants, personal knowledge navigators, and mobile phones, to name but a few.

Yet another object is to provide improved modules, systems and methods that support a range of applications.

Still yet another object is to provide such modules, systems and methods which are low-cost, low-power and/or support robust rapid-to-market implementations.

Yet still another object is to provide such modules, systems and methods which are suitable for use with desktop, laptop, mini computer, mainframe computer and other computing devices.

These and other aspects of the invention are evident in the discussion that follows and in the drawings.

SUMMARY OF THE INVENTION*Digital Data Processor with Cache-Managed Memory*

The foregoing are among the objects attained by the invention which provides, in some aspects, an improved digital data processing system with cache-controlled system memory. A system according to one such aspect of the invention includes one or more nodes, e.g., processor modules or otherwise, that include or are otherwise coupled to cache, physical or other memory (e.g., attached flash drives or other mounted storage devices)—collectively, “system memory.”

At least one of the nodes includes a cache memory system that stores data (and/or instructions) recently accessed (and/or expected to be accessed) by the respective node, along with tags specifying addresses and statuses (e.g., modified, reference count, etc.) for the respective data (and/or instructions). The caches may be organized in multiple hierarchical levels (e.g., a level 1 cache, a level 2 cache, and so forth), and the addresses may form part of a "system" address that is common to multiple ones of the nodes.

The system memory and/or the cache memory may include additional (or "extension") tags. In addition to specifying system addresses and statuses for respective data (and/or instructions), the extension tags specify physical address of those data in system memory. As such, they facilitate translating system addresses to physical addresses, e.g., for purposes of moving data (and/or instructions) between system memory (and, specifically, for example, physical memory—such as attached drives or other mounted storage) and the cache memory system.

Related aspects of the invention provide a system, e.g., as described above, in which one extension tag is provided for each addressable datum (or data block or page, as the case may be) in system memory.

Further aspects of the invention provide a system, e.g., as described above, in which the extension tags are organized as a tree in system memory.

Related aspects of the invention provide such a system in which one or more of the extension tags are cached in the cache memory system of one or more nodes. These may include, for example, extension tags for data recently accessed (or expected to be accessed) by those nodes following cache “misses” for that data within their respective cache memory systems.

Further related aspects of the invention provide such a system that comprises a plurality of nodes that are coupled for communications with one another as well, preferably, as with the memory system, e.g., by a bus, network or other media. In related aspects, this comprises a ring interconnect.

A node, according to still further aspects of the invention, can signal a request for a datum along that bus, network or other media following a cache miss within its own internal cache memory system for that datum. System memory can satisfy that request, or a subsequent related request for the datum, if none of the other nodes do so.

In related aspects of the invention, a node can utilize the bus, network or other media to communicate to other nodes and/or the memory system updates to cached data and/or extension tags.

Further aspects of the invention provide a system, e.g., as described above, in which one or more nodes, includes a first level of cache that contains frequently and/or recently used data and/or instructions, and at least a second level of cache that contains a superset of data and/or instructions in the first level of cache.

Other aspects of the invention provide systems e.g., as described above, that utilize fewer or greater than the two levels of cache within the nodes. Thus, for example, the system nodes may include only a single level of cache, along with extension tags of the type described above.

Still further aspects of the invention provide systems, e.g., as described above, wherein the nodes comprise, for example, processor modules, memory modules, digital data processing systems (or interconnects thereto), and/or a combination thereof.

Yet still further aspects of the invention provide such systems where, for example, one or more levels of cache (e.g., the first and second levels) are contained, in whole or in part, on one or more of the nodes, e.g., processor modules.

Advantages of digital data modules, systems and methods according to the invention are that all system addresses are treated as if cached in the memory system. Accordingly an addressable item that is present in the system—regardless, for example, of whether it is in cache memory, physical memory (e.g., an attached flash drive or other mounted storage device)—has an entry in one of the levels of cache. An item that is not present in any cache (and the memory system), i.e., is not reflected in any of the cache levels, is then not present in the memory system. Thus the memory system can be filled sparsely in a way that is natural to software and operating system, without the overhead of tables on the processor.

Advantages of digital data modules, systems and methods according to the invention are that they afford efficient utilization of memory, esp., where that might be limited, e.g., on mobile and consumer devices.

Further advantages are that digital data modules, systems and methods experience performance improvements of all memory being managed as cache without on-chip area penalty. This in turn enables memory, e.g., of mobile and consumer devices, to be expanded by another networked device. It can also be used, by way of further non limiting example, to manage RAM and FLASH memory, e.g., on more recent portable devices such as net books.

General Purpose Processor With Dynamic Assignment of Events to Threads

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which a processing module comprises a plurality of processing units that each execute processes or threads (collectively, “threads”). An event table maps events—such as, by way of non-limiting example, hardware interrupts, software interrupts and memory events—to respective threads. Devices and/or software (e.g., applications, processes and/or threads) register, e.g., with a default system thread or otherwise, to identify event-processing services that they require and/or that they can provide. That thread or other mechanism continually matches those

and updates the event table to reflect a mapping of events to threads, based on the demands and capabilities of the overall environment.

Related aspects of the invention provide systems and methods incorporating a processor, e.g., as described above, in which code utilized by hardware devices or software to register their event-processing needs and/or capabilities is generated, for example, by a preprocessor based on directives supplied by a developer, manufacturer, distributor, retailer, post-sale support personnel, end user or otherwise about actual or expected runtime environments in which the processor is or will be used.

Further related aspects of the invention provide such a method in which such code can be inserted into the individual applications' respective runtime code by the preprocessor, etc.

General Purpose Processor With Location-Independent Shared Execution Environment

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, that permit application and operating system-level threads to be transparently executed across different devices (including mobile devices) and which enable such device to automatically off load work to improve performance and lower power consumption.

Related aspects of the invention provide such modules, systems and methods in which events detected by a processor executing on one device can be routed for processing to a processor, e.g., executing on another device.

Other related aspects of the invention provide such modules, systems and methods in which threads executing on one device can be migrated, e.g., to a processor on another device and, thereby, for example, to processor events local to that other device and/or to achieve load balancing, both way way of example. Thus, for example, threads can migrated, e.g., to less busy devices, to better suited devices or, simply, to a device where most of events are expected to occur.

Further aspects of the invention provide modules, systems and methods, e.g., as described above in which events are routed and/or threads are migrated between and among processors in multiple different devices and/or among multiple processors on a single device.

Yet still other aspects of the invention provide modules, systems and methods, e.g., as described above in which tables for routing events are implemented in novel memory/cache structures, e.g., such that the tables of cooperating processor modules (e.g., those on a local area network) comprise single shared hierarchical table.

General Purpose Processor With Provision of Quality of Service Through Thread Instantiation, Maintenance and Optimization

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which a processor comprises a plurality of processing units that each execute processes or threads (collectively, “threads”). An event delivery mechanism delivers events—such as, by way of non-limiting example, hardware interrupts, software interrupts and memory events—to respective threads. A preprocessor (or other functionality), e.g., executed by a designer, manufacturer, distributor, retailer, post-sale support personnel, end-user, or other responds to expected core and/or site resource availability, as well as to user prioritization, to generate default system thread code, link parameters, etc., that optimize thread instantiation, maintenance and thread assignment at runtime.

Related aspects of the invention provide modules, systems and methods executing threads, e.g., a default system thread, created as discussed above.

Still further related aspects of the invention provide modules, systems and methods executing threads that are compiled, linked, loaded and/or invoked in accord with the foregoing.

Yet still further related aspects of the invention provide modules, systems and methods, e.g., as described above, in which the default system thread or other functionality insures instantiation of an appropriate number of threads at an appropriate time, e.g., to meet quality of service requirements.

Further related aspects of the invention provide such a method in which such code can be inserted into the individual applications' respective source code by the preprocessor, etc.

General Purpose Processor with JPEG2000 Bit Plane Stripe Column Encoding

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, that include an arithmetic logic or other execution unit that is in communications coupling with one or more registers. That execution unit executes a selected processor-level instruction by encoding and storing to one (or more) of the register(s) a stripe column for bit plane coding within JPEG2000 EBCOT (Embedded Block Coding with Optimized Truncation).

Related aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the execution unit generates the encoded stripe column based on specified bits of a column to be encoded and on bits adjacent thereto.

Further related aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the execution unit generates the encoded stripe column from four bits of the column to be encoded and on the bits adjacent thereto.

Still further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the execution unit generates the encoded stripe column in response to execution of an instruction that specifies, in addition to the bits of the column to be encoded and adjacent thereto, a current coding state of at least one of the bits to be encoded.

Yet still further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the coding state of each bit to be encoded is represented in three bits.

Still further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the execution unit generates the encoded stripe column in response to execution of an instruction that specifies an encoding pass that includes any of a significance

propagation pass (SP), a magnitude refinement pass (MR), a cleanup pass, and a combined MR and CP pass.

Yet still further related aspects of the invention provides processor modules, systems and methods, e.g., as described above, in which the execution unit selectively generates and stores to one or more registers an updated coding state of at least one of the bits to be encoded.

General Purpose Processor with JPEG2000 Binary Arithmetic Code Lookup

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which an arithmetic logic or other execution unit that is in communications coupling with one or more registers executes a selected processor-level instruction by storing to that/those register(s) value(s) from a JPEG2000 binary arithmetic coder lookup table.

Related aspects of the invention provide processor modules, systems and methods as described above in which the JPEG2000 binary arithmetic coder lookup table is a Qe-value and probability estimation lookup table.

Related aspects of the invention provide processor modules, systems and methods as describe above in which the execution unit responds to such a selected processor-level instruction by storing to said one or more registers one or more function values from such a lookup table, where those functions are selected from a group Qe-value, NMPS, NLPS and SWITCH functions.

In further related aspects, the invention provides processor modules, systems and methods, e.g., as described above, in which the execution logic unit stores said one or more values to said one or more registers as part of a JPEG2000 decode or encode instruction sequence.

General Purpose Processor with Arithmetic Operation Transpose Parameter

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which an arithmetic logic or other execution unit that is in communications coupling with one or more registers executes a selected processor-level instruction specifying arithmetic operations *with transpose* by performing the specified arithmetic operations on one or

more specified operands, e.g., longwords, words or bytes, contained in respective ones of the registers to generate and store the result of that operation in transposed format, e.g., across multiple specified registers.

In related aspects, the invention provides processor modules, systems and methods, e.g., as described above, in which the arithmetic logic unit writes the result, for example, as a one-quarter word column of four adjacent registers or, by way of further example, a byte column of eight adjacent registers.

In further related aspects, the invention provides processor modules, systems and methods, e.g., as described above, in which the arithmetic logic unit breaks the result (e.g., longwords, words or bytes) into separate portions (e.g., words, bytes or bits) and puts them into separate registers, e.g., at a specific common byte, bit or other location in each of those registers.

In further related aspects, the invention provides processor modules, systems and methods, e.g., as described above, in which the selected arithmetic operation is an addition operation.

In further related aspects, the invention provides processor modules, systems and methods, e.g., as described above, in which the selected arithmetic operation is a subtraction operation.

General Purpose Processor with Cache Control Instruction Set and Cache-Initiated Optimization

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, with improved cache operation. A processor module according to such aspects, for example, can include an arithmetic logic or other execution unit that is in communications coupling with one or more registers, as well as with cache memory. Functionality associated with the cache memory works cooperatively with the execution unit to vary utilization of the cache memory in response to load, store and other requests that effect data and/or instruction exchanges between the registers and the cache memory.

Related aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the (aforesaid functionality associated with the) cache memory varies

replacement and modified block writeback selectively in response to memory reference instructions (a term that is used interchangeably herein, unless otherwise evident from context, with the term “memory reference instructions”) executed by the execution unit.

Further related aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the (aforesaid functionality associated with the) cache memory varies a value of a “reference count” that is associated with cached instructions and/or data selectively in response to such memory reference instructions.

Still further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the (aforesaid functionality associated with the) cache memory forces the reference count value to a lowest value in response to selected memory reference instructions, thereby, insuring that the corresponding cache entry will be a next one to be replaced.

Related aspects of the invention provide such processor modules, systems and methods in which such instructions include parameters (e.g., the “reuse/no-reuse cache hint”) for influencing the reference counts accordingly. These can include, by way of example, any of load, store, “fill” and “empty” instructions and, more particularly, by way of example, can include one or more of LOAD (Load Register), STORE (Store to Memory), LOADPAIR (Load Register Pair), STOREPAIR (Store Pair to Memory), PREFETCH (Prefetch Memory), LOADPRED (Load Predicate Register), STOREPRED (Store Predicate Register), EMPTY (Empty Memory), and FILL (Fill Memory) instructions.

Yet still further aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the (aforesaid functionality associated with the) cache memory works cooperatively with the execution unit to prevent large memory arrays that are not frequently accessed from removing other cache entries that are frequently used.

Other aspects of the invention provide processor modules, systems and methods with functionality that varies replacement and writeback of cached data/instructions and updates in accord with (a) the access rights of the acquiring cache, and (b) the nature of utilization of such

data by in other processor modules. This can be effected in connection memory access instruction execution parameters and/or via “automatic” operation of the caching subsystems (and/or cooperating mechanisms in the operating system).

Still yet further aspects of the invention provide processor modules, systems and methods, e.g., as described above, that include a novel virtual memory and memory system architecture features in which inter alia all memory is effectively managed as cache.

Other aspects of the invention provide processor modules, systems and methods, e.g., as described above, in which the (aforesaid functionality associated with the) cache memory works cooperatively with the execution unit to perform requested operations on behalf of an executing thread. On multiprocessor systems these operations can span to non-local level2 and level2 extended caches.

General Purpose Processor and Digital Data Processing System Executing a Pipeline of Software Components That Replace a Like Pipeline of Hardware Components

Further aspects of the invention provide processor modules, systems and methods, e.g., as described above, that execute pipelines of software components in lieu of like pipelines of hardware components of the type normally employed by prior art devices.

Thus, for example, a processor according to the invention can execute software components pipelined for video processing and including a H.264 decoder software module, a scalar and noise reduction software module, a color correction software module, a frame rate control software module—all in lieu of a like hardware pipeline, namely, one including a semiconductor chip that functions as a system controller with H.264 decoding, pipelined to a semiconductor chip that functions as a scaler and noise reduction module, pipelined to a semiconductor chip that functions for color correction, and further pipelined to a semiconductor chip that functions as a frame rate controller.

Related aspects of the invention provide such digital data processing systems and methods in which the processing modules execute the pipelined software components as separate respective threads.

Further related aspects of the invention provide digital data processing systems and methods, e.g., as described above, comprising a plurality of processing modules, each executing pipelines of software components in lieu of like hardware components.

Yet further related aspects of the invention provide digital data processing systems and methods, e.g., as described above, in which at least one of plural threads defining different respective components of a pipeline (e.g., for video processing) is executed on a different processing module than one or more threads defining those other respective components.

Still yet further related aspects of the invention provide digital data processing systems and methods, e.g., as described above, in which at least one of the processor modules includes an arithmetic logic or other execution unit and further includes a plurality of levels of cache, at least one of which stores some information on circuitry common to the execution unit (i.e., on chip) and which stores other information off circuitry common to the execution unit (i.e., off chip).

Yet still further aspects of the invention provide digital data processing systems and methods, e.g., as described above, in which plural ones of the processing modules include levels of cache as described above. The cache levels of those respective processors can, according, to related aspects of the invention, manage the storage and access of data and/or instructions common to the entire digital data processing system.

Advantages of processing modules, digital data processing systems, and methods according to the invention are, among others, that they enable a single processor to handle all application, image, signal and network processing—by way of example—of a mobile, consumer and/or other products, resulting in lower cost and power consumption. A further advantage is that they avoid the recurring complexity designing, manufacturing, assembling and testing hardware pipelines, as well as that of writing software for such hardware pipelined-devices.

These and other aspects of the invention are evident in the discussion that follows and in the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1 depicts a system including processor modules according to the invention;

Figure 2 depicts a system comprising two processor modules of the type shown in Figure 1;

Figure 3 depicts thread states and transitions in a system according to the invention;

Figure 4 depicts thread-instruction abstraction in a system according to the invention;

Figure 5 depicts event binding and processing in a processor module according to the invention;

Figure 6 depicts registers in a processor module of a system according to the invention;

Figures 7–10 depict add instructions in a processor module of a system according to the invention;

Figures 11–16 depict pack and unpack instructions in a processor module of a system according to the invention;

Figures 17–18 depict bit plane stripe instructions in a processor module of a system according to the invention;

Figure 19 depicts a memory address model in a system according to the invention;

Figure 20 depicts a cache memory hierarchy organization in a system according to the invention;

Figure 21 depicts overall flow of an L2 and L2E cache operation in a system according to the invention;

Figure 22 depicts organization of the L2 cache in a system according to the invention;

Figure 23 depicts the result of an L2E access hit in a system according to the invention;

Figure 24 depicts an L2E descriptor tree look-up in a system according to the invention;

Figure 25 depicts an L2E physical memory layout in a system according to the invention;

Figure 26 depicts a segment table entry format in a system according to the invention;

Figures 27–29 depict, respectively, L1, L2 and L2E Cache addressing and tag formats in an SEP system according to the invention;

Figure 30 depicts an IO address space format in a system according to the invention;

Figure 31 depicts a memory system implementation in a system according to the invention;

Figure 32 depicts a runtime environment provided by a system according to the invention for executing tiles;

Figure 33 depicts a further runtime environment provided by a system according to the invention;

Figure 34 depicts advantages of processor modules and systems according to the invention;

Figure 35 depicts typical implementation of a consumer (or other) device for video processing;

Figure 36 depicts implementation of the device of Figure 35 in a system according to the invention;

Figure 37 depicts use of a processor in accord with one practice of the invention for parallel execution of applications and other components of the runtime environment;

Figure 38 depicts a system according to the invention that permits dynamic assignment of events to threads;

Figure 39 depicts a system according to the invention that provides a location-independent shared execution environment;

Figure 40 depicts migration of threads in a system according to the invention with a location-independent shared execution environment and with dynamic assignment of events to threads;

Figure 41 is a key to symbols used in Figure 40;

Figure 42 depicts a system according to the invention that facilitates the permits of quality of service through thread instantiation, maintenance and optimization;

Figure 43 depicts a system according to the invention in which the functional units execute selected arithmetic operations concurrently with transposes;

Figure 44 depicts a system according to the invention in which the functional units execute processor-level instructions by storing to register(s) value(s) from a JPEG2000 binary arithmetic coder lookup table;

Figure 45 depicts a system according to the invention in which the functional units execute processor-level instructions by encoding a stripe column of values in registers for bit plane coding within JPEG2000 EBCOT;

Figure 46 depicts a system according to the invention wherein a pipeline of instructions executing on cores serve as software equivalents of corresponding hardware pipelines of the type traditionally practiced in the prior art; and

Figures 47 and 48 show the effect of memory access instructions with and without a no-reuse hint on caches in a system according to the invention.

DETAILED DESCRIPTION OF THE ILLUSTRATED EMBODIMENT

OVERVIEW

Figure 1 depicts a system 10 including processor modules (generally, referred to as "SEP" and/or as "cores" elsewhere herein) 12, 14, 16 according to one practice of the invention. Each of these is generally constructed, operated, and utilized in the manner of the "processor module" disclosed, e.g., as element 5, of Figure 1, and the accompanying text of United States Patents US 7,685,607 and US 7,653,912, entitled "General Purpose Embedded Processor" and "Virtual Processor Methods and Apparatus With Unified Event Notification and Consumer-Producer Memory Operations," respectively, and further details of which are disclosed in Figures 2–26 and the accompanying text of those two patents, the teachings of which figures and text are incorporated herein by reference, and a copy of US 7,685,607 of which is filed herewith by example as **Appendix A**, as adapted in accord with the teachings hereof.

Thus, for example, the illustrated cores 12–16 include functional units 12A–16A, respectively, that are generally constructed, operated, and utilized in the manner of the "execution units" (or "functional units") disclosed, by way of non-limiting example, as elements 30–38, of Figure 1 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, and further details of which are disclosed, by way of non-limiting example, in Figures 13, 16 (branch unit), 17 (memory unit), 20, 21–22 (integer and compare units), 23A–23B (floating point unit) and the accompanying text of those two patents, the teachings of which figures and text (and others of which pertain to the functional or execution units) are incorporated herein by reference, as adapted in accord with the teachings hereof. The functional units 12A–16A are labelled "ALU" for arithmetic logic unit in the drawing, although they may serve other functions instead or in addition (e.g., branching, memory, etc.).

By way of further example, cores 12–16 include thread processing units 12B–16B, respectively, that are generally constructed, operated, and utilized in the manner of the "thread processing units (TPUs)" disclosed, by way of non-limiting example, as elements 10–20, of Figure 1 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, and further details of which are disclosed, by way of non-limiting example, in Figures 3, 9, 10, 13 and the

accompanying text of those two patents, the teachings of which figures and text (and others of which pertain to the thread processing units or TPUs) are incorporated herein by reference, as adapted in accord with the teachings hereof.

Consistent with those teachings, the respective cores 12–16 may have one or more TPUs and the number of those TPUs per core may differ (here, for example, core 12 has three TPUs 12B; core 14, two TPUs 14B; and, core 16, four TPUs 16B). Moreover, although the drawing shows a system 10 with three cores 12–16, other embodiments may have a greater or lesser number of cores.

By way of still further example, cores 12–16 include respective event lookup tables 12C–16C, which are generally constructed, operated and utilized in the manner of the “event-to-thread lookup table” (also referred to as the “event table” or “thread lookup table,” or the like) disclosed, by way of non-limiting example, as element 42 in Figure 4 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to the “event-to-thread lookup table”) are incorporated herein by reference, as adapted in accord with the teachings hereof, e.g., to provide for matching events to threads executing within or across processor boundaries (i.e., on other processors).

The tables 12C–16C are shown as a single structure within each core of the drawing for sake of convenience; in practice, they may be shared in whole or in part, logically, functionally and/or physically, between and/or among the cores (as indicated by dashed lines)—and which, therefore, may be referred to herein as “virtual” event lookup tables, “virtual” event-to-thread lookup tables, and so forth. Moreover, those tables 12C–16C can be implemented as part of a single hierarchical table that is shared among cooperating processor modules within a “zone” of the type discussed below and that operates in the manner of the novel virtual memory and memory system architecture discussed here.

By way of yet still further example, cores 12–16 include respective caches 12D–16D, which are generally constructed, operated and utilized in the manner of the “instruction cache,” the “data cache,” the “Level1 (L1)” cache, the “Level2 (L2)” cache, and/or the “Level2 Extended (L2E)”

cache disclosed, by way of non-limiting example, as elements 22, 24, 26 (26a, 26b) respectively, in Figure 1 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, and further details of which are disclosed, by way of non-limiting example, in Figures 5, 6, 7, 8, 10, 11, 12, 13, 18, 19 and the accompanying text of those two patents, the teachings of which figures and text (and others of which pertain to the instruction, data and other caches) are incorporated herein by reference, as adapted in accord with the teachings hereof, e.g., to support a novel virtual memory and memory system architecture features in which inter alia all memory is effectively managed as cache, even though off-chip memory utilizes DDR DRAM or otherwise.

The caches 12D–16D are shown as a single structure within each core of the drawing for sake of convenience. In practice, one or more of those caches may constitute one or more structures within each respective core that are logically, functionally and/or physically separate from one another and/or, as indicated by the dashed lines connecting caches 12D–16D, that are shared in whole or in part, logically, functionally and/or physically, between and/or among the cores. (As a consequence, one or more of the caches are referred to elsewhere herein as “virtual” instruction and/or data caches.) For example, as shown in **Figure 2**, each core may have its own respective L1 data and L1 instruction caches, but may share L2 and L2 extended caches with other cores.

By way of still yet further example, cores 12–16 include respective registers 12E–16E that are generally constructed, operated and utilized in the manner of the general-purpose registers, predicate registers and control registers disclosed, by way of non-limiting example, in Figures 9 and 20 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to registers employed in the processor modules) are incorporated herein by reference, as adapted in accord with the teachings hereof.

Moreover, one or more of the illustrated cores 12–16 may include on-chip DRAM or other “system memory” (as elsewhere herein), instead of or in addition to being coupled to off-chip DRAM or other such system memory—as shown, by way of non-limiting example, in the embodiment of Figure 31 and discussed elsewhere herein. In addition, one or more of those

cores may be coupled to flash memory (which may be on-chip, but is more typically off-chip), again, for example, as shown in Figure 31, or other mounted storage (not shown). Coupling of the respective cores to such DRAM (or other system memory) and flash memory (or other mounted storage) may be effected in the conventional manner known in the art, as adapted in accord with the teachings hereof.

The illustrated elements of the respective cores, e.g., 12A–12G, 14A–14G, 16A–16G, are coupled for communication to one another directly and/or indirectly via hardware and/or software logic, as well, as with the other cores, e.g., 14, 16, as evident in the discussion below and in the other drawings. For sake of simplicity, such coupling is not shown in **Figure 1**. Thus, for example, the arithmetic logic units, thread processing units, virtual event lookup table, virtual instruction and data caches of each core 12–16 may be coupled for communication and interaction with other elements of their respective cores 12–16, and with other elements of the system 10 in the manner of the “execution units” (or “functional units”), “thread processing units (TPUs),” “event-to-thread lookup table,” and “instruction cache”/“data cache,” respectively, disclosed in the aforementioned figures and text, by way of non-limiting example, of aforementioned, incorporated-by-reference US Patents 7,685,607 and US 7,653,912, as adapted in accord with the teachings hereof.

Cache-Controlled Memory System—Introduction

The illustrated embodiment provides a system 10 in which the cores 12–16 utilize a cache-controlled system memory (e.g., cache-based management of all memory stores that form the system, whether as cache memory within the cache subsystems, attached physical memory such as flash memory, mounted drives or otherwise). Broadly speaking, that system can be said to include one or more nodes, here, processor modules or cores 12–16 (but, in other embodiments, other logic elements) that include or are otherwise coupled to cache memory, physical memory (e.g., attached flash drives or other mounted storage devices) or other memory—collectively, “system memory”—as shown, for example, in **Figure 31** and discussed elsewhere herein. The nodes 12–16 (or, in some embodiments, at least one of them) provide a cache memory system that stores data (and, preferably, in the illustrated embodiment, instructions) recently accessed

(and/or expected to be accessed) by the respective node, along with tags specifying addresses and statuses (e.g., modified, reference count, etc.) for the respective data (and/or instructions). The data (and instructions) in those caches and, more generally, in the "system memory" as a whole are preferably referenced in accord with a "system" addressing scheme that is common to one or more of the nodes and, preferably, to all of the nodes.

The caches, which are shown in **Figure 1** hereof for simplicity as unitary respective elements 12D–16D are, in the illustrated embodiment, organized in multiple hierarchical levels (e.g., a level 1 cache, a level 2 cache, and so forth)—each, for example, organized as shown in **Figure 20** hereof.

Those caches may be operated as virtual instruction and data caches that support a novel virtual memory system architecture in which inter alia all system memory (whether in the caches, physical memory or otherwise) is effectively managed as cache, even though for example, off-chip memory may utilize DDR DRAM. Thus, for example, instructions and data may be copied, updated and moved among and between the caches and other system memory (e.g., physical memory) in a manner paralleling that disclosed, by way of example, patent publications of Kendall Square Research Corporation, including, US 5,055,999, US 5,341,483, and US 5,297,265, including, by way of example, Figures 2A, 2B, 3, 6A–7D and the accompanying text of US 5,055,999, the teachings of which figures and text (and others of which pertain to data movement, copying and updating) are incorporated herein by reference, as adapted in accord with the teachings hereof. The foregoing is likewise true of extension tags, which can also be copied, updated and moved among and between the caches and other system memory in like manner.

The system memory of the illustrated embodiment stores additional (or "extension") tags that can be used by the nodes, the memory system and/or the operating system like cache tags. In addition to specifying system addresses and statuses for respective data (and/or instructions), the extension tags also specify physical address of those data in system memory. As such, they facilitate translating system addresses to physical addresses, e.g., for purposes of moving data

(and/or instructions) between physical (or other system) memory and the cache memory system (a/k/a the “caching subsystem,” the “cache memory subsystem,” and so forth).

Selected extension tags of the illustrated system are cached in the cache memory systems of the nodes, as well as in the memory system. These selected extension tags include, for example, those for data recently accessed (or expected to be accessed) by those nodes following cache “misses” for that data within their respective cache memory systems. Prior to accessing physical (or other system memory) for data following a local cache miss (i.e., a cache miss within its own cache memory system), such a node can signal a request for that data to the nodes, e.g., along bus, network or other media (e.g., the Ring Interconnect shown in Figure 31 and discussed elsewhere herein) on which they are coupled. A node that updates such data or its corresponding tag can likewise signal the other nodes and/or the memory system of the update via the interconnect.

Referring back to **Figure 1**, the illustrated cores 12–16 may form part of a general purpose computing system, e.g., being housed in mainframe computers, mini computers, workstations, desktop computers, laptop computers, and so forth. As well, they may be embedded in a consumer, commercial or other device (not shown), such as a television, cell phone, or personal digital assistant, by way of example, and may interact with such devices via various peripherals interfaces and/or other logic (not shown, here).

A single or multiprocessor system embodying processor and related technology according to the illustrated embodiment—which processor and/or related technology is occasionally referred to herein by the mnemonic “SEP” and/or by the name “Pave Processor,” “Pave SDF,” or the like—is optimized for applications with large data processing requirements, e.g., real time embedded applications which have a high degree of media processing requirements. SEP is general purpose in multiple aspects:

- Software defined processing, rather than dedicated hardware for special purpose functions
 - o Standard languages and compilers like gcc
- Standard OS like Linux, no real time OS required

- High performance for a large range of media and general purpose applications.
- Leverage parallelism to scale applications and performance on today's and future implementation. SEP is designed to scale single thread performance, thread parallel performance and multiprocessor performance
- Gain high efficiency of software algorithms and utilization of underlying hardware capability.

The types of products and applications of SEP are limitless, but the focus of the discussion here is on mobile products for sake of simplicity and without loss of generality. Such applications are network- and Internet-aware and could include, by way of non-limiting example:

- Universal Networked Display
- Networked information appliance
- PDA & Personal Knowledge Navigator (PKN) with voice and graphical user interface with capabilities such as real time voice recognition, camera (still, video) recorder, MP3 player, game player, navigation and broadcast digital video (MP4?). This device might not look like a PDA.
- G3 mobile phone integrated with other capabilities.
- Audio and video appliances including video server, video recorder and MP3 server.
- Network-aware appliances in general

These exemplary target applications are, by way of non-limiting example, inherently parallel. In addition, they have or include one or more of the following:

- High computational requirements
- Real time application requirements
- Multi-media applications
- Voice and graphical user interface
- Intelligence
- Background tasks to aid the user (like intelligent agents)
- Interactive nature
- Transparent Internet, networking and Peer to Peer (P2P access)

- Multiple applications executing concurrently to provide the device/user function.

A class of such target applications are multi-media and user interface-driven applications that are inherently parallel at the multi-tasking and multi-processing levels (including peer-to-peer).

Discussed in the preceding sections and below are architectural, processing and other aspects of SEP, along with structures and mechanisms in support of those features. It will be appreciated that the processors, systems and methods shown in the illustrations and discussed here are examples of the invention and that other embodiments, incorporating variations on those here, are contemplated by the invention, as well.

The illustrated SEP embodiment directly supports 64 bit address, 64/32/18/8 bit data-types, large general purpose register set and general purpose predicate register set. In preferred embodiments (such as illustrated here), instructions are predicated to enable the compiler to eliminate many conditional branches. Instruction encodings support multi-threading and dynamic distributed shared execution environment features.

SEP simultaneous multi-threading provides flexible multiple instruction issue. High utilization of execution units is achieved through simultaneous execution of multiple process or threads (collectively, "threads") and eliminating the inefficiencies of memory misses, and memory/branch dependencies. High utilization yields high performance and lower power consumption.

Events are handled directly by the corresponding thread without OS intervention. This enables real-time capability utilizing a standard OS like Linux. Real time OS is not required.

The illustrated SEP embodiment supports a broad spectrum of parallelism to dynamically attain the right range and granularity of parallelism for a broad mix of applications, as discussed below.

- Parallelism within an instruction
 - Instruction set uniformly enables single 64 bit, dual 32 bit, quad 16 bit and octal 8 bit operations to support high performance image processing, video processing, audio processing, network processing and DSP applications

- Multiple Instruction Execution within a single thread
 - o Compiler specifies the instruction grouping within a single thread that can execute during a single cycle. Instruction encoding directly supports specification of grouping. The illustrated SEP architecture enables scalable instruction level parallelism across implementations- one or more integer, floating point, compare, memory and branch classes.
- Simultaneous multi-threading
 - o SEP implements the ability to simultaneously execute one or more instructions from multiple threads. Each cycle, the SEP schedules one or more instructions from multiple threads to optimally utilize available execution unit resources. SEP multi-threading enables multiple application and processing threads to operate and interoperate concurrently with low latency, low power consumption, high performance and reduced implementation complexity. See "Generalized Events and Multi-Threading," hereof.
- Generalized Event architecture
 - o SEP provides to mechanisms that enable efficient multi-threaded, multiple processor and distributed P2P environments: unified event mechanism and software transparent consumer producer memory capability.
 - o The largest degradation of real-time performance of standard OS, like Linux is that all interrupts and events must be handled by the kernel before being handled by the actual event or application event handler. This lowers the quality of real-time applications like audio and video. Every SEP event is transparently wakes up the appropriate thread without kernel intervention. Unified events enable all events (HW interrupts, SW events and others) to be handled directly by the user level thread, eliminating virtually all OS kernel latency. Thus the real time performance of standard OS is significantly improved.
 - o Synchronization overhead and programming difficulty of implemented the natural data based processing flow between threads or processors (for multiple steps of image processing for example) is very high. SEP memory instructions enable threads to wait on the availability of data and transparently wake up when another thread indicates the data is available. Software transparent consumer-producer memory operations enables higher performance fine grained thread level parallelism with an efficient data oriented, consumer-producer programming style.

- Single Processor replaces multiple embedded processors
 - o Most embedded systems require separate special purpose processors (or dedicated hardware resources) for application, image, signal and network processing. Also, the software development complexity with multiple special purpose processors is high. In general multiple embedded processors adds to the cost and power consumption of the end product.
 - o The multi-threading and generalized event architecture enables a single SEP processor to handle all application image, signal and network processing for a mobile product, resulting in lower cost and power consumption.
- Cache based Memory System
 - o In preferred embodiments (such as illustrated here), all system memory is managed as cache. This enables an efficient mechanism to manage a large sparse address and memory space across a single and multiple mobile devices. This also eliminates address translation bottleneck from first level cache and TLB miss penalty. Efficient operation of SEP across multiple devices is an integrated feature, not an afterthought.
- Dynamic distributed shared execution environment (remote P2P technology)
 - o Generally, OS level threads and application threads cannot be transparently executed across different devices. Generalized event, consumer-producer memory, multi-threading enables seamless distributed shared execution environment across processors including: distributed shared memory/objects, distributed shared events and distributed shared execution. This enables the mobile device to automatically off load work to improve performance and lower power consumption.

The architecture supports scalability, including:

- Instruction extension with additional functional units or programmable functional units
- Increasing the number of functional units improves the performance of individual threads more significantly the performance of simultaneously executing threads.
- Multi-processor- Adding additional processors to an SEP chip.
- Increases in cache and memory size.
- Improvements in semiconductor technology.

GENERALIZED EVENTS AND MULTI-THREADING

Generalized SEP event and multi-threading model are both unique and powerful. A thread is a stateful fully independent flow of control. Threads communicate through sharing memory, like a shared memory multi-processor or through events. SEP has special behavior and instructions that optimize memory performance, performance of threads interacting through memory and event signaling performance. SEP event mechanism enables device (or software) events (like interrupts) to be signaled directly to the thread that is designated to handle the event, without requiring OS interaction.

The generalized multi-thread model works seamlessly across one or more physical processors. Each processor 12, 14 implements one or more Thread Processing Units (TPU) 12B, 14B, which are bound to one thread at any given instant. Thread Processing Units behave like virtual processors and execute concurrently. As shown in the drawing, TPUs executing on a single processor usually share level1 (L1 Instruction & L1 Data) and level2 (L2) cache (which may be shared with the TPU of the other processor, as well). The fact that they share caches is software transparent, thus multiple threads can execute on a single or multiple processors in a transparent manner.

Each implementation of the SEP processor has some number (e.g., one or more) of Thread Processing Units (TPUs) and some number of execution (or functional) units. Each TPU contains the full state of each thread including general registers, predicate registers, control registers and address translation.

The foregoing may be appreciated by reference to **Figure 2**, which depicts a system 10' comprising two processor modules of the type shown in Figure 1 and labelled, here, as 12, 14. As discussed above, these include respective functional units 12A-14A, thread processing units 12B-14B, and respective caches 12D-14D, here, arranged as separate respective Level1 instruction and data caches for each module and as shared Level2 and Level2 Extended caches, as shown. Such sharing may be effected, for example, by interface logic that is coupled, on the one hand, to the respective modules 12-14 and, more particularly, to their respective L1 cache

circuitry and, on the other hand, to on-chip (in the case, e.g., of the L2 cache) and/or off-chip (in the case, e.g., of the L2E cache) memory making up the L2 and L2E caches, respectively.

The processor modules shown in Figure 2 additionally include respective address translation functionality 12G–14G, here, shown associated with the respective thread processing units 12B–14B, that provide for address translation in a manner like that disclosed, by way of non-limiting example, in connection with TPU elements 10–20 of Figure 1, in connection with Figure 5 and the accompanying text, and in connection with branch unit 38 of Figure 13 and the accompanying text, all of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to the address translation) are incorporated herein by reference, as adapted in accord with the teachings hereof.

Those processor modules additionally include respective launch and pipeline control units 12F–14F that are generally constructed, operated, and utilized in the manner of the “launch and pipeline control” or “pipeline control” unit disclosed, by way of non-limiting example, as elements 28 and 130 of Figure 1 and 13–14, respectively and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to the launch and pipeline control) are incorporated herein by reference, as adapted in accord with the teachings hereof.

During each cycle the dispatcher schedules instructions from the threads in “executing” state in the Thread Processing Units such as to optimize utilization of the execution units. In general with a small number of active threads, utilization can typically be quite high, typically >80-90%. During each cycle SEP schedules the TPUs requests for execution units (based on instructions) on a round robin bases. Each cycle the starting point of the round robin is rotated among TPUs to assure fairness. Thread priority can be adjusted on an individual thread basis to increase or decrease the priority of an individual thread to bias the relative rate that instructions are dispatched for that thread.

Across implementations the amount of instruction parallelism within a thread and across a thread can vary based on the number of execution units, TPUs and processors, all transparently to software.

Contrasting superscalar vs. SEP multithreaded architecture, in a superscalar processor, instructions from a single executing thread are dynamically scheduled to execute on available execution units based on the actual parallelism and dependencies within the program. This means that on the average most execution units are not able to be utilized during each cycle. As the number of execution units increases the percentage utilization typically goes down. Also execution units are idle during memory system and branch prediction misses/waits. In contrast, multithreaded SEP instructions from multiple threads (shown in different colors) execute simultaneously. Each cycle, the SEP schedules instructions from multiple threads to optimally utilize available execution unit resources. Thus the execution unit utilization and total performance is higher, totally transparent to software.

The underlying rationales for supporting multiple active threads (virtual processors) per processor are:

- Functional capability
 - o Enables single multi-threaded processor to replace multiple application, media, signal processing and network processors
 - o Enable multiple threads corresponding to application, image, signal processing and networking to operate and interoperate concurrently with low latency and high performance. Context switch and interfacing overhead is minimized. Even within a single image processing application like MP4 decode threads can easily operate simultaneously in a pipelined manner to for example prepare data for frame $n+1$ while frame n is being composed.
- Performance
 - o Increase the performance of the individual processor by better utilizing functional units and tolerating memory and other event latency. It is not unusual to gain a 3x or more performance increase for supporting up to 4-6 simultaneously

executing threads. Power consumption and die size increases are negligible so that performance per unit power and price performance are improved.

- o Lower the performance degradation due to branches and cache misses by having another thread execute during these events
- o Eliminates most context switch overhead
- o Lowers latency for real time activities
- o General, high performance event model.
- Implementation
 - o Simplification of pipeline and overall design
 - o No complex branch predication- another thread can run!!
 - o Lower cost of single processor chip vs. multiple processor chips.
 - o Lower cost when other complexities are eliminated.
 - o Improve performance per unit power.

THREAD STATE

Threads are disabled and enabled by the thread enable field of the Thread State Register (discussed below, in connection with “Control Registers.”) When a thread is disabled: no thread state can change, no instructions are dispatched and no events are recognized. System software can load or unload a thread into a TPU by restoring or saving thread state, when the thread is disabled. When a thread is enabled: instructions can be dispatched, events can be recognized and thread state can change based on instruction completion and/or events.

Thread states and transitions are illustrated in **Figure 3**. These include:

- **Executing:** Thread context is loaded into a TPU and is currently executing instructions.
 - o A thread transitions to waiting when a memory instruction must wait for cache to complete an operation, e.g. miss or not empty/full (producer-consumer memory)
 - o A thread transitions to idle when a event instruction is executed.
- **Waiting:** Thread context is loaded into a TPU, but is currently not executing instructions. Thread transitions to executing when an event it is waiting for occurs:

- o Cache operation is completed that would allow the memory instruction to proceed.
- **Waiting_IO:** Thread context is loaded into a TPU, but is currently not executing instructions. Thread transitions to executing when one of the following events occurs:
 - o Hardware or software event.

Figure 4 ties together instruction execution, thread and thread state. The dispatcher dispatches instructions from threads in “executing” state. Instructions either are retired- complete and update thread state (like general purpose (gp) registers); or transition to waiting because the instruction is not able to complete yet because it is blocked. Example of an instruction blocking is a cache miss. When an instruction becomes unblocked, the thread is transitioned from waiting to executing state and the dispatcher takes over from there. Examples of other memory instructions that block are empty and full.

Next asynchronous signals, called events which can occur in idle or executing states is introduced.

EVENTS

Event is an asynchronous signal to a thread. SEP events are unique in that any type of event can directly signal any thread, user or system privilege, without processing by the OS. In all other systems, interrupts are signaled to the OS, which then dispatches the signal to the appropriate process or thread. This adds the latency of the OS and latency of signaling another thread to the interrupt latency. This typically requires a highly tuned real-time OS and advanced software tuning for the application. For SEP, since the event gets delivered directly to a thread, the latency is virtually zero, since the thread can respond immediately and the OS is not involved. A standard OS and no application tuning is necessary.

Two types of SEP events are shown in **Figure 5**, which depicts event binding and processing in a processor module, e.g., 12-16, according to the invention. More particularly, that drawing illustrates functionality provided in the cores 12-16 of the illustrated embodiment and how they are used to process and bind device events and software events to loaded threads (e.g., within the

same core and/or, in some embodiments, across cores, as discussed elsewhere herein). Each physical event or interrupt is represented as a physical event number (16 bits). The event table maps the physical event number to a virtual thread number (16 bits). If the implementation has more than one processor, the event table also includes an eight bit processor number. An Event To Thread Delivery mechanism delivers the event to the mapped thread, as disclosed, by way of non-limiting example, in connection with element 40–44 of Figure 4 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to event-to-thread delivery) are incorporated herein by reference, as adapted in accord with the teachings hereof. The events are then queued. Each TPU corresponds to a virtual thread number as specified in its corresponding ID register. The virtual thread number of the event is compared to that of each TPU. If there is a match the event is signaled to the corresponding TPU and thread. If there is not a match, the event is signaled to the default system thread in TPU zero.

The routing of memory events to threads by the cores 12–16 of the illustrated embodiment is handled in the manner disclosed, by way of non-limiting example, in connection with elements 44, 50 of Figure 4 and the accompanying text of aforementioned US Patents 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain to memory event processing) are incorporated herein by reference, as adapted in accord with the teachings hereof.

In order to process an event, a thread takes the following actions. If the thread is in waiting state, the thread is waiting for a memory event to complete and the thread will recognize the event immediately. If the thread is in waiting_IO state, the thread is waiting for an IO device operation to complete and will recognize the event immediately. If the thread is in executing state the thread will stop dispatching instructions and recognize the event immediately.

On recognizing the event, the corresponding thread saves the current value of Instruction Pointer into System or Application Exception IP register and saves the event number and event status into System or Application Exception Status Register. System or Application registers are utilized based on the current privilege level. Privilege level is set to system and application trap enable is reset. If the previous privilege level was system, the system trap enable is also reset.

The Instruction Pointer is then loaded with the exception target address (Table 8) based on the previous privilege level and execution starts from this instruction.

Operations of other threads are unaffected by an event.

Threads run at two privilege levels, System and Application. System threads can access all state of its thread and all other threads within the processor. An application thread can only access non-privileged state corresponding to it. On reset TPU 0 runs thread 0 at system privilege. Other threads can be configured for privilege level when they are created by a system privilege thread.

EVENT FORMAT FOR HARDWARE AND SOFTWARE EVENTS

63	32	31	16	15	4	3	2	1	0
		threadnum	eventnum				how	priv	

Bit	Field	Description
0	priv	Privilege that the event will be signaled at: 1 System privilege 2 Application privilege
1	how	Specifies how the event is signaled if the thread is not in idle state. If the thread is in idle state, this field is ignored and the event is directly signalled 1 Wait for thread in idle state. All events after this event in the queue wait also. 2 Trap thread immediately

15:4	eventnum	Specifies the logical number for this event. The value of this field is captured in detail field of the system exception status or application exception status register.
31:16	threadnum	Specifies the logical thread number that this event is signaled to.

EXAMPLE EVENT OPERATIONS

Reset Event Handling

Reset event causes the following actions:

- Event handling queues are cleared.
- Thread State Register for each thread has reset behavior as specified. System exception status register will indicate reset. Thread 0 will start execution from virtual address 0x0. Since address translation is disabled at reset, this will also be System Address 0x0. The memcore is always configured as core 0, so 0x0 offset at memcore will address address 0x0 of flash memory. See sections "Addressing" and "Standard Device Registers" in "Virtual Memory and Memory System," hereof.
- All other threads are disabled on reset.
- No configuration for flash access after reset is required. Flash memory accessed directly by processor address is not cached and placed directly into the thread instruction queue.
- Cacheable address space must not be accessed until L1 instruction, L1 data and L2 caches are initialized. Only a single thread should be utilized until caches are initialized. L1 caches can be initialized through Instruction or Data Level1 Cache Tag Pointer (ICTP, DCTP) and Instruction or Data Level1 Cache Tag Entry (ICTE, DCTE) control registers. Tag format is provided in Cache organization and entry description section of "Virtual Memory and Memory System," hereof. L2 cache can be initialized through L2 standard device registers and formats described in "Virtual Memory and Memory System," hereof

Thread Event Handling

- Reset event handling must configure the event queue. There is a single event queue per chip, independent of the number of cores. The event queue is associated with core 0.
- For each event type, an entry is placed into event queue lookup table. All events with no value in the event queue lookup table are queued to thread 0.
- Each time that a thread is loaded or unloaded from a thread processing unit (hardware thread), the corresponding event queue lookup table entry should be updated. Sequence should be:
 - o Remove entry from event queue lookup table
 - o Disable thread, unload thread. Note if an event is signaled in the window between removing the entry and disabling the thread it will be presented to thread 0 for action.
 - o Add new entry event queue lookup table
 - o Load new thread into TPU.
- Operation is identical for single and multiple threads and TPUs

Dynamic Assignment Of Events To Threads

Referring to **Figure 38**, an SEP processor module (e.g, 12) according to some practices of the invention permits devices and/or software (e.g., applications, processes and/or threads) to register, e.g., with a default system thread or other logic to identify event-processing services that they require and/or event-handling capabilities they provide. That thread or other logic (e.g., event table manager 106', below) continually matches those requirements (or "needs") to capabilities and updates the event-to-thread lookup table to reflect an optimal mapping of events to threads, based on the requirements and capabilities of the overall system 10—so that, when those events occur, the table can be used (e.g., by the event-to-thread delivery mechanism, as discussed in the section "Events," hereof) to map and route them to respective virtual threads and to signal the TPUs that are executing them. In addition to matching to one another the needs and capabilities registered with it by the devices and/or software, the default system thread or other logic can match registered needs with other capabilities known to it (whether or not

registered) and, likewise, can match registered capabilities with other needs known to it (again, whether or not registered, per se).

This can be advantageous over matching of events to threads based solely on "hardcoded" or fixed assignments. Those arrangements may be more than adequate for applications where the software and hardware environment can be reasonably predicted by the software developers. However, they might not best serve processing and throughput demands of dynamically changing systems, e.g., where processing-capable devices (e.g., those equipped with SEP processing modules or otherwise) come into and out of communications coupling with one another and with other processing-demanding software or devices). By way of non-limiting example is a SEP core-equipped phone for gaming applications. When the phone is isolated, it processes all gaming threads (as well as telephony, etc., threads) on its own. However, if the phone comes into range of another core-equipped device, it offloads appropriate software and hardware interrupt processing to that other device.

Referring to **Figure 38**, a preprocessor of the type known in the art—albeit as adapted in accord with the teachings hereof—inserts into source code (or intermediate code, or otherwise) of applications, library code, drivers, etc. that will be executed by the system 10 event-to-thread lookup table management code that upon execution (e.g., upon interpretation and/or following compilation, linking, etc.) causes the executed code to register event-processing services that it will require and/or capabilities that it will provide at runtime. That event-to-thread lookup table management code can be based on directives supplied by the developer (as well, potentially, by the manufacturer, distributor, retailer, post-sale support personnel, end user or other) to reflect one or more of: the actual or expected requirements (or capabilities) of the respective source, intermediate or other code, as well as about the expected runtime environment and the devices or software potentially available within that environment with potentially matching capabilities (or requirements).

The drawing illustrates this by way of source code of three applications 100–104 which would normally be expected to require event-processing services; although, that and other software may provide event-handling capabilities, instead or in addition—e.g., as in the case of codecs, special-

purpose library routines, and so forth, which may have event-handling capabilities for service events from other software (e.g., high-level applications) or of devices. As shown, the exemplary applications 100–104 are processed by the preprocessor to generate “preprocessed apps” 100’–104’, respectively, each with event-to-thread lookup table management code inserted by the preprocessor.

The preprocessor can likewise insert into device driver code or the like (e.g., source, intermediate or other code for device drivers) event-to-thread lookup table management code detailing event-processing services that their respective devices will require and/or capabilities that those devices will provide upon insertion in the system 10.

Alternatively or in addition to being based on directives supplied by the developer (manufacturer, distributor, retailer, post-sale support personnel, end user or other), event-to-thread lookup table management code can be supplied with the source, intermediate or other code by the developers (manufacturers, distributors, retailers, post-sale support personnel, end users or other) themselves—or, still further alternatively or in addition, can be generated by the preprocessor based on defaults or other assumptions/expectations of the expected runtime environment. And, although event-to-thread lookup table management code is discussed here as being inserted into source, intermediate or other code by the preprocessor, it can, instead or in addition, be inserted by any downstream interpreters, compilers, linkers, loaders, etc. into intermediate, object, executable or other output files generated by them.

Such is the case, by extension, of the event table manger code module 106’, i.e., a module that that, at runtime, updates the event-to-thread table based on the event-processing services and event-handling capabilities registered by software and/or devices at runtime. Though that module may be provided in source code format (e.g., in the manner of files 100–104), in the illustrated embodiment, it is provided as a prepackaged library or other intermediate, object or other code module compiled and/or that is linked into the executable code. Those skilled in the art will appreciate that this is by way of example and that, in other embodiments the functionality of module 106’ may be provided otherwise.

With further reference to the drawing, a compiler/linker of the type known in the art—albeit as adapted in accord with the teachings hereof—generates executable code files from the preprocessed apps 100'–104' and module 106' (as well as from any other software modules) suitable for loading into and execution by module 12 at runtime. Although that runtime code is likely to comprise one or more files that are stored on disk (not shown), in L2E cache or otherwise, it is depicted, here, for convenience, as threads 100''–106'' it will ultimately be broken into upon execution.

In the illustrated embodiment, that executable code is loaded into the instruction/data cache 12D at runtime and is staged for execution by the TPUs 12B (here, labelled, TPU[0,0]–TPU[0,2]) of processing module 12 as described above and elsewhere herein. The corresponding enabled (or active) threads are shown here with labels 100''', 102''', 104'''. That corresponding to event table manager module 106' is shown, labelled as 106'''.

Threads 100'''–104''' that require event-processing services (e.g., for software interrupts) and/or that provide event-processing capabilities register, e.g., with event table manager module 106''', here, by signalling that module to identify those needs and/or capabilities. Such registration/signalling can be done as each thread is instantiated and/or throughout the life of the thread (e.g., if and as its needs and/or capabilities evolve). Devices 110 can do this as well and/or can rely on interrupt handlers to do that registration (e.g., signalling) for them. Such registration (here, signalling) is indicated in the drawing by notification arrows emanating from thread 102''' of TPU[0,1] (labelled, here, as “thread regis” for thread registration); thread 104''' of TPU[0,2] (software interrupt source registration); device 110 Dev 0 (device 0 registration); and, device 110 Dev 1 (device 1 registration) for routing to event table manager module 106'''. In other embodiments, the software and/or devices may register, e.g., with module 106''', in other ways.

The module 106''' responds to the notifications by matching the respective needs and/or capabilities of the threads and/or devices, e.g., to optimize operation of the system 10, e.g., on any of many factors including, by way of non-limiting example, load balancing among TPUs and/or cores 12–16, quality of service requirements of individual threads and/or classes of threads (e.g., data throughput requirements of voice processing threads vs. web data transmission

threads in a telephony application of core 12), energy utilization (e.g., for battery operation or otherwise), actual or expected numbers of simultaneous events, actual or expected availability of TPUs and/or cores capable of processing events, and so forth, all by way of example). The module 106"" updates the event lookup table 12C accordingly so that subsequently occurring events can be mapped to threads (e.g., by the event-to-thread delivery mechanism, as discussed in the section "Events," hereof) in accord with that optimization.

Location-Independent Shared Execution Environment

Figure 39 depicts configuration and use of the system 10 of **Figure 1** to provide a location-independent shared execution environment and, further, depicts operation of processor modules 12–16 in connection with migration of threads across core boundaries to support such a location-independent shared execution environment. Such configurations and uses are advantageous, among other reasons, in that they facilitate optimization of operation of the system 10—e.g., to achieve load balancing among TPUs and/or cores 12–16, to meet quality of service requirements of individual threads, classes of threads, individual events and/or classes of events, to minimize energy utilization, and so forth, all by way of example—both in static configurations of the system 10 and in dynamically changing configurations, e.g., where processing-capable devices come into and out of communications coupling with one another and with other processing-demanding software or devices. By way of overview, the system 10 and, more particularly, the cores 12–16 provide for migration of threads across core boundaries by moving data, instructions and/or thread (state) between the cores, e.g., in order to bring event-processing threads to the cores (or nearer to the cores) whence those events are generated or detected, to move event-processing threads to cores (or nearer to cores) having the capacity to process them, and so forth, all by way of non-limiting example.

Operation of the illustrated processor modules in support of location-independent shared execution environment and migration of threads across processor 12–16 boundaries is illustrated in **Figure 39**, in which the following steps (denoted in the drawings as numbers in dashed-line ovals) are performed. It will be appreciated that these are by way of example and that other embodiments may perform different steps and/or in different orders:

In step 120, core 12 is notified of an event. This may be a hardware or software event, and it may be signaled from a local device (i.e., one directly coupled to core 12), a locally executing thread, or otherwise. In the example, the event is one to which no thread has yet been assigned. Such notification may be effected in a manner known in the art and/or utilizing mechanisms disclosed in incorporated-by-reference patents US 7,685,607 and US 7,653,912, as adapted in accord with the teachings hereof.

In step 122, the default system thread executing on one of the TPUs local to core 12, here, TPU[0,0] is notified of the newly received event and, in step 123, that default thread can instantiate a thread to handle the incoming event and subsequent related events. This can include, for example, setting state for the new thread, identifying event handler or software sequence to process the event, e.g., from device tables, and so forth, all in the manner known in the art and/or utilizing mechanisms disclosed in incorporated-by-reference patents US 7,685,607 and US 7,653,912, as adapted in accord with the teachings hereof. (The default system thread can, in some embodiments, process the incoming event directly and schedule a new thread for handling subsequent related events.) The default system thread likewise updates the event-to-thread table to reflect assignment of the event to the newly created thread, e.g., a manner known in the art and/or utilizing mechanisms disclosed in incorporated-by-reference patents US 7,685,607 and US 7,653,912, as adapted in accord with the teachings hereof; see step 124.

In step 125, the thread that is handling the event (e.g., the newly instantiated thread or, in some embodiments, the default system thread) attempts to read the next instruction of the event-handling instruction sequence for that event from cache 12D. If that instruction is not present in the local instruction cache 12D, it (and, more typically, a block of instruction “data” including it and subsequent instructions of the same sequence) is transferred (or “migrated”) into it, e.g., in the manner described in connection with the sections entitled “Virtual Memory and Memory System,” “Cache Memory System Overview,” and “Memory System Implementation,” hereof, all by way of example; see step 126. And, in step 127, that instruction is transferred to the TPU 12B to which the event-handling thread is assigned, e.g., in accord with the discussion at “Generalized Events and Multi-Threading,” hereof, and elsewhere herein.

In step 128a, the instruction is dispatched to the execution units 12A, e.g., as discussed in “Generalized Events and Multi-Threading,” hereof, and elsewhere herein, for execution, along with the data required for such execution—which the TPU 12B and/or the assigned execution unit 12A can also load from cache 12D; see step 128b. As above, if that data is not present in the local data cache 12D, it is transferred (or “migrated”) into it, e.g., in the manner referred to above in connection with the discussion of step 126.

Steps 125–128b are repeated, e.g., while the thread is active (e.g., until processing of the event is completed) or until it is thrown into a waiting state, e.g., as discussed above in connection with “Thread State” and elsewhere herein. They can be further repeated if and when the TPU 12B on which the thread is executing is notified of further related events, e.g., received by core 12 and routed to that thread (e.g., by the event-to-thread delivery mechanism, as discussed in the section “Events,” hereof).

Steps 130–139 illustrate migration of that thread to core 16, e.g., in response to receipt of further events related to it. While such migration is not necessitated by systems according to the invention, it (migration) too can facilitate optimization of operation of the system as discussed above. The illustrated steps 130–139 parallel the steps described above, albeit steps 130–139 are executed on core 16.

Thus, for example, step 130 parallels step 120 vis-a-vis receipt of an event notification by core 16.

Step 132 parallels step 122 vis-a-vis notification of the default system thread executing on one of the TPUs local to core 16, here, TPU[2,0] of the newly received event.

Step 133 parallels step 123 vis-a-vis instantiation of a thread to handle the incoming event. However, unlike step 123 which instantiates a new thread, step 133 effects transfer (or migration) of a pre-existing thread to core 16 to handle the event—in this case, the thread instantiated in step 123 and discussed above in connection with processing of the event received in step 120. To that end, in step 133, the default system thread executing in TPU[2,0] signals and cooperates with the default system thread executing in TPU[0,0] to transfer the pre-existing thread’s register

state, as well as of the remainder of thread state based in memory, as discussed in “Thread (Virtual Processor) State,” hereof; see step 133b. In some embodiments, the default system thread identifies the pre-existing thread and the core on which it is (was) executing, e.g., by searching local and a remote components of the event lookup table show, e.g., in the breakout of Figure 40, below. Alternatively, one or more of the operations discussed here, in connection with steps 133 and 133b and be handled by logic (dedicated or otherwise) that is separate and apart from the TPU’s, e.g., by the event-to-thread delivery mechanism (discussed in the section “Events,” hereof) or the like.

Step 134 parallels step 124 vis-a-vis updating of the event-to-thread table of core 16 to reflect assignment of the event to the transferred thread.

Steps 135–137 parallel steps 125–127, respective, vis-a-vis reading the next instruction of the event-handling instruction sequence from the cache, here, cache 16D, migrating that instruction to that cache if not already present there, and transferring that instruction to the TPU, here, 16B, to which the event-handling thread is assigned.

Steps 138a–138b parallel steps 128a–128b vis-a-vis dispatching of the instruction for execution and loading the requisite data in connection therewith.

As above, steps 135–138b are repeated, e.g., while the thread is active (e.g., until processing of the event is completed) or until it is thrown into a waiting state, e.g., as discussed above in connection with “Thread State” and elsewhere herein. They can be further repeated if and when the TPU 16B on which the thread is executing is notified of further related events, e.g., received by core 16 and routed to that thread (e.g., by the event-to-thread delivery mechanism, as discussed in the section “Events,” hereof).

Figure 40 depicts further systems 10’ and methods according to practice of the invention wherein the processor modules (here, all labelled 12 for simplicity) of Figure 39 are embedded in consumer, commercial or other devices 150–164 for cooperative operation—e.g., routing and processing of events among and between modules within zones 170–174. The devices shown in

the illustration are televisions 152, 164 and set top boxes 154 cell phones 158, 162, and personal digital assistants 168, remote controls 156, though, these are only by way of example. In other embodiments, the modules may be embedded in other devices instead or in addition; for example, they may be included in desktop, laptop, or other computers.

The zones 170–174 shown in the illustration are defined by local area networks, though, again, these are by way of example. Such cooperative operation may occur within or across zones that defined in other ways. Indeed, in some embodiments, cooperative operation is limited to cores 12 within a given device (e.g., within a television 152), while in other embodiments that operation extends across networks even more encompassing (e.g., wider ranging) than LANs or less encompassing.

The embedded processor modules 12 are generally denoted in Figure 40 by the graphic symbol shown in **Figure 41A**. Along with those modules are symbolically depicted peripheral and/or other logic with which those modules 12 interact in their respective devices (i.e., within the respective devices within which they are embedded). The graphic symbol for those peripheral and/or other logic is provided in **Figure 41B**, but the symbols are otherwise left unlabeled in Figure 40 to avoid clutter.

A detailed breakout (indicated by dashed lines) of such a core 12 is shown in the upper left of Figure 40. That breakout does not show caches or functional units (ALU's) of the core 12 for ease of illustration. However, it does show the event lookup table 12C of that module (which is generally constructed, operated and utilized as discussed above, e.g., in connection with Figures 1 and 39) as including two components: a local event table 182 to facilitate matching events to locally executing threads (i.e., threads executing on one of the TPUs 12B of the same core 12) and a remote event table 184 to facilitate matching events to remotely executing threads (i.e., threads executing on another or the cores—e.g., within the same zone 170 or within another zone 172–174, depending upon implementation. Though shown as two separate components 182, 184 in the drawings, these may comprise a greater or lesser number of components in other embodiments of the invention.

Moreover, though described here as “tables,” it will be appreciated that the event lookup tables may comprise or be coupled with other functional components—such as, for example, an event-to-thread delivery mechanism, as discussed in the section “Events,” hereof—and that those tables and/or components may be entirely local to (i.e., disposed within) the respective core or otherwise. Thus, for example, the remote event lookup table 184 (like the local event lookup table 182) may comprise logic for effecting the lookup function. Moreover, table 184 may include and/or work cooperatively with logic resident not only in the local processor module but also in the other processor modules 14–16 for exchange of information necessary to route events to them (e.g., thread id’s, module id’s/addresses, event id’s, and so forth). To this end, the remote event lookup “table” is also referred to in the drawing as a “remote event distribution module.”

The results of matching locally occurring events, e.g., local software event 186 and local memory event 188, against the local event table 182 are depicted in the drawing. Specifically, as indicated by arrow labelled “in-core processing” those events are routed to a TPU of the local core for processing by a pre-existing or newly created thread. This is reflected in detail in the upper left of Figure 41.

Conversely, if a locally occurring event does not match an entry in the local event table 182 but does match one in the remote event table 184 (e.g., as determined by parallel or *in seriatim* applications of an incoming event ID against those tables), the latter can return a thread id, module id/address (collectively, “address”) of the core and thread responsible for processing that event. The event-to-thread delivery mechanism and/or the default system thread (for example) of the core in which the event is detected can utilize that address to route the event for processing by that responsible core/thread. This is reflected in Figure 40, by way of example, by hardware event 190, which matches an entry in table 184, which returns the address of a remote core responsible for handling that event—in this case, a core 12 embedded in device 154. The event-to-thread delivery mechanism and/or the default system thread (or other logic) of the core 12 that detected the event 190 utilizes that address to route the event to that remote core, which processes the event, e.g., as described above, e.g., in connection with steps 120–128b.

While routing of events to which threads are already assigned can be based on “current” thread location, that is, on the location of the core 12 on which the assigned thread is currently resident, events can be routed to other modules instead, e.g., to achieve load balancing (as discussed above). In some embodiments, this is true for both “new” events, i.e., those to which no thread is yet assigned, as well as for events to which threads are already assigned. In the latter regard (and, indeed, in both regards), the cores can utilize thread migration (e.g., as shown in Figure 39 and discussed above) to effect processing of the event of the module to which the event is so routed. This is illustrated, by way of non-limiting example, in the lower right-hand corner of Figure 40, wherein device 158 and, more particularly, its respective core 12, is shown transferring a “thread” (and, more precisely, thread state, instructions, and so forth—in accord with the discussion of Figure 39).

In some embodiments, a “master” one of the processor modules 12 within a zone 170–174 and/or within the system as a whole (depending on implementation), however, is responsible for routing events to preexisting threads and for choosing which modules/devices (including, potentially, the local module) will handle new events—e.g., in cooperation with default system threads running on the cores 12 within which those preexisting threads are executing (e.g., as discussed above in connection with Figure 39). Master status can be conferred on an *ad hoc* basis or otherwise and, indeed, it can rotate (or otherwise dynamically vary) among processors within a zone. Indeed, in some embodiments distribution is effected on a peer-to-peer basis, e.g., such that each module is responsible for routing events that it receives (e.g., assuming the module does not take up processing of the event itself).

Systems constructed in accord with the invention can effect downloading of software to the illustrated embedded processor modules. As shown in Figure 40, this can be effected from a “vendor” server to modules that are deployed “in the field” (i.e., embedded in devices that are installed in business, residences or otherwise). However, it can similarly be effected to modules pre-deployment, e.g., during manufacture, distribution and/or at retail. Moreover, it need be effected by a server but, rather, can be carried out by other functionality suitable for transmitting and/or installing requisite software on the modules. Regardless, as shown in the upper-right corner of Figure 40, the software can be configured and downloaded, e.g., in response to requests

from the modules, their operators, installers, retailers, distributors, manufacturers, or otherwise, that specify requirements of applications necessary (and/or desired) on each such module and the resources available on that module (and/or within the respective zone) to process those applications. This can include, not only the processing capabilities of the processor module to which the code will be downloaded, but also those of other processor modules with which it cooperates in the respective zone, e.g., to offload and/or share processing tasks.

General Purpose Embedded Processor With Provision Of Quality Of Service Through Thread Instantiation, Maintenance And Optimization

In some embodiments, threads are instantiated and assigned to TPUs on an as-needed basis. Thus, for example, events (including, for example, memory events, software interrupts and hardware interrupts) received or generated by the cores are mapped to threads and the respective TPUs are notified for event processing, e.g., as described in the section “Events,” hereof. If no thread has been assigned to a particular event, the default system thread is notified, and it instantiates a thread to handle the incoming event and subsequent related events. As noted above, such instantiation can include, for example, setting state for the new thread, identifying event handler or software sequence to process the event, e.g., from device tables, and so forth, all in the manner known in the art and/or utilizing mechanisms disclosed in incorporated-by-reference patents US 7,685,607 and US 7,653,912, as adapted in accord with the teachings hereof.

Such as-needed instantiation and assignment of events to threads is more than adequate for many applications. However, in an overly burdened system with one or more cores 12–16, the overhead required for setting up a thread and/or the reliance on a single critical service-providing thread may starve operations necessary to achieve a desired quality of service. By way of example is use of an embedded core 12 to support picture-in-a-picture display on a television. While a single JPEG 2000 decoding thread may be adequate for most uses, it may be best to instantiate multiple such threads if the user requests an unduly large number of embedded pictures—lest one or more of the displays appears jagged in the face of substantial on-screen motion. Another example might be a lower-power core 12 that is employed as the primary

processor in a cell phone and that is called upon to provide an occasional support processing role when the phone is networked with a television (or other device) that is executing an intensive gaming application on a like (though, potentially more powerful, core). If the phone's processor is too busy in its support role, the user who is initiating a call may notice degradation in phone responsiveness.

To this end, an SEP processor module (e.g., 12) according to some practices of the invention, utilizes a preprocessor of the type known in the art—albeit as adapted in accord with the teachings hereof—to insert into source code (or intermediate code, or otherwise) of applications, library code, drivers, or otherwise that will be executed by the system 10 thread management code that, upon execution, causes the default system thread (or other functionality within system 10) to optimize thread instantiation, maintenance and thread assignment at runtime. This can facilitate instantiation of an appropriate number of threads at an appropriate time, e.g., to meet quality of service requirements of individual threads, classes of threads, individual events and/or classes of events with respect to one or more of the factors identified above, among others, and including, by way of non-limiting example

- data processing requirements of voice processing events, applications and/or threads,
- data throughput requirements of web data transmission events, applications and/or threads,
- data processing and display requirements of gaming events, applications and/or threads,
- data processing and display requirements of telepresence events, applications and/or threads,
- decoding, scaler & noise reduction, color correction, frame rate control and other processing and display requirements of audiovisual (e.g., television or video) events, applications and/or threads,
- energy utilization requirements of the system 5, as well as of events, applications and/or threads processed thereon, and/or
- processing of actual or expected numbers of simultaneous events by individual threads, classes of threads, individual events and/or classes of events
- prioritization of the processing of threads, classes of threads, events and/or classes of events over other threads, classes of threads, events and/or classes of events

Referring to **Figure 42**, this is illustrated by way of source code modules of applications 200–204, the functions performed by which, during execution, have respective quality-of-service requirements. Paralleling the discussion above in connection with Figure 38, as shown in Figure 42, the applications 200–204 are processed by preprocessor of the type known in the art—albeit as adapted in accord with the teachings hereof—to generate “preprocessed apps” 200’–204’, respectively, into which preprocessor inserts thread management code based on directives supplied by the developer, manufacturer, distributor, retailer, post-sale support personnel, end user or other about one or more of: quality-of-service requirements of functions provided by the respective applications 200–204, the frequency and duration with which those functions are expected to be invoked at runtime (e.g., in response to actions by the end user or otherwise), the expected processing or throughput load (e.g., in MIPS or other suitable terms) that those functions and/or the applications themselves are expected to exert on the system 10 at runtime, the processing resources required by those applications, the relative prioritization of those functions as to each other and to others provided within the executing system, and so forth.

Alternatively or in addition to being based on directives, event management code can be supplied with the application 200–204 source or other code itself—or, still further alternatively or in addition, can be generated by the preprocessor based on defaults or other assumptions/expectations about one or more of the foregoing, e.g., quality-of-service requirements of the applications functions, frequency and duration of their use at runtime, and so forth. And, although event management code is discussed here as being inserted into source, intermediate or other code by the preprocessor, it can, instead or in addition, be inserted by any downstream interpreters, compilers, linkers, loaders, etc. into intermediate, object, executable or other output files generated by them.

Such is the case, by extension, of the thread management code module 206’, i.e., a module that that, at runtime, supplements the default system thread, event management code inserted into preprocessed applications 200’–204’, and/or other functionality within system 10 to facilitate thread creation, assignment and maintenance so as to meet the quality-of-service requirements of functions of the respective applications 200–204 in view of the other factors identified above (frequency and duration of their use at runtime, and so forth) and in view of other demands on

the system 10, as well, as its capabilities. Though that module may be provided in source code format (e.g., in the manner of files 200–204), in the illustrated embodiment, it is provided as a prepackaged library or other intermediate, object or other code module compiled and/or that is linked into the executable code. Those skilled in the art will appreciate that this is by way of example and that, in other embodiments, the functionality of module 206' may be provided otherwise.

With further reference to the drawing, a compiler/linker of the type known in the art—albeit as adapted in accord with the teachings hereof—generates executable code files from the preprocessed applications 200'–204' and module 206' (as well as from any other software modules) suitable for loading into and execution by module 12 at runtime. Although that runtime code is likely to comprise one or more files that are stored on disk (not shown), in L2E cache or otherwise, it is depicted, here, for convenience, as threads 200''–206'' it will ultimately be broken into upon execution.

In the illustrated embodiment, that executable code is loaded into the instruction/data cache 12D at runtime and is staged for execution by the TPUs 12B (here, labelled, TPU[0,0]–TPU[0,2]) of processing module 12 as described above and elsewhere herein. The corresponding enabled (or active) threads are shown here with labels 200'''–204'''. That corresponding to thread management code 206' is shown, labelled as 206'''.

Upon loading of the executable, thread instantiation and/or throughout their lives, threads 200'''–204''' cooperate with thread management code 206''' (whether operating as a thread independent of the default system thread or otherwise) to insure that the quality-of-service requirements of functions provided by those threads 200'''–204''' is met. This can be done a number of ways, e.g., depending on the factors identified above (e.g., frequency and duration of their use at runtime, and so forth), on system implementation, demands on and capabilities of the system 10, and so forth.

For example, in some instances, upon loading of the executable code, thread management code 206''' will generate a software interrupt or otherwise invoke threads 200'''–204'''—potentially,

long before their underlying functionality is demanded in the normal course, e.g., as a result of user action, software or hardware interrupts or so forth—hence, insuring that when such demand occurs, the threads will be more immediately ready to service it.

By way of further example, one or more of the threads 200'''–204''' may, upon invocation by module 206''' or otherwise, signal the default system thread (e.g., working with the thread management code 206''' or otherwise) to instantiate multiple instances of that same thread, mapping each to different respective upcoming events expected occur, e.g., in the near future. This can help insure more immediate servicing of events that typically occur in batches and for which dedication of additional resources is appropriate, given the quality-of-service demands of those events. *C.f.* the example above regarding use of JPEG 2000 decoding threads for support of picture-in-a-picture display.

By way of still further example, the thread management code 206''' can periodically, sporadically, episodically, randomly or otherwise or generate software interrupts or otherwise invoke one or more of threads 200'''–204''' to prevent them from going inactive, even after apparent termination of their normal processing following servicing of normal events incurred as a result of user action, software or hardware interrupts or so forth—again, insuring that when such events occurs, the threads will be more immediately ready to service it.

PROGRAMMING MODEL

ADDRESSING MODEL AND DATA ORGANIZATION

The illustrated SEP architecture utilizes a single flat address space. The SEP supports both big-endian and little-endian addresses spaces and are configured through a privileged bit in the processor configuration register. All memory data types can be aligned at any byte boundary, but performance is greater if a memory data type is aligned on a natural boundary.

Table 1-Address Space

Memory Format	Address space
Signed and unsigned Integer Byte (8 bits)	2^{64} bytes
Signed and unsigned Integer $\frac{1}{4}$ Word (16 bits)	$2^{63} \frac{1}{4}$ words
Signed and unsigned Integer $\frac{1}{2}$ Word (32 bits)	$2^{62} \frac{1}{2}$ words
Signed and unsigned Integer Word (64 bits)	2^{61} words
IEEE single precision floating point format (32bits)	$2^{62} \frac{1}{2}$ words
IEEE double precision floating point format (64bits)	2^{61} words
Instruction Doubleword	2^{60} doublewords
Compressed instructions- Huffman encoded Byte stream in Memory	2^{64} bytes (not implemented)

In the illustrated embodiment, all data addresses are byte address format; all data types must be aligned by natural size and addresses by natural size; and, all instruction addresses are instruction doublewords. Other embodiments may vary in one or more of these regards.

THREAD (VIRTUAL PROCESSOR) STATE

Each application thread includes the register state shown in **Figure 6**. This state in turn provides pointers to the remainder of thread state based in memory. Threads at both system and application privilege levels contain identical state, although some thread state is only visible when at system privilege level.

Register Sizing implementation note:

Architectural Resource	Architecture Size	Min Goal	Desired Goal
Thread General Purpose Registers	128	48	64
Predicate Registers	64	24	32
Number active threads	256	6	8
Pending memory event table	512	16	16
Pending memory events/ thread		2	
Event Queue			256
Event to Thread lookup table	256	16	32

General Purpose Registers

Each thread has up to 128 general purpose registers depending on the implementation. General Purpose registers 3-0 (GP[3:0]) are visible only at system privilege level and can be utilized for event stack pointer and working registers during early stages of event processing.

GP registers are organized and normally accessed as a single or adjacent pair of registers analogous to a matrix row. Some instructions have a Transpose (T) option to write the destination as a ¼ word column of 4 adjacent registers or a byte column of 8 adjacent registers. This option can be useful for accelerated matrix transpose and related types of operations.

Predication Registers

The predicate registers are part of the general purpose illustrated SEP predication mechanism. The execution of each instruction is conditional based on the value of the reference predicate register.

The illustrated SEP provides up to 64 one bit predicate registers as part of thread state. Each predicate register holds what is called a predicate, which is set to 1 (true) or reset to 0 (false) based on the result of executing a compare instruction. Predicate registers 3-1 (PR[3:1]) are visible at

system privilege level and can be utilized for working predicates during early stages of event processing. Predicate register 0 is read only and always reads as 1, true. It is by instructions to make their execution unconditional.

Control Registers

Thread State Register

63 24	23 16	15	14	13	12	11	10	9	8	7 6	5	4	3	2	1	0
	mod		dbg	see	daddr	iaddr	align	endian	mem step l	Thread bias	Thread state		priv	tenable	atrapi n	strapi n

Bit	Field	Description	Privilege	Per	Design Usage
0	strapi n	System trap enable. On reset cleared. Signalling of system trap resets this bit and atrapi n until it is set again by software when it is once again re-entrant. 1 System traps disabled 2 Events enabled	system_r w	Threa d	Branch

1	atrapen	Application trap enable. On reset cleared. Signalling of application trap resets this bit until it is set again by software when it is once again re-entrant. Application trap is caused by an event that is marked as application level when the privilege level is also application level 1 Events disabled (events are disabled on event delivery to thread) 2 Events enabled	app_rw	Thread	
2	tenable	Thread Enable. On reset set for thread 0, cleared for all other threads Thread operation is disabled. System thread can load or store thread state. Thread operation is enabled.	System_rw	Thread	Branch
3	priv	Privilege level. On reset cleared. 1 System privilege 2 Application privilege	System_rw App_r	Thread	Branch
5:4	state	Thread State. On reset set to "executing" for thread0, set to "idle" for all other threads. 1 Idle 2 reserved 3 Waiting 4 Executing	System_rw	Thread	Branch

7:6	bias	Thread execution bias. Higher value gives a bias to the corresponding thread for dispatching instructions. A high bias guarantees a higher dispatch rate, but the exact rate is determined by bias of other active threads	System_rw	Thread	Pipe
8	Memstep1	Memory step1- Unaligned memory reference instructions which cross L1 cache block boundry require two L1 cache cycles to complete. Indicates the first step of a load or store memory reference instruction has completed. For IO space reads, indicates that the data is available. Memory Reference Staging Register (MRSR) contains the special state when Memstep1 is set.	App_rw	Thread	Mem
9	endian	Endian Mode- On reset cleared. 1 little endian 2 big endian	System_rw App_r	Proc	Mem
10	align	Alignment check- When clear, unaligned memory references are allowed. When set, all un-aligned memory references result in unaligned data reference fault. On reset cleared.	System_rw App_r	Proc	Mem
11	iaddr	Instruction address translation enable. On reset cleared. 1 disabled 2 enabled	System_rw App_r	Proc	Branch

12	daddr	Data address translation enable. On reset cleared. 1 disabled 2 enabled	System_r w App_r	Proc	Mem
13	see	Enable Software event enqueue for application privilege for corresponding thread. When executing at system privilege sw events are always enabled. 1 Disabled- Corresponding thread, when executing at application privilege can not directly enqueue sw events. 2 Enabled – Corresponding thread, when executing at application privilege can directly enqueue sw events through the enqueue sw event control register	System_r w App_r	Threa d	Branc h
14	dbg	Debug enable. On reset cleared. 0- Disabled- debug mode disabled 1- Enabled- debug mode enabled	System_r w	Proc	Branc h
15	Reserved				

23:1 6	mod[7:0]]	GP Registers Modified. Cleared on reset.	App_rw	Thread	Pipe
		bit <u>modified for registers</u>			
		8 registers 0-15			
		9 registers 16-31			
		10 registers 32-47			
		11 registers 48-63			
		12 registers 63-79			
		13 registers 80-95			
		14 registers 96-111			
		15 registers 112-127			

ID Register

63	32	39	32	31	16	15	8	7	0
				thread_id		id		type	

Bit	Field	Description	Privilege	Per
7:0	type	Processor type and revision[7:0]	read only	Proc
15:8	id	Processor ID[7:0]- Virtual processor number	read only	Thread
31:1 6	thread_id	Virtual Thread Number[15:0]	System_rw App_ro	Thread

Instruction Pointer Register

63	3	0
4	1	

Doubleword	Mask2:0	
------------	---------	--

Specifies the 64 bit virtual address of the next instruction to be executed.

Bit	Field	Description	Privilege	Per
63:4	Doubleword	Doubleword address of instruction doubleword	app	thread
4:1	Mask2:0	Indicates which instructions within instruction doubleword remain to be executed. • Bit0- first instruction doubleword 0, bit[40:00] • Bit1- second instruction doubleword 0, bit[81:41] • Bit2- third instruction doubleword 0, bit[122:82]	app	thread
0	reserved			

System Exception Status Register

63	51	36	35	32	31	0
52						
	detail	etype				tstate

Bit	Field	Description	Privilege	Per
-----	-------	-------------	-----------	-----

31:0	tstate	Thread State register at time of exception	read only	Thread
35:3 2	etype	Exception Type 1 none 2 event 3 timer event 4 SW event 5 reset 6 SystemCall 7 Single Step 8 Protection Fault 9 Protection Fault, system call 10 Memory reference Fault 11 HW fault 12 others	read only	Thread
51:3 6	detail	Fault details- Valid for the following exception types: - Memory reference fault details (type 5) 1 None 2 page fault 3 waiting for fill 4 waiting for empty 5 waiting for completion of cache miss 6 memory reference error - event (type 1) - Specifies the 16 bit event number		

Application Exception Status Register

63	51	35	31
52	36	32	0

	detail	etype	tstate
--	--------	-------	--------

Bit	Field	Description	Privilege	Permission
31:0	tstate	Thread State register at time of exception	read only	Thread
35:3 2	etype	Exception Type 1 none 2 event 3 timer event 4 SW event 5 Others	read only	Thread
51:3 6	detail	Protection Fault details- Valid for the following exception types: • event (type 1) - Specifies the 16 bit event number		

System Exception IP

63	61	4	3	1	0
62					
mask5:4	Quadword			Mask2:0	

Address of instruction corresponding to signaled exception to system privilege.

Bit	Field	Description	Privilege	Permission
61:5	Doubleword	Quadword address of instruction doubleword with address[63:62] equal zero.	system	thread

3:1	Mask2:0	Indicates which instructions within instruction doubleword remain to be executed. • Bit0- first instruction doubleword 0, bit[40:00] • Bit1- second instruction doubleword 0, bit[81:41] • Bit2- third instruction doubleword 0, bit[122:82]	system	thread
0	reserved			

Address of instruction corresponding to signaled exception.

Application Exception IP

63	62	61	4	3	1	0
mask5:4	Quadword				Mask2:0	

Address of instruction corresponding to signaled exception to application privilege.

Bit	Field	Description	Privilege	Per
61:5	Doubleword	Quadwork address of instruction doubleword with address[63:62] equal zero.	system	thread
3:1	Mask2:0	Indicates which instructions within instruction doubleword remain to be executed. • Bit0- first instruction doubleword 0, bit[40:00] • Bit1- second instruction doubleword 0, bit[81:41] • Bit2- third instruction doubleword 0, bit[122:82]	system	thread

0	reserved			
---	----------	--	--	--

Exception Mem Address

63		0
Address		

Address of memory reference that signaled exception. Valid only for memory faults. Holds the address of the pending memory operation when the Exception Status register indicates memory reference fault, waiting for fill or waiting for empty.

Instruction Seg Table Pointer (ISTP), Data Seg Table Pointer (DSTP)

63	32	31	6	5	1	0
		reserved		ste number		field

Utilized by ISTE and DSTE registers to specify the ste and field that is read or written.

Bit	Field	Description	Privilege	Per
0	field	Specifies the low (0) or high (1) portion of Segment Table Entry	system	thread
5:1	ste number	Specifies the STE number that is read into STE Data Register.	system	thread

Instruction Segment Table Entry (ISTE), Data Segment Table Entry (DSTE)

63	
0	

data

When read the STE specified by ISTE register is placed in the destination general register. When written, the STE specified by ISTE or DSTE is written from the general purpose source register. The format of segment table entry is specified in "Virtual Memory and Memory System," hereof, section titled Translation Table organization and entry description.

Instruction or Data Level1 Cache Tag Pointer (ICTP, DCTP)

63	15	13	6	1
32	14	7	2	0
	reserved	index	bank	

Specifies the Instruction Cache Tag entry that is read or written by the ICTE or DCTE.

Bit	Field	Description	Privilege	Per
6:2	bank	Specifies the bank that is read from Level1 Cache Tag Entry. The first implementation has valid banks 0x0-f.	system	thread
13:7	index	Specifies the index address within a bank that is read from Level1 Cache Tag Entry	System	thread

Instruction or Data Level1 Cache Tag Entry (ICTE, DCTE)

63
0
data

When read the Cache Tag specified by ICTP or DCTP register is placed in the destination general register. When written, the Cache Tag specified by ICTP or DCTP is written from the general purpose source register. The format of cache tag entry is specified in "Virtual Memory and Memory System," hereof, section titled Translation Table organization and entry description.

Memory Reference Staging Register (MRSR0, MRSR1)

63	0
data	

Memory Reference Staging Registers provide a 128 bit staging register for some memory operations. MRSR0 corresponds to low 64 bits.

Instruction	Condition	Usage
Load, LoadPair, Store, StorePair	Aligned access or aligned access which does not cross a level1 cache block	Not used
Load, LoadPair	Unaligned access which crosses a level1 cache block	Holds the portion of the load from the lower addressed cache block which the upper address cache block is accessed
Store, StorePair	Unaligned access which crosses a level1 cache block	Not used
Load, LoadPair	IO Space	Holds the value of IO space read

Enqueue SW Event Register

63	15	0
16		
reserved		Eventnum

Writing to the enqueue SW Event register en-queues an event onto the Event Queue to be handled by a thread.

Bit	Field	Description	Privilege	Per
15:0	Eventnum	Number of clock cycles since processor reset	See ese	proc
63:16	reserved	Reserved for expansion of event number	See ese	proc

Timers And Performance Monitor

All timer and performance monitor registers are accessible at application privilege.

Clock

63
0
clock

Bit	Field	Description	Privilege	Per
63:0	clock	Number of clock cycles since processor reset	app	proc

Instructions executed

63	32	31	0
		count	

Bit	Field	Description	Privilege	Per
31:0	count	Saturating count of the number of instruction executed. Cleared on read. Value of all 1's indicates that the count has overflowed.	app	thread

Thread execution clock

63	31
32	0
active	

Bit	Field	Description	Privilege	Per
31:0	active	Saturating count of the number of cycles the thread is in active-executing state. Cleared on read. Value of all 1's indicates that the count has overflowed.	app	thread

Wait Timeout Counter

63	31
32	0
	timeout

Bit	Field	Description	Privilege	Per
31:0	timeout	Count of the number of cycles remaining until a timeout event is signaled to thread. Decrements by one, each clock cycle.	app	thread

INSTRUCTION SET OVERVIEW**OVERALL CONCEPTS****Thread Is Basic Control Flow Of Instruction Execution**

The thread is the basic unit of control flow for illustrated SEP embodiment. It can execute multi-threads concurrently in a software transparent manner. Threads can communicate through shared memory, producer-consumer memory operations or events independent of whether they are executing on the same physical processor and/or active at that instant. The natural method of building SEP applications is through communicating threads. This is also a very natural style for Unix and Linux. See "Generalized Events and Multi-Threading," hereof, and/or the discussions of individual instructions for more information.

Instruction Grouping And Ordering

The SEP architecture requires the compiler to specify what instructions can be executed within a single cycle for a thread. The instructions that can be executed within a single cycle for a single thread are called an instruction group. An instruction group is delimited by setting the stop bit, which is present in each instruction. The SEP can execute the entire group in a single cycle or can break that group up into multiple cycles if necessary because of resource constraints, simultaneous multi-thread or event recognition. There is no limit to the number of instructions that can be specified within an instruction group. Instruction groups do not have any alignment requirements with respect to instruction doublewords.

In the illustrated embodiment, branch targets must be the beginning of an instruction doubleword; other embodiments may vary in this regard.

Result Delay

Instruction result delay is visible to instructions and thus the compiler. Most instructions have no result delay, but some instructions have 1 or 2 cycle result delay. If an instruction has a zero result delay, the result can be used during the next instruction grouping. If an instruction has a result delay of one, the result of the instruction can be first utilized after one instruction grouping. In the rare occurrence that no instruction that can be scheduled within an instruction grouping, a one instruction grouping consisting of a NOP (with stop bit set to delineate end of group) can be used. The NOP instruction does not utilize any processor execution resources.

Predication

In addition to general purpose register file, SEP contains a predicate register file. In the illustrated embodiment, each predicate register is a single bit (though, other embodiments may vary in this regard). Predicate registers are set by compare and test instructions. In the illustrated embodiment, every SEP instruction specifies a predicate register number within its encoding

(and, again, other embodiments may vary in this regard). If the value of the specified predicate register is true the instruction is executed, otherwise the instruction is not executed. The SEP compiler utilizes predicates as a method of conditional instruction execution to eliminate many branches and allow more instructions to be executed in parallel than might otherwise be possible.

Operand Size And Elements

Most SEP instructions operate uniformly across a single word, two $\frac{1}{2}$ words, four $\frac{1}{4}$ words and eight bytes. An element is a chunk of the 64 bit register that is specified by the operand size.

Low Power Instruction Set

The instruction set is organized to minimize power consumption- accomplishing maximal work per cycle rather than minimal functionality to enable maximum clock rate.

Exceptions

Exceptions are all handled through the generalized event architecture. Depending on how event recognition is set up, a thread can handle its own events or a designated system thread can handle an events. This event recognition can be set up on an individual event basis.

Just In Time Compilation Parallelism

The SEP architecture and instruction set is a powerful general purpose 64 bit instruction set. When couple with the generalized event structure, high performance virtual environments can be set up to execute Java or ARM for example.

INSTRUCTION CLASSES

This section will be expanded to overview the instruction classes

Memory Access

Instruction	Description
Load	Load memory operand into general purpose register
Store	Store memory operand from general purpose register
Load Pair	Load two word memory operand into two general purpose registers
Store Pair	Store two word memory operand from two general purpose registers
Prefetch	Hint to memory system that memory address will be required soon
Translation probe	Indicates whether a specified System Address has access privilege in this thread in a specific thread context (privileged)
Load predicate	Loads the predicate registers from memory
Store predicate	Stores the predicate registers to memory
Empty	Usually executed by the consumer of a memory object to indicate that object at the corresponding address has been consumed
Fill	Usually executed by the producer of a memory object to indicate that the object at the corresponding address has been consumed.
Cache Allocate	Software based cache allocation.

Compare And Test

Parallel compares eliminates the artificial delay in evaluating complex conditional relationships.

Instruction	Description
CMP	Compare integer word and set predicate registers

CMPMS	Compare multiple integer elements and set predicate register based on summary of compares
CMPM	Compare multiple integer elements and set general purpose register with the result of compares
FCMP	Compare floating point element and set predicate registers
FCMPM	Compare multiple floating point elements and set general purpose register with the result of compares
FCLASS	Classify floating point elements and set predicate registers based on result
FCLASSM	Classify multiple floating point elements and set general purpose register based on result.
TESTB	Test specified bit and set predicate registers based on result
TESTBM	Test specified bit of each element and set general purpose register based on result.

Operate And Immediate

Instruction	Description
ADD	Add integer elements
LOGIC	Logical and, or, xor or andc between integer elements
SHIFTBYTE	Shift integer elements the specified number of bytes.
SHIFT	Shift integer elements the specified number of bits.
PACK	Two registers are concatenated and elements packed into a single destination register
UNPACK	Each element of source is unpacked to the next larger size.
EXTRACT	A field is extract from each element and right justified in each element of destination

DEPOSIT	Bit field for each element of 2 nd source is merged with first source
SPLAT	Contents a source element are extended and placed in each element of destination.
POP	Count the number of bits set to value 1.
FINDF	For each element find the first chunk that match criterion.
MUL	Multiply integer elements
MULSEL	Multiply integer elements and select result field for each element
MIN/MAX	integer minimum and maximum for each element
AVE	Add the elements from two sources and calculate average for each element
FMIN/FMAX	Floating point minimum and maximum
FROUND	Round floating point elements
CONVERT	Convert to or from floating point elements to integer elements
EST	Floating point estimate functions including reciprocal, reciprocal square root, log and power.
FADD	Floating point addition.
FMULADD	Multiply and add floating point elements
MULADD	Multiply and add integer elements
MULSUM	Multiply and sum integer elements
SUM	Sum integer elements
MOVI	Integer and floating point move immediate, 21 or 64 bits
Control field	Modifies specific control register fields
MOVECTL	Move to or from control register and general register.

Branch, SW Events

Instruction	Description
-------------	-------------

BR	Branch instruction
Event	Poll the event queue
SWEVENT	Initiate a software event

INSTRUCTION SET

MEMORY ACCESS INSTRUCTIONS

LOAD REGISTER LOAD

42 38	37 36	35	34 28	27 26	25 24	23	22	21	20 14	13 7	6 1	0
0000 0	lsiz	0	dreg	*	cach e	ls2	u	0	ireg	breg	ps	stop
0000 0	lsiz	1	dreg	disp[9 :8]	cach e	ls2	u	disp[7:0]		breg	ps	stop
0000 1	cach e	0	dreg	*	01	0	u	0	ireg	breg	ps	stop
0000 1	cach e	1	dreg	disp[9 :8]	01	0	u	disp[7:0]		breg	ps	stop

Format: ps LOAD.lsize.cache dreg, breg.u, ireg {,stop} register index form
 ps LOAD.lsize.cache dreg , breg.u, disp, {,stop} displacement form
 ps LOAD.splat32.cache dreg, breg.u, ireg {,stop} splat32 register index form
 ps LOAD.splat32.cache dreg , breg.u, disp, {,stop} splat32 displacement form

Description: A value consisting of lsize is read from memory starting at the effective address.
 The lsize value is then sign or zero extended to word size and placed in dreg
 (destination register). Splat32 form loads a 1/2 word into both the low and high 1/2
 words of dreg.
 For the register index form, the effective address is calculated by adding breg (base

register) and ireg (index register). For the displacement form, the effective address is calculated by adding breg (base register) and disp (displacement) shifted by lsize:

byte: $EA = breg[63:0] + disp[9:0]$
 $\frac{1}{4}$ word: $EA = breg[63:0] + (disp[9:0] < 1)$
 $\frac{1}{2}$ word: $EA = breg[63:0] + (disp[9:0] < 2)$
word: $EA = breg[63:0] + (disp[9:0] < 3)$
Double-word: $EA = breg[63:0] + (disp[9:0] < 4)$

Both aligned and unaligned effective address are supported. Aligned and unaligned access which does not cross an L1 cache block boundry execute in a single cycle. Unaligned access requires a second cycle to access the second cache block.. Aligned effective address is recommended where possible, but unaligned effective addressing is statistically high performance.

lsize	Offset with respect to L1 block		Probability within L1 block	
	within	across	random access	sequential access
Byte	0-127	none	100%	100%
$\frac{1}{4}$ word	0-126	127	99%	98%
$\frac{1}{2}$ word	0-124	125-127	98%	96%
word	0-120	121-127	95%	94%
double word	0-112	113-127	88%	88%

Operands and Fields:

- ps* The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
- stop* 0 Specifies that an instruction group is not delineated by this instruction.

	1	Specifies that an instruction group is delineated by this instruction.
<i>cache</i>	0	read only with reuse cache hint
	1	read/write with reuse cache hint
	2	read only with no-reuse cache hint
	3	read/write with no-reuse cache hint
<i>u</i>	0	Base register (breg) is not modified
	1	Write base register (breg) with base plus index register (or displacement) address calculation..
<i>lsiz</i> [2:0]	0	Load byte and sign extend to word size
	1	Load 1/4 word and sign extend to word size
	2	Load 1/2 word and sign extend to word size
	3	Load word
	4	Load byte and zero extend to word size
	5	Load 1/4 word and zero extend to word size
	6	Load 1/2 word and zero extend to word size
	7	Load pair into (dreg[6:1],0) and (dreg[6:1],1)
<i>ireg</i>		Specifies the index register of the instruction.
<i>breg</i>		Specifies the base register of the instruction.
<i>disp</i> [9:0]		Specifies the two's complement displacement constant (10 bits) for memory reference instructions.
<i>dreg</i>		Specifies the destination register of the instruction.

Exceptions: TLB faults
Page not present fault

STORE TO MEMORY STORE

42	37	35	34	27	25	24	23	22	21	20	13	6	0
38	36		28	26						14	7	1	

0000 1	size	0	s1reg	*	ru	0	sz2	u	0	ireg	breg	predicat e	stop
0000 1	size	1	s1reg	disp[9 :8]	ru	0	sz2	u	disp[7:0]		breg	predicat e	stop

Format: ps STORE.size.ru s1reg, breg.u, ireg {,stop} register index form
 ps STORE.size.ru s1reg , breg.u, disp,{,stop} displacement form

Description: A value consisting of least significant ssize bits of the value in s1reg is written to memory starting at the effective address. For the register index form, the effective address is calculated by adding breg (base register) and ireg (index register). For the displacement form, the effective address is calculated by adding breg (base register) and disp (displacement) shifted by lsize:

byte: EA = breg[63:0] + disp[9:0]

1/4 word: EA = breg[63:0] + (disp[9:0] < 1)

1/2 word: EA = breg[63:0] + (disp[9:0] < 2)

word: EA = breg[63:0] + (disp[9:0] < 3)

Double-word: EA = breg[63:0] + (disp[9:0] < 4)

Both aligned and unaligned effective address are supported. Aligned and unaligned access which does not cross an L1 cache block boundary execute in a single cycle. Unaligned access requires a second cycle to access the second cache block.. Aligned effective address is recommended where possible, but unaligned effective addressing is statistically high performance.

	Offset with respect to L1 block		Probability within L1 block	
	within	across	random access	sequential access
lsize				
Byte	0-127	none	100%	100%
1/4 word	0-126	127	99%	98%
1/2 word	0-124	125-127	98%	96%

word	0-120	121-127	95%	94%
double	0-112	113-127	88%	88%
word				

Operands and Fields:

- ps* The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
- stop* 0 Specifies that an instruction group is not delineated by this instruction.
1 Specifies that an instruction group is delineated by this instruction.
- ru* 0 resuse cache hint
1 no-reuse cache hint
- u* 0 Base register (breg) is not modified
1 Write base register (breg) with base plus index register (or displacement) address calculation..
- size[2:0]* 0 Store byte
1 Store 1/4 word
2 Store 1/2 word
3 Store word
4-6 reserved
7 Store register pair (dreg[6:1],0) and (dreg[6:1],1) into memory
- ireg* Specifies the index register of the instruction.
- breg* Specifies the base register of the instruction.
- disp* Specifies the two's complement displacement constant (10 bits) for memory reference instructions
- s1reg* Specifies the register that contains the first operand of the instruction.

Exceptions: TLB faults

Page not present fault

CACHE OPERATION CACHEOP

42 38	37 35	34	28	27 24	23	22	22	20	14	13	7	6	1	0
00001	010	dreg	1***	*	0	1		*	breg	ps	stop			
00001	010	dreg	1***	*	1	1		slreg	breg	ps	stop			

Format: ps.CacheOp.pr dreg = breg {,stop} address form
 ps.CacheOp.pr dreg = breg,slreg {,stop} address-source form

Description: Instructs the local level2 and level2 extended cache to perform an operation on behalf of the issuing thread. On multiprocessor systems these operations can span to non-local level2 and level2 extended caches. Breg specifies the operation and address corresponding to the operation. The optional slreg specifies an additional source operand which depends on the operation. The return value specified by the issued CacheOp is placed into dreg. CacheOp always causes the corresponding thread to transition from executing to wait state.

Table 2- CacheOp breg format

	63 14	13 0
Cache Allocate	Page address	0x0000
reserved	*	0x0001-0x3fff

Table 3- CacheOp operand description

	Address form privilege	Source form privilege	sreg	dreg
Cache Allocate	system	reserved	reserved	See Table 4
reserved	reserved	reserved	reserved	

Table 4- Cache Allocate dreg description

	63 30	29 14	13 0
Success	reserved	L2E page number	0x0000
Already allocated	Reserved	L2E page number	0x0001
No space available	*	*	0x0002
reserved	*	*	0x0003-0x3fff

Operands and Fields:

- ps* The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
- stop* 0 Specifies that an instruction group is not delineated by this instruction.
 1 Specifies that an instruction group is delineated by this instruction.
- s1reg* Specifies the source register for the address-source version of CacheOp instruction.
- dreg* Specifies the destination register for the CacheOp instruction.

Exceptions:

Privilege exception when accessing system control field at application privilege level.

OPERATE INSTRUCTIONS

Most operate instructions are very symmetrical, except for the operation performed.

ADD INTEGER OPERATIONS ADD, SUB, ADDSATU, ADDSAT, SUBSATU, SUBSAT, RSUBSATU, RSUBSAT, RSUB

Figure 43 depicts a core 12 constructed and operated as discussed elsewhere herein in which the functional units 12A, here, referred to as ALUs (arithmetic logic units), execute selected arithmetic operations concurrently with transposes.

In operation, arithmetic logic units 12A of the illustrated core 12 execute conventional arithmetic instructions, including unary and binary arithmetic instructions which specify one or more operands 230 (e.g., longwords, words or bytes) contained in respective registers by storing results of the designated operations in in a single register 232, e.g., typically in the same format as one or more of the operands (e.g., longwords, words or bytes). An example of this is shown in the upper right of Figure 43 and more examples are shown in **Figures 7–10**.

The illustrated ALUs, however, execute such arithmetic instructions that include a *transpose* (T) parameter (e.g., as specified, here, by a second bit contained in the *addop* field—but, in other embodiments, as specified elsewhere and otherwise) by transposing the results and storing them across multiple specified registers. Thus, as noted below, when the value of the *T* bit of the *addop* field is 0 (meaning no transpose), the result is stored in normal (i.e., non-transposed) register format, which is logically equivalent to a matrix row. However, when that bit is 1 (meaning transpose), the result is stored in transpose format, i.e., across multiple registers 234–240, which is logically equivalent to storing the result in a matrix column—as further discussed below. In this regard, the ALUs apportion results of the specified operations across multiple specified registers, e.g., at a common word, byte, bit or other starting point. Thus, for example, an ALU may execute an ADD (with transpose) operation that write the results, for example, as a one-quarter word column of four adjacent registers or, by way of further example, a byte column

of eight adjacent registers. The ALUs similarly execute other arithmetic operations—binary, unary or otherwise—with such concurrent transposes.

Logic gates, timing, and the other structural and operational aspects of operation of the ALUs 12E of the illustrated embodiment effecting arithmetic operations with optional transpose in response to the aforesaid instructions may be implemented in the conventional manner of known in the art as adapted in accord with the teachings hereof.

42 38	37 36	35	34 28	27	26 22	21	20	14	13 7	6 1	0
01010	osize	0	dreg	0	addop	0	s2reg	s1reg	predicate	stop	
01010	osize	1	dreg	0	addop	immediate8		s1reg	predicate	stop	
00010	osize	0	dreg	immediate14				s1reg	predicate	stop	

Format: ps.addop.T.osize. dreg = s1reg, s2reg {,stop} register form
 ps.addop.T.osize dreg = s1reg, immediate8, {,stop} immediate form
 ps.add.T.osize dreg= s1reg, immediate14 {,stop} long immediate form

Description: The two operands are operated on as specified by addop and osize fields and the result placed in destination register dreg. The add instruction processes a full 64 bit word as a single operation or as multiple independent operations based on the natural size boundaries as specified in the osize field and illustrated in **Figures 7–10**.

Operands and Fields:

addop

addop[5:0]	Mnemonic	Description	Register usage
0T000	ADD	signed add	dreg=s1reg + s2reg dreg=s1reg + immediate8
0T001		reserved	
0T010	ADDSAT	signed saturated add	dreg=s1reg + s2reg dreg=s1reg + immediate
0T011	ADDSAT U	unsigned saturated add	dreg=s1reg + s2reg dreg=s1reg + immediate
0T100	SUB	signed subtract	dreg=s1reg - s2reg dreg=s1reg - immediate
0T101		reserved	
0T110	SUBSAT	signed saturated subtract	dreg=s1reg - s2reg dreg=s1reg - immediate
0T111	SUBSAT U	unsigned saturated subtract	dreg=s1reg - s2reg dreg=s1reg - immediate
10000	RSUB	reverse signed subtract	dreg=s2reg - s1reg dreg=immediate - s1reg
10001		reserved	
10010	RSUBSAT	reverse signed saturated subtract	dreg=s2reg - s1reg dreg=immediate - s1reg
10011	RSUBSAT U	reverse unsigned saturated subtract	dreg=s2reg - s1reg dreg=immediate - s1reg
10100	Addhigh	Take the carry out of unsigned addition and place it into result register	dreg=carry(s1reg + s2reg) dreg=carry(s1reg + immediate)
10101	Subhigh	Take the carry out of unsigned subtract and place it into result register	dreg=carry(s1reg - s2reg) dreg=carry(s1reg - immediate)
10110		Logic instructions	
11111		reserved for other instructions	

- ps* The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
- stop* 0 Specifies that an instruction group is not delineated by this instruction.
1 Specifies that an instruction group is delineated by this instruction.
- osize* 0 Eight independent byte operations
1 Four independent $\frac{1}{4}$ word operations
2 Two independent $\frac{1}{2}$ word operations
3 Single word operation
- immediate8* Specifies the immediate8 constant which is zero extended to operation size for unsigned operations and sign extended to operation size for signed operations. Applied independently to each sub operation.
- Immediate14* Specifies the immediate14 constant which is sign extended to operation size. Applied independently to each sub operation.
- s1reg* Specifies the register that contains the first source operand of the instruction.
- s2reg* Specifies the register that contains the second source operand of the instruction.
- dreg* Specifies the destination register of the instruction.

T (transpose)

Transpos e[0]	Mnemonic	Description
0	nt	Default. Store result in normal register format, which would be logically equivalent to a matrix row.

Transpose[0]	Mnemonic	Description
1	t	Store result in transpose format. Transpose format is logically equivalent to storing the result in a matrix column. Valid for osize equal 0 (byte operations) or 1 ($\frac{1}{4}$ word operations). For byte operations, the destination for each byte is specified by [dreg[6:3],byte[2:0]], where byte[2:0] is the corresponding byte in the destination. Thus only one byte in 8 contiguous registers is updated. For $\frac{1}{4}$ word operations, the destination for each $\frac{1}{4}$ word is specified by [dreg[6:2],qw[1:0]], where qw[1:0] is the corresponding $\frac{1}{4}$ word in the destination. Thus only one $\frac{1}{4}$ word in 4 contiguous registers is updated.

TRANSPOSE BITS TRAN

42 38	37 36	35	34 28	27 23	22	21	20 14	13 7	6 1	0
01010	mode	0	dreg	11000	mod e	1	s2reg	s1reg	predicate	stop
01101	01	qw	dreg	s3reg		s2reg		s1reg	predicate	stop

Format: $\text{ps.tran.mode dreg} = \text{s1reg, s2reg \{,stop\}}$ fixed form
 $\text{ps.tran.qw dreg} = \text{s1reg, s2reg, s3reg \{,stop\}}$ variable form

Description: For the fixed form, bits within each $\frac{1}{4}$ word (QW) or byte element are bit transposed based on mode to the dreg register. For the variable form, bits within each $\frac{1}{4}$ word (QW) or byte element are bit transposed based on qw and s3reg bit positions to the dreg register.

See Figures 11–16

mode

mode[2:0]	Mnemonic	Description
100	PackB	Within for the n^{th} bit in the m^{th} byte element: For bit $\text{dreg}[(n*8)+m] = \text{s1reg}[(m*8)+n]$
101	reserved	
110	VPackB	S2reg specifies the bit position within each byte of sreg for each byte within dreg. Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: For bit $\text{dreg}[(n*8)+m] = \text{s1reg}[(m*8)+\text{s2reg}[(m*8)+2:(m*8)]]$
111	reserved	
000	PackQW_Low	Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: • For bit $\text{dreg}[(n*16)+m] = \text{s1reg}[(m*16)+n]$

mode[2:0]	Mnemonic	Description
010	UnPackQW_Low	Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: For bit $\text{dreg}[(m*16)+n] = \text{s1reg}[(n*16)+m]$
001	PackQW_High	Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: For bit $\text{dreg}[(n*16)+m] = \text{s1reg}[(m*16)+n+8]$
011	UnPackQW_High	Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: For bit $\text{dreg}[(m*16)+n] = \text{s1reg}[(n*16)+m+8]$

qw

Qw[0]	Mnemonic	Description
0	VPackQW	Let $\text{sreg}[127:0] = (\text{s2reg}[63:0], \text{s1reg}[63:0])$ S3reg specifies the bit position within each QW of sreg for each byte within dreg. Within for the n^{th} bit in the m^{th} $\frac{1}{4}$ word element: For bit $\text{dreg}[(n*8)+m] = \text{sreg}[(m*16)+\text{s3reg}[(m*8)+3:(m*8)]]$

Qw[0]	Mnemonic	Description
1	VUnPackQW	Let $sreg[127:0] = (s2reg[63:0], s1reg[63:0])$ S3reg specifies which $\frac{1}{2}$ byte goes into each bit position of each QW of dreg. Within for the n^{th} bit in the m^{th} byte element: • For bit $dreg[(m*16)+n] = sreg[sreg3[(n*8)+3:(n*8)]+m]$

<i>stop</i>	0	Specifies that an instruction group is not delineated by this instruction.
	1	Specifies that an instruction group is delineated by this instruction.
<i>s1reg</i>		Specifies the register that contains the first source operand of the instruction.
<i>s2reg</i>		Specifies the register that contains the second source operand of the instruction.
<i>s3reg</i>		Specifies the register that contains the third source operand of the instruction.
<i>dreg</i>		Specifies the destination register of the instruction.

BINARY ARITHMETIC CODER LOOKUP

BAC

Figure 44 depicts a core 12 constructed and operated as discussed elsewhere herein in which the functional units 12A, here, referred to as ALUs (arithmetic logic units), execute processor-level instructions (here, referred to as BAC instructions) by storing to register(s) 12E value(s) from a JPEG2000 binary arithmetic coder lookup table.

More particularly, referring to the drawing, the ALUs 12A of the illustrated core 12 execute processor-level instructions, including JPEG2000 binary arithmetic coder table lookup instructions (BAC instructions) that facilitate JPEG2000 encoding and decoding. Such instructions include, in the illustrated embodiment, parameters specifying one or more function

values to lookup in such a table 208, as well as values upon which such lookup is based. The ALU responds to such an instruction by loading into a register in 12E (Figure 44) a value from a JPEG2000 binary arithmetic coder Qe-value and probability estimation lookup table.

In the illustrated embodiment, the lookup table is as specified in Table 7.7 of Tinku Acharya & Ping-Sing Tsai, "JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures", Wiley, 2005, reprinted in Appendix C hereof. Moreover, the functions are the Qe-value, NMPS, NLPS and SWITCH function values specified in that table. Other embodiments may utilize variants of this table and/or may provide lesser (or additional) functions. A further appreciation of the aforesaid functions may be appreciated by reference to the cited text, the teachings of which are incorporated herein by reference.

The table 208, whether from the cited text or otherwise, may be hardcoded and/or may, itself, be stored in registers. Alternatively or in addition, return values generated by the ALUs on execution of the instruction may be from an algorithmic approximation of such a table.

Logic gates, timing, and the other structural and operational aspects of operation of the ALUs 12E of the illustrated embodiment effecting storage of value(s) from a JPEG2000 binary arithmetic coder lookup table in response to the aforesaid instructions implement the lookup table specified in Table 7.7 of Tinku Acharya & Ping-Sing Tsai, "JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures", Wiley, 2005, which table is incorporated herein by reference and a copy of which is attached **Exhibit D** hereto. The ALUs of other embodiments may employ logic gates, timing, and other structural and operational aspects that implement other algorithmic such tables.

A more complete understanding of an instruction for effecting storage of value(s) from a JPEG2000 binary arithmetic coder lookup table according to the illustrated embodiment may be attained by reference to the following specification of instruction syntax and effect:

42	37	36	35	34		27	23	21	20		13	6	0
38				28		24	22		14		7	1	

01010	*	*	0	dreg	1001	type	1	s2reg	000010 0	predicate	stop
-------	---	---	---	------	------	------	---	-------	-------------	-----------	------

Format: ps.bac.fs dreg = s2reg {,stop} register form

Description: A table lookup, as specified by *type*, of the value range 0-46 in s2reg is placed into corresponding element of dreg. Returned values for s2reg outside the value range are undefined.

Operands and Fields:

type

type	Mnemonic	Description
00	bac.qe	MQ-coder binary arithmetic coder probability function. Returns a 16 bit value. See table 7.7 of Tinku Acharya & Ping-Sing Tsai, "JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures", Wiley, 2005
01	bac.nmps	NMPS function. (See Acharya, et al, <i>supra</i>). Returns a value between 0-46.
10	bac.nlps	NLPS function. (See Acharya, et al, <i>supra</i>). Returns a value between 0-46.
11	bac.switch	SWITCH function. (See Acharya, et al, <i>supra</i>). Returns 0x0 or 0x1.

ps The predicate source register in element 12E that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).

<i>stop</i>	0	Specifies that an instruction group is not delineated by this instruction.
	1	Specifies that an instruction group is delineated by this instruction.

<i>S2reg</i>	Specifies the register in element 12E that contains the second source operand of the instruction.
<i>dreg</i>	Specifies the destination register in element 12E of the instruction.

BIT PLANE STRIPE COLUMN CODE BPSCCODE

Figure 45 depicts a core 12 constructed and operated as discussed elsewhere herein in which the functional units 12A, here, referred to as ALUs (arithmetic logic units), execute processor-level instructions (here, referred to as BPSCCODE instructions) by encoding a stripe column of values in registers 12E for bit plane coding within JPEG2000 EBCOT (or, put another way, bit plane coding in accord with the EBCOT scheme). EBCOT stands for “Embedded Block Coding with Optimal Truncation.” Those instructions specify, in the illustrated embodiment, four bits of the column to be coded and the bits immediately adjacent to each of those bits. The instructions further specify the current coding state (here, in three bits) for each of the four column bits to be encoded.

As reflected by element 210 of the drawing, according to one variant of the instruction (as determined by a so-called “cs” parameter), the ALUs 12E of the illustrated embodiment respond to such instructions by generating and storing to a specified register the column coding specified by a “pass” parameter of the instruction. That parameter, which can have values specifying significance propagation pass (SP), a magnitude refinement pass (MR), a cleanup pass, and a combined MR and CP pass, determines the stage of encoding performed by the ALUs 12E in response to the instruction.

As reflected by element 212 of the drawing, according to another variant of the instruction (again, as determined by the “cs” parameter), the ALUs 12E of the illustrated embodiment respond to an instruction as above by alternatively (or in addition) generating and storing to a register updated values of the coding state, e.g., following execution of a specified pass.

Logic gates, timing, and other structural and operational aspects of ALUs 12E of the illustrated embodiment for effecting the encoding of stripe columns in response to the aforesaid instructions implement an algorithmic/methodological approach disclosed in Amit Gupta, Saeid Nooshabadi & David Taubman, "Concurrent Symbol Processing Capable VLSI Architecture for Bit Plane Coder of JPEG2000", IEICE Trans. Inf. & System, Vol. E88-D, No. 8, August 2005, the teachings of which are incorporated herein by reference, and a copy of which is attached **Exhibit D** hereto. The ALUs of other embodiments may employ logic gates, timing, and other structural and operational aspects that implement other algorithmic and/or methodological approaches.

A more complete understanding of an instruction for encoding a stripe column for bit plane coding within JPEG2000 EBCOT according to the illustrated embodiment may be attained by reference to the following specification of instruction syntax and effect:

42 38	37 36	35	34 28	27 23	22	21	20 14	13 7	6 1	0
01010	pass	0	dreg	11010	cs	1	s2reg	s1reg	predicate	stop

Format: ps.bpsccode.pass.cs dreg = s1reg, s2reg {,stop} register form

Description: Used to encode a 4 bit stripe column for bit plane coding within JPEG2000 EBCOT(Embedded Block Coding with Optimized Truncation). (See Amit Gupta, Saeid Nooshabadi & David Taubman, “Concurrent Symbol Processing Capable VLSI Architecture for Bit Plane Coder of JPEG2000”, IEICE Trans. Inf. & System, Vol. E88-D, No. 8, August 2005). S1reg specifies the 4 bits of the column from registers 12E (Figure 45) to be coded and the bits immediately adjacent to each of these bits. S2reg specifies the current coding state (3 bits) for each the 4 column bits. Column coding as specified by pass and cs is returned in dreg, a destination in registers 12E.

See Figures 17–18.

Operands and Fields:

<i>ps</i>		The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
<i>pass</i>	0	Significance propagation pass (SP)
	1	Magnitude refinement pass (MR)
	2	Cleanup pass (CP)
	3	combined MR and CP
<i>cs</i>	0	Dreg contains column coding, CS, D pairs.
	1	Dreg contains new value of state bits for column.
<i>stop</i>	0	Specifies that an instruction group is not delineated by this instruction.
	1	Specifies that an instruction group is delineated by this instruction.
<i>s1reg</i>		Specifies the register in element 12E (Figure 45) in that contains the first source operand of the instruction.
<i>S2reg</i>		Specifies the register in element 12E that contains the first source operand of the instruction.
<i>dreg</i>		Specifies the destination register in element 12E of the instruction.

VIRTUAL MEMORY AND MEMORY SYSTEM

SEP utilizes a novel Virtual Memory and Memory System architecture to enable high performance, ease of programming, low power and low implementation cost. Aspects include:

- 64 bit Virtual Address (VA)
- 64 bit System Address (SA). As we shall see this address has different characteristics than a standard physical address.

- Segment model of Virtual Address to System Address translation with a sparsely fill VA or SA.
- The VA to SA translation is on a segment basis. The System addresses are then cached in the memory system. So a SA that is present in the memory system has an entry in one of the levels of cache. An SA that is not present in any cache (and the memory system) is then not present in the memory system. Thus the memory system is filled sparsely at the page (and subpage) granularity in a way that is natural to software and OS, without the overhead of page tables on the processor.
- All memory is effectively managed as cache, even though off chip memory utilizes DDR DRAM. The memory system includes two logical levels. The level1 cache, which is divided into separate data and instruction caches for optimal latency and bandwidth. The level2 cache includes an on chip portion and off chip portion referred to as level2 extended. As a whole the level2 cache is the memory system for the individual SEP processor(s) and contributes to a distributed all cache memory system for multiple SEP processors. The multiple processors do not have to be physically sharing the same memory system, chips or buses and could be connected over a network.

Some additional benefits of this architecture are:

- Directly supports Distributed Shared:
 - Memory (DSM)
 - Files (DSF)
 - Objects (DSO)
 - Peer to Peer (DSP2P)
- Scalable cache and memory system architecture
- Segments can easily be shared between threads
- Fast level1 cache since lookup is in parallel with tag access, no complete virtual to physical address translation or complexity of virtual cache.

VIRTUAL MEMORY OVERVIEW

Referring to **Figure 19**, virtual address is the 64 bit address constructed by memory reference and branch instructions. The virtual address is translated on a per segment basis to a system address which is used to access all system memory and IO devices. Table 6 specifies system address assignments. Each segment can vary in size from 2^{24} to 2^{48} bytes.

The virtual address is used to match an entry in the segment table. The matched entry specifies the corresponding system address, segment size and privilege. System memory is a page level cache of the System Address space. Page level control is provided in the cache memory system, rather at address translation time at the processor. The operating system virtual memory subsystem controls System memory on a page basis through L2 Extended Cache (L2E Cache) descriptors. The advantage of this approach is that the performance overhead of processor page tables and page level TLB is avoided.

When the address translation is disabled, the segment table is bypassed and all addresses are truncated to the low 32 bits and require system privilege.

CACHE MEMORY SYSTEM OVERVIEW

Introduction

With reference to **Figure 20**, the data and instruction caches of cores 12–16 the illustrated embodiment are organized as shown. L1 data and instruction caches are both 8-way associative. Each 128 byte block has a corresponding entry. This entry describes the system address of the block, the current l1 cache state, whether the block has been modified with respect to the l2 cache and whether the block has been referenced. The modified bit is set on each store to the block. The referenced bit is set by each memory reference to the block, unless the reuse hint indicates no reuse. The no-reuse hint allows the program to access memory locations once, without them displacing other cache blocks that will be reused. The referenced bit is periodically cleared by the L2 cache controller to implement a level1 cache working set algorithm. The modified bit is clear when the L2 cache control updates its data with the modified data in the block.

The level2 cache consists of an on-chip and off chip extended L2 Cache (L2E). The on-chip L2 cache, which may be self-contained on respective core, distributed among multiple cores, and/or contained (in whole or in part) on DDRAM on a “gateway” (or “IO bridge”) interconnects to other processors (e.g., of types other than those shown and discussed here) and/or systems, consists of the tag and data portions. Each 128 byte data block is described by a corresponding descriptor within the tag portion. The descriptor keeps track of cache state, whether the block has been modified with respect to L2E, whether the block is present in L1 cache, an LRU count to keep how often the block is being used by L1 and tag mode.

The off-chip DDR dram memory is called L2E Cache because it acts as an extension to the L2 cache. The L2E Cache may contained within a single device (e.g., a memory board with an integral controller (e.g., a DDR3 controller) or distributed among multiple devices associated with the respective cores or otherwise. Storage within the L2E cache is allocated on a page basis and data is transferred between L2 and L2E on a block basis. The mapping of System Address to a particular L2E page is specified by an L2E descriptor. These descriptors are stored within fixed locations in the System Address space and in external ddr2 dram. The L2E descriptor specifies the location with system memory or physical memory (e.g., an attached flash drive or other mounted storage device) that the corresponding page is stored. The operating system is responsible for initializing and maintaining these descriptors as part of the virtual memory subsystem of the OS. As a whole, the L2E descriptors specify the sparse pages of System Address space that are present (cached) in physical memory. If a page and corresponding L2E descriptor is not present in, then a page fault exception is signaled.

The L2 cache references the L2E descriptors to search for a specific system address, to satisfy a L2 miss. Utilizing the organization of L2E descriptors the L2 cache is required to access 3 blocks to access the referenced block, 2 blocks to traverse the descriptor tree and 1 block for the actual data. In order to optimize performance the L2 cache, caches the most recently used descriptors. Thus the L2E descriptor can most likely be referenced by the L2 directly and only a single L2E reference is required to load the corresponding block.

L2E descriptors are stored within the data portion of a L2 block as shown in Figure 85. The tag-mode bit within an L2 descriptor within the tag indicates that the data portion consists of 16 tags for Extended L2 Cache. The portion of the L2 cache which is used to cache L2E descriptors is set by OS and is normally set to one cache group, or 256 blocks for a 0.5m L2 Cache. This configuration results descriptors corresponding to 212 L2E pages being cached, this is equivalent to 256 Mbytes.

Although shown in use in connection with like processor modules (e.g., of the type detailed elsewhere herein), it will be appreciated that caching structures, systems and/or mechanisms according to the invention practiced with other processor modules, memory systems and/or storage systems, e.g., as illustrated Figure 31.

Advantages of embodiments utilizing caching of the type described herein are

- Caching of in memory directory
- Eliminating translation lookahead buffer (TLB) & TLB overhead at processor
- Single sparse address space enables single level store
- Encompassing dram, flash & cache as single optimized memory system
- Providing distributed coherence & working set management
- Affording Transparent state management
- Accelerating performance and lowering power by dynamically keeping data close to where it is needed and being able to utilize lower cost denser storage technologies.

Cache Memory System Continued

Level1 caches are organized as separate level1 instruction cache and level1 data cache to maximize instruction and data bandwidth. Both level1 caches are proper subsets of level2 cache. The overall SEP memory organization is shown in **Figure 20**. This organization is parameterized within the implementation and is scalable in future designs.

The L1 data and instruction caches are both 8 way associative. Each 128 byte block has a corresponding entry. This entry describes the system address of the block, the current L1 cache

state, whether the block has been modified with respect to the L2 cache and whether the block has been referenced. The modified bit is set on each store to the block. The referenced bit is set by each memory reference to the block, unless the reuse hint indicates no reuse. The no-reuse hint allows the program to access memory locations once, without them displacing other cache blocks that will be reused. The referenced bit is periodically cleared by the L2 cache controller to implement a level1 cache working set algorithm. The modified bit is clear when the L2 cache control updates its data with the modified data in the block.

The level2 cache includes an on-chip and off chip extended L2 Cache (L2E). The on-chip L2 cache includes the tag and data portions. Each 128 byte data block is described by a corresponding descriptor within the tag portion. The descriptor keeps track of cache state, whether the block has been modified with respect to L2E, whether the block is present in L1 cache, an LRU count to keep how often the block is being used by L1 and tag mode. The organization of the L2 cache is shown in **Figure 22**.

The off chip DDR DRAM memory is called L2E Cache because it acts as an extension to the L2 cache. Storage within the L2E cache is allocated on a page basis and data is transferred between L2 and L2E on a block basis. The mapping of System Address to a particular L2E page is specified by an L2E descriptor. These descriptors are stored within fixed locations in the System Address space and in external ddr2 dram. The L2E descriptor specifies the location within offchip L2E DDR DRAM that the corresponding page is stored. The operating system is responsible for initializing and maintaining these descriptors as part of the virtual memory subsystem of the OS. As a whole, the L2E descriptors specify the sparse pages of System Address space that are present (cached) in physical memory. If a page and corresponding L2E descriptor is not present in, then a page fault exception is signaled.

L2E descriptors are organized as a tree as shown in **Figure 24**.

Figure 25 depicts an L2E physical memory layout in a system according to the invention;

The L2 cache references the L2E descriptors to search for a specific system address, to satisfy a L2 miss. Utilizing the organization of L2E descriptors the L2 cache is required to access 3 blocks to access the referenced block, 2 blocks to traverse the descriptor tree and 1 block for the actual data. In order to optimize performance the L2 cache, caches the most recently used descriptors. Thus the L2E descriptor can most likely be referenced by the L2 directly and only a single L2E reference is required to load the corresponding block.

L2E descriptors are stored within the data portion of a L2 block as shown in **Figure 23**. The tag-mode bit within an L2 descriptor within the tag indicates that the data portion includes 16 tags for Extended L2 Cache. The portion of the L2 cache which is used to cache L2E descriptors is set by OS and is normally set to one cache group (SEP implementations are not required to support caching L2E descriptors in all cache groups. A minimum of 1 cache group is required), or 256 blocks for a 0.5m L2 Cache. This configuration results descriptors corresponding to 2^{12} L2E pages being cached, this is equivalent to 256 Mbytes.

Figure 21 illustrates overall flow of L2 and L2E operation. Psuedo-code summary of L2 and L2E cache operation:

```

L2_tag_lookup;
if (L2_tag_miss) {
    L2E_tag_lookup;
    if (L2E_tag_miss) {
        L2E_descriptor_tree_lookup;
        if (descriptor_not_present) {
            signal_page_fault;
            break;
        } else allocate_L2E_tag;
    }
    allocate_L2_tag;
    load_dram_data_into_l2

```

```

    }
    respond_data_to_l1_cache;

```

TRANSLATION TABLE ORGANIZATION AND ENTRY DESCRIPTION

Figure 26 depicts a segment table entry format in an SEP system according to one practice of the invention.

CACHE ORGANIZATION AND ENTRY DESCRIPTION

Figures 27–29 depict, respectively, L1, L2 and L2E Cache addressing and tag formats in an SEP system according to one practice of the invention.

The Ref (Referenced) count field is utilized to keep track of how often an L2 block is referenced by the L1 cache (and processor). The count is incremented when a block is move into L1. It can be used likewise in the L2E cache (vis-a-vis movement to the L2 cache) and the L1 cache (vis-a-vis references by the functional units of the local core or of a remote core).

In the illustrated embodiment, the functional or execution units, e.g., 12A–16A within the cores, e.g., 12–16, execute memory reference instructions that influence the setting of reference counts within the cache and which, thereby, influence cache management including replacement and modified block writeback. Thus, for example, the reference count set in connection with a typical or normal memory access by an execution unit is set to a middle value (e.g., in the example below, the value 3) when the corresponding entry (e.g., data or instruction) is brought into cache. As each entry in the cache is referenced, the reference count is incremented. In the background the cache scans and decrements reference counts on a periodic basis. As new data/instructions are brought into cache, the cache subsystem determines which of the already-cached entries to remove based on their corresponding reference counts (i.e., entries with lower reference counts are removed first).

The functional or execution units, e.g., 12A, of the illustrated cores, e.g., 12, can selectively force the reference counts of newly accessed data/instructions to be purposely set to low values,

thereby, insuring that the corresponding cache entries will be the next ones to be replaced and will not supplant other cache entries needed longer term. To this end, the illustrated cores, e.g., 12, support an instruction set in which at least some of the memory access instructions include parameters (e.g., the “no-reuse cache hint”) for influencing the reference counts accordingly.

In the illustrated embodiment, the setting and adjusting of reference counts—which, themselves, are maintained along with descriptors of the respective data in the so-called tag portions (as opposed to the so-called data portions) or the respective caches—is automatically carried out by logic within the cache subsystem, thus, freeing the functional units, e.g., 12A–16A, from having to set or adjust those counts themselves. Put another way, in the illustrated embodiment, execution of memory reference instructions (e.g., with or without the no-reuse hint) by the functional or execution units, e.g., 12A–16A, causes the caches (and, particularly, for example, the local L2 and L2E caches) to perform operations (e.g., the setting and adjustment of reference counts in accord with the teachings hereof) on behalf of the issuing thread. On multicore systems these operations can span to non-local level2 and level2 extended caches.

The aforementioned mechanisms can also be utilized, in whole or part, to facilitate cache-initiated performance optimization, e.g., independently of memory access instructions executed by the processor. Thus, for example, the reference counts for data newly brought into the respective caches can be set (or, if already set, subsequently adjusted) in accord with (a) the access rights of the acquiring cache, and (b) the nature of utilization of such data by the processor modules—local or remote.

By way of example, where a read-only datum brought into a cache is expected to be frequently updated on a remote cache (e.g., by a processing node with write rights), the acquiring cache can set the reference count low, thereby, insuring that (unless that datum is access frequently by the acquiring cache) the corresponding cache entry will be replaced, obviating the need for needless updates from the remote cache. Such setting of the reference count can be effected via memory access instructions parameters (as above) and/or “cache initiated” via automatic operation of the caching subsystems (and/or cooperating mechanisms in the operation system).

By way of further example, where a write-only datum maintained in a cache is not shared on a read-only (or other) basis in any other cache, the caching subsystems (and/or cooperating mechanisms in the operation system) can delay or suspend entirely signalling to the other caches or memory system of updates to that datum, at least, until the processor associated with the maintaining cache has stopped using the datum.

The foregoing can be further appreciated with reference to **Figure 47**, showing the effect on the L1 data cache, by way of non-limiting example, of execution of a memory “read” operation sans the no-reuse hint (or, put another way, with the re-use parameter set to “true”) by application, e.g., 200 (and, more precisely, threads thereof, labelled 200’’) on core 12. Particularly, the virtual address of the data being read, as specified by the thread 200’’, is converted to a system address, e.g., in the manner shown in Figure 19, by way of non-limiting example, and discussed elsewhere herein.

If the requested datum is in the L1 Data cache, an L1 Cache lookup and, more specifically, a lookup comparing that system address against the tag portion of the L1 data cache (e.g., in the manner paralleling that shown in **Figure 22** vis-a-vis the L2 Data cache) results in a hit that returns the requested block, page, etc. (depending on implementation) to the requesting thread. As shown in the right-hand corner of Figure 47, the reference count maintained in the descriptor of the found data is incremented in connection with the read operation.

On a periodic basis the reference count is decremented if it is still present in L1 (e.g., assuming it has not been accessed by another memory access operation). The blocks with the highest reference counts have the highest current temporal locality within L2 cache. The blocks with the lowest reference counts have been accessed the least in the near past and are targeted as replacement blocks to service L2 misses, i.e., the bringing in of new blocks from L2E cache. In the illustrated embodiment, the ref count for a block is normally initialized to a middling value of 3 (by way of non-limiting example), when the block is brought in from L2E cache. Of course, other embodiments may vary not only as to the start values of these counts, but also in the amount and timing of increases and decreases to them.

As noted above, setting of the referenced bit can be influenced programmatically, e.g., by application 200''', e.g., when it uses memory access instructions that have a no-reuse hint that indicates "no reuse" (or, put another way, a reuse parameter set to "false"), i.e., that the referenced data block will not be reused (e.g., in the near term) by the thread. For example, in the illustrated embodiment, if the block is brought into a cache (e.g., the L1 or L2 caches) by a memory reference instruction that specifies no-reuse, the ref count is initialized to a value of 2 (instead of 3 per the normal case discussed above)—and, by way of further example, if that block is already in cache, its reference count is not incremented as a result of execution of the instruction (or, indeed, can be reduced to, say, that start value of 2 as a result of such execution). Again, of course, other embodiments may vary in regard to these start values and/or in setting or timing of changes in the reference count as a result of execution of a memory access instruction with the no-reuse hint.

This can be further appreciated with reference to **Figure 48**, which parallels Figure 47 insofar as it, too, shows the effect on the data caches (here, the L1 and L2 caches), by way of non-limiting example, of execution of a memory "read" operation that includes a no-reuse hint by application thread 200''' on core 12. As above, the virtual address of the data requested, as specified by the thread 200'', is converted to a system address, e.g., in the manner shown in Figure 19, by way of non-limiting example, and discussed elsewhere herein.

If the requested datum is in the L1 Data cache (which is not the case shown here), it is returned to the requesting program 200'', but the reference count for its descriptor is not updated in the cache (because of the no-reuse hint)—and, indeed, in some embodiments, if it is greater than the default initialization value for a no-reuse request, it may be set to that value, here, 2).

If the requested datum is not in the L1 Data cache (as shown here), that cache signals a miss and passes the request to the L2 Data cache. If the requested datum is in the L2 Data cache, an L2 Cache lookup and, more specifically, a lookup comparing that system address against the tag portion of the L2 data cache (e.g., in the manner shown in Figure 22) results in a hit that returns the requested block, page, etc. (depending on implementation) to the L1 Data cache, which allocates a descriptor for that data and which (because of the no-reuse hint) sets its reference

count to the default initialization value for a no-reuse request, it may be set to that value, here, 2). The L1 Data cache can, in turn, pass the requested datum back to the requesting thread.

It will be appreciated that the operations shown in Figures 47 and 48, though, shown and discussed here for simplicity with respect to read operations involving two levels of cache (L1 and L2) can likewise be extended to additional levels of cache (e.g., L2E) and to other memory operations, as well, e.g., write operations. In the illustrated embodiment, other such operations can include, by way of non-limiting example, the following memory access instructions (and their respective reuse/no-reuse cache hints), e.g., among others: LOAD (Load Register), STORE (Store to Memory), LOADPAIR (Load Register Pair), STOREPAIR (Store Pair to Memory), PREFETCH (Prefetch Memory), LOADPRED (Load Predicate Register), STOREPRED (Store Predicate Register), EMPTY (Empty Memory), and FILL (Fill Memory) instructions. Other embodiments may provide other instructions, instead or instead or in addition, that utilize such parameters or that otherwise provide for influencing reference counts, e.g., in accord with the principles hereof.

Table 5- Level2 (L2) and Level2 Extended (L2E) block state

Nmeumonic	Value	Description
Invalid	000	Invalid
reserved	001	reserved
c_empty_ro	010	Copy, Empty, read only
c_full_ro	011	Copy, Full, read only
o_empty_ro	100	Owner, Empty, Read Only
o_empty_rw	101	Owner, Empty, Read/Write
o_full_ro	110	Owner, Full, Read Only
o_full_rw	111	Owner Full, Read/Write

Level2 Extended (L2E) Cache tags are addressed in a indexed, set associative manner. L2E data can be placed at arbitrary locations in off-chip memory.

ADDRESSING

Figure 30 depicts an IO address space format in an SEP system according to one practice of the invention.

Table 6- System Address Ranges

Range	Description
0x0000000000000000-0x0fffffffffffff	IO Devices
0x1000000000000000-0xfffffffffffff	Cache Memory

Table 7- IO Address Space Ranges

Device (SA[46:41])	Description
0x00	Flash memory
0x01-0x3f	IO Device 1-63

Table 8- Exception target address

Address	Description
0x0000000000000000	System privilege exception address
0x0000000001000000	Application privilege exception address

STANDARD DEVICE REGISTERS

IO devices include standard device registers and device specific registers. Standard device registers are described in the next sections.

Device Type Register

63 16	31 16	15 0
device specific	revision	device type

Identifies the type of device. Enables devices to be dynamically configured by software reading the type register first. Cores provide a device type of 0x0000 for all null devices.

Bit	Field	Description	type
15:0	device type	Value identifies the type of device. Value Description . 0x0000 Null device 0x0001 L2 and L2E memory controller 0x0002 Event Table 0x0003 DRAM Memory 0x0004 DMA Controller 0x0005 FPGA-Ethernet 0x0006 FPGA-DVI 0x0007 HDMI 0x0008 LCD Interface 0x0009 PCI 0x000a ATA 0x000b USB2 0x000c 1394 0x000d Ethernet 0x000e Flash memory 0x000f Audio out 0x0010 Power Management 0x0011-0xffff Reserved	read-only

31:16	revision	Value indentifies device revision	read-only
63:32	device specific	Additional device specific information	read-only

IO DEVICES

For each IO device the functionality, address map and detailed register description are provided.

Event Table

Table 9- Event Table Addressing

Device Offset	Register
0x00000000-0x0000ffff	Device type register
0x00010000-0x0001ffff	Event Queue Register
0x00020000-0x0002ffff	Event Queue Operation Register
0x00030000-0x0003ffff	Event-Thread Lookup Register
0x00040000-0xffffffff	Reserved

Event Queue Register

63	15
16	0
reserved	event

The Event Queue Register (EQR) enables read and write access to the event queue. The Event Queue location is specified by bits[15:0] of the device offset of IO address. First implementation contains 16 locations.

Bit	Field	Description	Privilege	Per
15:0	event	For writes specifies the virtual event number written or pushed onto the queue. For read operations contains the event number read from the queue	system	proc
63:16	Reserved	Reserved for future expansion of virtual event number	System	proc

Event Queue Operation Register

63:17	16	15	0
	empty	event	

The Event Queue Operation Register (EQR) enables an event to be pushed onto or popped from the event queue. Store to EQR is used for push and load from EQR is used for pop.

Bit	Field	Description	Privilege	Per
15:0	event	For writes specifies the event number written or pushed onto the queue. For read operations contains the event number read from the queue	system	proc

16	empty	For pop operation indicates whether the queue was empty prior to the current operation. If the queue was empty for pop operation, the event field is undefined. For push operation indicates whether the queue was full prior to the push operation. If the queue was full for the push operation, the push operation is not completed.	system	proc
----	-------	---	--------	------

Event-Thread Lookup Table Register

63	31	16	15	0
41				
	thread	event		

The Event to Thread lookup table establishes a mapping between an event number presented by a hardware device or event instruction and the preferred thread to signal the event to. Each entry in the table specifies an event number and a corresponding virtual thread number that the event is mapped to. In the case where the virtual thread number is not loaded into a TPU, or the event mapping is not present, the event is then signaled to the default system thread. See "Generalized Events and Multi-Threading," hereof, for further description.

The Event-Thread Lookup location is specified by bits[15:0] of the device offset of IO address. First implementation contains 16 locations.

Bit	Field	Description	Privilege	Per
-----	-------	-------------	-----------	-----

15:0	event	For writes specifies the event number written at the specified table address. For read operations contains the event number at the specified table address	system	proc
31:1 6	thread	Specifies virtual thread number corresponding to event	System	proc

L2 And L2E Memory Controller

Table 10- L2 and L2E Memory Controller

Device Offset	Register
0x00000000-0x0000ffff	Device type register
0x00010000-0x00ffffff	Reserved
0x01000000-0x01ffffff	L2 Tag
0x02000000-0x02ffffff	L2E Tag and Data
0x03000000-0xffffffff	Reserved

Power Management

SEP utilizes several types of power management:

- SEP processor instruction scheduler puts units that are not required during a given cycle in a low power state.
- IO controllers can be disabled if not being used
- Overall Power Management includes the following states
 - o Off- All chip voltages are zero
 - o Full on- A chip voltages and subsystems are enabled

- o Idle- Processor enters a low power state when all threads are in WAITING_IO state
- o Sleep- Clock timer, some other misc registers and auto-dram refresh are enabled. All other subsystems are in a low power state.

EXAMPLE MEMORY SYSTEM OPERATIONS

Adding The Removing Segments

SEP utilizes variable size segments to provide address translation (and privilege) from the Virtual to System address spaces. Specification of a segment does not in itself allocate system memory within the System Address space. Allocation and deallocation of system memory is on a page basis as described in the next section.

Segments can be viewed as mapped memory space for code, heap, files, etc.

Segments are defined on a per-thread basis. Segments are added enabling an instruction or data segment table entry for the corresponding process. These are managed explicitly by software running at system privilege. The segment table entry defines the access rights for the corresponding thread for the segment. Virtual to System address mapping for the segment can be defined arbitrary at the size boundary.

A segment is removed by disabling the corresponding segment table entry.

Allocating And Deallocating Pages

Pages are allocated on a system wide basis. Access privilege to a page is defined by the segment table entry corresponding to the page system address. By managing pages on a system shared basis, coherency is automatically maintained by the memory system for page descriptors and page contents. Since SEP manages all memory and corresponding pages as cache, pages are allocated and deallocated at the shared memory system, rather than per thread.

Valid pages and the location where they are stored in memory are described by the in memory hash table shown in figure 86, L2E Descriptor Tree Lookup. For a specific index the descriptor tree can be 1, 2 or 3 levels. The root block starts at 0 offset. System software can create a segment that maps virtual to system at 0x0 and create page descriptors that directly map to the address space so that this memory is within the kernel address space.

Pages are allocated by setting up the corresponding NodeBlock, TreeNode and L2E Cache Tag. The TreeNode describes the largest SA within the NodeBlocks that it points to. The TreeNodes are arranged within a NodeBlock in increasing SA order. The physical page number specifies the storage location in dram for the page. This is effectively a b-tree organization.

Pages are deallocated by marking the entries invalid.

MEMORY SYSTEM IMPLEMENTATION

Referring to **Figure 31**, the memory system implementation of the illustrated SEP architecture enables an all-cache memory system which is transparently scalable across cores and threads. The memory system implementation includes:

- Ring Interconnect (RI) provides packet transport for cache memory system operations. Each device includes a RI port. Such a ring interconnect can be constructed, operated, and utilized in the manner of the “cell interconnect” disclosed, by way of non-limiting example, as elements 10–13, in Figure 1 and the accompanying text of United States Patent US 5,119,481, entitled “Register Bus Multiprocessor System with Shift,” and further details of which are disclosed, by way of non-limiting example, in Figures 3–8 and the accompanying text of that patent, the teachings of which are incorporated herein by reference, and a copy of which is filed herewith by example as Appendix B, as adapted in accord with the teachings hereof.
- External Memory Cache Controller provides interface between the RI and external DDR3 dram and flash memory.
- Level2 Cache Controller provides interface between the RI and processor core.

- IO Bridge provides a DMA and programmed IO interface between the RI and IO busses and devices.

The illustrated memory system is advantageous, e.g., in that it can serve to combine high bandwidth technology with bandwidth efficiency, and in that it scales across cores and/or other processing modules (and/or respective SOC's or systems in which they may respectively be embodied) and external memory (DRAM & flash)

RING INTERCONNECT (RI) GENERAL OPERATION

RI provides a classic layered communication approach:

- Caching protocol- provides integrated coherency for all-cache memory system including support for events
- Packet contents- Payload consisting of data, address, command, state and signalling
- Physical transport- Mapping to signals. Implementations can have different levels of parallelism and bandwidth

PACKET CONTENTS

Packet includes the following fields:

- SystemAddress[63:7] – Block address corresponding the data transfer or request. All transfers are in units of a single 128 byte block.
- RequestorID[31:0]- RI interface number of requestor. ReqID[2:0] implemented in first implementation, remainder reserved. The value of each RI is hardwired as part of the RI interface implementation.
- Command

Value	Command	State Field	Data
0x0	Nop	Invalid	invalid
0x1	Read only request	Invalid	invalid
0x2	Writable read request	Invalid	invalid

0x3	Exclusive read request	Invalid	invalid
0x4	Invalidate	Invalid	invalid
0x5	Update	Invalid	valid
0x6	Response ro request	Valid	valid
0x7	Response writeable request	Valid	Valid
0x8	Response exclusive request	Valid	valid
0x9	Read IO request	Invalid	invalid
0xa	Response IO	Invalid	valid
0xb	Write IO	Invalid	valid
0xc-0xf	reserved		

State- Cache state associated with the command.

Value	State & Description
0x0	Invalid
0x1	Reserved
0x2	C_EMPTY_RO- Read only copy, empty
0x3	C_FULL_RO- Read only copy, full
0x4	O_EMPTY_RW- Owner, writeable, empty
0x5	O_EMPTY_RWE- Owner, writeable, no other copies
0x6	O_FULL_RW- Owner, writeable, full
0x7	O_FULL-RWE- Owner, writeable, no other copies

- Early Valid- Boolean that indicates that the corresponding packet slot contains a valid command. Bit is present early in the packet. Both early and late valid Booleans must be true for packet to be valid.
- Early Busy- Boolean that indicates that the command could not be processed by RI interface. The command must be re-tried by initiator. The packet is considered busy if either early busy or late busy is set.
- Late Valid – Boolean that indicated that the corresponding packet slot contains a valid command. Bit is present late in the packet. Both early and late valid Booleans must be true for packet to be valid. When an RI interface is passing a packet through it should attempt clear early valid if late valid is false.

- Late Busy- Boolean that indicates that the command could not be processed by RI interface. The command must be re-tried by the initiator. The packet is considered busy if either early busy or late busy is set. When an RI interface is passing a packet through it should attempt to set early busy if late busy is true.

PHYSICAL TRANSPORT

The Ring Interconnect bandwidth is scalable to meet the needs of scalable implementations beyond 2-core. The RI can be scaled hierarchically to provide virtually unlimited scalability.

The Ring Interconnect physical transport is effectively a rotating shift register. The first implementation utilizes 4 stages per RI interface. A single bit specifies the first cycle of each packet (corresponding to cycle 1 in table below) and is initialized on reset.

For a two-core SEP implementation, example, there can be a 32 byte wide data payload path and a 57 bit address path that also multiplexes command, state, flow control and packet signaling.

Cycle	Data payload path (32 bytes wide)	Address payload path (57 bits)
1	Previous packet....	SystemAddress[63:7]
2	Databytes[31:0]	Command, ReqID[31:0], State, EarlyValid, EarlyBusy
3	Databytes[63:32]	Not used
4	Databytes[95:64]	LateValid, LateBusy
5	Databytes[127:96]	Next packet.....

INSTRUCTION SET EXPANDABILITY

Provides a capability to define programmable instructions, which are dedicated to a specific application and/or algorithm. These instructions can be add in two ways:

- Dedicated functional unit- Fixed instruction capability. This can be an additional functional unit or an addition to an existing unit.
- Programmable functional unit- Limited FPGA type functionality to tailor the hardware unit to the specifics of the algorithm. This capability is loaded from a privileged control register and is available to all threads.

ADVANTAGES AND FURTHER EMBODIMENTS

Systems constructed in accord with the invention can be employed to provide a runtime environment for executing tiles, e.g., as illustrated in **Figure 32** (*sans* graphical details identifying separate processor or core boundaries):

Those tiles can be created, e.g., applications, attendant software libraries, etc., and assigned to threads in the conventional manner known in the art, e.g., as discussed in US 5,535,393 (“System for Parallel Processing That Compiles a [Tiled] Sequence of Instructions Within an Iteration Space”), the teachings of which are incorporated herein by reference. Such tiles can beneficially utilize memory access instructions discussed herein, as well those disclosed, by way of non-limiting example, in Figures 24A-24B and the accompanying text (e.g., in the section entitled “CONSUMER-PRODUCER MEMORY”) of incorporated-by-reference patents US 7,685,607 and US 7,653,912, the teachings of which figures and text (and others of which pertain memory access instructions and particularly, for example, the *Empty* and *Fill* instructions) are incorporated herein by reference, as adapted in accord with the teachings hereof.

An exemplary, non-limiting software architecture utilizing a runtime environment of the sort provided by systems according to the invention is shown in **Figure 33**, to wit, a TV/set-top application providing simultaneously running one or more of television, telepresence, gaming and other applications (apps) by way of example, that (a) execute over a common applications framework of the type known in the art as adapted in accord with the teachings hereof and that, in turn (b) executes on media (e.g., video streams, etc.) of the type known in the art utilizing a media framework (e.g., codecs, OpenGL, scaling and noise reduction functionality, color

conversion & correction functionality, and frame rate correction functionality, all by way of example) of the type known in the art (e.g., Linux core services) as adapted in accord with the teachings hereof and that, in turn, (c) executes on core services of the type known in the art as adapted in accord with the teachings hereof and that, in turn, (d) executes on a core operating system (e.g., Linux) of the type known in the art as adapted in accord with the teachings hereof.

Processor modules, systems and methods of the illustrated embodiment are well suited for executing digital cinema, integrated telepresence, virtual hologram based gaming, hologram-based medical imaging, video intensive applications, face recognition, user-defined 3D presence, software applications, all by way of non-limiting example, utilizing a software architecture of the type shown in **Figure 33**.

Advantages of processor modules and systems according to the invention are that, among other things, they provide the flexibility & programmability of “all software” logic solutions combined with the performance equal or better to that of “all hardware” logic solutions, as depicted in **Figure 34**.

A typical implementation of a consumer (or other) device for video processing using a prior art processor is shown in **Figure 35**. Generally speaking, such implementations demand that new hardware (e.g., additional hardware processor logic) be added for each new function in the device. By comparison, there is shown in **Figure 36** a corresponding implementation using a processor module of the illustrated embodiment. As evident from comparing the drawings, what has typically required a fixed hardwired solution in prior art implementations can be effected by a software pipeline in solutions in accord with the illustrated embodiment. This is also shown in **Figure 46**, wherein a pipeline of instructions executing on each of cores 12–16 serve as software equivalents of corresponding hardware pipelines of the type traditionally practiced in the prior art. Thus, for example, a pipeline of instructions 220 executing on the TPUs 12B of core 12 perform the same functionality as and take place of a hardware pipeline 222; software pipeline 224 executing on TPUs 14B of core 14 take perform the same functionality as and take place of a hardware pipeline 226; and, software pipeline 228 executing on TPUs 14B of core 14 take

perform the same functionality as and take place of a hardware pipeline 230, all by way of non-limiting example.

In addition to executing software pipelines that perform the same functionality as and take place of corresponding hardware pipelines, new functions can be added to these cores 12–16 without the addition of new hardware as those functions can often be accommodated via the software pipeline.

To these ends, **Figure 37** illustrates use of an SEP processor in accord with the invention for parallel execution of applications, ARM binaries, media framework (here, e.g., H.264 and JPEG 2000 logic) and other components of the runtime environment of a system according to the invention, all by way of example.

Referring to Figure 46, the illustrated cores are general purpose processors capable of executing pipelines of software components in lieu of like pipelines of hardware components of the type normally employed by prior art devices. Thus, for example, core 14 executes, by way of non-limiting example, software components pipelined for video processing and including a H.264 decoder software module, a scalar and noise reduction software module, a color correction software module, a frame rate control software module, e.g., as shown. This is in lieu of inclusion execution of a like hardware pipeline 226 on dedicated chips, e.g., a semiconductor chip that functions as a system controller with H.264 decoding, pipelined to a semiconductor chip that functions as a scalar and noise reduction module, pipelined to a semiconductor chip that functions for color correction, and further pipelined to a semiconductor chip that functions as a frame rate controller.

In operation, each of the respective software components, e.g., of pipeline 224, executes as one or more threads, all of which for a given task may execute on a single core or which may be distributed among multiple cores.

To facilitate the foregoing, cores 12–16 operate as discussed above and each supports one or more of the following features, all by way of non-limiting example, dynamic assignment of

events to threads, a location-independent shared execution environment, the provision of quality of service through thread instantiation, maintenance and optimization, JPEG2000 bit plane stripe column encoding, JPEG2000 binary arithmetic code lookup, arithmetic operation transpose, a cache control instruction set and cache-initiated optimization, and a cache managed memory system.

Shown and described herein are processor modules, systems and methods meeting the objects set forth above, among others. It will be appreciated that the illustrated embodiments are merely examples of the invention and that other embodiments embodying changes thereto fall within the scope of the invention.

Appendix A

US 7,685,607



US007685607B2

(12) **United States Patent**
Frank et al.

(10) **Patent No.:** **US 7,685,607 B2**
(45) **Date of Patent:** **Mar. 23, 2010**

(54) **GENERAL PURPOSE EMBEDDED
PROCESSOR**

(75) Inventors: **Steven Frank**, 116 Pleasant St., Apt. #1,
Easthampton, MA (US) 01027; **Shigeki
Imai**, Nara (JP)

(73) Assignees: **Steven Frank**, Florence, MA (US);
Sharp Corporation, Osaka (JP)

(*) Notice: Subject to any disclaimer, the term of this
patent is extended or adjusted under 35
U.S.C. 154(b) by 633 days.

(21) Appl. No.: **10/735,610**

(22) Filed: **Dec. 12, 2003**

(65) **Prior Publication Data**

US 2004/0244000 A1 Dec. 2, 2004

Related U.S. Application Data

(63) Continuation of application No. 10/449,732, filed on
May 30, 2003.

(51) Int. Cl.
G06F 13/00 (2006.01)

(52) U.S. Cl. **719/318; 718/102; 718/104;
710/260**

(58) **Field of Classification Search** **719/318;
713/320; 718/102, 104; 710/260**
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,689,739 A * 8/1987 Federico et al. 710/264
5,692,193 A * 11/1997 Jagannathan et al. 718/106
5,721,855 A * 2/1998 Hinton et al. 712/218
6,219,780 B1 4/2001 Lipasti
6,240,508 B1 * 5/2001 Brown et al. 712/219
6,272,520 B1 8/2001 Sharangpani et al.

6,408,381 B1 6/2002 Gearty et al.
6,427,195 B1 7/2002 McGowen et al.
6,470,443 B1 10/2002 Emer et al.
6,493,741 B1 * 12/2002 Emer et al. 718/107

(Continued)

FOREIGN PATENT DOCUMENTS

JP 10-242833 A 9/1998

OTHER PUBLICATIONS

"Microsoft Computer Dictionary," Fifth Edition, Microsoft Press,
2002; page including "branch instruction" definition; retrieved from
<safaribooksonline.com> on Aug. 27, 2008.*

(Continued)

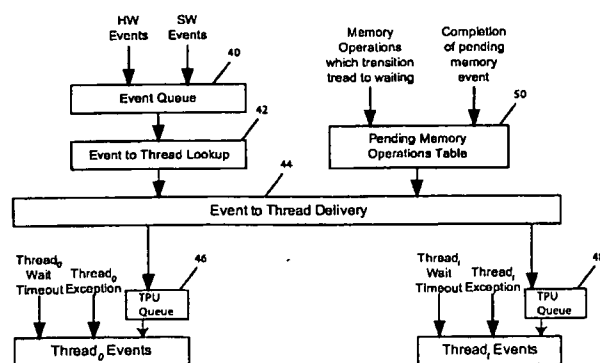
Primary Examiner—Andy Ho

(74) Attorney, Agent, or Firm—Nutter McClennen & Fish
LLP; David J. Powsner

(57) **ABSTRACT**

The invention provides an embedded processor architecture comprising a plurality of virtual processing units that each execute processes or threads (collectively, "threads"). One or more execution units, which are shared by the processing units, execute instructions from the threads. An event delivery mechanism delivers events—such as, by way of non-limiting example, hardware interrupts, software-initiated signaling events ("software events") and memory events—to respective threads without execution of instructions. Each event can, per aspects of the invention, be processed by the respective thread without execution of instructions outside that thread. The threads need not be constrained to execute on the same respective processing units during the lives of those threads—though, in some embodiments, they can be so constrained. The execution units execute instructions from the threads without needing to know what threads those instructions are from. A pipeline control unit which launches instructions from plural threads for concurrent execution on plural execution units.

61 Claims, 26 Drawing Sheets



US 7,685,607 B2

Page 2

U.S. PATENT DOCUMENTS

6,799,317 B1 * 9/2004 Heywood et al. 719/313
6,912,647 B1 * 6/2005 Blandy 712/24
7,082,519 B2 * 7/2006 Kelsey et al. 712/228
7,363,474 B2 * 4/2008 Rodgers et al. 712/227
2001/0016879 A1 * 8/2001 Sekiguchi et al. 709/319
2003/0120896 A1 * 6/2003 Gosior et al. 712/32
2004/0049672 A1 * 3/2004 Nollet et al. 713/100

OTHER PUBLICATIONS

Susan J. Eggers, et al. "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro* (Sep./Oct. 1997) pp. 12-19.

Japanese Office Action, issued Apr. 22, 2009, in corresponding application of the instant case, 4 pages.

* cited by examiner

U.S. Patent

Mar. 23, 2010

Sheet 1 of 26

US 7,685,607 B2

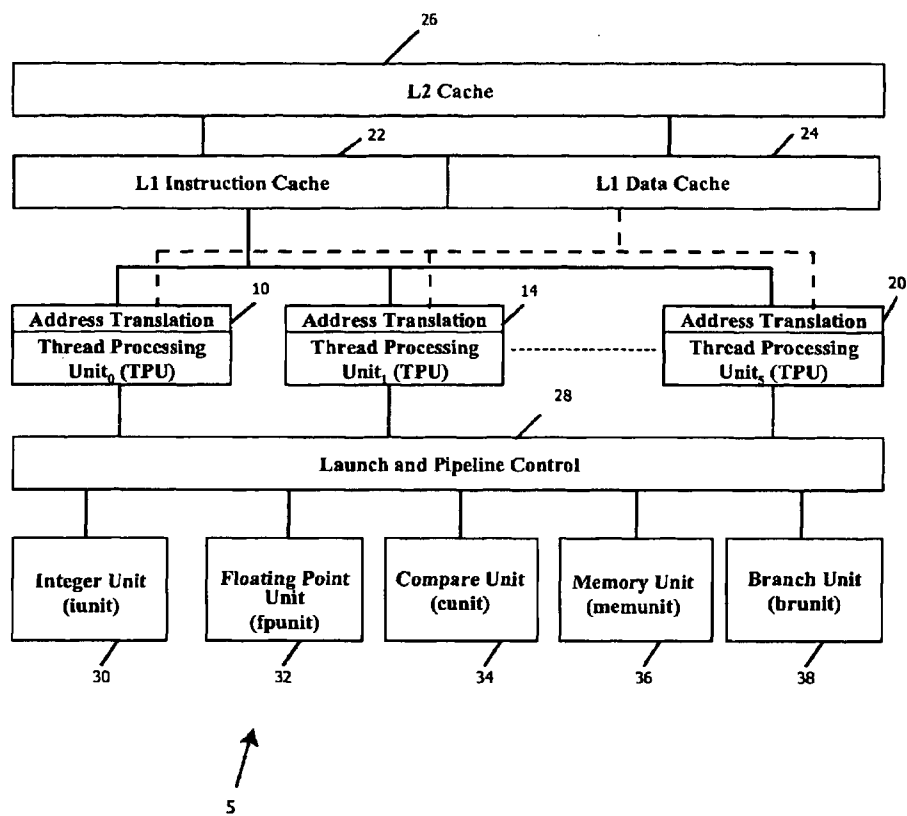


Figure 1

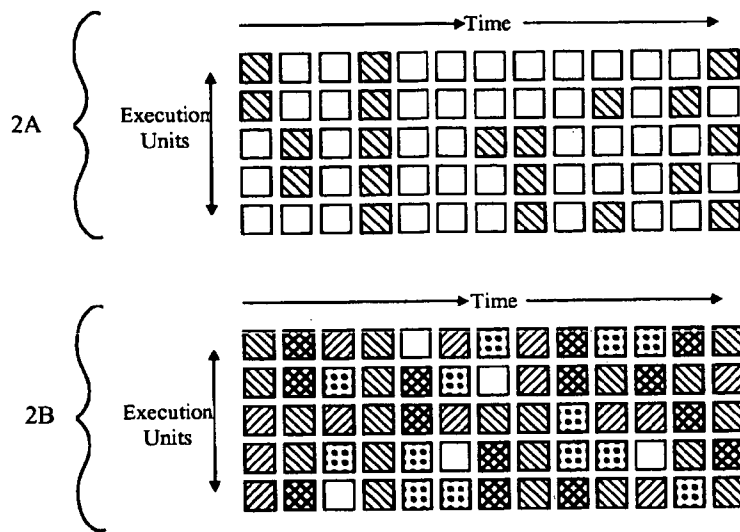


Figure 2

U.S. Patent

Mar. 23, 2010

Sheet 3 of 26

US 7,685,607 B2

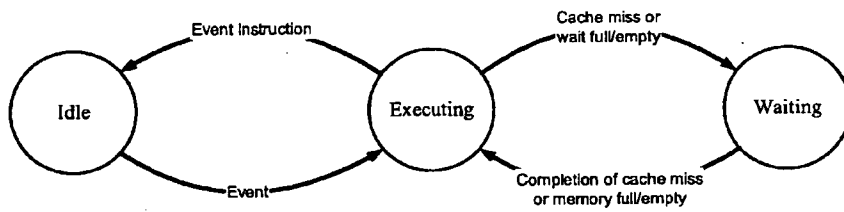


Figure 3

U.S. Patent

Mar. 23, 2010

Sheet 4 of 26

US 7,685,607 B2

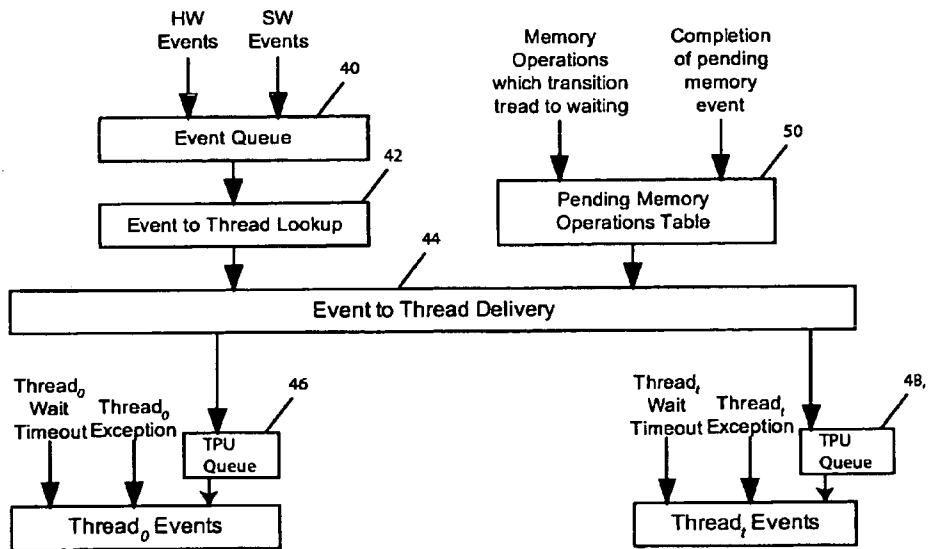


Figure 4

U.S. Patent

Mar. 23, 2010

Sheet 5 of 26

US 7,685,607 B2

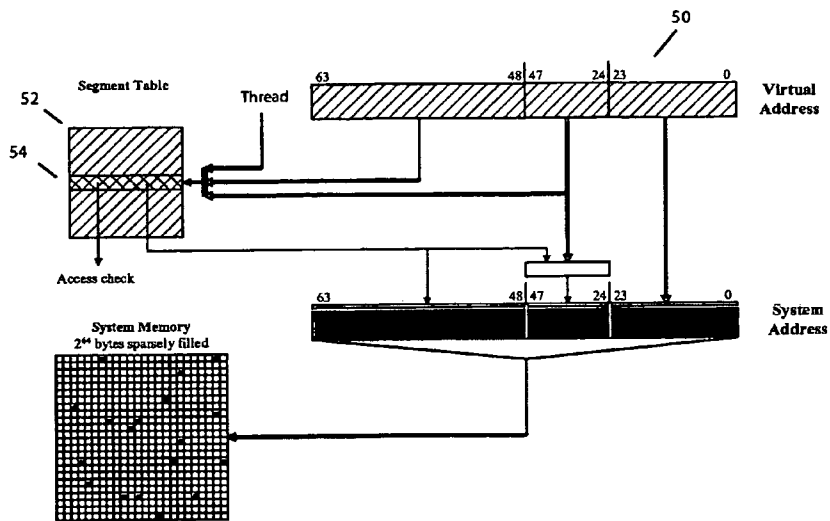


Figure 5

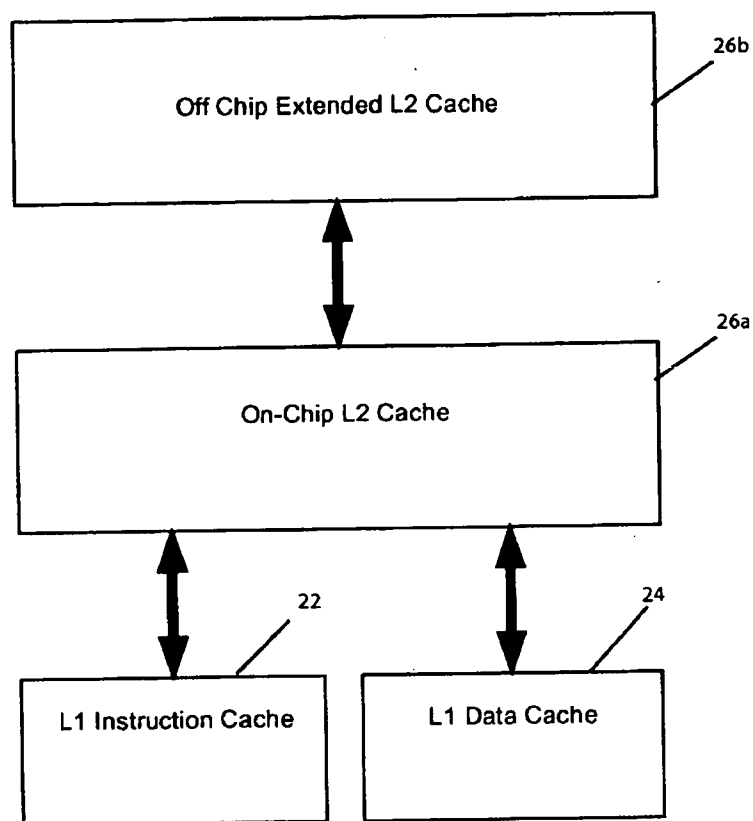


Figure 6

U.S. Patent

Mar. 23, 2010

Sheet 7 of 26

US 7,685,607 B2

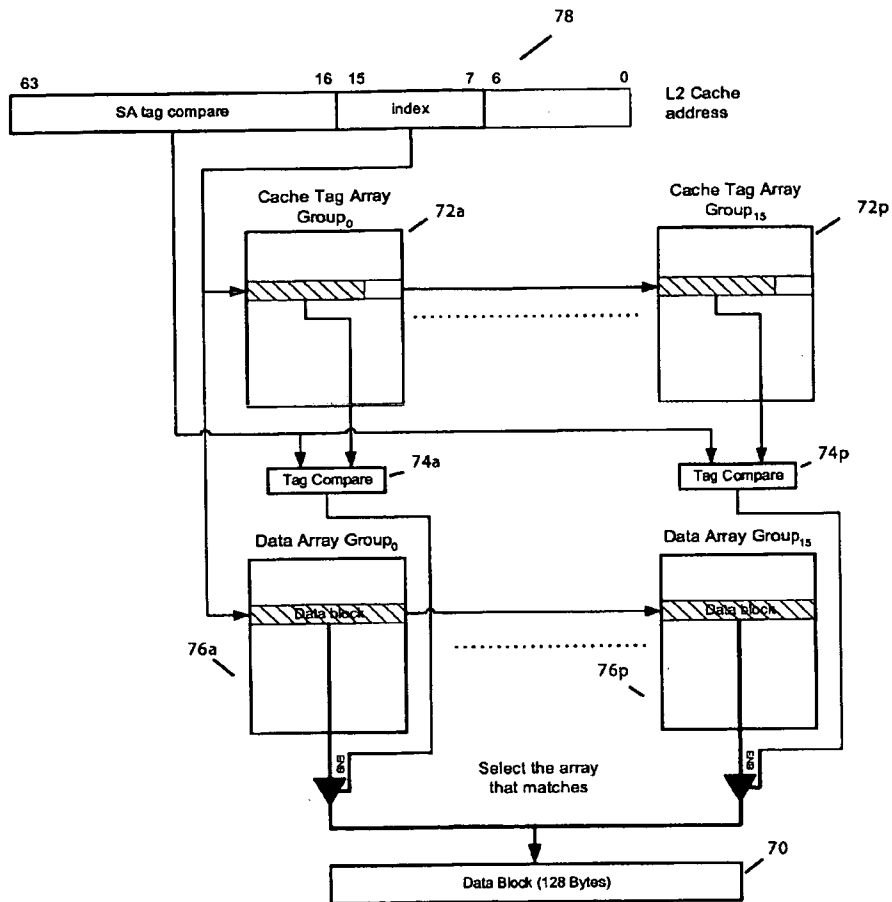


Figure 7

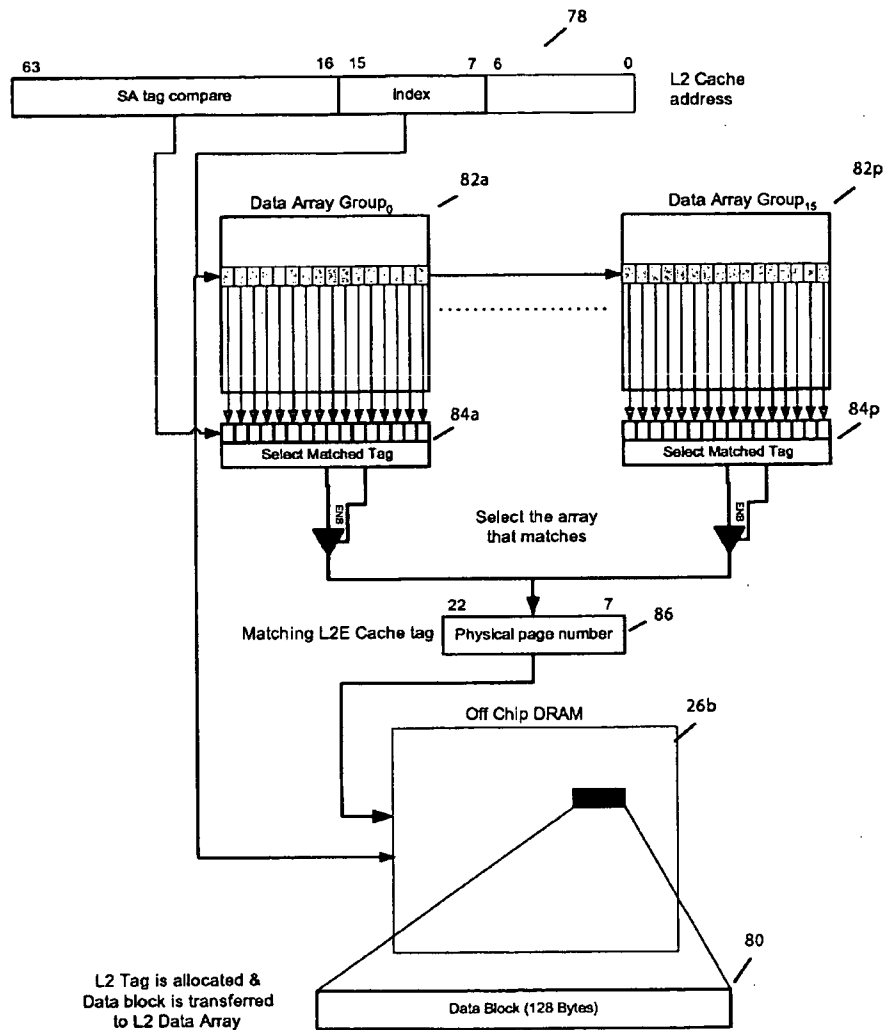


Figure 8

U.S. Patent

Mar. 23, 2010

Sheet 9 of 26

US 7,685,607 B2

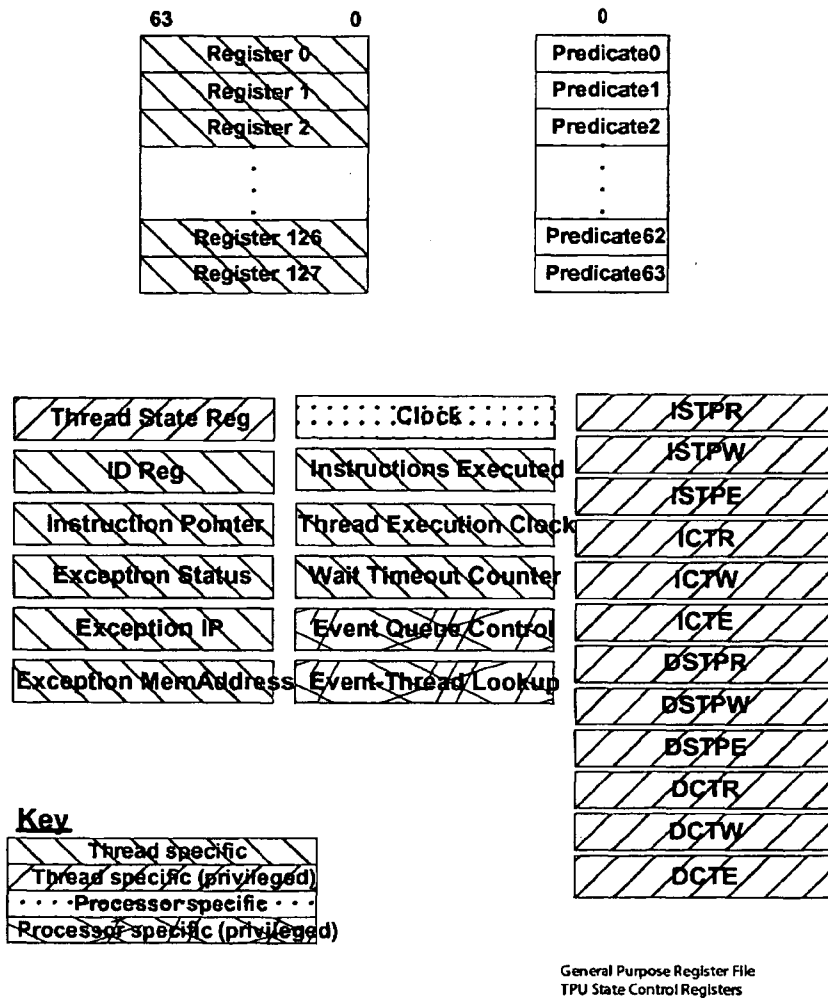


Figure 9

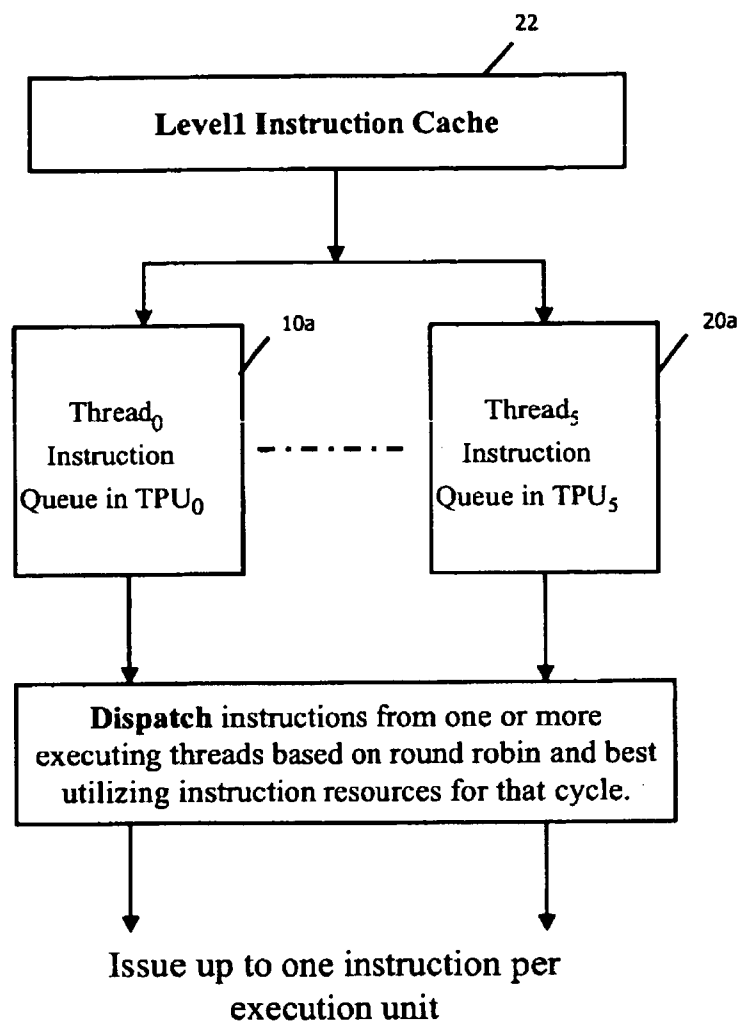


Figure 10

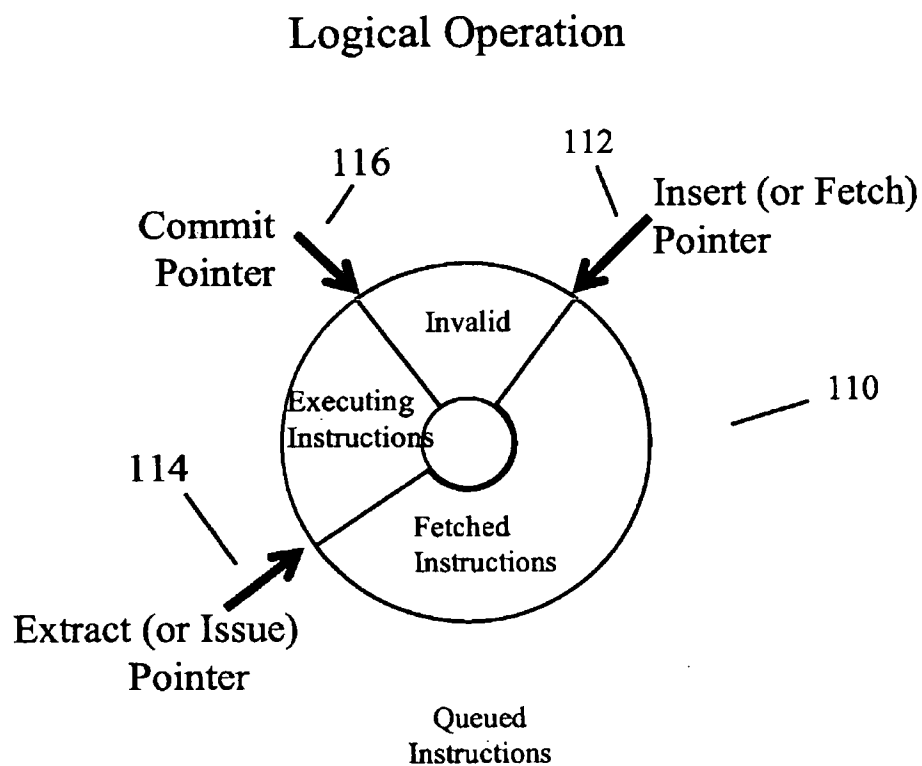


Figure 11

U.S. Patent

Mar. 23, 2010

Sheet 12 of 26

US 7,685,607 B2

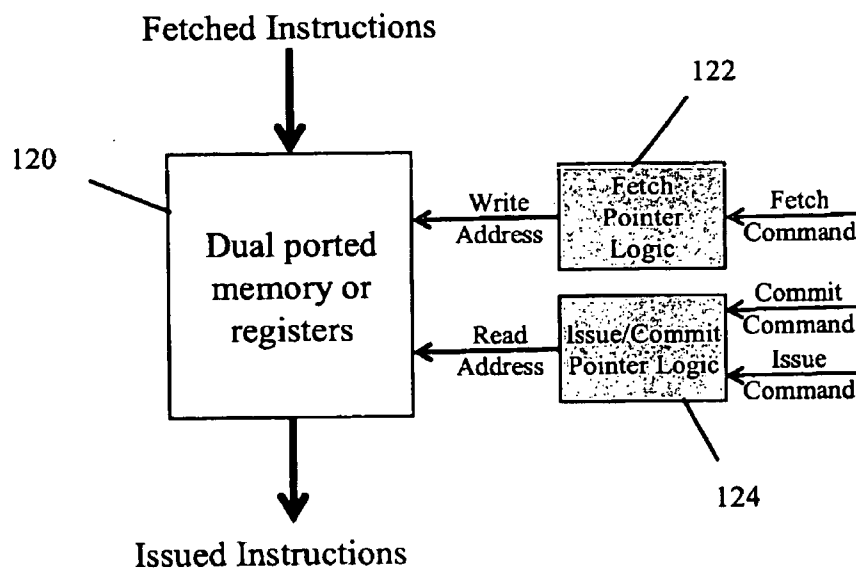


Figure 12

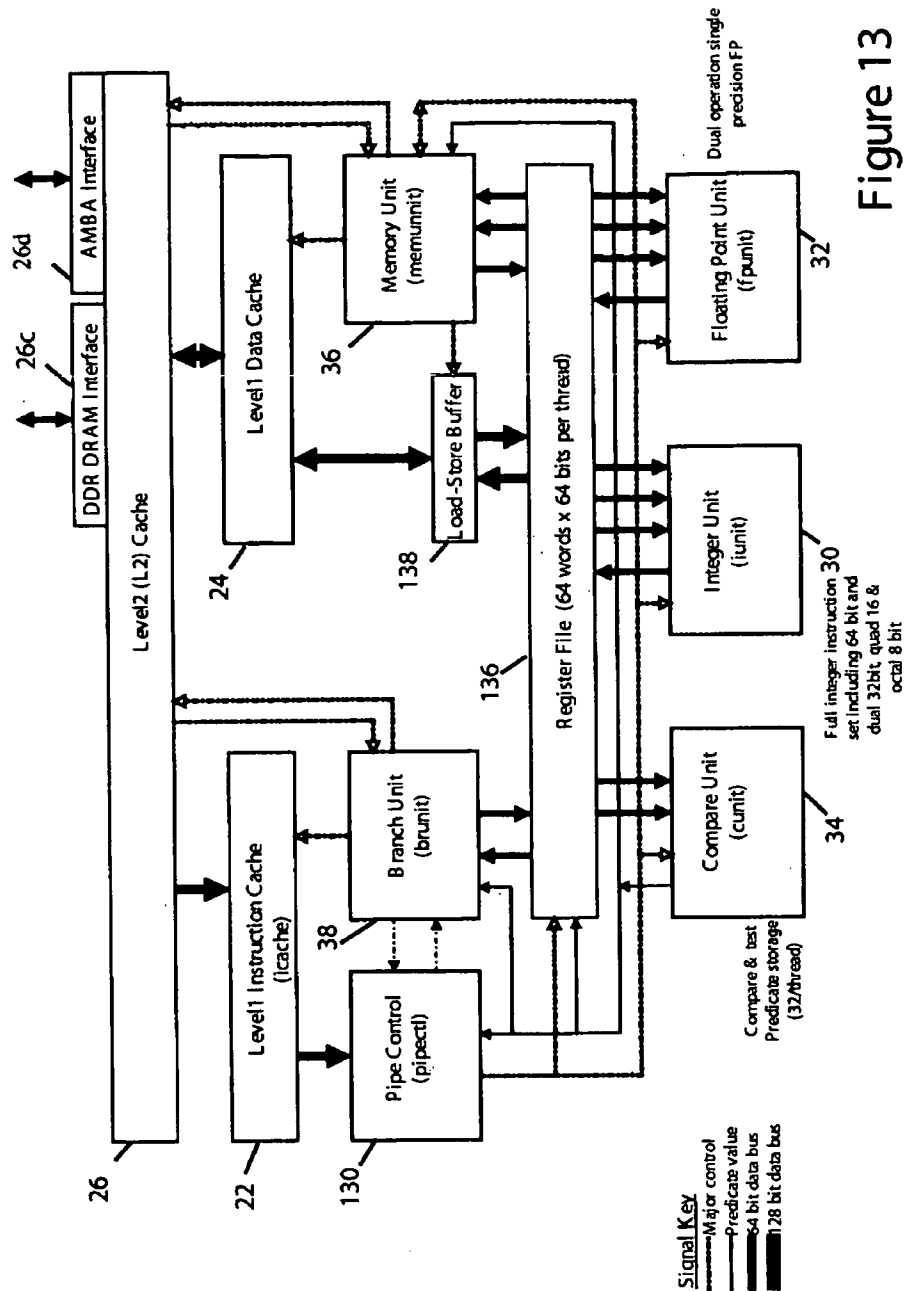


Figure 13

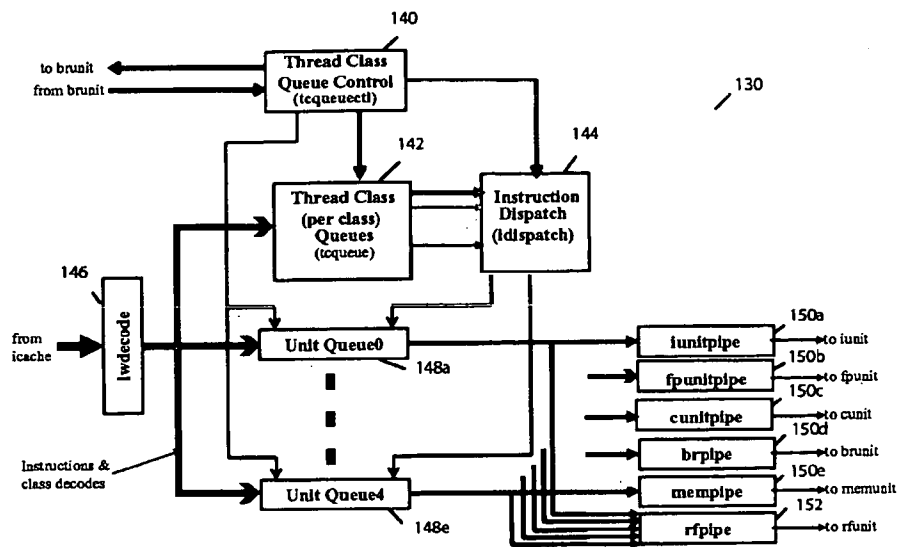


Figure 14

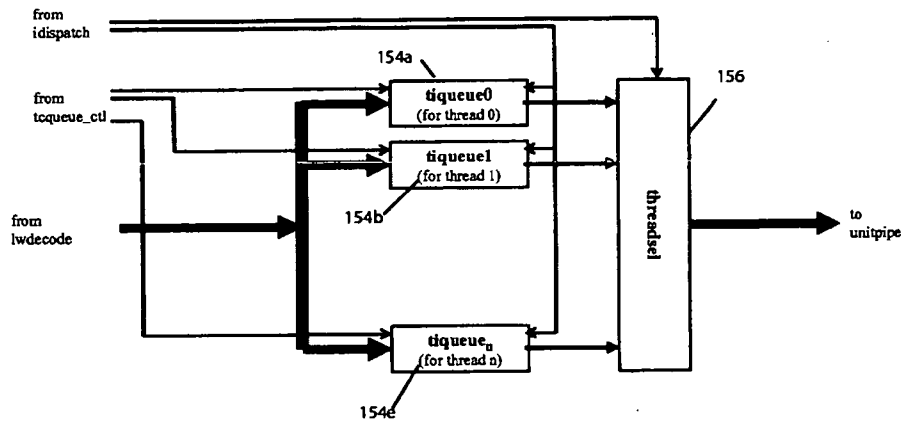


Figure 15

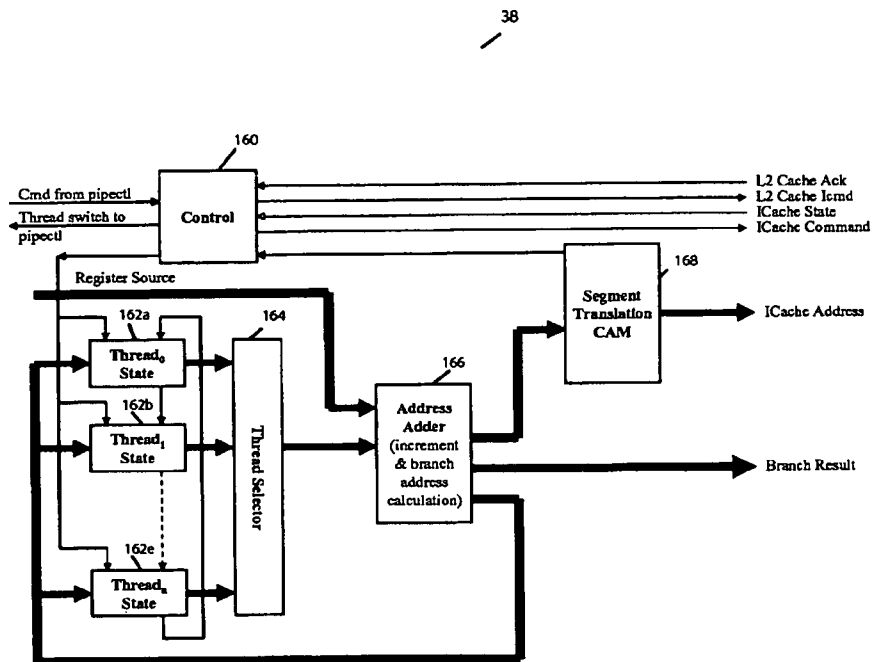


Figure 16

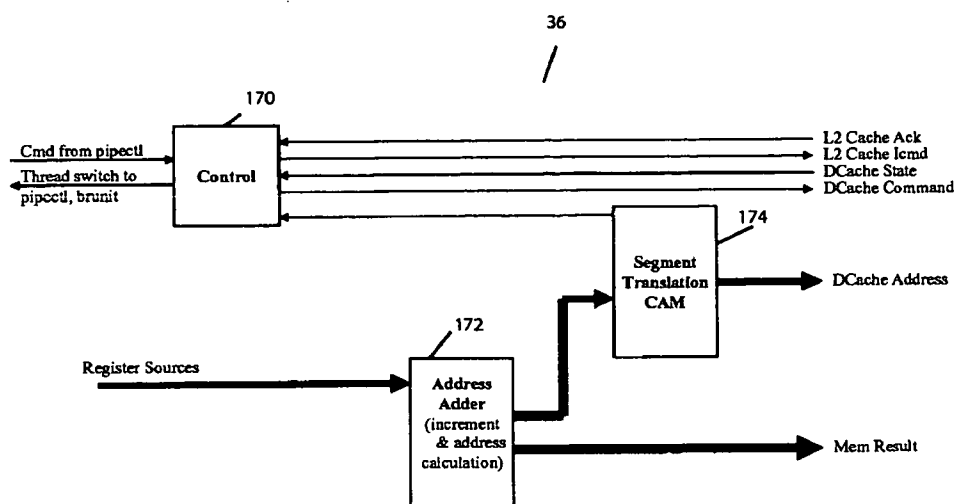


Figure 17

U.S. Patent

Mar. 23, 2010

Sheet 18 of 26

US 7,685,607 B2

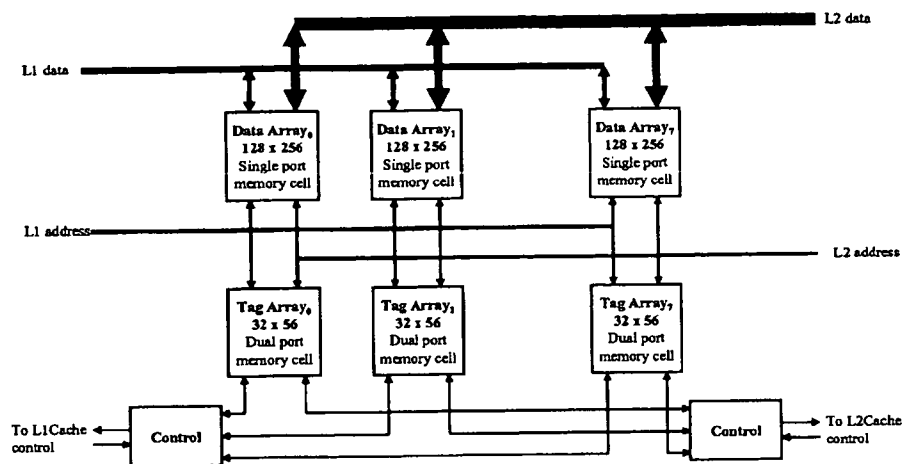


Figure 18

U.S. Patent

Mar. 23, 2010

Sheet 19 of 26

US 7,685,607 B2

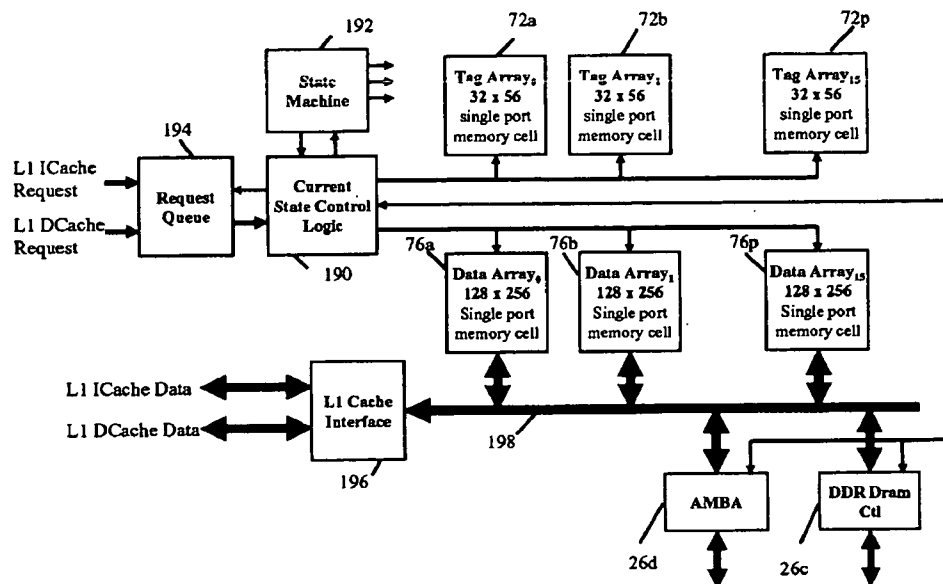


Figure 19

U.S. Patent

Mar. 23, 2010

Sheet 20 of 26

US 7,685,607 B2

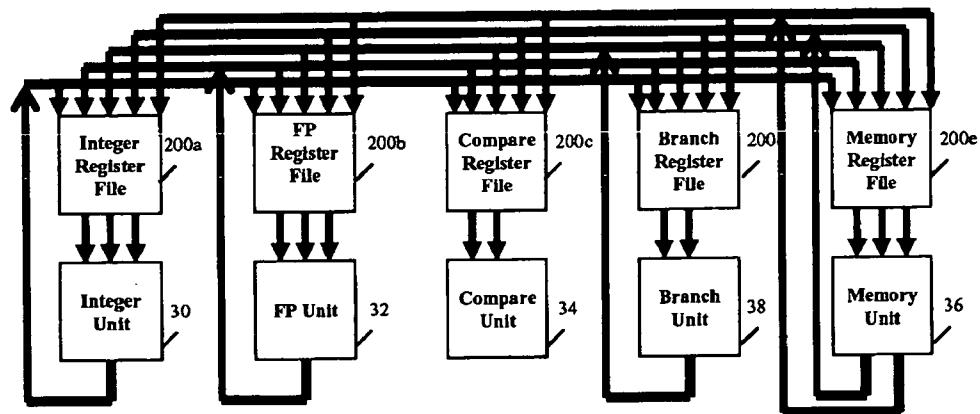


Figure 20

U.S. Patent

Mar. 23, 2010

Sheet 21 of 26

US 7,685,607 B2

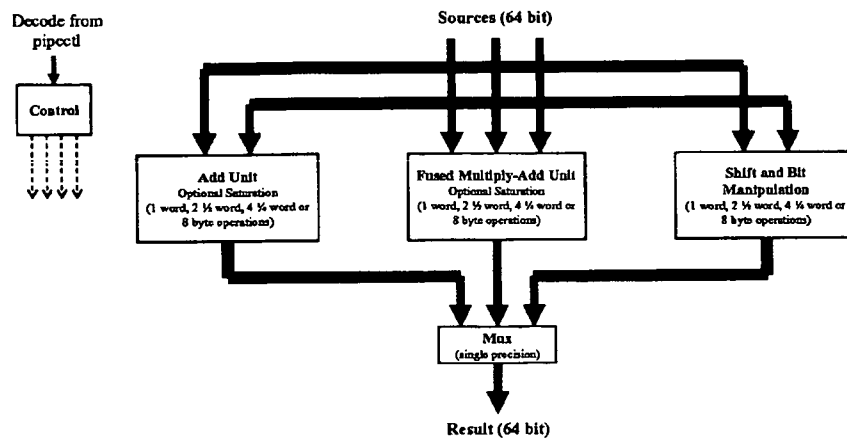


Figure 21

U.S. Patent

Mar. 23, 2010

Sheet 22 of 26

US 7,685,607 B2

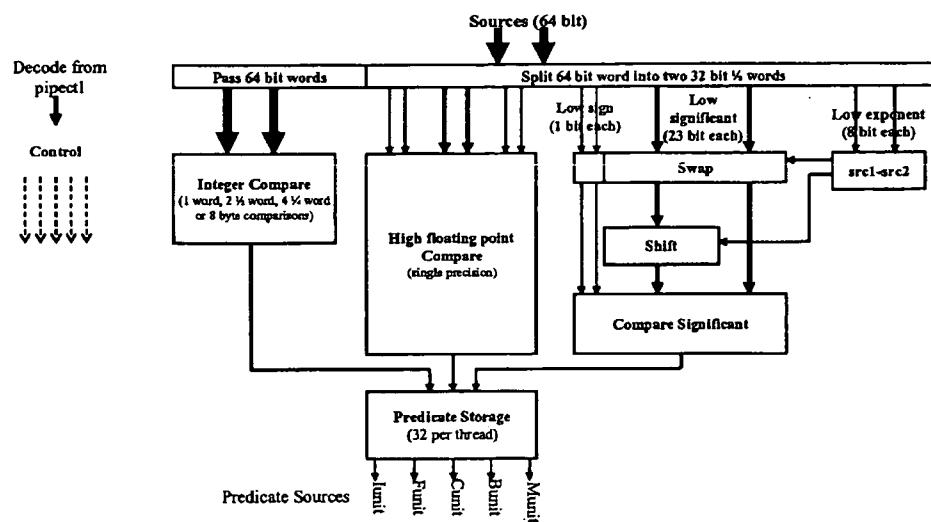


Figure 22

U.S. Patent

Mar. 23, 2010

Sheet 23 of 26

US 7,685,607 B2

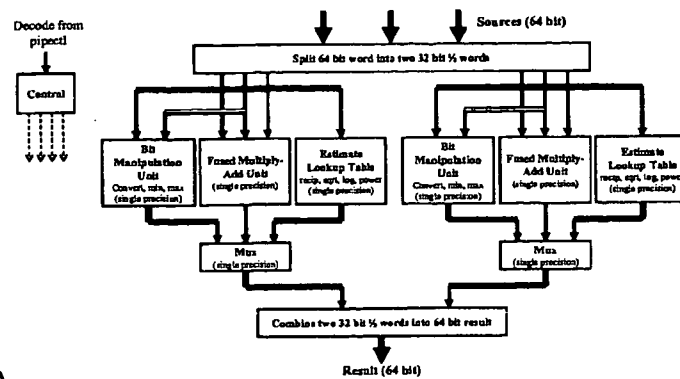


Figure 23A

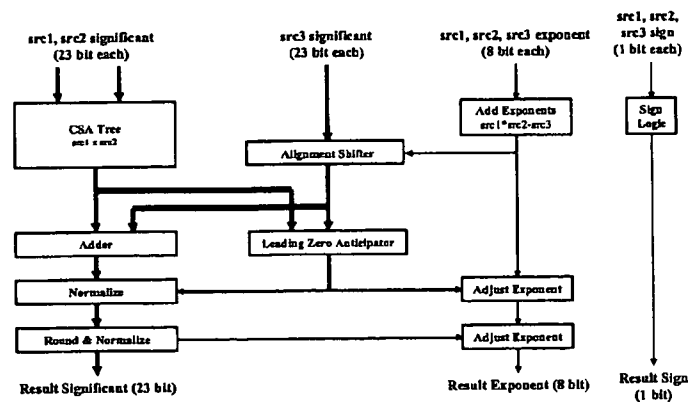


Figure 23B

U.S. Patent

Mar. 23, 2010

Sheet 24 of 26

US 7,685,607 B2

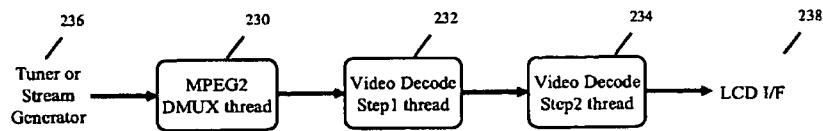


Figure 24A

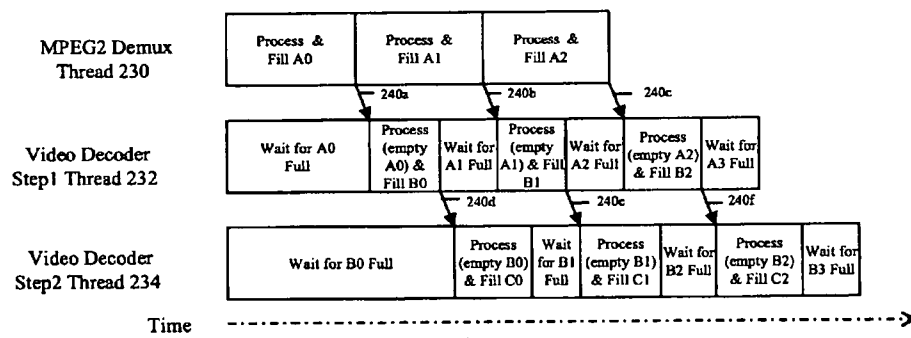


Figure 24B

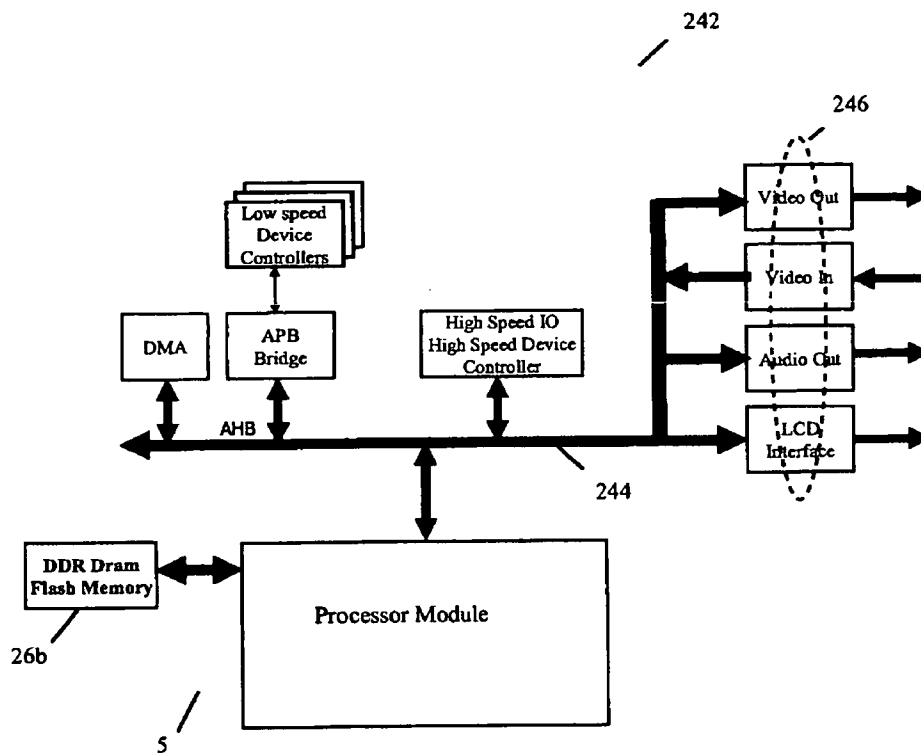


Figure 25

U.S. Patent

Mar. 23, 2010

Sheet 26 of 26

US 7,685,607 B2

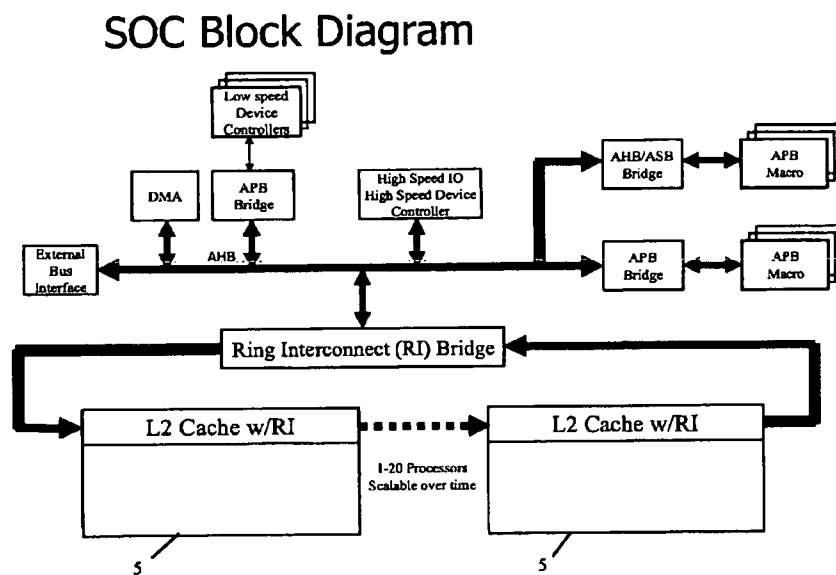


Figure 26

US 7,685,607 B2

1

GENERAL PURPOSE EMBEDDED PROCESSOR

This application is a continuation of, and claims the benefit of priority of, copending, commonly-assigned U.S. patent application Ser. No. 10/449,732, filed May 30, 2003, entitled "Virtual Processor Methods and Apparatus with Unified Event Notification And Consumer-producer Memory Operations," the teachings of which are incorporated herein by reference.

BACKGROUND

The invention pertains to digital data processing and, more particularly, to embedded processor architectures and operation. The invention has application in high-definition digital television, game systems, digital video recorders, video and/or audio players, personal digital assistants, personal knowledge navigators, mobile phones, and other multimedia and non-multimedia devices.

Prior art embedded processor-based or application systems typically combine: (1) one or more general purpose processors, e.g., of the ARM, MIPS or x86 variety, for handling user interface processing, high level application processing, and operating system tasks, with (2) one or more digital signal processors (DSPs), including media processors, dedicated to handling specific types of arithmetic computations at specific interfaces or within specific applications, on real-time/low latency bases. Instead of, or in addition to, the DSPs, special-purpose hardware is often provided to handle dedicated needs that a DSP is unable to handle on a programmable basis, e.g., because the DSP cannot handle multiple activities at once or because the DSP cannot meet needs for a very specialized computational element.

Examples of these prior art systems include personal computers, which typically combine a main processor with a separate graphics processor and a separate sound processor; game systems, which typically combine a main processor and separately programmed graphics processor; digital video recorders, which typically combine a general purpose processor, mpeg2 decoder and encoder chips, and special-purpose digital signal processors; digital televisions, which typically combine a general purpose processor, mpeg2 decoder and encoder chips, and special-purpose DSPs or media processors; mobile phones, which typically combine a processor for user interface and applications processing and special-purpose DSPs for mobile phone GSM, CDMA or other protocol processing.

Prior art patents include U.S. Pat. No. 6,408,381, disclosing a pipeline processor utilizing snapshot files with entries indicating the state of instructions in the various pipeline stages. As the instructions move within the pipeline, the corresponding snapshot file entries are changed. By monitoring those files, the device determines when interim results from one pipe-line stage can be directly forwarded to another stage over an internal operand bus, e.g., without first being stored to the registers.

The prior art also includes U.S. Pat. No. 6,219,780, which concerns improving the throughput of computers with multiple execution units grouped in clusters. This patent suggests identifying "consumer" instructions which are dependent on results from "producer" instructions. Multiple copies of each producer instruction are then executed, one copy in each cluster that will be subsequently used to execute dependent consumer instructions.

The reasons for the general prior art approach—combining general purpose processors with DSPs and/or special-pur-

2

pose hardware—is the need to handle multiple activities (e.g., events or threads) simultaneously on a real-time basis, where each activity requires a different type of computational element. However, no single prior art processor has the capacity to handle all of the activities. Moreover, some of the activities are of such a nature that no prior art processor is capable of properly handling more than a single one of them. This is particularly true of real-time activities, to which timely performance is degraded, if not wholly prevented, by operating system intervention.

One problem with the prior art approach is hardware design complexity, combined with software complexity in programming and interfacing heterogeneous types of computing elements. Another problem is that both hardware and software must be re-engineered for every application. Moreover, prior art systems do not load balance: capacity cannot be transferred from one hardware element to another.

An object of this invention is to provide improved apparatus and methods for digital data processing. A further object of the invention is to provide such apparatus and methods as support multiple activities, real-time or otherwise, to be executed on a single processor, as well to provide multiple such processors that are capable of working together. A related object is to provide such apparatus and methods as are suitable for an embedded environment or application. Another related object is to provide such apparatus and methods as facilitate design, manufacture, time-to-market, cost and/or maintenance.

A further object of the invention is to provide improved apparatus and methods for embedded (or other) processing that meet the computational, size, power and cost requirements of today's and future appliances, including by way of non-limiting example, digital televisions, digital video recorders, video and/or audio players, personal digital assistants, personal knowledge navigators, and mobile phones, to name but a few.

Yet another object is to provide improved apparatus and methods that support a range of applications.

Still yet another object is to provide such apparatus and methods which are low-cost, low-power and/or support robust rapid-to-market implementations.

SUMMARY

These and other objects are attained by the invention which provides, in one aspect, an embedded processor comprising a plurality of processing units that each execute processes or threads (collectively, "threads"). One or more execution units are shared by the processing units and execute instructions from the threads. An event delivery mechanism delivers events—such as, by way of non-limiting example, hardware interrupts, software-initiated signaling events ("software events") and memory events—to respective threads without execution of instructions. Each event can be processed by the respective thread without execution of instructions outside that thread.

According to related aspects of the invention, the threads need not be constrained to execute on the same respective processing units during the lives of those threads. In still other related aspects, the execution units execute instructions from the threads without needing to know what threads those instructions are from.

The invention provides, in other aspects, an embedded processor as described above that additionally comprises a pipeline control unit which launches instructions from plural threads for concurrent execution on plural execution units. That pipeline control unit can comprise a plurality of instruc-

US 7,685,607 B2

3

tion queues, each associated with a respective one of the virtual processing units, from which queues instructions are dispatched. In addition to decoding instruction classes from the instruction queues, the pipeline control unit can control access by the virtual processing units to a resource that provides source and destination registers for the dispatched instructions.

According to other related aspects, among the plural execution units is a branch execution unit. This is responsible for any of instruction address generation, address translation and instruction fetching. The branch execution unit can also maintain state for the virtual processing units. It can be controlled by the pipeline control unit, which signals the branch execution unit as each virtual processing unit instruction queue is emptied.

According to other aspects of the invention, plural virtual processing units as discussed above can execute on one embedded processor. While, in related aspects, those plural virtual processing units can execute on multiple embedded processors.

Further aspects of the invention provide an embedded processor comprising a plurality of processing units, each executing one or more threads, as described above. An event delivery mechanism delivers events as described above to respective threads with which those events are associated, without execution of instructions. As above, the processing units can be virtual processing units. And, as above, they can execute on one or more embedded processors.

Embedded processors and systems as described above are capable of executing multiple threads and processing multiple events simultaneously with little or no latency and with no operating system switching overhead. Threads can range from real time video processing, to Linux operating system functions, to end-use applications or games. As such, embedded processor and systems have application, by way of non-limiting example, in supporting the direct execution of the diverse functions required for multimedia or other devices, like a high-definition digital television, game systems, digital video recorders, video and/or audio players, personal digital assistants, personal knowledge navigators, mobile phones, and other multimedia and non-multimedia devices.

Moreover, embedded processors and systems as described above enable all functions to be developed and executed in a single programming and execution environment without the need for special purpose hardware accelerators, special purpose DSPs, special purpose processors or other special purpose hardware. Where multiple such embedded processors are added together, they work seamlessly, providing greater overall capacity including both more execution capacity and more concurrent thread handling capacity. The addition of those processors is transparent from the perspective of both threads and events that are directed to specific threads. Load balancing is also transparent, in part, because of the way the processor assigns events and threads for processing.

Other aspects of the invention provide digital data processing system having structures and operating as described above, albeit not in an embedded platform.

Yet further aspects of the invention provide methods paralleling the operation of the systems described above.

These and other aspects invention are evident in the drawings and the description follows.

BRIEF DESCRIPTION OF THE DRAWINGS

A more complete understanding of the invention may be attained by reference to the drawings, in which:

4

FIG. 1 depicts a processor module constructed and operated in accord with one practice of the invention;

FIG. 2 contrasts thread processing by a conventional super-scalar processor with that by a processor module constructed and operated in accord with one practice of the invention;

FIG. 3 depicts potential states of a thread executing in a virtual processing unit (or thread processing unit (TPU)) in a processor constructed and operated in accord with one practice of the invention;

FIG. 4 depicts an event delivery mechanism in a processor module constructed and operated in accord with one practice of invention;

FIG. 5 illustrates a mechanism for virtual address to system address translation in a system constructed and operated in accord with one practice of the invention;

FIG. 6 depicts the organization of Level1 and Level2 caches in a system constructed and operated in accord with one practice the invention;

FIG. 7 depicts the L2 cache and the logic used to perform a tag lookup in a system constructed and operated in accord with one practice of invention;

FIG. 8 depicts logic used to perform a tag lookup in the L2 extended cache in a system constructed and operated in accord with one practice invention;

FIG. 9 depicts general-purpose registers, predicate registers and thread state or control registers maintained for each thread processing unit (TPU) in a system constructed and operated in accord with one practice of the invention;

FIG. 10 depicts a mechanism for fetching and dispatching instructions executed by the threads in a system constructed and operated in accord with one practice of the invention;

FIGS. 11-12 illustrate a queue management mechanism used in system constructed and operated in accord with one practice of the invention;

FIG. 13 depicts a system-on-a-chip (SoC) implementation of the processor module of FIG. 1 including logic for implementing thread processing units in accord with one practice of the invention;

FIG. 14 is a block diagram of a pipeline control unit in a system constructed and operated in accord with one practice of the invention;

FIG. 15 is a block diagram of an individual unit queue in a system constructed and operated in accord with one practice of the invention;

FIG. 16 is a block diagram of the branch unit in a system constructed and operated in accord with one practice of the invention;

FIG. 17 is a block diagram of a memory unit in a system constructed and operated in accord with one practice of the invention;

FIG. 18 is a block diagram of a cache unit implementing any of the L1 instruction cache or L1 data cache in a system constructed and operated in accord with one practice of the invention;

FIG. 19 depicts an implementation of the L2 cache and logic of FIG. 7 in a system constructed and operated in accord with one practice of the invention;

FIG. 20 depicts the implementation of the register file in a system constructed and operated in accord with one practice of the invention;

FIGS. 21 and 22 are block diagrams of an integer unit and a compare unit in a system constructed and operated in accord with one practice of the invention;

FIGS. 23A and 23B are block diagrams of a floating point unit in a system constructed and operated in accord with one practice of the invention;

US 7,685,607 B2

5

FIGS. 24A and 24B illustrate use of consumer and producer memory instructions in a system constructed and operated in accord with one practice of the invention;

FIG. 25 is a block diagram of a digital LCD-TV subsystem in a system constructed and operated in accord with one practice of the invention; and

FIG. 26 is a block diagram of a digital LCD-TV or other application subsystem in a system constructed and operated in accord with one practice of the invention.

DETAILED DESCRIPTION

FIG. 1 depicts a processor module 5 constructed and operated in accord with one practice of the invention and referred to occasionally throughout this document and the attached drawings as "SEP". The module can provide the foundation for a general purpose processor, such as a PC, workstation or mainframe computer—though, the illustrated embodiment is utilized as an embedded processor.

The module 5, which may be used singly or in combination with one or more other such modules, is suited inter alia for devices or systems whose computational requirements are parallel in nature and that benefit from multiple concurrently executing applications and/or instruction level parallelism. This can include devices or systems with real-time requirements, those that execute multi-media applications, and/or those with high computational requirements, such as image, signal, graphics and/or network processing. The module is also suited for integration of multiple applications on a single platform, e.g., where there is concurrent application use. It provides for seamless application execution across the devices and/or systems in which it is embedded or otherwise incorporated, as well as across the networks (wired, wireless, or otherwise) or other medium via which those devices and/or systems are coupled. Moreover, the module is suited for peer-to-peer (P2P) applications, as well as those with user interactivity. The foregoing is not intended to be an extensive listing of the applications and environments to which the module 5 is suited, but merely one of examples.

Examples of devices and systems in which the module 5 can be embedded include inter alia digital LCD-TVs, e.g., type shown in FIG. 24, wherein the module 5 is embodied in a system-on-a-chip (SOC) configuration. (Of course, it will be appreciated that the module need not be embodied on a single chip and, rather, can be embodied in any of a multitude of form factors, including multiple chips, one or more circuit boards, one or more separately-housed devices, and/or a combination of the foregoing). Further examples include digital video recorders (DVR) and servers, MP3 servers, mobile phones, applications which integrate still and video cameras, game platforms, universal networked displays (e.g., combinations of digital LCD-TV, networked information/Internet appliance, and general-purpose application platform), G3 mobile phones, personal digital assistants, and so forth.

The module 5 includes thread processing units (TPUs) 10-20, level one (L1) instruction and data caches 22, 24, level two (L2) cache 26, pipeline control 28 and execution (or functional units) 30-38, namely, an integer processing unit, a floating-point processing unit, a compare unit, a memory unit, and a branch unit. The units 10-38 are coupled as shown in the drawing and more particularly detailed below.

By way of overview, TPUs 10-20 are virtual processing units, physically implemented within processor module 5, that are each bound to and process one (or more) process(es) and/or thread(s) (collectively, thread(s)) at any given instant. The TPUs have respective per-thread state represented in

6

general purpose registers, predicate registers, control registers. The TPUs share hardware, such as launch and pipeline control, which launches up to five instructions from any combination of threads each cycle. As shown in the drawing, the TPUs additionally share execution units 30-38, which independently execute launched instructions without the need to know what thread they are from.

By way of further overview, illustrated L2 cache 26 is shared by all of the thread processing units 10-20 and stores instructions and data on storage both internal (local) and external to the chip on which the module 5 is embodied. Illustrated L1 instruction and data caches 22, 24, too, are shared by the TPUs 10-20 and are based on storage local to the aforementioned chip. (Of course, it will be appreciated that, in other embodiments, the level1 and level2 caches may be configured differently—e.g., entirely local to the module 5, entirely external, or otherwise).

The design of module 5 is scalable. Two or more modules 5 may be "ganged" in an SoC or other configuration, thereby, increasing the number of active threads and overall processing power. Because of the threading model used by the module 5 and described herein, the resultant increase in TPUs is software transparent. Though the illustrated module 5 has six TPUs 10-20, other embodiments may have a greater number of TPUs (as well, of course, as a lesser number). Additional functional units, moreover, may be provided, for example, boosting the number of instructions launched per cycle from five to 10-15, or higher. As evident in the discussion below of L1 and L2 cache construction, these too may be scaled.

Illustrated module 5 utilizes Linux as an application software environment. In conjunction with multi-threading, this enables real-time and non-real-time applications to run on one platform. It also permits leveraging of open source software and applications to increase product functionality. Moreover, it enables execution of applications from a variety of providers.

Multi-Threading

As noted above, TPUs 10-20 are virtual processing units, physically implemented within a single processor module 5, that are each bound to and process one (or more) thread(s) at any given instant. The threads can embody a wide range applications. Examples useful in digital LCD-TVs, for example, include MPEG2 signal demultiplexing, MPEG2 video decoding, MPEG audio decoding, digital-TV user interface operation, operating system execution (e.g., Linux). Of course, these and/or other applications may be useful in digital LCD TVs and the range of other devices and systems in which the module 5 may be embodied.

The threads executed by the TPUs are independent but can communicate through memory and events. During each cycle of processor module 5, instructions are launched from as many active-executing threads as necessary to utilize the execution or functional units 30-38. In the illustrated embodiment, a round robin protocol is imposed in this regard to assure "fairness" to the respective threads (though, in other embodiments, priority or other protocols can be used instead or in addition). Although one or more system threads may be executing on the TPUs (e.g., to launch application, facilitate thread activation, and so forth), no operating system intervention is required to execute active threads.

The underlying rationales for supporting multiple active threads (virtual processors) per processor are:

Functional Capability

Multiple active threads per processor enables a single multi-threaded processor to replace multiple application, media, signal processing and network processors. It also

US 7,685,607 B2

7

enables multiple threads corresponding to application, image, signal processing and networking to operate and inter-operate concurrently with low latency and high performance. Context switching and interfacing overhead is minimized. Even within a single image processing application, like MP4

decode, threads can easily operate simultaneously in a pipelined manner to for example prepare data for frame n+1 while frame n is being composited.

Performance

Multiple active threads per processor increases the performance of the individual processor by better utilizing functional units and tolerating memory and other event latency. It is not unusual to gain a 2x performance increase for supporting up to four simultaneously executing threads. Power consumption and die size increases are negligible so that performance per unit power and price performance are improved. Multiple active threads per processor also lowers the performance degradation due to branches and cache misses by having another thread execute during these events. Additionally, it eliminates most context switch overhead and lowers latency for real-time activities. Moreover, it supports a general, high performance event model.

Implementation

Multiple active threads per processor leads to simplification of pipeline and overall design. There is no need for a complex branch predication, since another thread can run. It leads to lower cost of single processor chips vs. multiple processor chips, and to lower cost when other complexities are eliminated. Further, it improves performance per unit power.

FIG. 2 contrasts thread processing by a conventional super-scalar processor with that of the illustrated processor module 5. Referring to FIG. 2A, in a super-scalar processor, instructions from a single executing thread (indicated by diagonal stippling) are dynamically scheduled to execute on available execution units based on the actual parallelism and dependencies within the code being executed. This means that on the average most execution units are not able to be utilized during each cycle. As the number of execution units increases the percentage utilization typically goes down. Also execution units are idle during memory system and branch prediction misses/waits.

In contrast, referring to FIG. 2B, in the module 5, instructions from multiple threads (indicated by different respective stippling patterns) execute simultaneously. Each cycle, the module 5 schedules instructions from multiple threads to optimally utilize available execution unit resources. Thus the execution unit utilization and total performance is higher, while at the same time transparent to software.

Events and Threads

In the illustrated embodiment, events include hardware (or device) events, such as interrupts; software events, which are equivalent to device events but are initiated by software instructions and memory events, such as completion of cache misses or resolution of memory producer-consumer (full-empty) transitions. Hardware interrupts are translated into device events which are typically handled by an idle thread (e.g., a targeted thread or a thread in a targeted group). Software events can be used, for example, to allow one thread to directly wake another thread.

Each event binds to an active thread. If a specific thread binding doesn't exist, it binds to the default system thread which, in the illustrated embodiment, is always active. That thread then processes the event as appropriate including scheduling a new thread on a virtual processor. If the specific thread binding does exist, upon delivery of a hardware or

8

software event (as discussed below in connection with the event delivery mechanism), the targeted thread is transitioned from idle to executing. If the targeted thread is already active and executing, the event is directed to default system thread for handling.

In the illustrated embodiment, threads can become non-executing (block) due to: Memory system stall (short term blockage), including cache miss and waiting on synchronization; Branch miss-prediction (very short term blockage); Explicitly waiting for an event (either software or hardware generated); and System thread explicitly blocking application thread.

In preferred embodiments of the invention, events operate across physical processors modules 5 and networks providing the basis for efficient dynamic distributed execution environment. Thus, for example, a module 5 executing in a digital LCD-TV or other device or system can execute threads and utilize memory dynamically migrated over a network (wireless, wired or otherwise) or other medium from a server or other (remote) device. The thread and memory-based events, for example, assure that a thread can execute transparently on any module 5 operating in accord with the principles hereof. This enables, for example, mobile devices to leverage the power of other networked devices. It also permits transparent execution of peer-to-peer and multi-threaded applications on remote networked devices. Benefits include increased performance, increased functionality and lower power consumption. Threads run at two privilege levels, System and Application. System threads can access all state of its thread and all other threads within the processor. An application thread can only access non-privileged state corresponding to itself. By default thread 0 runs at system privilege. Other threads can be configured for system privilege when they are created by a system privilege thread.

Referring to FIG. 3, in the illustrated embodiment, thread states are:

Idle (or Non-active)

Thread context is loaded into a TPU and thread is not executing instructions. An Idle thread transitions to Executing, e.g., when a hardware or software event occurs.

Waiting (or Active, Waiting)

Thread context is loaded into a TPU, but is currently not executing instructions. A Waiting thread transitions to Executing when an event it is waiting for occurs, e.g., a cache operation is completed that would allow the memory instruction to proceed.

Executing (or Active, Executing)

Thread context is loaded into a TPU and is currently executing instructions. A thread transitions to Waiting, e.g., when a memory instruction must wait for cache to complete an operation, e.g. a cache miss or an Empty/Fill (producer-consumer memory) instruction cannot be completed. A thread transitions to idle when an event instruction is executed.

A thread enable bit (or flag or other indicator) associated with each TPU disables thread execution without disturbing any thread state for software loading and unloading of a thread onto a TPU.

The processor module 5 load balances across active threads based on the availability of instructions to execute. The module also attempts to keep the instruction queues for each thread uniformly full. Thus, the threads that stay active the most will execute the most instructions.

Events

FIG. 4 shows an event delivery mechanism in a system according to the one practice of the invention. When an event

US 7,685,607 B2

9

is signaled to a thread, the thread suspends execution (if currently in the Executing state) and recognizes the event by executing the default event handler, e.g., at virtual address 0x0.

In the illustrated embodiment, there are five different event types that can be signaled to a specific thread:

Event	Description	Thread Delivery
Thread wait timeout	The timeout value from a wait instruction executed by thread _n has expired	thread _n
Thread exception	Executing instruction has signaled exception.	thread _n
HW Event	Event (like interrupt) generated by hardware device.	thread _n , as determined by event to thread lookup instruction specifies thread.
SW Event	Event (like sw interrupt) signaled by sw event instruction	If that thread is not Active, Waiting or Idle delivered to default system thread
Memory Event	All pending memory operations for a thread _n have completed.	thread _n

Illustrated Event Queue 40 stages events presented by hardware devices and software-based event instructions (e.g., software “interrupts”) in the form of tuples comprising virtual thread number (VTN) and event number:

63	32	31	16	15	4	3	2	1	0
		threadnum		eventnum			how	priv	

Bit	Field	Description
0	priv	Privilege that the event will be signaled at: 0. System privilege 1. Application privilege
1	how	Specifies how the event is signaled if the thread is not in idle state. If the thread is in idle state, this field is ignored and the event is directly signalled 0. Wait for thread in idle state. All events after this event in the queue wait also. 1. Trap thread immediately
15:4	eventnum	Specifies the logical number for this event. The value of this field is captured in detail field of the system exception status or application exception status register.
31:16	threadnum	Specifies the logical thread number that this event is signaled to.

Of course, it will be appreciated that the events presented by the hardware devices and software instructions may be presented in other forms and/or containing other information.

The event tuples are, in turn, passed in the order received to the event-to-thread lookup table (also referred to as the event table or thread lookup table) 42, which determines which TPU is currently handling each indicated thread. The events are then presented, in the form of “TPU events” comprised of event numbers, to the TPUs (and, thereby, their respective threads) via the event-to-thread delivery mechanism 44. If no thread is yet instantiated to handle a particular event, the corresponding event is passed to a default system thread active on one of the TPUs.

The event queue 40 can be implemented in hardware, software and/or a combination thereof. In the embedded, system-

10

on-a-chip (SoC) implementation represented by module 5, the queue is implemented as a series of gates and dedicated buffers providing the requisite queuing function. In alternate embodiments, it is implemented in software (or hardware) linked lists, arrays, or so forth.

The table 42 establishes a mapping between an event number (e.g., hardware interrupt) presented by a hardware device or event instruction and the preferred thread to signal the event to. The possible cases are:

No entry for event number: signal to default system thread. Present to thread: signal to specific thread number if thread is in Executing, Active or Idle, otherwise signal to specified system thread

The table 42 may be a single storage area, dedicated or otherwise, that maintains an updated mapping of events to threads. The table may also constitute multiple storage areas, distributed or otherwise. Regardless, the table 42 may be implemented in hardware, software and/or a combination thereof. In the embedded, SoC implementation represented by module 5, the table is implemented by gates that perform “hardware” lookups on dedicated storage area(s) that maintains an updated mapping of events to threads. That table is software-accessible, as well—for example, by system-level privilege threads which update the mappings as threads are newly loaded into the TPUs 10-20 and/or deactivated and unloaded from them. In turn embodiments, the table 42 is implemented by a software-based lookup of the storage area that maintains the mapping.

The event-to-thread delivery mechanism 44, too, may be implemented in hardware, software and/or a combination thereof. In the embedded, SoC implementation represented by module 5, the mechanism 44 is implemented by gates (and latches) that route the signaled events to TPU queues which, themselves, are implemented as a series of gates and dedicated buffers 46-48 for queuing be delivered events. As above, in alternate embodiments, the mechanism 44 is implemented in software (or other hardware structures) providing the requisite functionality and, likewise, the queues 46-48 are implemented in software (or hardware) linked lists, arrays, or so forth.

An outline of a procedure for processing hardware and software events (i.e., software-initiated signalling events or “software interrupts”) in the illustrated embodiment is as follows:

1. Event is signalled to the TPU which is currently executing active thread.
2. That TPU suspends execution of active thread. The Exception Status, Exception IP and Exception MemAddress control registers are set to indicate information corresponding to the event based on the type of event. All—Thread State is valid.
3. The TPU initiates execution at system privilege of the default event handler at virtual address 0x0 with event signaling disabled for the corresponding thread unit. GP registers 0-3 contain and predicate registers 0-1 are utilized as scratch 20 registers by the event handlers and are system privilege. By convention GP[0] is the event processing stack pointer.
4. The event handler saves enough state so that it can make itself re-entrant and re-enable event signaling for the corresponding thread execution unit.
5. Event handler then processes the event, which could just be posting the event to a SW based queue or taking some other action.
6. The event handler then restores state and returns to execution of the original thread.

US 7,685,607 B2

11

Memory-related events are handled only somewhat differently. The Pending (Memory) Event Table (PET) 50 holds entries for memory operations (from memory reference instructions) which transition a thread from executing to waiting. The table 50, which may be implemented like the event-to-thread lookup table 42, holds the address of the pending memory operation, state information and thread ID which initiated the reference. When a memory operation is completed corresponding to an entry in the PET and no other pending operations are in the PET for that thread, an PET event is signaled to the corresponding thread.

An outline of memory event processing according to the illustrated embodiment is as follows:

1. Event is signal to unit which is currently executing active thread
2. If the thread is in active-wait state and the event is a Memory Event the thread transitions to active-executing and continues execution at the current IP. Otherwise the event is ignored.

As further shown in the drawing, in the illustrated embodiment, thread wait timeouts and thread exceptions are signalled directly to the threads and are not passed through the event-to-thread delivery mechanism 44.

Traps

The goal of multi-threading and events is such that normal program execution of a thread is not disturbed. The events and interrupts which occur get handled by the appropriate thread that was waiting for the event. There are cases where this is not possible and normal processing must be interrupted. SEP supports trap mechanism for this purpose. A list of actions based on event types follows, with a full list of the traps enumerated in the System Exception Status Register.

Event Type	Thread State	Privilege Level	Action
Application event	Idle	Application or System	Recognize event by transitioning to execute state, application priv
System event	Idle	Application or System	System trap to recognize event, transition to execute state
Application event (wait for idle)	Waiting or executing	Application or System	Event stays queued until idle
Application event (trap if not idle)	Waiting or executing	Application or System	Application trap to recognize event
System event	Waiting or executing	Application	System trap to recognize event
System event (wait for idle)	Waiting or executing	System	Event stays queued until idle
System event (trap if not idle)	Waiting or executing	System	System trap to recognize event
ApplicationTrap	Any	Application	Application trap
ApplicationTrap	Any	System	System trap
System Trap	Any	Application	System trap, system privilege level
System Trap	Any	System	System trap

Illustrated processor module 5 takes the following actions when a trap occurs:

1. The IP (Instruction Pointer) specifying the next instruction to be executed is loaded in the Exception IP register.
2. The Privilege Level is stored into bit0 of Exception IP register.
3. The Exception type is loaded into Exception State register

12

4. If the exception is related to a memory unit instruction, the memory address corresponding to exception is loaded into Exception Memory Address register.
5. Current privilege level is set to system.
6. IP (Instruction Pointer) is cleared (zero).
7. Execution begins at IP 0.

Virtual Memory and Memory System

The illustrated processor module 5 utilizes a virtual memory and memory system architecture having a 64-bit Virtual Address (VA) space, a 64-bit System Address (SA) (having different characteristics than a standard physical address), and a segment model of virtual address to system address translation with a sparsely filled VA or SA.

All memory accessed by the TPUs 10-20 is effectively managed as cache, even though off-chip memory may utilize DDR DRAM or other forms of dynamic memory. Referring back to FIG. 1, in the illustrated embodiment, the memory system consists of two logical levels. The level1 cache, which is divided into separate data and instruction caches, 24, 22, respectively, for optimal latency and bandwidth. Illustrated level2 cache 26 consists of an on-chip portion and off-chip portion referred to as level2 extended. As a whole, the level2 cache is the memory system for the individual SEP processor(s) 5 and contributes to a distributed "all cache" memory system in implementations where multiple SEP processors 5 are used. Of course, it will be appreciated that those multiple processors would not have to be physically sharing the same memory system, chips or buses and could, for example, be connected over a network or otherwise.

FIG. 5 illustrates VA to SA translation used in the illustrated system, which translation is handled on a segment basis, where (in the illustrated embodiment) those segments can be of variable size, e.g., 2^{24} - 2^{48} bytes. The SAs are cached in the memory system. So an SA that is present in the memory system has an entry in one of the levels of cache 22/24, 26. An SA that is not present in any cache (and the memory system) is effectively not present in the memory system. Thus, the memory system is filled sparsely at the page (and subpage) granularity in a way that is natural to software and OS, without the overhead of page tables on the processor.

In addition to the foregoing the virtual memory and memory system architecture of the illustrated embodiment has the following additional features: Direct support for distributed shared: Memory (DSM), Files (DSF), Objects (DSO), Peer to Peer (DSP2P); Scalable cache and memory system architecture; Segments that can be shared between threads; Fast level1 cache, since lookup is in parallel with tag access, with no complete virtual-to-physical address translation or complexity of virtual cache.

Virtual Memory Overview

A virtual address in the illustrated system is the 64-bit address constructed by memory reference and branch instructions. The virtual address is translated on a per segment basis to a system address which is used to access all system memory and I/O devices. Each segment can vary in size from 2^{24} to 2^{48} bytes. More specifically, referring to FIG. 5, the virtual address 50 is used to match an entry in a segment table 52 in the manner shown in the drawing. The matched entry 54 specifies the corresponding system address, when taken in combination with the components of the virtual address identified in drawing. In addition, the matched entry 54 specifies the corresponding segment size and privilege. That system address, in turn, maps in to the system memory—which in the illustrated embodiment comprises 2^{64} bytes sparsely filled. The illustrated embodiment permits address translation to be

US 7,685,607 B2

13

disabled by threads with system privilege in which case the segment table is bypassed and all addresses are truncated to the low 32 bits.

Illustrated segment table 52 comprises 16-32 entries per thread (TPU). The table may be implemented in hardware, software and/or a combination thereof. In the embedded, SoC implementation represented by module 5, the table is implemented in hardware, with separated entries in memory being provided for each thread (e.g., a separate table per thread). A segment can be shared among two or more threads by setting up a separate entry for each thread that points to the same system address. Other hardware or software structures may be used instead, or in addition, for this purpose.

Cache Memory System Overview

As noted above, the Level1 cache is organized as separate level1 instruction cache 22 and level1 data cache 24 to maximize instruction and data bandwidth.

Referring to FIG. 6, the on-chip L2 cache 26a consists of the tag and data portions. In the illustrated embodiment, it is 0.5-1 Mbytes in size, with 128 blocks, 16-way associative. Each block stores 128 bytes data or 16 extended L2 tags, with 64 kbytes are provided to store the extended L2 tags. A tag-mode bit within the tag indicates that the data portion consists of 16 tags for Extended L2 Cache.

The extended L2 cache 26b is, as noted above, DDR DRAM-based, though other memory types can be employed. In the illustrated embodiment, it is up to 1 gbyte in size, 256-way associative, with 16 k byte pages and 128 byte subpages. For a configuration of 0.5 mbyte L2 cache 26a and 1 gbyte L2 extended cache 26b, only 12% of on-chip L2 cache is required to fully describe L2 extended. For larger on-chip L2 or smaller L2 extended sizes the percentage is lower. The aggregation of L2 caches (on-chip and extended) make up the distributed SEP memory system.

In the illustrated embodiment, both the L1 instruction cache 22 and the L1 data cache 24 are 8-way associative with 32 k bytes and 128 byte blocks. As shown in the drawing, both level1 caches are proper subsets of level2 cache. The level2 cache consists of an on-chip and off chip extended L2 Cache.

FIG. 7 depicts the L2 cache 26a and the logic used in the illustrated embodiment to perform a tag lookup in L2 cache 26a to identify a data block 70 matching an L2 cache address 78. In the illustrated embodiment, that logic includes sixteen Cache Tag Array Groups 72a-72p, corresponding Tag Compare elements 74a-74p and corresponding Data Array Groups 76a-76p. These are coupled as indicated to match an L2 cache address 78 against the Group Tag Arrays 72a-72p, as shown, and to select the data block 70 identified by the indicated Data Array Group 76a-76p, again, as shown.

The Cache Tag Array Groups 72a-72p, Tag Compare elements 74a-74p, corresponding Data Array Groups 76a-76p may be implemented in hardware, software and/or a combination thereof. In the embedded, SoC implementation represented by module 5, these are implemented in as shown in FIG. 19, which shows the Cache Tag Array Groups 72a-72p embodied in 32x256 single port memory cells and the Data Array Groups 76a-76p embodied in 128x256 single port memory cells, all coupled with current state control logic 190 as shown. That element is, in turn, coupled to state machine 192 which facilitates operation of the L2 cache unit 26a in a manner consistent herewith, as well as with a request queue 192 which buffers requests from the L1 instruction and data caches 22, 24, as shown.

The logic element 190 is further coupled with DDR DRAM control interface 26c which provides an interface to the off-chip portion 26b of the L2 cache. It is likewise coupled to AMBA interface 26d providing an interface to AMBA-

14

compatible components, such as liquid crystal displays (LCDs), audio out interfaces, video in interfaces, video out interfaces, network interfaces (wireless, wired or otherwise), storage device interfaces, peripheral interfaces (e.g., USB, USB2), bus interfaces (PCI, ATA), to name but a few. The DDR DRAM interface 26c and AMBA interface 26d are likewise coupled to an interface 196 to the L1 instruction and data caches by way of L2 data cache bus 198, as shown.

FIG. 8 likewise depicts the logic used in the illustrated embodiment to perform a tag lookup in L2 extended cache 26b and to identify a data block 80 matching the designated address 78. In the illustrated embodiment, that logic includes Data Array Groups 82a-82p, corresponding Tag Compare elements 84a-84p, and Tag Latch 86. These are coupled as indicated to match an L2 cache address 78 against the Data Array Groups 72a-72p, as shown, and to select a tag from one of those groups that matches the corresponding portion of the address 78, again, as shown. The physical page number from the matching tag is combined with the index portion of the address 78, as shown, to identify data block 80 in the off chip memory 26b.

The Data Array Groups 82a-82p and Tag Compare elements 84a-84p may be implemented in hardware, software and/or a combination thereof. In the embedded, SoC implementation represented by module 5, these are implemented in gates and dedicated memory providing the requisite lookup and tag comparison functions. Other hardware or software structures may be used instead, or in addition, for this purpose.

The following is a pseudo-code illustrates L2 and L2E cache operation in the illustrated embodiment:

```
L2 tag lookup, if hit respond back with data to L1 cache
else L2E tag lookup, if hit
    allocate tag in L2;
    access L2E data, store in corresponding L2 entry;
    respond back with data to L1 cache;
else extended L2E tag lookup
    allocate L2E tag;
    allocate tag in L2;
    access L2E data, store in corresponding L2 entry;
    respond back with data to L1 cache;
```

Thread Processign Unit State

Referring to FIG. 9, the illustrated embodiment has six TPUs supporting up to six active threads. Each TPU 10-20 includes general-purpose registers, predicate registers, and control registers, as shown in FIG. 9. Threads at both system and application privilege levels contain identical state, although some thread state information is only visible when at system privilege level—as indicated by the key and respective stippling patterns. In addition to registers, each TPU additionally includes a pending memory event table, an event queue and an event-to-thread lookup table, none of which are shown in FIG. 9.

Depending on the embodiment, there can be from 48 (or fewer) to 128 (or greater) general-purpose registers, with the illustrated embodiment having 128; 24 (or fewer) to 64 (or greater) predicate registers, with the illustrated embodiment having 32; six (or fewer) to 256 (or greater) active threads, with the illustrated embodiment having 8; a pending memory event table of 16 (or fewer) to 512 (or greater) entries, with the illustrated embodiment having 16; a number of pending memory events per thread, preferably of at least two (though potentially less); an event queue of 256 (or greater, or fewer); and an event-to-thread lookup table of 16 (or fewer) to 256 (or greater) entries, with the illustrated embodiment having 32.

US 7,685,607 B2

15

General Purpose Registers

In the illustrated embodiment, each thread has up to 128 general purpose registers depending on the implementation. General Purpose registers 3-0 (GP[3:0]) are visible at system privilege level and can be utilized for event stack pointer and working registers during early stages of event processing.

Predication Registers

The predicate registers are part of the general purpose SEP predication mechanism. The execution of each instruction is conditional based on the value of the reference predicate register.

16

The SEP provides up to 64 one-bit predicate registers as part of thread state. Each predicate register holds what is called a predicate, which is set to 1 (true) or reset to 0 (false) based on the result of executing a compare instruction. Predicate registers 3-1 (PR[3:1]) are visible at system privilege level and can be utilized for working predicates during early stages of event processing. Predicate register 0 is read only and always reads as 1, true. It is by instructions to make their execution unconditional.

Control Registers

Thread State Register

63	20	19	18	17	16	15	8	6	5	4	3	2	1	0
		daddr	iaddr	align	endian	mod			Thread state	priv	tenable	atrapen	strapen	

Bit	Field	Description	Privilege	Per	Design Usage
0	strapen	System trap enable. On reset cleared. Signalling of system trap resets this bit and atrapen until it is set again by software when it is once again re-entrant. 0- System traps disabled 1- Events enabled	system_rw	Thread	Branch
1	atrapen	Application trap enable. On reset cleared. Signalling of application trap resets this bit until it is set again by software when it is once again re-entrant. Application trap is cause by an event that is marked as application level when the privilege level is also application level 0- Events disabled (events are disabled on event delivery to thread) 1- Events enabled	app_rw	Thread	
2	tenable	Thread Enable. On reset set for thread 0, cleared for all other threads 0- Thread operation is disabled. System thread can load or store thread state. 1- Thread operation is enabled.	System_rw	Thread	Branch
3	priv	Privilege level. On reset cleared. 0- System privilege 1- Application privilege	System_rw App_r	Thread	Branch
5:4	state	Thread State. On reset set to "executing" for thread0, set to "idle" for all other threads. 0- Idle 1- reserved 2- Waiting 3- Executing	System_rw	Thread	Branch
15:8	mod[7:0]	GP Registers Modified. Cleared on reset. bit 8 registers 0-15 bit 9 registers 16-31 bit 10 registers 32-47 bit 11 registers 48-63 bit 12 registers 63-79 bit 13 registers 80-95 bit 14 registers 96-111 bit 15 registers 112-127	App_rw	Thread	Pipe
16	endian	Endian Mode-On reset cleared. 0- little endian 1- big endian	System_rw App_r	Proc	Mem
17	align	Alignment check-When clear, unaligned memory references are allowed. When set, all un-aligned memory references result in unaligned data reference fault. On reset cleared.	System_rw App_r	Proc	Mem
18	iaddr	Instruction address translation enable. On reset cleared. 0- disabled 1- enabled	System_rw App_r	Proc	Branch
19	daddr	Data address translation enable. On reset cleared. 0- disabled 1- enabled	System_rw App_r	Proc	Mem

US 7,685,607 B2

17

ID Register

63	32	39	32	31	16	15	8	7	0
				thread_id		id		type	

Bit	Field	Description	Privilege	Per
7:0	type	Processor type and revision[7:0]	read only	Proc
15:8	id	Processor ID[7:0] - Virtual processor number	read only	Thread
31:16	thread_id	Thread ID[15:0]	System_rw App_ro	Thread

Instruction Pointer Register

63	4	3	1	0
Doubleword	mask		0	

Specifies the 64-bit virtual address of the next instruction to be executed.

Bit	Field	Description	Privilege	Per
63:4	Doubleword	Address of instruction doubleword	app	thread
3:2	mask	Indicates which instructions within instruction doubleword remain to be executed. Bit1- first instruction doubleword bit[40:00] Bit2- second instruction doubleword bit[81:41] Bit3- third instruction doubleword, bit[122:82]	app	thread
0		Always read as zero	app	thread

System Exception Status Register

63	4	15	4	3	0
		detail		etype	

Bit	Field	Description	Privilege	Per
3:0	etype	Exception Type 0. none 1. event 2. timer event 3. SystemCall 4. Single Step 5. Protection Fault 6. Protection Fault, system call 7. Memory reference Fault 8. SW event 9. HW fault 10. others	read only	Thread

18

-continued

Bit	Field	Description	Privilege	Per
5	15:4	detail		
10		Fault details- Valid for the following exception types: Memory reference fault details (type 5) 1. None 2. waiting for fill 3. waiting for empty 4. waiting for completion of cache miss 5. memory reference error event (type 1) - Specifies the 12 bit event number		

Application Exception Status Register

63	4	15	4	3	0
		detail		etype	

Bit	Field	Description	Privilege	Per
3:0	etype	Exception Type 0. none 1. event 2. timer event 3. SystemCall 4. Single Step 5. Protection Fault 6. Protection Fault, system call 7. Memory reference Fault 8. SW event 9. HW fault 10. Others	read only	Thread
15:4	detail	Protection Fault details- Valid for the following exception types: event (type 1) - Specifies the 12 bit event number		

System Exception IP

63	4	3	1	0
Doubleword	mask		priv	

Address of instruction corresponding to signaled exception to system privilege. Bit[0] is the privilege level at the time of the exception.

Bit	Field	Description	Privilege	Per
63:4	Doubleword	Address of instruction doubleword which signaled exception	system	thread
3:1	mask	Indicates which instructions within instruction doubleword remain to be executed. Bit1- first instruction doubleword bit[40:00] Bit2- second instruction doubleword bit[81:41]	system	thread

US 7,685,607 B2

19

-continued

Bit	Field	Description	Privilege	Per
		Bit3- third instruction doubleword, bit[122:82]		
0	priv	Privilege level of thread at time of exception	system	thread

Address of instruction corresponding to signaled exception. Bit[0] is the privilege level at the time of the exception.
Application Exception IP

63	4	3	1	0
Doubleword			mask	

Address of instruction corresponding to signaled exception to application privilege.

Bit	Field	Description	Privilege	Per
63:4	Doubleword	Address of instruction doubleword which signaled exception	system	thread
3:1	mask	Indicates which instructions within instruction doubleword remain to be executed. Bit1- first instruction doubleword bit[40:00] Bit2- second instruction doubleword bit[81:41] Bit3- third instruction doubleword, bit[122:82]	system	thread

Address of instruction corresponding to signaled exception. Bit[0] is the privilege level at the time of the exception.
Exception Mem Address

63	0
Address	

Address of memory reference that signaled exception. Valid only for memory faults. Holds the address of the pending memory operation when the Exception Status register indicates memory reference fault, waiting for fill or waiting for empty.

Instruction Seg Table Pointer (ISTP), Data Seg Table Pointer (DSTP)

63	32	31	6	5	1	0
	reserved		ste number		field	

20

Utilized by ISTE and ISTE registers to specify the ste and field that is read or written.

Bit	Field	Description	Privilege	Per
0	field	Specifies the low (0) or high (1) portion of Segment Table Entry	system	thread
10	5:1 ste number	Specifies the STE number that is read into STE Data Register.	system	thread

Instruction Segment Table Entry (ISTE), Data Segment Table Entry (DSTE)

63	0
data	

When read the STE specified by ISTE register is placed in the destination general register. When written, the STE specified by ISTE or DSTE is written from the general purpose source register. The format of segment table entry is specified in Chapter 6-section titled Translation Table organization and entry description.

Instruction or Data Level1 Cache Tag Pointer (ICTP, DCTP)

63	32	15	14	13	7	6	2	1	0
	reserved			index		bank			

Specifies the Instruction Cache Tag entry that is read or written by the ICTE or DCTE.

Bit	Field	Description	Privilege	Per
6:2	bank	Specifies the bank that is read from Level1 Cache Tag Entry. The first implementation has valid banks 0x0-f.	system	thread
13:7	index	Specifies the index address within a bank that is read from Level1 Cache Tag Entry	System	thread

Instruction or Data Level1 Cache Tag Entry (ICTE, DCTE)

63	0
data	

When read the Cache Tag specified by ICTP or DCTP register is placed in the destination general register. When written, the Cache Tag specified by ICTP or DCTP is written from the general purpose source register. The format of cache tag entry is specified in Chapter 6-section titled Translation Table organization and entry description.

US 7,685,607 B2

21

Event Queue Control Register

63	32	31	19	18	17	2	1	0
		address		offset	event		reg_op	

The Event Queue Control Register (EQCR) enables normal and diagnostic access to the event queue. The sequence for using the register is a register write followed by a register read. The contents of the reg_op field specifies the operation for the write and the next read. The actual register modification or read is triggered by the write.

Bit	Field	Description	Privilege	Per
1:0	reg_op	Specifies the register operation for that write and the next read. Valid for register read. 0- read 1- write 2- push onto queue 3- pop from queue	system	proc
17:2	event	For writes and push specifies the event number written or pushed onto the queue. For read and pop operations contains the event number read or popped from the queue	system	proc
18	empty	Indicates whether the queue was empty prior to the current operation.	system	proc
31:19	address	Specifies the address for read and write queue operations. Address field is don't care for push and pop operations.	system	proc

Event-Thread Lookup Table Control

63	9	48	41	40	33	32	17	32	1	0
		address		thread		mask		event		reg_op

The Event to Thread lookup table establishes a mapping between an event number presented by a hardware device or event instruction and the preferred thread to signal the event to. Each entry in the table specifies an event number with a bit mask and a corresponding thread that the event is mapped to.

Bit	Field	Description	Privilege	Per
0	reg_op	Specifies the register operation for that write and the next read. Valid for register read. 0- read 1- write	system	proc
16:1	event[15:0]	For writes specifies the event number written at the specified table address. For read operations contains the event number at the specified table address	system	proc
32:17	mask[15:0]	Specifies whether the corresponding event bit is significant. 0- significant 1- don't care	system	proc

22

-continued

Bit	Field	Description	Privilege	Per
40:33	thread			
48:41	address	Specifies the table address for read and write operations. Address field is don't care for push and pop operations.	system	proc

Timers and Performance Monitor

In the illustrated embodiment, all timer and performance monitor registers are accessible at application privilege.

Clock

63		0
	clock	

Bit	Field	Description	Privilege	Per
63:0	clock	Number of clock cycles since processor reset	app	proc

Instructions Executed

63	32	31	0
		count	

Bit	Field	Description	Privilege	Per
31:0	count	Saturating count of the number of instruction executed. Cleared on read. Value of all 1's indicates that the count has overflowed.	app	thread

Thread Execution Clock

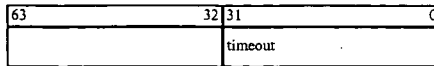
63	32	31	0
		active	

Bit	Field	Description	Privilege	Per
31:0	active	Saturating count of the number of cycles the thread is in active-executing state. Cleared on read. Value of all 1's indicates that the count has overflowed.	app	thread

US 7,685,607 B2

23

Wait Timeout Counter



Bit	Field	Description	Privilege	Per
31:0	timeout	Count of the number of cycles remaining until a timeout event is signaled to thread. Decrements by one, each clock cycle.	app	thread

Virtual Processor and Thread ID

In the illustrated embodiment, each active thread corresponds to a virtual processor and is specified by a 8-bit active thread number (activethread[7:0]). The module 5 supports a 16-bit thread ID (threaded[15:0]) to enable rapid loading (activation) and unloading (de-activation) of threads. Other embodiments may support thread IDs of different sizes.

Thread-Instruction Fetch Abstraction

As noted above, the TPUs 10-20 of module 5 share L1 instruction cache 22, as well as pipeline control hardware that launches up to five instructions each cycle from any combination of the threads active in those TPUs. FIG. 10 is an abstraction of the mechanism employed by module 5 to fetch and dispatch those instructions for execution on functional units 30-38.

As shown in that drawing, during each cycle, instructions are fetched from the L1 cache 22 and placed in instruction queues 10a-20a associated with each respective TPU 10-20. This is referred to as the fetch stage of the cycle. In the illustrated embodiment, three to six instructions are fetched for each single thread, with an overall goal of keeping thread queues 10a-20a at equal levels. In other embodiments, different numbers of instructions may be fetched and/or different goals set for relative filling of the queues. Also during the fetch stage, the module 5 (and, specifically, for example, the event handling mechanisms discussed above) recognize events and transition corresponding threads from waiting to executing.

During the dispatch stage—which executes in parallel with the fetch and execute/retire stages—instructions from each of one or more executing threads are dispatched to the functional units 30-38 based on a round-robin protocol that takes into account best utilization of those resources for that cycle. These instructions can be from any combination of threads. The compiler specifies, e.g., utilizing “stop” flags provided in the instruction set, boundaries between groups of instructions within a thread that can be launched in a single cycle. In other embodiments, other protocols may be employed, e.g., ones that prioritize certain threads, ones that ignore resource utilization, and so forth.

During the execute & retire phase—which executes in parallel with the fetch and dispatch stages—multiple instructions are executed from one or more threads simultaneously. As noted above, in the illustrated embodiment, up to five instructions are launched and executed each cycle, e.g., by the integer, floating, branch, compare and memory functional units 30-38. In other embodiments, greater or fewer instructions

24

tions can be launched, for example, depending on the number and type of functional units and depending on the number of TPUs.

An instruction is retired after execution if it completes: its result is written and the instruction is cleared from the instruction queue.

On the other hand, if an instruction blocks, the corresponding thread is transitioned from executing to waiting. The blocked instruction and all instructions following it for the corresponding thread are subsequently restarted when the condition that caused the block is resolved. FIG. 11 illustrates a three-pointer queue management mechanism used in the illustrated embodiment to facilitate this.

Referring to that drawing, an instruction queue and a set of three pointers is maintained for each TPU 10-20. Here, only a single such queue 110 and set of pointers 112-116 is shown. The queue 110 holds instructions fetched, executing and retired (or invalid) for the associated TPU—and, more particularly, for the thread currently active in that TPU. As instructions are fetched, they are inserted at the queue's top, which is designated by the Insert (or Fetch) pointer 112. The next instruction for execution is identified by the Extract (or Issue) pointer 114. The Commit pointer 116 identifies the last instruction whose execution has been committed. When an instruction is blocked or otherwise aborted, the Commit pointer 116 is rolled back to quash instructions between Commit and Extract pointers in the execution pipeline. Conversely, when a branch is taken, the entire queue is flushed and the pointers reset.

Though the queue 110 is shown as circular, it will be appreciated that other configurations may be utilized as well. The queuing mechanism depicted in FIG. 11 can be implemented, for example, as shown in FIG. 12. Instructions are stored in dual ported memory 120 or, alternatively, in a series of registers (not shown). The write address at which each newly fetched instruction is stored is supplied by Fetch pointer logic 122 that responds to a Fetch command (e.g., issued by the pipeline control) to generate successive addresses for the memory 120. Issued instructions are taken from the other port, here, shown at bottom. The read address from which each instruction is taken is supplied by Issue/Commit pointer logic 124. That logic responds to Commit and Issue commands (e.g., issued by the pipeline control) to generate successive addresses and/or to reset, as appropriate.

Processor Module Implementation

FIG. 13 depicts an SoC implementation of the processor module 5 of FIG. 1 including, particularly, logic for implementing the TPUs 10-20. As in FIG. 1, the implementation of FIG. 13 includes L1 and L2 caches 22-26, which are constructed and operated as discussed above. Likewise, the implementation includes functional units 30-34 comprising an integer unit, a floating-point unit, and a compare unit, respectively. Additional functional units can be provided instead or in addition. Logic for implementing the TPUs 10-20 includes pipe-line control 130, branch unit 38, memory unit 36, register file 136 and load-store buffer 138. The components shown in FIG. 13 are interconnected for control and information transfer as shown, with dashed lines indicating major control, thin solid lines indicating predicate value control, thicker solid lines identifying a 64-bit data bus and still thicker lines identifying a 128-bit data bus. It will be appreciated that FIG. 13 represents one implementation of a processor module 5 according to invention and that other implementations may be realized as well.

US 7,685,607 B2

25

Pipeline Control Unit

In the illustrated embodiment, pipeline control 130 contains the per-thread queues discussed above in connection with FIGS. 11-12. There can be parameterized at 12, 15 or 18 instructions per thread. The control 130 picks up instructions from those queues on a round robin basis (though, as also noted, this can be performed on other bases as well). It controls the sequence of accesses to the register file 136 (which is the resource which provides source and destination registers for the instructions), as well as to the functional units 30-38. The pipeline control 130 decodes basic instruction classes from the per-thread queues and dispatches instructions to the functional units 30-38. As noted above, multiple instructions from one or more threads can be scheduled for execution by those functional units in the same cycle. The control 130 is additionally responsible for signaling the branch unit 38 as it empties the per-thread construction queues, and for idling the functional units when possible, e.g., on a cycle by cycle basis, to decrease our consumption.

FIG. 14 is a block diagram of the pipeline control unit 130. The unit includes control logic 140 for the thread class queues, the thread class (or per-thread) queues 142 themselves, an instruction dispatch 144, a longword decode unit 146, and functional units queues 148a-148e, connected to one another (and to the other components of module 5) as shown in the drawing. The thread class (per-thread) queues are constructed and operated as discussed above in connection with FIGS. 11-12. The thread class queue control logic 140 controls the input side of those queues 142 and, hence, provides the Insert pointer functionality shown in FIGS. 11-12 and discussed above. The control logic 140 is also responsible for controlling the input side of the unit queues 148a-148e, and for interfacing with the branch unit 38 to control instruction fetching. In this latter regard, logic 140 is responsible for balancing instruction fetching in the manner discussed above (e.g., so as to compensate for those TPUs that are retiring the most instructions).

The instruction dispatch 144 evaluates and determines, each cycle, the schedule of available instructions in each of the thread class queues. As noted above, in the illustrated embodiment the queues are handled on a round robin basis with account taken for queues that are retiring instructions more rapidly. The instruction dispatch 144 also controls the output side of the thread class queues 142. In this regard, it manages the Extract and Commit pointers discussed above in connection with FIGS. 11-12, including updating the Commit pointer when instructions have been retired and rolling that pointer back when an instruction is aborted (e.g., for thread switch or exception).

The longword decode unit 146 decodes incoming instruction longwords from the L1 instruction cache 22. In the illustrated embodiment, each such longword is decoded into the instructions. This can be parameterized for decoding one or two longwords, which decode into three and six instructions, respectively. The decode unit 146 is also responsible for decoding the instruction class of each instruction.

Unit queues 148a-148e queue actual instructions which are to be executed by the functional units 30-38. Each queue is organized on a per-thread basis and is kept consistent with the class queues. The unit queues are coupled to the thread class queue control 140 and to the instruction dispatch 144 for control purposes, as discussed above. Instructions from the queues 148a-148e are transferred to corresponding pipelines 150a-150b en route to the functional units themselves 30-38. The instructions are also passed to the register file pipeline 152.

26

FIG. 15 is a block diagram of an individual unit queue, e.g., 148a. This includes one instruction queue 154a-154e for each TPU. These are coupled to the thread class queue control 140 (labeled *tcqueue_ctl*) and the instruction dispatch 144 (labeled *idispach*) for control purposes. These are also coupled to the longword decode unit 146 (labeled *lwdecode*) for instruction input and to a thread selection unit 156, as shown. That unit controls thread selection based on control signals provided by instruction dispatch 144, as shown. Output from unit 156 is routed to the corresponding pipeline 150a-150e, as well as to the register file pipeline 152.

Referring back to FIG. 14, integer unit pipeline 150a and floating-point unit pipeline 150b decode appropriate instruction fields for their respective functional units. Each pipeline also times the commands to that respective functional units. Moreover, each pipeline 150a, 150b applies squashing to the respective pipeline based on branching or aborts. Moreover, each applies a powerdown signal to its respective functional unit when it is not used during a cycle. Illustrated compare unit pipeline 150c, branch unit pipeline 150d, and memory unit pipeline 150e, provide like functionality for their respective functional units, compare unit 34, branch unit 38 and memory unit 36. Register file pipeline 150 also provide like functionality with respect to register file 136.

Referring, now, back to FIG. 13, illustrated branch unit 38 is responsible for instruction address generation and address translation, as well as instruction fetching. In addition, it maintains state for the thread processing units 10-20. FIG. 16 is a block diagram of the branch unit 38. It includes control logic 160, thread state stores 162a-162e, thread selector 164, address adder 166, segment translation content addressable memory (CAM) 168, connected to one another (and to the other components of module 5) as shown in the drawing.

The control logic drives unit 38 based on a command signal from the pipeline control 130. It also takes as input the instruction cache 22 state and the L2 cache 26 acknowledgment, as illustrated. The logic 160 outputs a thread switch to the pipeline control 130, as well as commands to the instruction cache 22 and the L2 cache, as illustrated. The thread state stores 162a-162e store thread state for each of the respective TPUs 10-20. For each of those TPUs, it maintains the general-purpose registers, predicate registers and control registers shown in FIG. 3 and discussed above.

Address information obtained from the thread state stores is routed to the thread selector, as shown, which selects the thread address from which and address computation is to be performed based on a control signal (as shown) from the control 160. The address adder 166 increments the selected address or performs a branch address calculation, based on output of the thread selector 164 and addressing information supplied by the register file (labelled *register source*), as shown. In addition, the address adder 166 outputs a branch result. The newly computed address is routed to the segment translation memory 168, which operates as discussed above in connection with FIG. 5, which generates a translated instruction cache address for use in connection with the next instruction fetch.

Functional Units

Turning back to FIG. 13, memory unit 36 is responsible for memory referents instruction execution, including data cache 24 address generation and address translation. In addition, unit 36 maintains the pending (memory) event table (PET) 50, discussed above. FIG. 17 is a block diagram of the memory unit 36. It includes control logic 170, address adder 172, and segment translation content addressable memory (CAM) 174, connected to one another (and to the other components of module 5) as shown in the drawing.

US 7,685,607 B2

27

The control logic drives 170 unit 36 based on a command signal from the pipeline control 130. It also takes as input the data cache 22 state and the L2 cache 26 acknowledgment, as illustrated. The logic 170 outputs a thread switch to the pipeline control 130 and branch unit 38, as well as commands to the data cache 24 and the L2 cache, as illustrated. The address adder 172 increments addressing information provided from the register file 136 or performs a requisite address calculation. The newly computed address is routed to the segment translation memory 174, which operates as discussed above in connection with FIG. 5, which generates a translated instruction cache address for use in connection with a data access. Though not shown in the drawing, the unit 36 also includes the PET, as previously mentioned.

FIG. 18 is a block diagram of a cache unit implementing any of the L1 instruction cache 22 or L2 data cache 24. The unit includes sixteen 128x256 byte single port memory cells 180a-180p serving as data arrays, along with sixteen corresponding 32x56 byte dual port memory cells 182a-182p serving as tag arrays. These are coupled to L1 and L2 address and data buses as shown. Control logic 184 and 186 are coupled to the memory cells and to L1 cache control and L2 cache control, also as shown.

Returning, again, to FIG. 13, the register file 136 serves as the resource for all source and destination registers accessed by the instructions being executed by the functional units 30-38. The register file is implemented as shown in FIG. 20. As shown there, to reduce delay and wiring overhead, the unit 136 is decomposed into a separate register file instance per functional unit 30-38. In the illustrated embodiment, each instance provides forty-eight 64-bit registers for each of the TPUs. Other embodiments may vary, depending on the number of registers allotted the TPUs, the number of TPUs and the sizes of the registers.

Each instance 200a-200e has five write ports, as illustrated by the arrows coming into the top of each instance, via which each of the functional units 30-38 can simultaneously write output data (thereby insuring that the instances retain consistent data). Each provides a varying number of read ports, as illustrated by the arrows emanating from the bottom of each instance, via which their respective functional units obtain data. Thus, the instances associated with the integer unit 30, the floating point unit 32 and the memory unit all have three read ports, the instance associated with the compare unit 34 has two read ports, and the instance associated with the branch unit 38 has one port, as illustrated.

The register file instances 200-200e can be optimized by having all ports read for a single thread each cycle. In addition, storage bits can be folded under wires to port access.

FIGS. 21 and 22 are block diagrams of the integer unit 30 and the compare unit 34, respectively. FIGS. 23A and 23B are block diagrams, respectively, of the floating point unit 32 and the fused multiply-add unit employed therein. The construction and operation of these units is evident from the components, interconnections and labelling supplied with the drawings.

Consumer-Producer Memory

In prior art multiprocessor systems, the synchronization overhead and programming difficulty to implement data-based processing flow between threads or processors (for multiple steps of image processing for example) is very high. The processor module 5 provides memory instructions that permit this to be done easily, enabling threads to wait on the availability of data and transparently wake up when another thread indicates the data is available. Such software transparent consumer-producer memory operations enable higher

28

performance fine grained thread level parallelism with an efficient data oriented, consumer-producer programming style.

The illustrated embodiment provides a "Fill" memory instruction, which is used by a thread that is a data producer to load data into a selected memory location and to associate a state with that location, namely, the "full" state. If the location is already in that state when the instruction is executed, an exception is signalled.

The embodiment also provides an "Empty" instruction, which is used by a data consumer to obtain data from a selected location. If the location is associated with the full state, the data is read from it (e.g., to a designated register) and the instruction causes the location to be associated with an "empty" state. Conversely, if the location is not associated with the full state at the time the Empty instruction is executed, the instruction causes the thread that executed it to temporarily transition to the idle (or, in an alternative embodiment, an active, non-executing) state, re-transitioning it back to the active, executing state—and executing the Empty instruction to completion—once it becomes so associated. Using the Empty instruction enables a thread to execute when its data is available with low overhead and software transparency.

In the illustrated embodiment, it is the pending (memory) event table (PET) 50 that stores status information regarding memory locations that are the subject of Fill and Empty operations. This includes the addresses of those locations, their respective full or empty states, and the identities of the "consumers" of data for those locations, i.e., the threads that have executed Empty instructions and are waiting for the locations to fill. It can also include the identities of the producers of the data, which can be useful, for example, in signalling and tracking causes of exceptions (e.g., as where to successive Fill instructions are executed for the same address, with no intervening Empty instructions).

The data for the respective locations is not stored in the PET 50 but, rather, remains in the caches and/or memory system itself, just like data that is not the subject of Fill and/or Empty instructions. In other embodiments, the status information is stored in the memory system, e.g., alongside the locations to which it pertains and/or in separate tables, linked lists, and so forth.

Thus, for example, when an Empty instruction is executed on a given memory location, the PET is checked to determine whether it has an entry indicating that same location is currently in the full state. If so, that entry is changed to empty and a read is effected, moving data from the memory location to the register designated by the Empty instruction.

If, on the other hand, when the Empty instruction is executed, there no entry in the PET for the given memory location (or if any such entry indicates that the location is currently empty) then an entry is created (or updated) in the PET to indicate that the given location is empty and to indicate that the thread which executed the Empty instruction is a consumer for any data subsequently stored to that location by a Fill instruction.

When a Fill instruction is subsequently executed (presumably, by another thread), the PET is checked to determine whether it has an entry indicating that same location is currently in the empty state. Upon finding such an entry, its state is changed to full, and the event delivery mechanism 44 (FIG. 4) is used to route a notification to the consumer-thread identified in that entry. If that thread is in an active, waiting state in a TPU, the notification goes to that TPU, which enters active, executing state and reexecutes the Empty instruction—this time, to completion (since the

US 7,685,607 B2

29

memory location is now in the full state). If that thread is in the idle state, the notification goes to the system thread (in whatever TPU it is currently executing), which causes the thread to be loaded into a TPU in the executing, active state so that the Empty instruction can be reexecuted.

In the illustrated embodiment, this use of the PET for consumer/producer-like memory operations is only effected with respect to selected memory instructions, e.g., Fill and Empty, but not with the more conventional Load and Store memory instructions. Thus, for example, even if a Load instruction is executed with respect to a memory location that is currently the subject of an Empty instruction, no notification is made to the thread that executed that Empty instruction so that the instruction can be reexecuted. Other embodiments may vary in this regard.

FIGS. 24A depicts three interdependent threads, 230, 232 and 234, the synchronization of and data transfer between which can be facilitated by Fill and Empty instructions according to the invention. By way of example, thread 230 is an MPEG2 demultiplexing thread 230, responsible for demultiplexing an MPEG2 signal obtained, for example, from an MPEG2 source 236, e.g., a tuner, a streaming source or otherwise. It is assumed to be in an active, executing state on TPU 10, to continue the example. Thread 232 is a video decoding Step 1 thread, responsible for a first stage of decoding a video signal from a demultiplexed MPEG2 signal. It is assumed to be in an active, executing state on TPU 12. Thread 234 is a video decoding Step 2 thread, responsible for a second stage of decoding a video signal from a demultiplexed MPEG2 signal for output via an LCD interface 238 or other device. It is assumed to be in an active, executing state on TPU 14.

To accommodate data streaming from the source 236 in real-time, each of the threads 230-234 continually process data provided by its upstream source and does so in parallel with the other threads. FIG. 24B illustrates use of the Fill and Empty instructions to facilitate this in a manner which insures synchronization and facilitates data transfer between the threads.

Referring to the drawing, arrows 240a-240g indicate fill dependencies between the threads and, particularly, between data locations written to (filled) by one thread and read from (emptied) by another thread. Thus, thread 230 processes data destined for address A0, while thread 232 executes an Empty instruction targeted to that location and thread 234 executes an Empty instruction targeted to address B0 (which thread 232 will ultimately Fill). As a result of the Empty instructions, thread 232 enters a wait state (e.g., active, non-executing or idle) while awaiting completion of the Fill of location A0 and thread 234 enters a wait state while awaiting completion of the Fill of location B0.

On completion of thread 230's Fill of A0, thread 232's Empty completes, allowing that thread to process the data from A0, with the result destined for B0 via a Fill instruction. Thread 234 remains in a wait state, still awaiting completion of that Fill. In the meanwhile, thread 230 begins processing data destined for address A1 and thread 232 executes the Empty instruction, placing it in a wait state while awaiting completion of the Fill of A1.

When thread 232 executes the Fill demand for B0, thread 234's Empty completes allowing that thread to process the data from B0, with the result destined for C0, whence it is read by the LCD interface (not shown) for display to the TV viewer. The three threads 230, 232, 234 continue process and executing Fill and Empty instruction in this manner—as illustrated in the drawing—until processing of the entire MPEG2 stream is completed.

30

A further appreciation of the Fill and Empty instructions may be attained by review of their instruction formats.

Empty

Format: ps EMPTY.cache.threads dreg, breg, ireg {,stop}

Description: Empty instructs the memory system to check the state of the effective address. If the state is full, empty instruction changes the state to empty and loads the value into dreg. If the state is already empty, the instruction waits until the instruction is full, with the waiting behavior specified by the thread field.

Operands and Fields:

15	ps	The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
	stop	0 Specifies that an instruction group is not delineated by this instruction. 1 Specifies that an instruction group is delineated by this instruction.
20	thread	0 unconditional, no thread switch 1 unconditional thread switch 2 conditional thread switch on stall (block execution of thread) 3 reserved
	scache	0 tbd with reuse cache hint 1 read/write with reuse cache hint 2 tbd with no-reuse cache hint 3 read/write with no-reuse cache hint
25	im	0 Specifies index register (ireg) for address calculation 1 Specifies disp for address calculation
	ireg	Specifies the index register of the instruction.
	breg	Specifies the base register of the instruction.
30	disp	Specifies the two's complement displacement constant (8-bits) for memory reference instructions
	dreg	Specifies the destination register of the instruction.

35 Fill

Format: ps FILL.cache.threads s1reg, breg, ireg {,stop}

Description: Register s1reg is written to the word in memory at the effective address. The effective address is calculated by adding breg (base register) and either ireg (index register) or disp (displacement) based on the im (immediate memory) field. The state of the effective address is changed to full. If the state is already full an exception is signaled.

Operands and Fields:

	ps	The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
50	stop	0 Specifies that an instruction group is not delineated by this instruction. 1 Specifies that an instruction group is delineated by this instruction.
	thread	0 unconditional, no thread switch 1 unconditional thread switch 2 conditional thread switch on stall (block execution of thread) 3 reserved
55	scache	0 tbd with reuse cache hint 1 read/write with reuse cache hint 2 tbd with no-reuse cache hint 3 read/write with no-reuse cache hint
	im	0 Specifies index register (ireg) for address calculation 1 Specifies disp for address calculation
	ireg	Specifies the index register of the instruction.
	breg	Specifies the base register of the instruction.
	disp	Specifies the two's complement displacement constant (8-bits) for memory reference instructions
60	s1reg	Specifies the register that contains the first operand of the instruction.
65		

US 7,685,607 B2

31

Software Events

A more complete understanding of the processing of hardware and software events may be attained by review of their instruction formats:

Event

Format: ps EVENT s1reg{,stop}

Description: The EVENT instruction polls the event queue for the executing thread. If an event is present the instruction completes with the event status loaded into the exception status register. If no event is present in the event queue, the thread transitions to idle state.

Operands and Fields:

ps	The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
stop	0 Specifies that an instruction group is not delineated by this instruction. 1 Specifies that an instruction group is delineated by this instruction.
s1reg	Specifies the register that contains the first source operand of the instruction.

SW Event

Format ps SWEVENT s1reg {,stop}

Description: The SWEVENT instruction en-queues an event onto the Event Queue to be handled by a thread. See xxx for the event format.

Operands and Fields:

ps	The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
stop	0 Specifies that an instruction group is not delineated by this instruction. 1 Specifies that an instruction group is delineated by this instruction.
s1reg	Specifies the register that contains the first source operand of the instruction.

CtlFLD

Format: ps.CtlFld.ti cfield, {,stop}

Description: The Control Field instruction modifies the control field specified by cfield. Other fields within the control register are unchanged.

Operands and Fields:

ps	The predicate source register that specifies whether the instruction is executed. If true the instruction is executed, else if false the instruction is not executed (no side effects).
stop	0 Specifies that an instruction group is not delineated by this instruction. 1 Specifies that an instruction group is delineated by this instruction.
ti	0 Specifies access to this threads control registers. 1 Specifies access to control register of thread specified by ID.t_indirect field. (thread indirect) (privileged)
cfield	cfield[4:0] control field privilege

32

000nn	Thread state	value	application
nn	nn	value	
00	idle		
01	reserved		
10	waiting		
11	executing		
0010S	System trap enable		system
0011S	Application trap enable		application
0100S	Thread Enable		system
0101S	Privilege Level		System
0110S	Registers Modified		application
0111S	Instruction address translation enable		system
1000S	Data address translation enable		system
1001S	Alignment Check		System
1010S	Endian Mode		system
1011S	reserved		
11**S			

S = 0 clear,
S = 1 set

Devices Incorporating Processor Module 5

FIG. 25 is a block diagram of a digital LCD-TV subsystem 242 according to the invention embodied in a SoC format. The subsystem 242 includes a processor module 5 constructed as described above and operated to execute simultaneously execute threads providing MPEG2 signal demultiplexing, MPEG2 video decoding, MPEG audio decoding, digital-TV user interface operation, and operating system execution (e.g., Linux), e.g., as described above. The module 5 is coupled to DDR DRAM flash memory comprising the off-chip portion of the L2 cache 26, also as discussed above. The module includes an interface (not shown) to an AMBA AHB bus 244, via which it communicates with "intellectual property" or "IP" 246 providing interfaces to other components of the digital LCD-TV, namely, a video input interface, a video output interface, an audio output interface and LCD interface. Of course other IP may be provided in addition or instead, coupled to the module 5 via the AHB bus 5 or otherwise. For example, in the drawing, illustrated module 5 communicates with optional IP via which the digital LCD-TV obtains source signals and/or is controlled, such as DMA engine 248, high speed I/O device controller 250 and low speed device controllers 252 (via APB bridge 254) or otherwise.

FIG. 26 is a block diagram of a digital LCD-TV or other application subsystem 256 according to the invention, again, embodied in a SoC format. The illustrated subsystem is configured as above, except insofar as it is depicted with APB and AHB/APB bridges and APB macros 258 in lieu of the specific IP shown 246 shown in FIG. 24. Depending on application needs, elements 258 may comprise a video input interface, a video output interface, an audio output interface and an LCD interface, as in the implementation above, or otherwise.

The illustrated subsystem further includes a plurality of modules 5, e.g., from one to twenty such modules (or more) that are coupled via an interconnect that interfaces with and, preferably, forms part of the off-chip L2 cache 26b utilized by the modules 5. That interconnect may be in the form of a ring interconnect (RI) comprising a shift register bus shared by the modules 5 and, more particularly, by the L2 caches 26. Alternatively, it may be an interconnect of another form, proprietary or otherwise, that facilitates the rapid movement of data within the combined memory system of the modules 5. Regardless, the L2 caches are preferably coupled so that the L2 cache for any one module 5 is not only the memory system for that individual processor but also contributes to a distributed all cache memory system for all of the processor mod-

US 7,685,607 B2

33

ules 5. Of course, as noted above, the modules 5 do not have to physically sharing the same memory system, chips or buses and could, instead, be connected over a network or otherwise.

Described above is apparatus, systems and methods meeting the desired objects. It will be appreciated that the embodiments described herein are examples of the invention and that other embodiments, incorporating changes therein, fall within the scope of the invention, of which we claim:

The invention claimed is:

1. An embedded processor, comprising

A. a plurality of processing units, each of which execute one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") and one or more of which execute a plurality of threads,

B. one or more execution units that are shared by, and in communication coupling with, the plurality of processing units, the execution units executing instructions from the threads,

C. an event delivery mechanism that delivers events to respective threads with which those events are associated, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, wherein the event delivery mechanism

i. is in communication coupling with the plurality of processing units, and

ii. delivers each such event to the respective thread without execution of instructions by said processing units.

2. The embedded processor of claim 1, wherein the thread to which an event is delivered processes that event without execution of instructions outside that thread.

3. The embedded processor of claim 1, wherein the execution units execute instructions from the threads without need to know what thread they are from.

4. The embedded processor of claim 1, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread.

5. The embedded processor of claim 1, wherein at least one of the processing units is a virtual processing unit.

6. An embedded processor, comprising

A. a plurality of virtual processing units, each executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads"), wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread,

B. a plurality of execution units,

C. a pipeline control that is in communication coupling with the plurality of processing units and with the plurality of execution units, the pipeline control launching instructions from plural ones of the threads for concurrent execution on plural ones of the execution units,

D. an event delivery mechanism that is in communication coupling with the plurality of processing units and that delivers events to respective threads with which those events are associated without execution of instructions by said processing units, where the events include any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered,

34

E. wherein a thread to which such an event is delivered processes that event without execution of instructions outside that thread.

7. The embedded processor of claim 6, where the pipeline control comprises a plurality of instruction queues, each associated with a respective virtual processing unit.

8. The embedded processor of claim 7, where the pipeline control decodes instruction classes from the instruction queues.

9. The embedded processor of claim 7, where the pipeline control controls access by the processing units to a resource providing source and destination registers for the instructions dispatched from the instruction queues.

10. The embedded processor of claim 7, wherein the execution units include a branch execution unit responsible for any of instruction address generation, address translation and instruction fetching.

11. The embedded processor of claim 10, wherein the branch execution unit maintains state for the virtual processing units.

12. The embedded processor of claim 6, where the pipeline control controls access by the virtual processing units to the execution units.

13. The embedded processor of claim 6, where the pipeline control signals a branch execution unit that is shared by the virtual processing unit as the instruction queue for each virtual processing unit is emptied.

14. The embedded processor of claim 6, where the pipeline control idles the execution units to decrease power consumption.

15. The embedded processor of claim 6, wherein the plurality of execution units include any of integer, floating, branch, compare and memory units.

16. An embedded processor system, comprising

A. a plurality of embedded processors,

B. a plurality of virtual processing units executing on the plurality of embedded processors, each virtual processing unit executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") such that one or more embedded processors has plural threads executing thereon, each thread being any of constrained or not constrained to execute on a same virtual processing unit and/or a same processor during a life of that thread,

C. one or more execution units that are shared by, and in communication coupling with, the plurality of virtual processing units, the execution units executing instructions from the threads, the execution units including any of integer, floating, branch, compare and memory execution units,

D. an event delivery mechanism that delivers events to respective threads with which those events are associated, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, wherein the event delivery mechanism

i. is in communication coupling with the plurality of virtual processing units, and

ii. delivers each such event to the respective thread without execution of instructions by said virtual processing units.

17. The embedded processor system of claim 16, wherein the thread to which an event is delivered processes that event without execution of instructions outside that thread.

US 7,685,607 B2

35

18. The embedded processor system of claim 17, wherein the branch unit is responsible for fetching instructions that are to be executed for the threads.

19. The embedded processor system of claim 18, wherein the branch unit is additionally responsible for any of instruction address generation and address translation.

20. The embedded processor system of claim 19, wherein the branch unit comprises thread state stores that store thread state for each of the respective virtual processing units.

21. The embedded processor system of claim 17, comprising

- A. a pipeline control that is in communication coupling with the plurality of processing units and with the plurality of execution units, the pipeline control dispatching instructions from plural ones of the threads for concurrent execution on plural ones of the execution units,
- B. the pipeline control comprises a plurality of instruction queues, each associated with a respective virtual processing unit, and wherein
- C. instructions fetched by the branch execution unit are placed in the instruction queues associated with the respective virtual processing unit in which the corresponding thread is executed.

22. The embedded processor system of claim 21, wherein one or more instructions are fetched at a time for a said thread with a goal of keeping the instructions queues at equal levels.

23. The embedded processor system of claim 22, wherein the pipeline control dispatches one or more instructions at a time from a given instruction queue for execution.

24. The embedded processor system of claim 23, wherein a number of instructions dispatched by the pipeline control at a given time from a given instruction queue is controlled by a stop flag in a sequence of instructions in that queue.

25. The embedded processor system of claim 21, wherein the pipeline control launches, and the execution units execute, multiple instructions from one or more threads simultaneously.

26. An embedded processor, comprising

- A. a plurality of processing units, each of which execute one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") and one or more of which execute a plurality of threads,
- B. an event delivery mechanism that delivers events to respective threads with which those events are associated, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, and wherein the event delivery mechanism
 - i. is in communication coupling with the plurality of processing units, and
 - ii. delivers each such event to the respective thread without execution of instructions by said processing units.

27. The embedded processor of claim 26, wherein the thread to which an event is delivered processes that event without execution of instructions outside that thread.

28. The embedded processor of claim 26, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread.

29. The embedded processor of claim 26, wherein at least one of the processing units is a virtual processing unit.

30. An embedded processor system, comprising

- A. a plurality of embedded processors,
- B. a plurality of virtual processing units executing on the plurality of embedded processors, each virtual process-

36

ing executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") such that one or more embedded processors has plural threads executing thereon,

- C. an event delivery mechanism that delivers events to respective threads with which those events are associated, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, and wherein the event delivery mechanism
 - i. is in communication coupling with the plurality of virtual processing units, and
 - ii. delivers each such event to the respective thread without execution of instructions by said virtual processing units.

31. The embedded processor system of claim 30, wherein the thread to which an event is delivered processes that event without execution of instructions outside that thread.

32. The embedded processor system of claim 31, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread.

33. A method of embedded processing, comprising the steps of

- A. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of processing units, such that plural threads are executing on one or more of those processing units,
- B. executing instructions from the threads in one or more execution units that are shared by the plurality of processing units,
- C. delivering events to respective threads with which those events are associated, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered
- D. wherein step (C) is effected without executing instructions by said processing units.

34. The method of claim 33, comprising the step of processing the event by the thread to which the event is delivered without execution of instructions outside that thread.

35. The method of claim 33, wherein the step of executing instructions from the threads in one or more execution units does not necessitate the execution units knowing what threads the respective instructions are from.

36. The method of claim 33, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread.

37. The method of claim 33, wherein at least one of the processing units is a virtual processing unit.

38. A method of embedded processing, comprising the steps of

- A. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of virtual processing units, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread,
- B. launching instructions from plural ones of the threads for concurrent execution on a plurality of execution units,
- C. delivering events to respective threads with which those events are associated without execution of instructions

US 7,685,607 B2

37

by said virtual processing units, where the events include any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, and

D. processing the event to which the thread is delivered without execution of instructions outside that thread.

39. The method of claim 38, where the launching step includes decoding instruction classes from the instruction queues.

40. The method of claim 38, where the launching step includes controlling access by the virtual processing units to a resource providing source and destination registers for the instructions dispatched from the instruction queues.

41. The method of claim 38, comprising executing any of the steps of instruction address generation, address translation and instruction fetching using a branch execution unit that is shared by all of the virtual processing units.

42. The method of claim 41, comprising maintaining state for the virtual processing units using the branch execution unit.

43. The method of claim 38, wherein the execution units include any of integer, floating, branch, compare and memory units.

44. A method of embedded processing, comprising the steps of

- A. executing a plurality of virtual processing units on a plurality of embedded processors,
- B. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of virtual processing units such that one or more embedded processors has plural threads executing thereon, each thread being any of constrained or not constrained to execute on a same virtual processing unit and/or a same embedded processor during a life of that thread,
- C. executing instructions from the threads in one or more execution units that are shared by the plurality of virtual processing units, the execution units including any of integer, floating, branch, compare and memory execution units, and
- D. delivering events to respective threads with which those events are associated without execution of instructions by said virtual processing units, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered, and
- E. wherein step (D) is effected without executing instructions by said processing units.

45. The method of claim 44, comprising processing the event by the thread to which the event is delivered without execution of instructions outside that thread.

46. The method of claim 45, comprising fetching instructions that are to be executed for the threads using the branch execution unit.

47. The method of claim 46, comprising executing with the branch execution unit any of instruction address generation and address translation.

48. The method of claim 45, comprising

- A. dispatching instructions from plural ones of the threads for concurrent execution on plural ones of the execution units,
- B. the dispatching step including placing instructions fetched for each of the respective threads in instruction

38

queues associated with the virtual processing units in which that thread is executed.

49. The method of claim 48, wherein the dispatching step includes fetching one or more instructions at a time for a given thread with a goal of keeping the instructions queues at equal levels.

50. The method of claim 49, wherein the dispatching step includes dispatching one or more instructions at a time from a given instruction queue for execution by the execution units.

51. The method of claim 50, wherein a number of instructions dispatched by the pipeline control at a given time from a given instruction queue is controlled by a stop flag in a sequence of instructions in that queue.

52. The method of claim 48, comprising launching and executing multiple instructions from one or more threads simultaneously.

53. A method of embedded processing, comprising the steps of

- A. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of processing units such that plural threads are executing on one or more processing units,
- B. delivering events to respective threads with which those events are associated without execution of instructions by said processing units, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered.

54. The method of claim 53, comprising the step of processing the event to which the thread is delivered without execution of instructions outside that thread.

55. The method of claim 53, wherein each thread is any of constrained or not constrained to execute on a same processing unit during a life of that thread.

56. The method of claim 53, wherein at least one of the processing units is a virtual processing unit.

57. A method of embedded processing, comprising the steps of

- A. executing a plurality of virtual processing units on a plurality of embedded processors,
- B. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of virtual processing units such that one or more embedded processors has plural threads executing thereon, and
- C. delivering events to respective threads with which those events are associated without execution of instructions by said virtual processing units, wherein a said event is any of (i) a hardware interrupt generated other than by the processing unit that is executing the thread to which that hardware interrupt is delivered, (ii) a software interrupt generated other than by the thread to which that software interrupt is delivered.

58. The method of claim 57, wherein processing the event by the thread to which the event is delivered without execution of instructions outside that thread.

59. The method of claim 58, wherein each thread is any of constrained or not constrained to execute on a same virtual processing unit and/or a same embedded processor during a life of that thread.

60. An embedded processor, comprising

- A. a plurality of processing units, each of which execute one or more processes or threads (which one or more

US 7,685,607 B2

39

processes or threads are collectively referred to as "threads") and one or more of which execute a plurality of threads,

- B. an event delivery mechanism that delivers events to respective threads with which those events are associated, wherein a said event is any of (i) loading of cache memory following a cache miss by the thread to which that event is delivered, (ii) filling of a memory location by a thread, other than the thread to which that notification is delivered, in response to a memory instruction issued by the thread to which that notification is delivered, and wherein the event delivery mechanism:
- i. is in communication coupling with the plurality of processing units, and
 - ii. delivers each such event to the respective thread without execution of instructions by said processing units.

40

61. A method of embedded processing, comprising the steps of

- A. executing one or more processes or threads (which one or more processes or threads are collectively referred to as "threads") on each of a plurality of processing units such that plural threads are executing on one or more processing units,
- B. delivering events to respective threads with which those events are associated without execution of instructions by said processing units, wherein a said event is any of (i) loading of cache memory following a cache miss by the thread to which that event is delivered, (ii) filling of a memory location by a thread, other than the thread to which that notification is delivered, in response to a memory instruction issued by the thread to which that notification is delivered.

* * * * *

Appendix B

US 5,119,481



US005119481A

United States Patent [19][11] **Patent Number:** 5,119,481

Frank et al.

[45] **Date of Patent:** Jun. 2, 1992[54] **REGISTER BUS MULTIPROCESSOR SYSTEM WITH SHIFT****FOREIGN PATENT DOCUMENTS**

[75] **Inventors:** Steven J. Frank, Hopkinton; Henry Burkhardt, III, Manchester; Frederick D. Weber, Concord, all of Mass.

2178205 4/1987 United Kingdom .

Primary Examiner—Eddie P. Chan
Attorney, Agent, or Firm—Lahive & Cockfield

[73] **Assignee:** Kendall Square Research Corporation, Waltham, Mass.

[57] **ABSTRACT**[21] **Appl. No.:** 696,291[22] **Filed:** Apr. 26, 1991**Related U.S. Application Data**

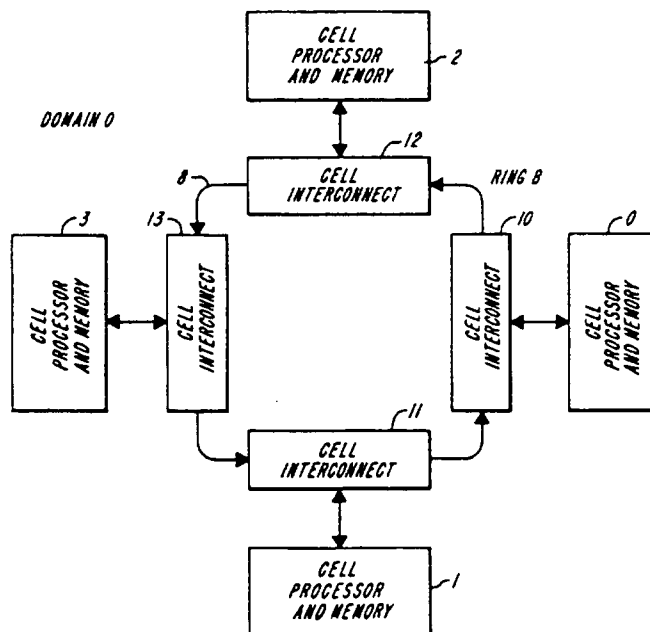
[63] Continuation of Ser. No. 509,480, Apr. 13, 1990, abandoned, which is a continuation of Ser. No. 136,701, Dec. 22, 1987, abandoned.

[51] **Int. Cl.⁵** G06F 13/00; G06F 15/16[52] **U.S. Cl.** 395/325; 364/DIG. 1; 364/229.3; 364/259.5; 370/85.1; 370/85.15[58] **Field of Search** ... 364/200 MS File, 900 MS File; 370/85.1, 85.15, 60, 61, 91, 94.1[56] **References Cited****U.S. PATENT DOCUMENTS**

3,748,647 7/1973 Ashany et al. .
 4,011,545 3/1977 Nadiv .
 4,031,512 6/1977 Faber .
 4,334,305 6/1982 Girardi 364/200

The cell interconnect responds to applied digital clock cycle signals for simultaneously transferring at least a selected packet through successive stages of the bus at a rate responsive to the digital clock cycle rate, while performing the modifying, extracting, replicating and transferring operation.

17 Claims, 11 Drawing Sheets

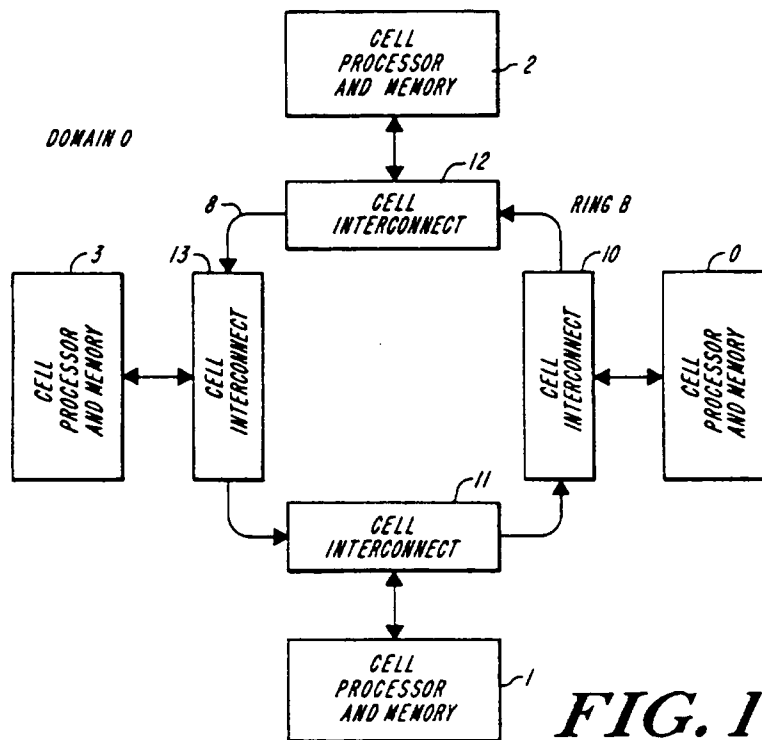
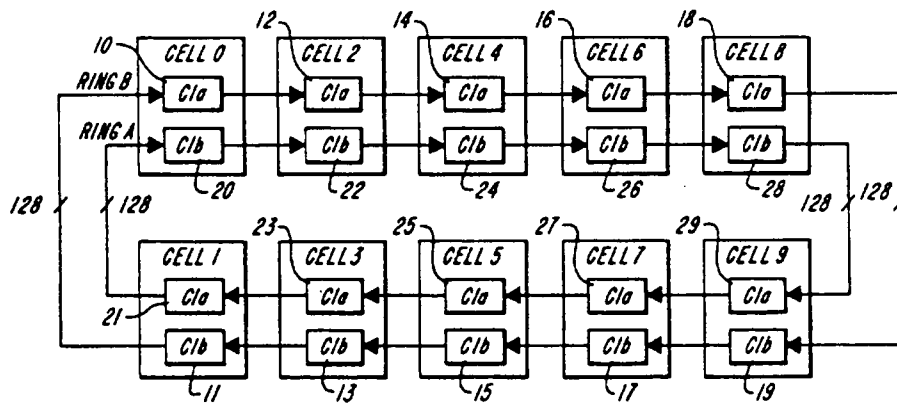


U.S. Patent

June 2, 1992

Sheet 1 of 11

5,119,481

**FIG. 1****FIG. 3**

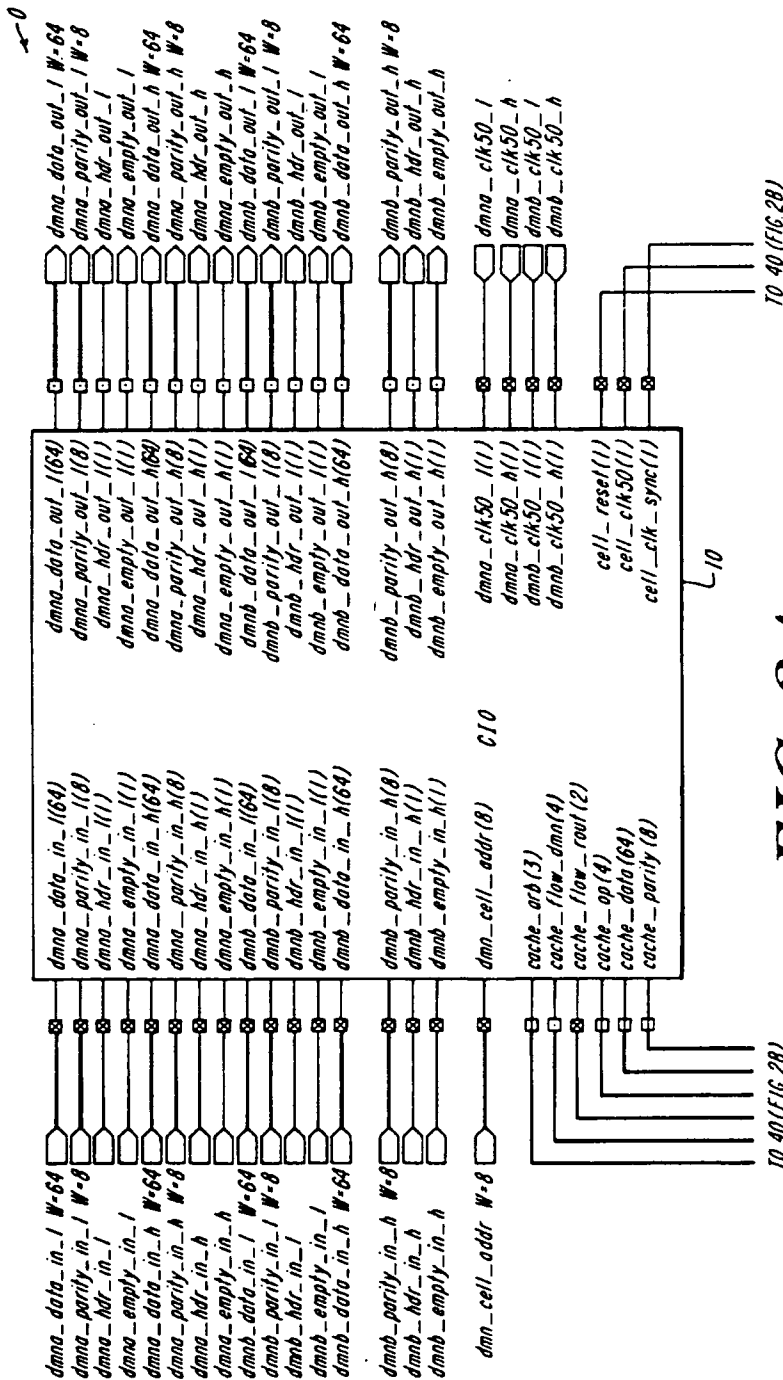


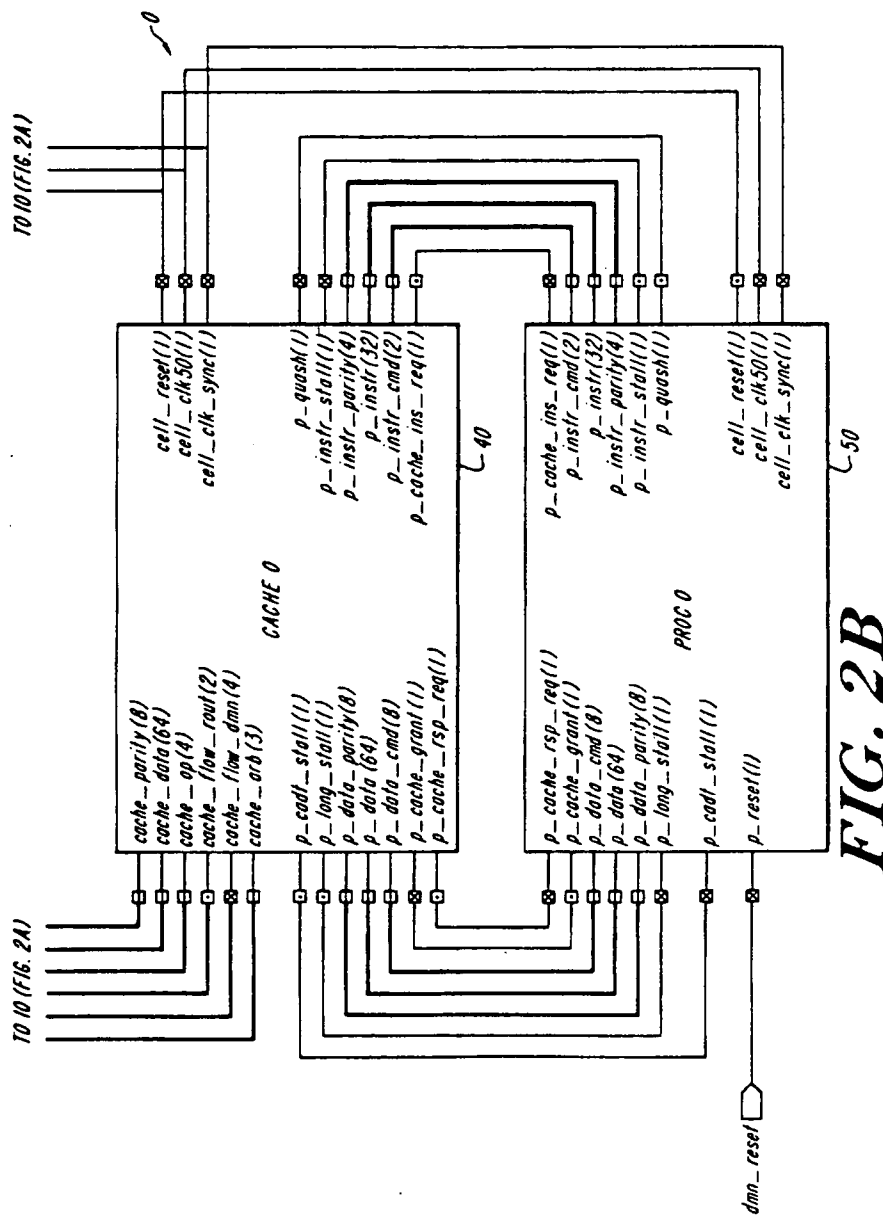
FIG. 2A

U.S. Patent

June 2, 1992

Sheet 3 of 11

5,119,481



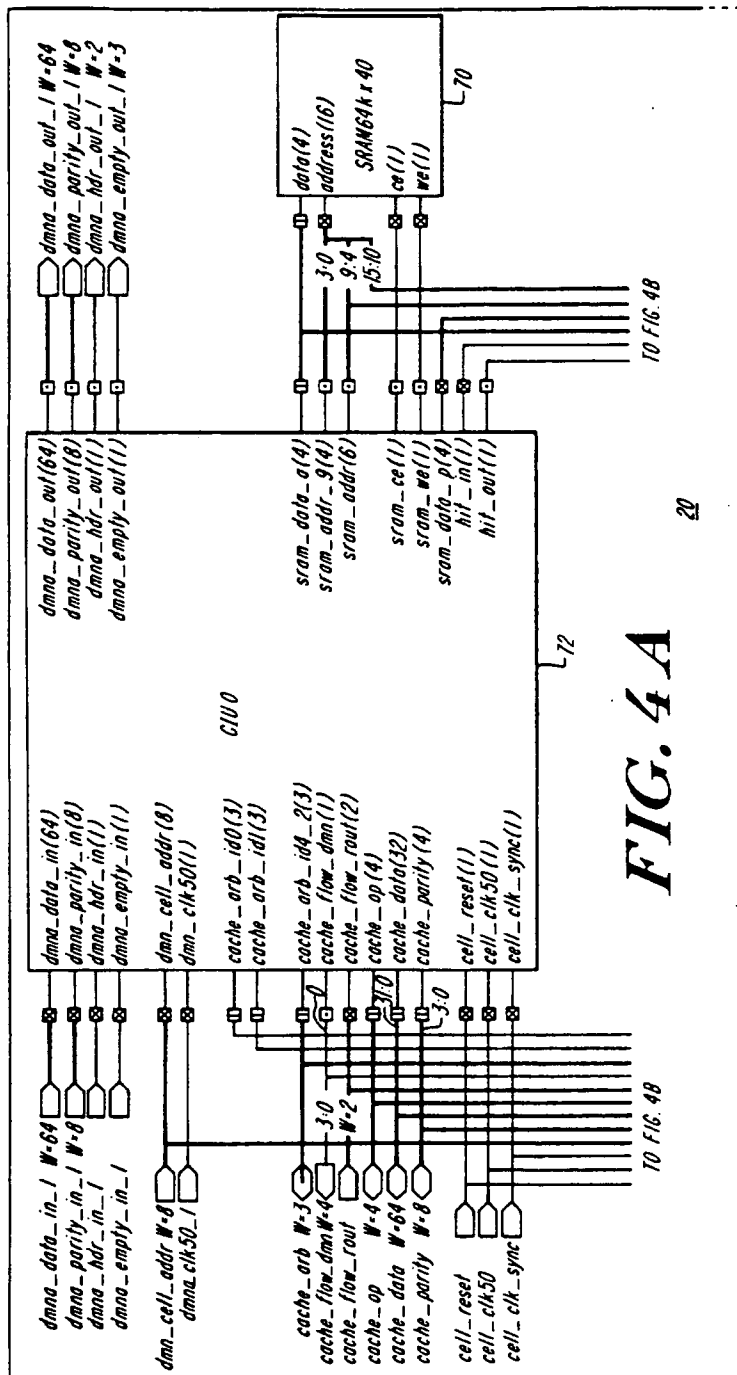


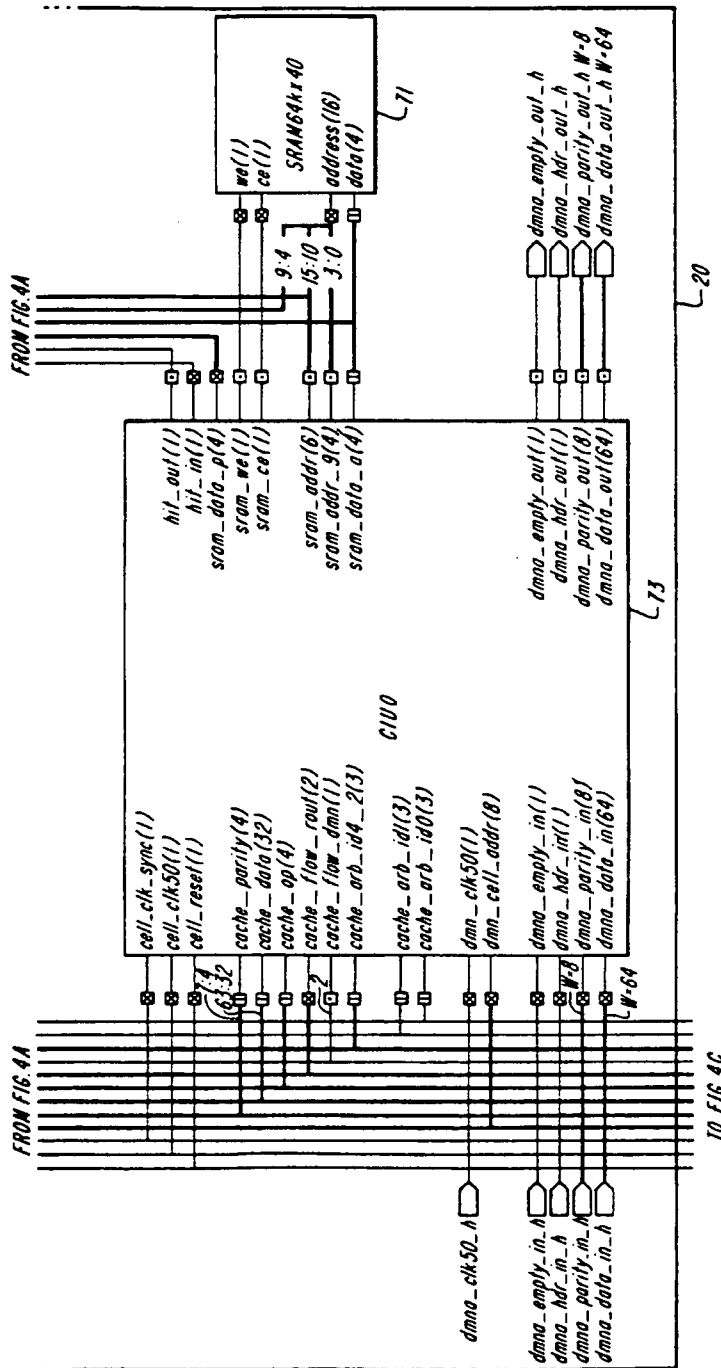
FIG. 4A

U.S. Patent

June 2, 1992

Sheet 5 of 11

5,119,481



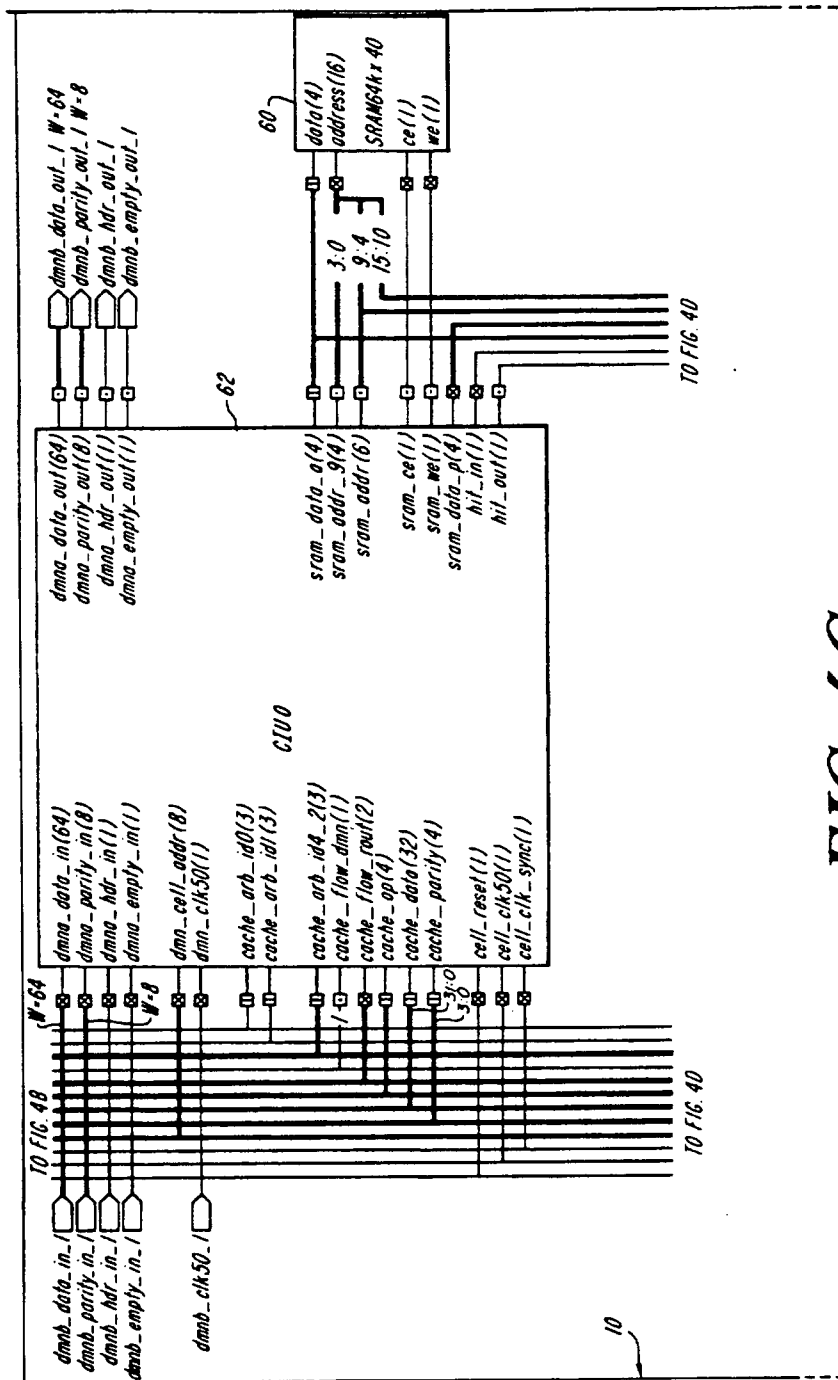


FIG. 4C

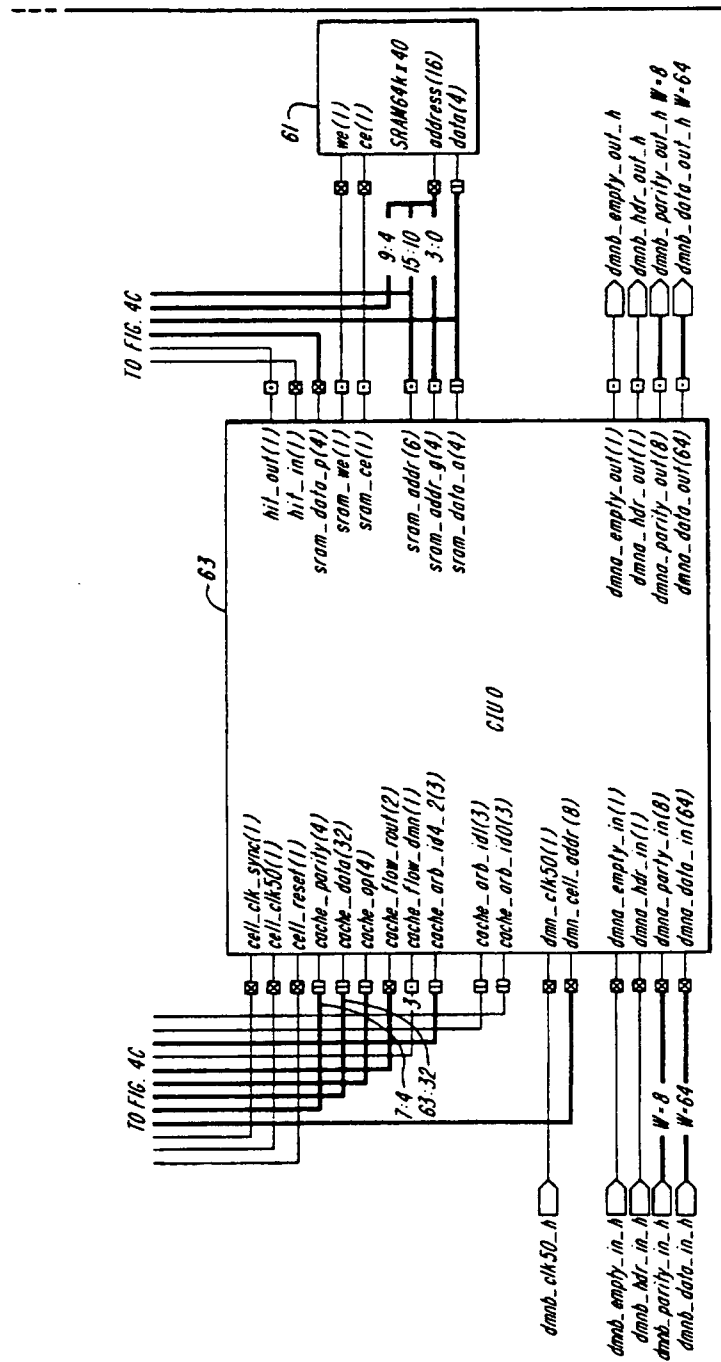


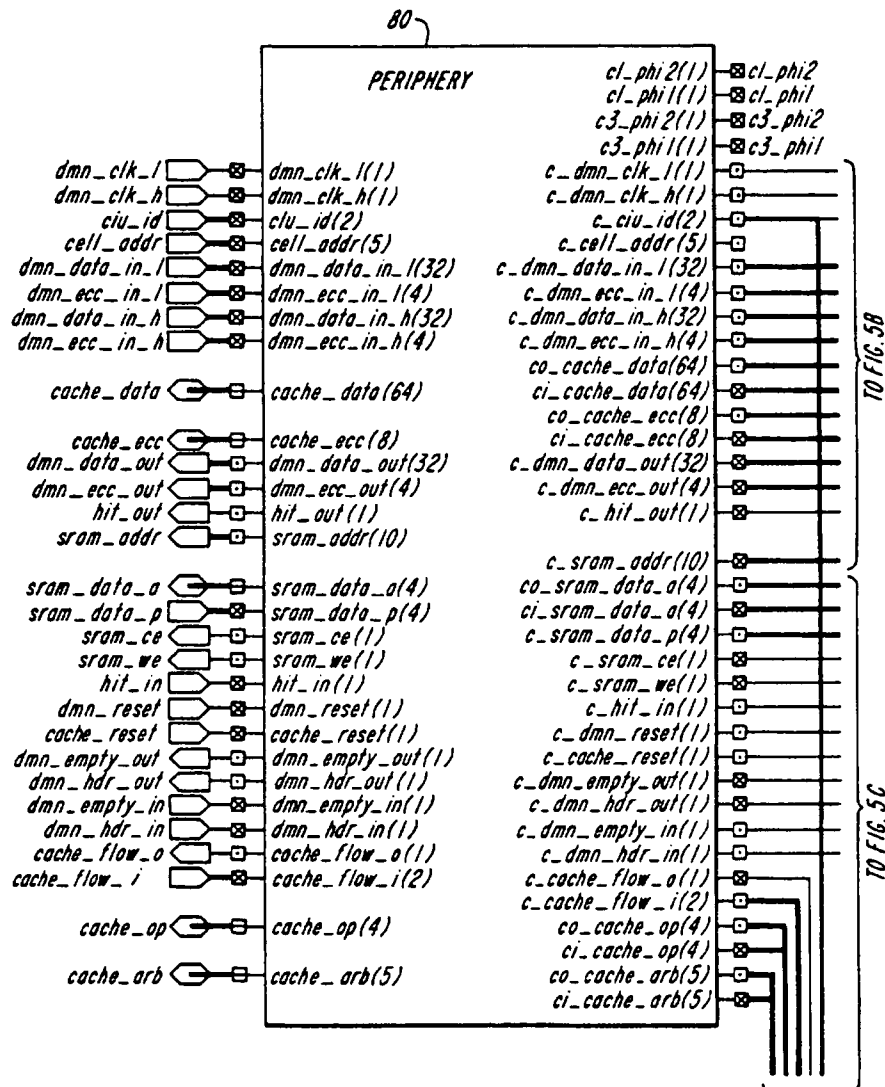
FIG. 4D

U.S. Patent

June 2, 1992

Sheet 8 of 11

5,119,481

**FIG. 5A**

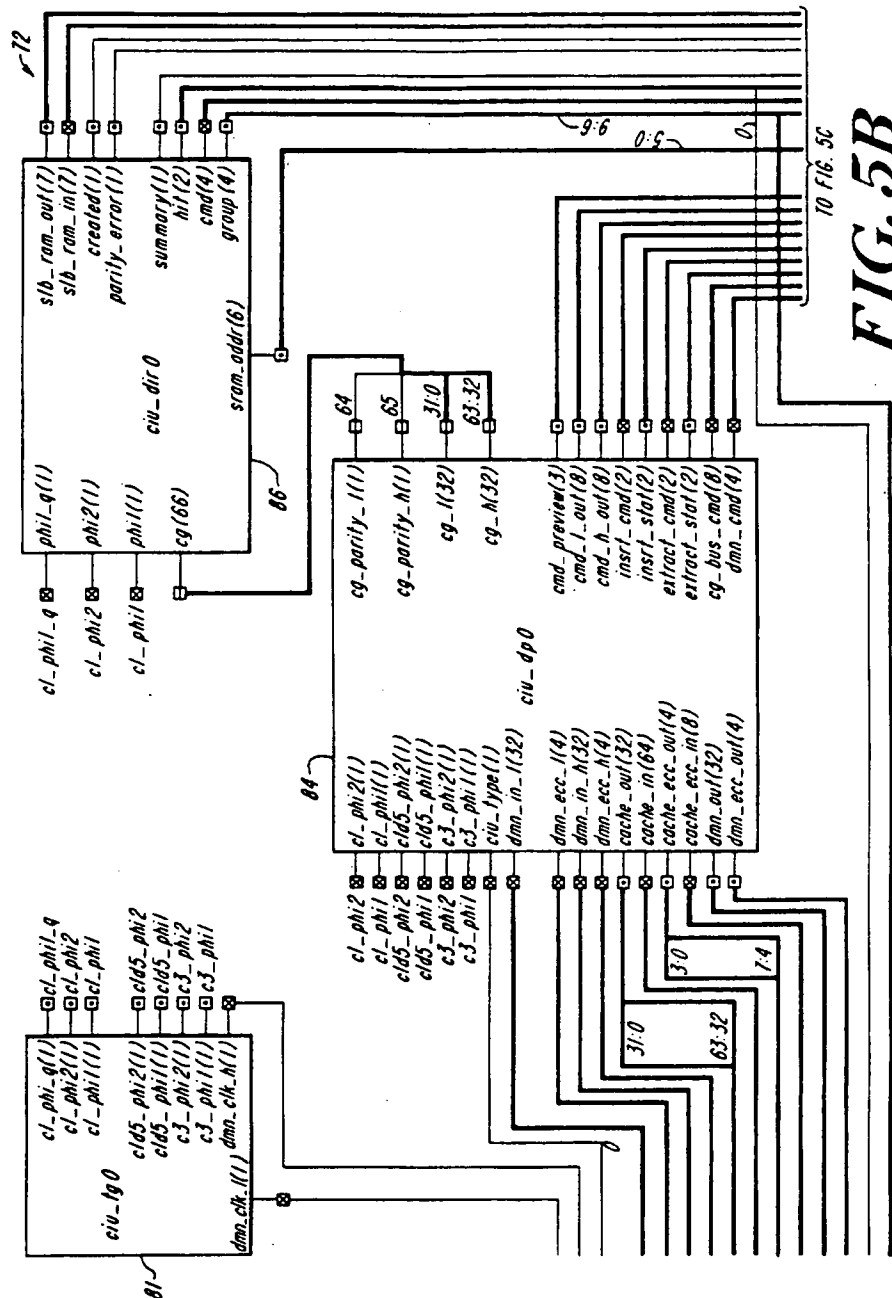
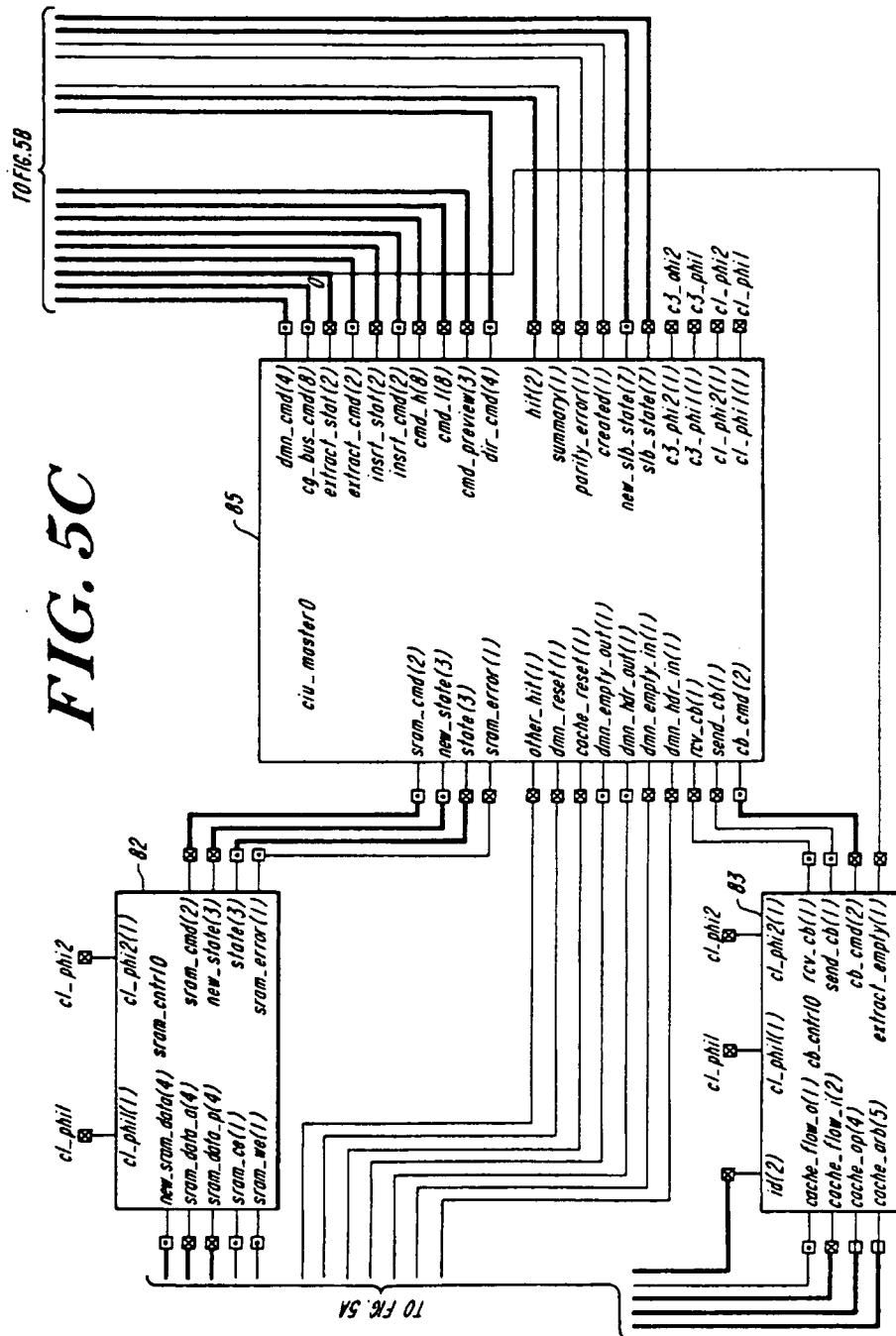


FIG. 5C



U.S. Patent

June 2, 1992

Sheet 11 of 11

5,119,481

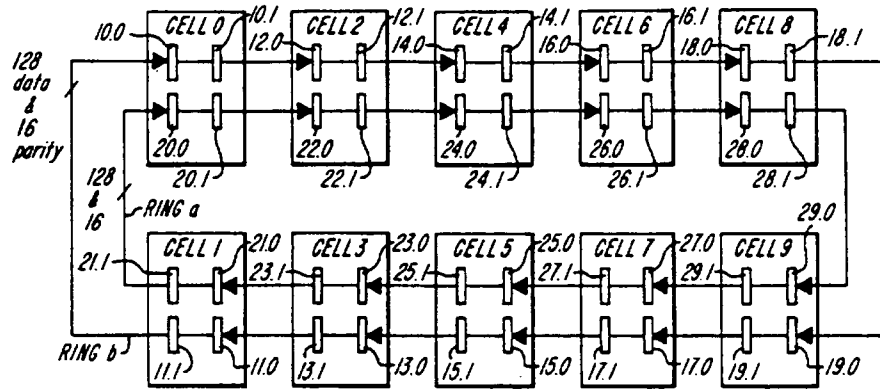


FIG. 6

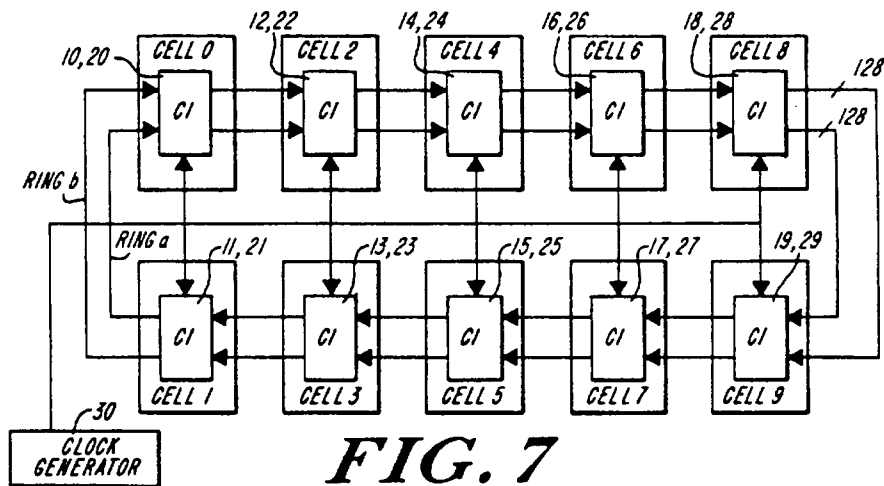


FIG. 7

FIG. 8

ADDR	ADDR
CMD	CMD
DATA 0	DATA 1
DATA 2	DATA 3
DATA 4	DATA 5
DATA 6	DATA 7

5,119,481

1

REGISTER BUS MULTIPROCESSOR SYSTEM WITH SHIFT

This is a continuation of copending Application Ser. No. 509,480 filed 13 Apr., 1990, which is abandoned, and which is a continuation of Patent Application Ser. No. 136,701 filed 22 Dec., 1987, which is abandoned.

BACKGROUND OF THE INVENTION

This invention relates generally to digital data processing systems, and, in particular, to a bus structure suitable for use with multiprocessor computer systems.

Multiprocessor computer systems provide multiple independent central processing units (CPUs) which can be coherently interconnected. Recent efforts in the multiprocessor field have concentrated on multiprocessor systems wherein each of a plurality of processors is equipped with a dedicated random access or cache memory unit. These multiple processors typically communicate with one another via a common system bus structure (i.e. shared bus systems), or by signalling within a shared memory address area (i.e. shared address space systems).

In recent years, a wide range of structures and methods have been proposed or developed to interconnect the processors of a shared bus system multiprocessor.

One such shared bus multiprocessor computer system is disclosed in United Kingdom Patent application No. 2,178,205 (published Feb. 4, 1987). That system is understood to comprise a plurality of processors, each having its own dedicated cache memory, and wherein the cache memories are connected to one another over a shared bus.

Conventional shared bus systems, however, lack adequate bandwidth to provide multiple processors with short effective access times during periods of high bus contention. Although a number of caching schemes have been proposed and developed for the purpose of reducing bus contention, bus saturation still limits the speed and size of multiprocessor computers.

Additionally, the speed of a conventional bus structure is limited by the speed of light and by bus length. In particular, as more processors are linked to a conventional bus, bus length increases and thus the time required for signal transfer increases.

Another class of interconnection systems, known as crossbar networks, avoid some of the limitations of conventional bus systems. In a crossbar network, however, the path taken by a given signal cannot be uniquely specified. Moreover, cost increases as the square of the number of interconnected processors. These characteristics make crossbar networks unsuitable for multiprocessor systems.

There accordingly exists a need for an interconnection system for multiprocessor computer systems which can accommodate the large volume of interconnect access requests generated by multiple processors. In particular, there exists a need for an interconnection system in which transfer speed is independent of the number of interconnected processors.

It is thus an object of the invention to provide an improved multiprocessor digital data processing system.

It is another object of the invention to provide an interconnection system for a multiprocessor digital computer structure which can provide multiple proces-

2

sors with short effective access times under all operating conditions.

It is yet another object of the invention to provide an interconnection system having the above characteristics which is applicable to both shared memory and non-shared memory multiprocessors.

It is a further object of the invention to provide an interconnection system having high bandwidth and the capability of transferring signals at rates sufficient to allow multiprocessors to operate at full speed.

It is another object of the invention to provide an interconnection system for a multiprocessor wherein bandwidth increases in proportion to the number of processors.

It is a further object of the invention to provide an interconnection system wherein transfer speed is independent of the number of interconnected processors, and is limited only by the switching speed of an individual interconnect.

Other general and specific objects of the invention will in part be obvious and will in part appear hereinafter.

SUMMARY OF THE INVENTION

The aforementioned objects are attained by the invention, which provides a digital data processing apparatus having a bus structure for transferring information-representative digital signals, the bus structure including a shift register element having a set of digital storage and transfer stages connected in series, for sequentially storing and transferring said information-representative digital signals. The invention also includes a plurality of processing cells, connected in a ring configuration to the bus structure, wherein at least one of the cells includes a central processing unit, an associated memory element for storing information-representative digital signals, coupled with the central processing unit for information transfer therebetween, and associated cell interconnect units, connected in circuit with the shift register element, and with an associated central processing unit, for transferring information-representative signals onto the shift register element.

In accordance with one aspect of the invention, the bus structure comprises unidirectional information-representative signal flow paths, and the cell interconnect units include elements for driving information-representative signals along the flow path defined by the bus structure.

In another aspect of the invention, each stage of the shift register element includes latch and register elements for storing a digital information-representative signal word of (n) digital bits, where (n) is a positive integer, and the cell interconnect units include timing control elements responsive to applied digital clock cycle signals for sequentially driving information-representative digital words through successive stages of the shift register element at a rate controlled by the digital clock cycle rate. In accordance with this aspect of the invention, the shift register element includes elements for storing in a given stage of the shift register element a given digital word for one applied digital clock cycle, and transferring a given digital word to a succeeding stage of the shift register element after an applied digital clock cycle. Moreover, in this aspect of the invention, each cell interconnect unit has associated therewith a subset of (s) stages of the shift register structure, where (s) is a positive integer, so that a given

5,119,481

3

digital word is resident in a stage associated with each cell interconnect unit for (s) applied digital clock cycles.

In another aspect of the invention, the shift register structure includes elements for sequentially transferring digital signal packets comprising (w) corresponding digital words, where (w) is a positive integer, so that a digital word corresponding to a given digital signal packet is resident in at least one stage associated with a given cell interconnect unit for (s) + (W-1) digital clock cycles.

The invention further provides data processing apparatus wherein the shift register structure includes elements for simultaneously transferring to successive shift register stages (p) digital signal packets, where (p) is a positive integer given by

$$(p) = (c)(s)/(w)$$

where (c) is the number of cell interconnect units, (s) is the number of shift register stages associated with each cell interconnect unit and (w) is the number of digital words in each digital signal packet. In accordance with the invention, as the number of cell interconnect units in a ring is increased, the flux of transfer operations through the ring is constant, and the number of bus operations which can be executed during a complete bus cycle increases linearly.

Another aspect of the invention provides data processing apparatus wherein at least one of the processing cells described above includes elements for generating and transmitting to an associated cell interconnect unit a cell interconnect control signal representative of a request to store an information-representative signal in a first associated stage of the shift register structure.

In accordance with this aspect of the invention, at least one of the processing cells includes elements for generating and transmitting to an associated cell interconnect unit a cell interconnect control signal representative of a request for access to an information-representative signal stored in a first associated stage of the shift register structure. The associated cell interconnect unit in turn includes elements, responsive to the cell interconnect control signal, for extracting the information-representative signal stored in the first associated stage of the shift register structure and for transferring the extracted information-representative signal to the associated cell.

In a digital data processing apparatus of the type described above, the associated cell interconnect unit can also include an element, responsive to the cell interconnect control signal, for replicating the information-representative signal stored in the first associated stage of the shift register structure and for transferring the replicated information-representative signal to the cell.

In yet another aspect of the invention, at least one of the processing cells includes an element for generating and transmitting to an associated cell interconnect unit, a cell interconnect control signal representative of a request to transfer, unchanged, an information-representative signal stored in a first associated stage of the shift register structure to a second, succeeding associated stage of the shift register structure.

In accordance with this aspect of the invention, the associated cell interconnect unit can include an element, responsive to the cell interconnect control signal, for enabling transfer to the second, succeeding associated stage of the shift register structure the information-

4

representative signal stored in the first associated stage of the shift register structure.

The invention further contemplates data processing apparatus of the type described above, wherein at least one of the processing cells include an element for generating and transmitting to an associated cell interconnect means a cell interconnect control signal representative of a request to identify a given digital word stored in an associated stage of the shift register structure as the first word of a data packet. In accordance with this aspect of the invention, the associated cell interconnect unit can include means, responsive to the cell interconnect control signal, for setting a portion of the given digital word to a selected value identifying the given digital word as the first word of the data packet.

Yet another aspect of the invention provides a digital data processing apparatus having a bus structure and plurality of processing cells of the type described above, wherein at least one of the cells includes an associated cell interconnect unit, connected in circuit with the shift register structure, and with an associated central processing unit, for transferring information-representative signals onto the shift register structure. the cell interconnect unit comprising a subset of serially connected stages of the shift register structure.

The invention accordingly comprises apparatus embodying features of construction, combinations of elements and arrangements of parts as exemplified in the following detailed disclosure, and the scope of the invention is indicated in the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

For a fuller understanding of the nature and objects of the invention, reference should be made to the following detailed description and the accompanying drawings, in which:

FIG. 1 depicts the structure of a multiprocessor computer system constructed in accordance with the invention;

FIG. 2 comprising 2A and 2B depicts detail of the structure of a processing cell illustrated in FIG. 1;

FIG. 3 depicts a plurality of the processing cells of FIG. 2 interconnected by a bus system constructed in accordance with the invention;

FIG. 4 comprising 4A, 4B, 4C and 4D depicts detail of the structure of a cell interconnect of FIG. 3;

FIG. 5 comprising 5A, 5B and 5C depicts detail of the structure of a cell interconnect unit in the cell interconnect of FIG. 4;

FIG. 6 depicts the shift register stages associated with the cell interconnects of FIG. 3;

FIG. 7 depicts clock signal distribution in the embodiment of FIG. 3; and

FIG. 8 depicts the contents of an exemplary data packet processed by the embodiment of FIG. 3.

DESCRIPTION OF THE ILLUSTRATED EMBODIMENT

Structure

FIG. 1 depicts a multiprocessor computer utilizing a bus system constructed in accordance with the invention. The multiprocessor system is hierarchically constructed from processors, cells and domains. Each of the processing cells 0, 1, 2 and 3 contains a processor and cache memory, as discussed below in connection with FIG. 2. The cells 0-3 are interconnected by cell interconnects (CIs) 10-13 and bus 8, thereby collec-

5,119,481

5

tively forming Domain 0. Domains, in turn, are interconnected by domain interconnects (not shown), to form a complete system. The structure of cell interconnects is described hereinafter in connection with FIGS. 4 and 5, and the structure and operation of the illustrated multiprocessor system is more fully discussed in U.S. patent application Ser. No. 136,930 filed on even date herewith, and incorporated herein by reference.

FIG. 2 depicts the components of processing cell 0, including processor (PROC) 50, cache 40 and cell interconnect (CI) 10. Data, parity, and control signals Passed between processor 50, cache 40 and cell interconnect 10 are indicated in FIG. 2. The datapath width associated with each respective signal is indicated by numerals in brackets. For example, cache data signals (cache_data [64]) passed between cell interconnect 10 and cache 40 have a 64 bit datapath width, as do processor data signals (p_data[64]) passed between cache 40 and processor 50.

As FIG. 2 illustrates, cell interconnect 10 receives 20 and transmits DOMAIN DATA signals (dmn_data), DOMAIN PARITY signals (dmn_parity), DOMAIN EMPTY signals (dmn_empty), DOMAIN HEADER signals (dmn_hdr), DOMAIN CELL ADDRESS signals (dmn_cell_addr), and DOMAIN CLOCK signals (dmn_clk50) discussed in greater detail hereinafter. In addition, cell interconnect 10 processes cache arbitration, routing, operation, and parity signals as indicated in FIG. 2. The structure of cell interconnect 10 is discussed in greater detail below in connection with FIG. 4. Moreover, further understanding of the logic components and structure of cache 40 and processor 50 may be had by reference to the schematics incorporated herein as Appendix A, and by reference to U.S. Ser. No. 136,930. Cell interconnect 10 provides interconnection of cell 0 into a multiple-cell domain like that depicted in FIG. 3.

FIG. 3 illustrates the configuration of a ten cell domain, containing cells 0-9 interconnected in accordance with the invention in a dual ring bus structure organized as ring A and ring B. Utilizing plural rings is an important feature of the invention, which enables the system to continue operating in the event of single point component failures, and increases the bandwidth of the interconnection system. In a preferred practice of the invention, utilizing two rings, A and B, ring A is configured for transfers involving even page addresses in memory, and ring B for odd page addresses in memory. This interleaving mode is discussed in greater detail hereinafter. Those skilled in the art will understand that the invention may be practiced in an embodiment having more than two rings.

Rings A and B are preferably 50 megahertz synchronous shift registers having plural data storage stages with a 128 bit datapath width, as indicated in FIG. 3. Each of the cells 0-9 communicates with rings A and B through two associated Cell Interconnects (CIs). As FIG. 3 illustrates, cell interconnects 10-19 connect cells 0-9, respectively, to ring B, while cell interconnects 20-29 connect cells 0-9, respectively, to ring A.

A preferred cell interconnect structure is illustrated in FIG. 4. Two cell interconnect units (CIUs) 72 and 73 and two 64K x 4 static RAMs (SRAMs) 70 and 71 are configured in pairs to form a single cell interconnect 20. Similarly, cell interconnect units 62 and 63, and SRAMs 60 and 61 are utilized to form cell interconnect 10. Each cell interconnect presents two 64 bit data connections from its cell to a respective ring (dmn_data) and one 64

6

bit connection to its cell cache bus (cache data). The structure and operation of such a cell cache bus are described in U.S. Ser. No. 136,930. Through these connections, the cell interconnect moves requests and responses between the cell and a respective ring.

The ring connections of each cell interconnect collectively form an input port and an output port. In operation, each cell interconnect moves the data on its input port through two stages (comprising four latches), modifies the data as required by a given cell interconnect unit operation and presents the data on its output port. Accordingly, when a number of cell interconnects are linked in a loop, the delay stages form a shift register such as Ring A or Ring B. Each cell interconnect receives data from the previous cell interconnect in its ring and forwards data to the next. An insertion and extraction protocol described in greater detail hereinafter allows the cell interconnects to pass data between cells.

As FIG. 4 illustrates, each cell interconnect unit (CI) is formed by two cell interconnect units (CIUs), and associated SRAMs for storing state bits. Each cell interconnect unit (CIU), in turn, is constructed from a plurality of integrated circuits. The integrated circuits which form cell interconnect unit (CIU) 72 are depicted in FIG. 5.

The cache bus connection of the cell interconnect is a bi-directional interface. The cell interconnect receives data from the cache bus to send to the ring and places data it receives from the ring onto the cache bus to be serviced by the cache control unit or domain routing unit. The structure and operation of preferred cache control and domain routing units are described in U.S. Ser. No. 136,930.

As illustrated in FIG. 6, each cell interconnect contributes two shift register stages to the shift register structures of rings A and B. For example, a ring with ten cell interconnects, such as rings A and B shown in FIG. 6, consists of twenty pipeline stages. Each pipeline stage is capable of selectively storing and transferring an information-representative signal representing one data word. All data words circulate through the ring by progressing, in parallel, at the rate of one stage per applied clock cycle. It is this feature of the invention which allows each cell to uniquely identify the source and destination of each data word on the bus, and determine appropriate processing steps for each data word. One example of a preferred clock signal distribution configuration is depicted in FIG. 7.

In accordance with the invention, cell interconnect unit (CIU) 72 is constructed from periphery unit 80, CIU tag unit 81, SRAM control unit 82, cache bus control unit 83, CIU data path unit 84, CIU master control unit 85, and CIU directory unit 86. The integrated circuits illustrated in FIG. 5 contain latches, FIFO buffers, multiplexors (MUXs) and other conventional logic elements.

In particular, the CIU datapath associated with CIU datapath circuit 84 is a 36 bit wide datapath including low and high cache group units, an extract FIFO and an insert FIFO. These four units collectively provide paths for (i) moving addresses from the domain interconnected by rings A and B, and from the cache bus, to the directory for address lookup, (ii) moving packets through the two pipeline stages of each CIU, (iii) moving packets from the domain to the cache bus, and (iv) moving packets from the cache bus to the domain.

5,119,481

7

The low and high cache group units direct appropriate addresses to the CIU directory circuit 86 for lookup, and provide modification of directory entries. In particular, the cache group units can pass domain, cache or recirculated addresses for lookup operations, modify directory entries, and move data from directory 86 to the associated cache bus.

The extract FIFO unit moves data from the CIU domain inputs into a holding register file, and subsequently passes the data to the associated cache bus. The insert FIFO unit moves data from the cache bus inputs into a holding register file, and subsequently passes this data to the CIU domain outputs. Additionally, the insert FIFO unit provides for modifying packets on the domain formed by rings A and B. The datapath control section associated with CIU datapath unit 84 receives commands from the master control unit 85 and converts them into command signals for use by the elements of the CIU datapath. Detailed schematics and timing diagrams for the elements of these integrated circuits are set forth in Appendix A, incorporated herein.

Top level control of the CIU 72 is managed by the CIU master control circuit 85, the SRAM control circuit 82, and the cache bus control circuit 83. The master control circuit 85 receives PACKET HEADER and EMPTY STATUS bits, and provides sequencing to the directory block to perform address lookups. The master control circuit 85 utilizes the results of these lookup operations to determine which of the PASS, EXTRACT and INSERT operations, discussed in greater detail hereinafter, is appropriate. The master control circuit 85 performs these operations based upon signals from the CIU data path circuit 84 and cache bus control circuit 83.

The SRAM control circuit 82 generates control signals for addressing the external SRAMS 70 and 71 used by the CIU 72 and illustrated in FIG. 4. The cache bus control circuit 83 manages arbitration and flow control on the cache bus, as described in U.S. Ser. No. 136,930. The cache bus control circuit 83 receives command signals from the master control circuit 85, and in turn, transmits status report signals to the master control circuit 85.

SIGNALS AND FIELDS

As FIG. 7 illustrates, a single domain clock signal (h,l), generated by clock generator 30, is distributed to the entire domain interconnect formed by rings A and B. Domain clock (h,l) provides 50 mhz synchronous timing information to the cell interconnects within the domain interconnect formed by rings A and B.

By properly distributing domain clock (h,l), the effective clock skew for a cell interconnect such as, for example, the cell interconnect 14 corresponding to cell 4, is the clock skew between that cell interconnect's input stage 14.0 and prior adjacent cell interconnect (cell 2) and its output stage 14.1 and next adjacent cell (cell 6). An important advantage of the invention is that clock skew is not accumulative, and propagation time between cell interconnects is independent of the number of cell interconnects or stages.

The fundamental result is that the clock cycle time —i.e. the inverse of the clock frequency—of the domain interconnect is simply the cycle time between two adjacent cell interconnects. The clock cycle time does not increase and frequency does not decrease as the number of cell interconnects is increased. Thus, as the number of cell interconnects in a ring is increased, the

8

flux of operations through the ring is constant, while the number of bus operations which can be executed during a complete bus cycle increases linearly. This is an important feature of the invention, which is ideally suited for multiprocessor structures.

The high speed nature of a domain interconnect constructed in accordance with the invention is further enhanced by two topological factors. First, the output (i.e. second) stage of each cell interconnect drives a single load, the input stage of the adjacent cell interconnect. Second, each cell interconnect requires connection to only its two neighboring cell interconnects, allowing close proximity of all directly connected cell interconnects. This combination of absolute minimal loading and very short physical distance between adjacent cells minimizes propagation time between cell interconnects.

Those skilled in the art will understand that while the embodiment described above in connection with FIG. 7 utilizes a synchronous clock, the invention can be practiced in connection with an asynchronous or self-timed clock configuration.

In accordance with the invention, data circulating through a given ring is divided into data packets of ten data words, corresponding to ten shift register ring stages. The number of shift register stages must be an exact multiple of the number of data words in a data packet. Given, for example, a ring with twenty cells and two register stages per cell, the ring consists of forty stages. Thus, four ten-word Packets can be transferred simultaneously in this ring. This property is generalized below in Table I.

TABLE I

Number of CI	Number of Stages	Number of Packets
5	10	1
10	20	2
15	30	3
20	40	4

The invention is preferably practiced in connection with the packet configuration shown in FIG. 8. The first data word in each half is an address, the second data word is a command and the remaining data words are data, as indicated in FIG. 8. Those skilled in the art will understand that alternative packet configurations are possible and within the scope of the invention.

In addition to the operations described above, a cell interconnect can modify the command field of a packet. For example, a cell interconnect can extract a packet by copying the packet from the ring and changing the command field to EMPTY. Alternatively, if a cell interconnect merely copies a packet, the command field would remain unchanged, allowing the packet to continue to circulate through the ring.

All packets circulate through the domain interconnect only once. This property results from an operational protocol in which each operation is created and retired by the same cell interconnect. Cells which extract a packet to add response data must later re-insert the packet.

The operations that the cell interconnect units can perform on packets thus include the following:

PASS PACKET: The cell interconnect unit passes a packet from its ring inputs to its ring outputs without any modification if the packet specifies an address of which the cell interconnect has no knowledge.

5,119,481

9

EXTRACT PACKET: The cell interconnect unit extracts a Packet from the ring if it represents a request the cell interconnect unit made to the ring or contains an address the cell interconnect unit must act upon. When a cell interconnect unit extracts a packet from the ring it modifies the command of the packet to indicate the extraction.

SEND PACKET to the Cache Bus: The cell interconnect unit sends each packet that it extracts from the ring to the cache bus for service by the cache controller.

RECEIVE PACKET from the Cache Bus: The cache controller can send a packet to the cell interconnect unit for insertion into the ring. The cell interconnect unit receives these packets and retains them until they can be inserted into the ring.

INSERT PACKET: The cell interconnect unit inserts a packet into the ring whenever it has a packet awaiting insertion and the current ring packet is marked as EMPTY.

In a preferred embodiment of the invention, the domain interconnect formed by rings A and B supports two sets of fields, referred to as the domain fields and the cell interconnect unit fields. The domain fields are established by serial connections from one cell interconnect to the next cell interconnect, which form a ring. Each cell interconnect has a separate receive and transmit port for the domain fields, as indicated in FIG. 4. The cell interconnect unit fields provide communication among the cell interconnect units of the cell. The domain fields are summarized below in Table II:

TABLE II

DOMAIN FIELDS
DOMAIN DATA
DOMAIN ECC
DOMAIN HEADER
DOMAIN CLOCK
CIU ID
CIU CELL ADDRESS
CIU EXTRACT

The DOMAIN DATA and DOMAIN ECC fields are responsible for moving the data of ring operations. Each operation is a packet of ten domain bus transfers. The DOMAIN HEADER field marks the beginning of an operation. The DOMAIN CLOCK field provides timing for the shift register structure in cooperation with the clock generator illustrated in FIG. 7. The CIU ID field identifies the type of cell interconnect unit involved in a given transfer. The CIU CELL ADDRESS field identifies the domain local address of the cell. The CIU EXTRACT field communicates information between cell interconnect units.

In operation, the DOMAIN DATA field transmits the address, command, and data information corresponding to an operation. The configuration of the DOMAIN DATA field consists first of an address, then a command, and finally eight data values, one or more of which may be empty.

The DOMAIN ECC field transmits a Hamming based error correction code (ECC) word for each domain data transfer. DOMAIN ECC is not generated or checked by the cell interconnect units, but passed unchanged to the target units.

The DOMAIN HEADER field designates a given word in the current domain data transfer as the first word of a packet.

10

The assertion of the DOMAIN RESET signal by the domain structure or domain power controller causes each cell interconnect and corresponding cell to enter a reset state. The reset state is described in U.S. Ser. No. 136,930.

The assertion of CELL RESET by a cell causes the corresponding cell and cell interconnect to reset. When reset, cell interconnects perform only PASS PACKET operations.

The cell interconnect unit control fields provide for communication specific to the cell interconnect units. These cell interconnect unit control fields are summarized below in Table III.

TABLE III

CIU Control Fields
CIU ID
CIU EXTRACT
CIU CELL ADDRESS

THE CIU ID field for each CIU is established by the configuration of electrical connections from the CIU to power and ground terminals. This configuration establishes a unique CIU identification number for each CIU. The interpretation of the identification number depends upon whether the plural ring structure is configured in a two-way or a four-way memory interleaving mode. Normally, ring A is configured for even page addresses in memory and ring B for odd page addresses in memory. However, those skilled in the art will appreciate that the shift register structure can be configured to pass all addresses on either ring. Page address interleaving is normally configured at system configuration time by control locations in the cell interconnects. Table IV below summarizes interpretation of id numbers in the two-way interleaved mode.

TABLE IV

TWO WAY INTERLEAVED	
ID VALUE	DEFINITION
0	Master CIU 1
1	Master CIU 1
2	Slave CIU 0
3	Slave CIU 1

Table V below summarizes interpretation of id numbers in the four-way interleaved mode:

TABLE V

FOUR WAY INTERLEAVED	
ID VALUE	DEFINITION
0	MASTER 0
1	Master CIU 1
2	Master CIU 2
3	Master CIU 3

When two CIUs are partnered as master and slave, the slave cell interconnect unit drives a one-bit CIU EXTRACT signal which is read by its partner master cell interconnect unit. The CIU EXTRACT signal is asserted or de-asserted according to whether the slave CIU identifies the current operation as requiring removal from the shift register structure.

The CIU CELL ADDRESS signal represents the address within the domain of the cell asserting the signal.

In accordance with the invention, all domain interconnect transfers form a single packet. The domain interconnect formed by the plural rings is initialized to

5,119,481

11

contain a fixed number of packets based on the number of cell interconnects. For example, a twenty cell domain interconnect contains eight packets per ring. Thus, in this example, eight packets per ring, or sixteen packets per domain interconnect, can be transferred in parallel.

In a preferred practice of the invention in conjunction with a multiprocessor structure like that described in U.S. Ser. No. 136,930, the cell interconnect performs two levels of interpretation in order to determine how to operate on a packet. First, the cell interconnect examines the address specified in the packet. The cell interconnect may be configured to operate as a Positive or a negative filter. As a positive filter, it operates on any System Virtual Address (SVA) that has an entry in its cache descriptors. The interaction of SVA addresses and cache descriptors is further described in U.S. Ser. No. 136,930. The positive filter configuration is used when a cell interconnect is connected to a cache either directly or by a remote interface. The negative filter configuration is used when the cell interconnect is connected to a router. In either configuration, the cell interconnect operates on SVA addresses directed to it.

Secondly, having recognized an address, the cell interconnect examines the command portion of the packet to determine if it can modify the response field of the command and allow the packet to proceed, or whether it must extract the packet from the domain interconnect.

A cell interconnect can insert a packet into the ring when an empty packet arrives. An empty packet is indicated by an IDLE operation type in the command word of the packet. Evenly distributed usage of the packets of the ring is provided by the invention because no cell interconnect can use a packet that it has just emptied by an extraction operation. In order to perform an insertion, the cell interconnect must place its operation into the ten consecutive stages of the empty packet.

It will be understood that given the plural ring structure of the invention, the cell interconnect which initially injects a particular operation into the ring will eventually receive that operation back. At that time, the

12

cell interconnect destroys the operation by changing the command word to IDLE. Any cell interconnect that removes an operation it did not create must return that operation to the ring.

A cell interconnect extracts an operation from the ring by copying its contents from the ten consecutive stages of its packet and writing the IDLE operation type into the command word of the packet. Any cell interconnect that extracts an operation it did not create must return that operation to the ring.

The bus structures are initialized in two steps. First, the domain stages are formatted into ten word packets by initializing the domain header signal. Secondly, each packet is initialized to an IDLE state. If the number of stages in either ring is not an even multiple of ten stages, or if the circular path is logically broken, the ring will not initialize.

In a preferred embodiment of the invention, bus initialization is performed by software, with cell interconnect assist. Within each cell interconnect is a DOMAIN HEADER STATUS bit which indicates whether the domain is properly formatted by verification of the DOMAIN HEADER signal. If the DOMAIN HEADER STATUS bit indicates that a given ring is improperly formatted, then a SETUP DOMAIN command issued by a given cell to a cell interconnect performs domain initialization.

It will thus be seen that the invention efficiently attains the objects set forth above, among those made apparent from the preceding description. It will be understood that changes may be made in the above construction and in the foregoing sequences of operation without departing from the scope of the invention. It is accordingly intended that all matter contained in the above description or shown in the accompanying drawings be interpreted as illustrative rather than in a limiting sense.

It is also understood that the following claims are intended to cover all of the generic and specific features of the invention as described herein, and all statements of the scope of the invention which, as a matter of language, might be said to fall therebetween.

U.S. Patent Application for INTERCONNECTION SYSTEM FOR MULTIPROCESSOR STRUCTURE

STEVEN J. FRANK
HENRY J. BURKHARDT III.
FREDERICK D. WEBER

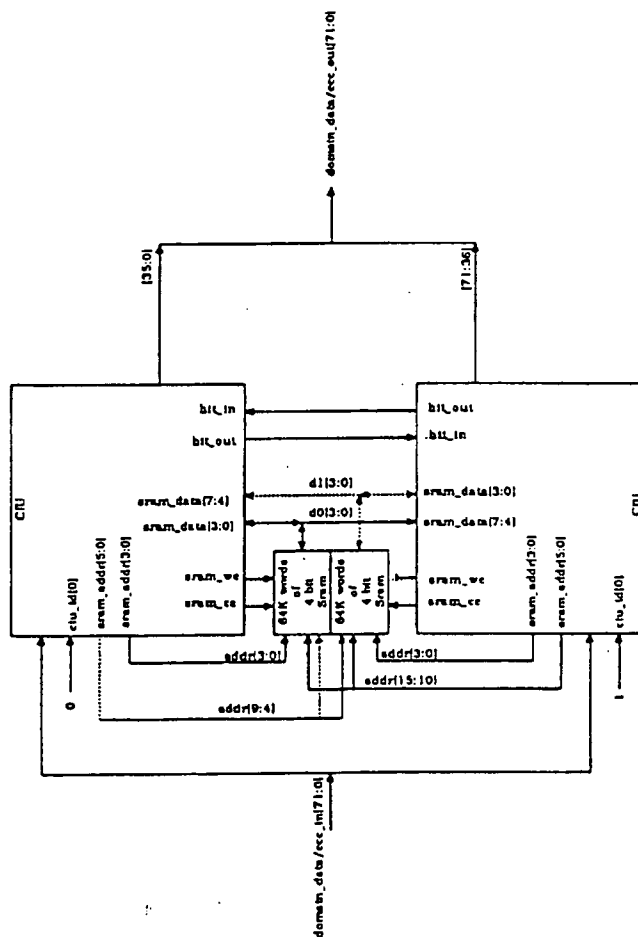
APPENDIX A

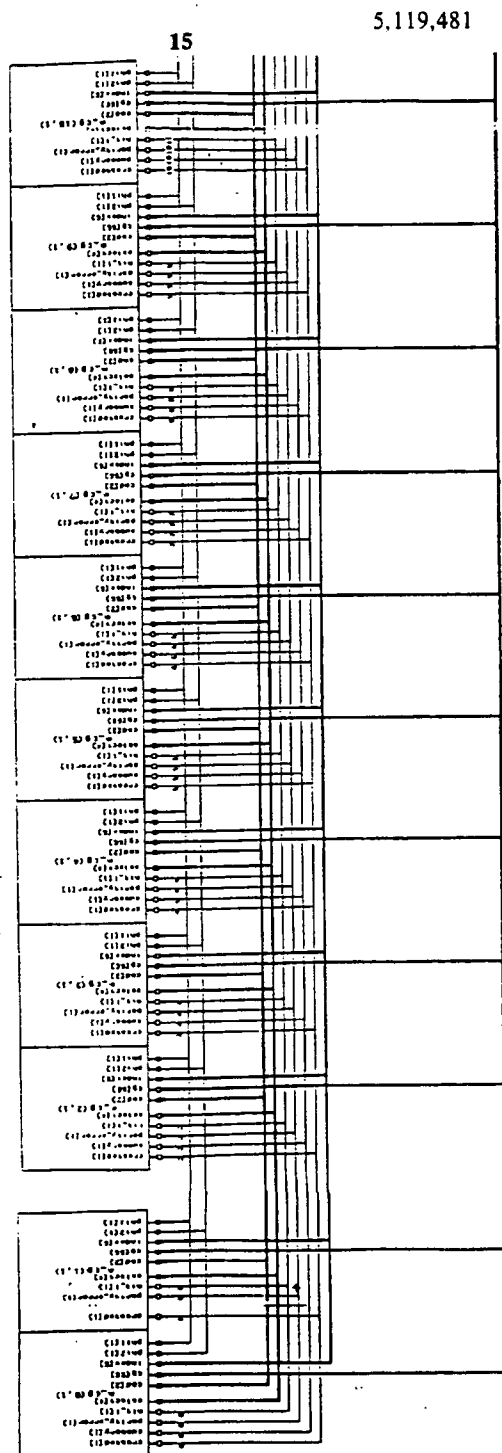
Cell Interconnect Schematics and Clock Diagrams

SCHEMATICS and DIAGRAMS

1. CI
2. CIU
3. directory
4. cache group
5. data path
6. data path cell
7. data path cache group low
8. data path cache group high
9. data path extract fifo
10. data path insert fifo
11. clocks

Cell Interconnect	
Revision 1	9/15/87
Proprietary and confidential to Kendall Square Research. Reproduction Forbidden.	



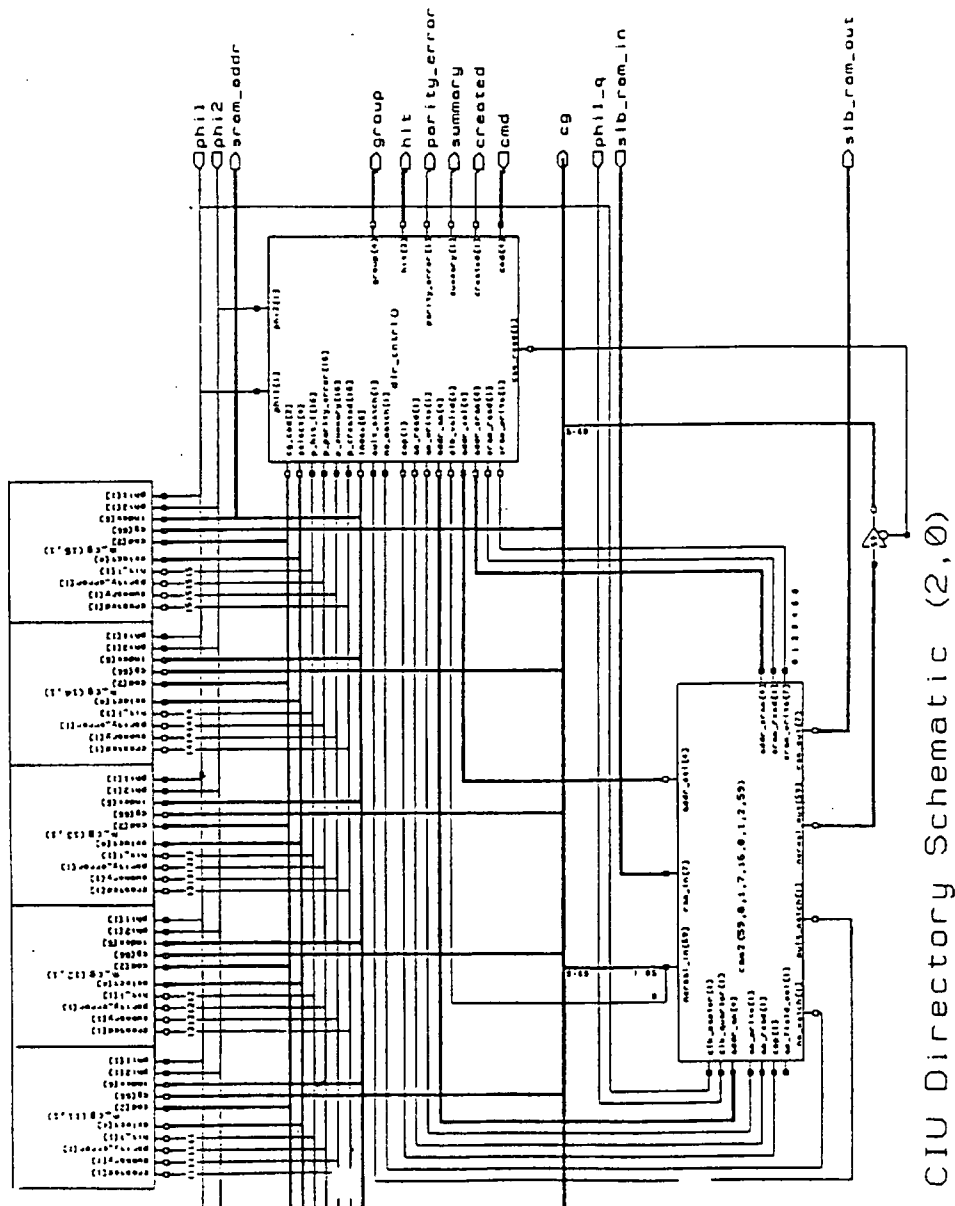


CIU Directory Schematic (1,0)

17

5,119,481

18

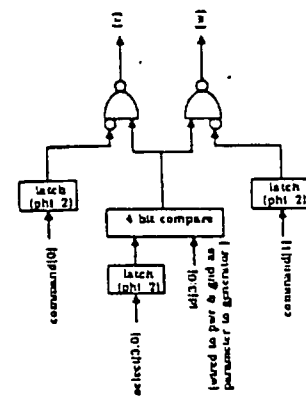
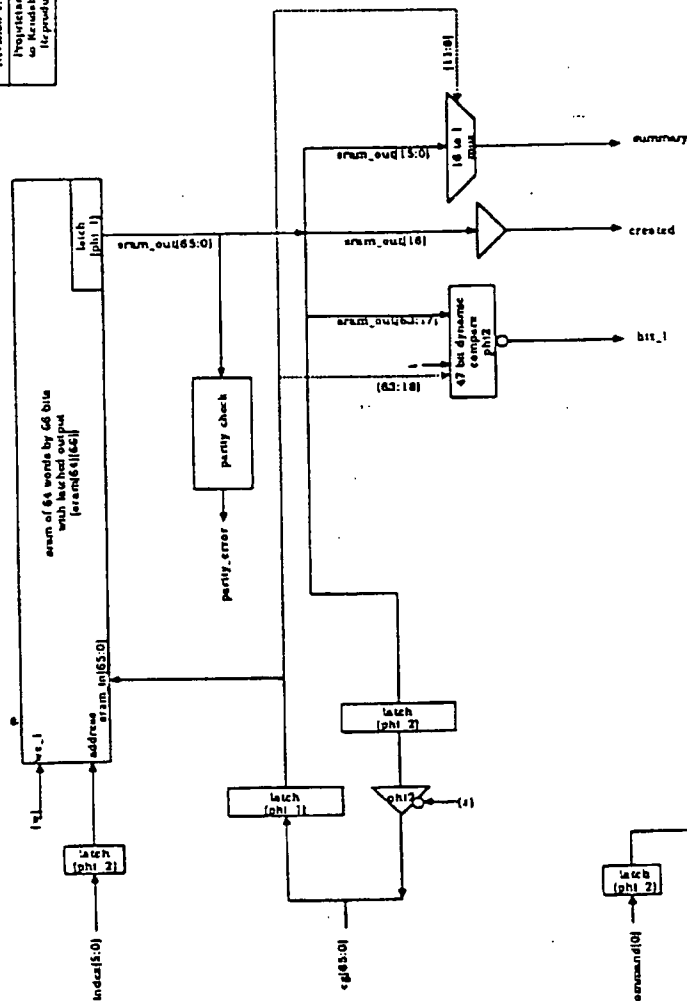


19

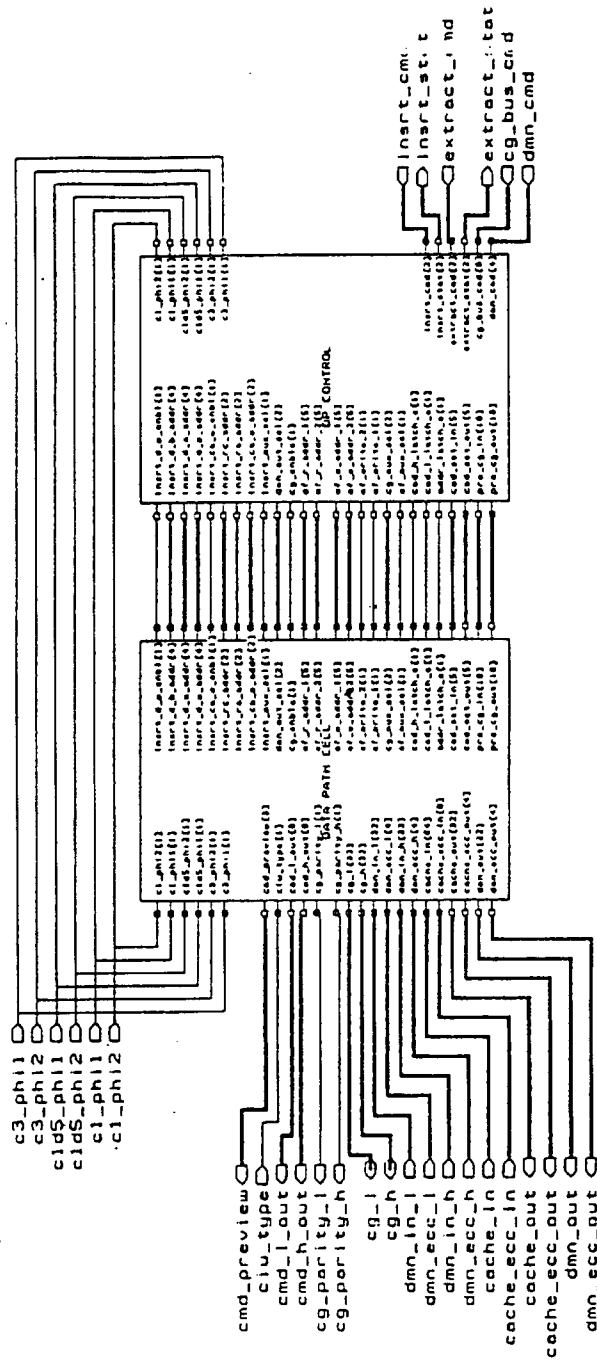
5,119,481

20

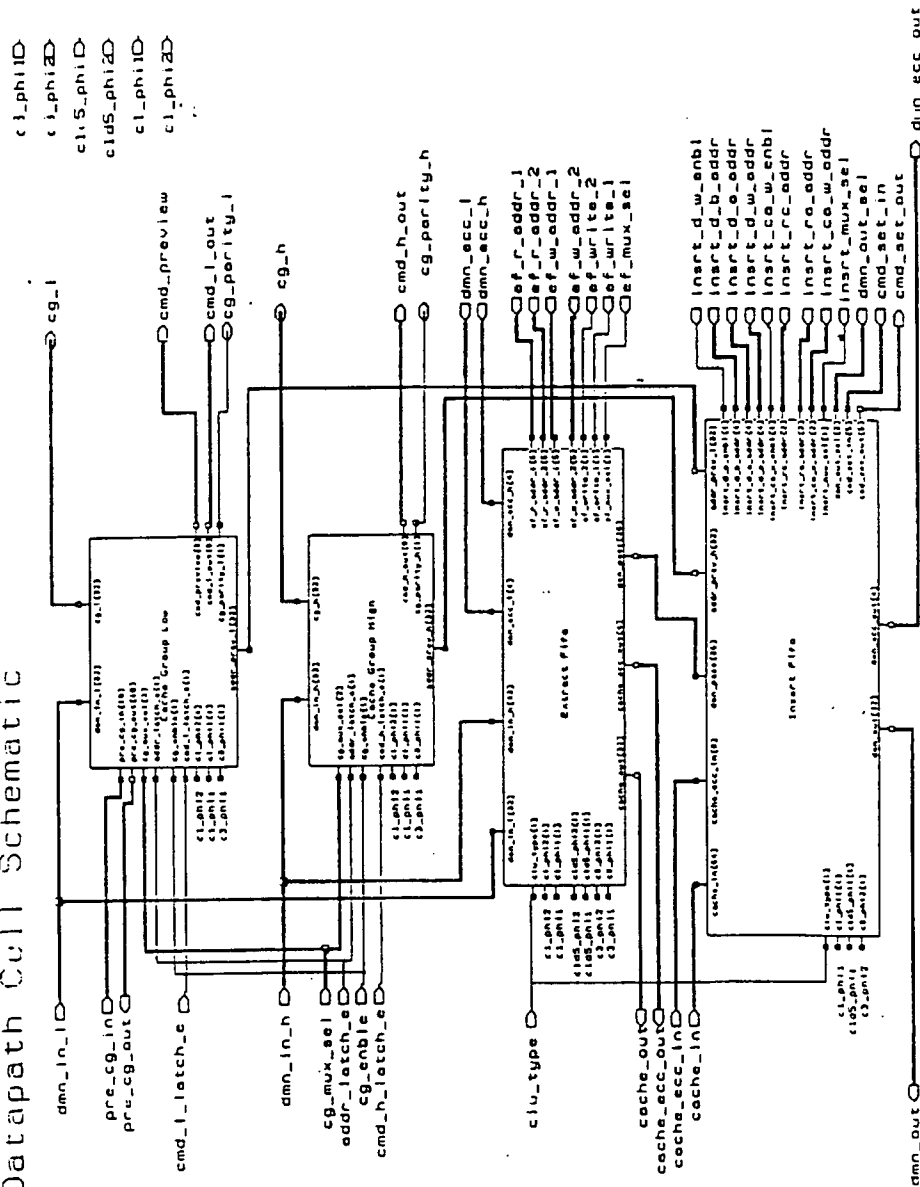
CIU Cache Group	10/	7/87
Proprietary and confidential to Merrill Lynch, Pierce, Fenner & Smith Inc.		



CIU: Datapath Schematic



CPU Datapath Cell Schematic

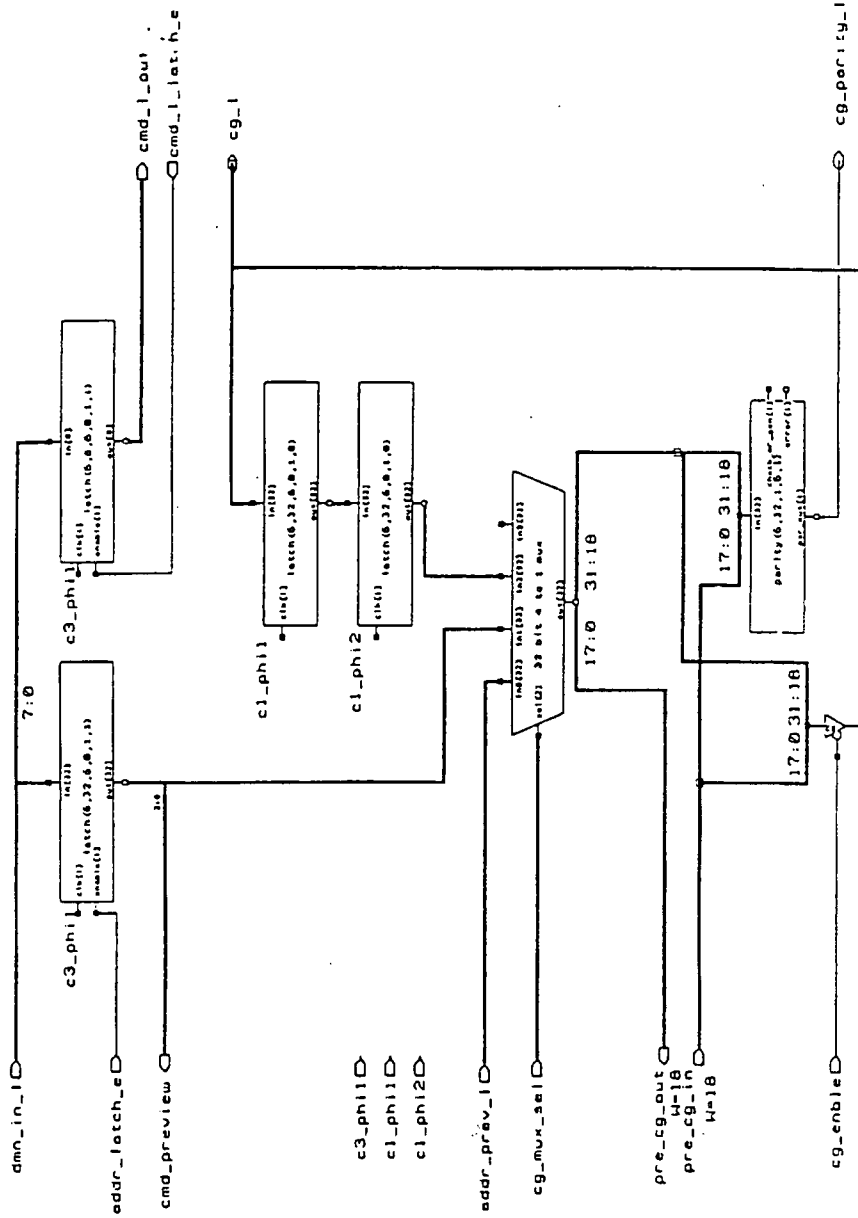


25

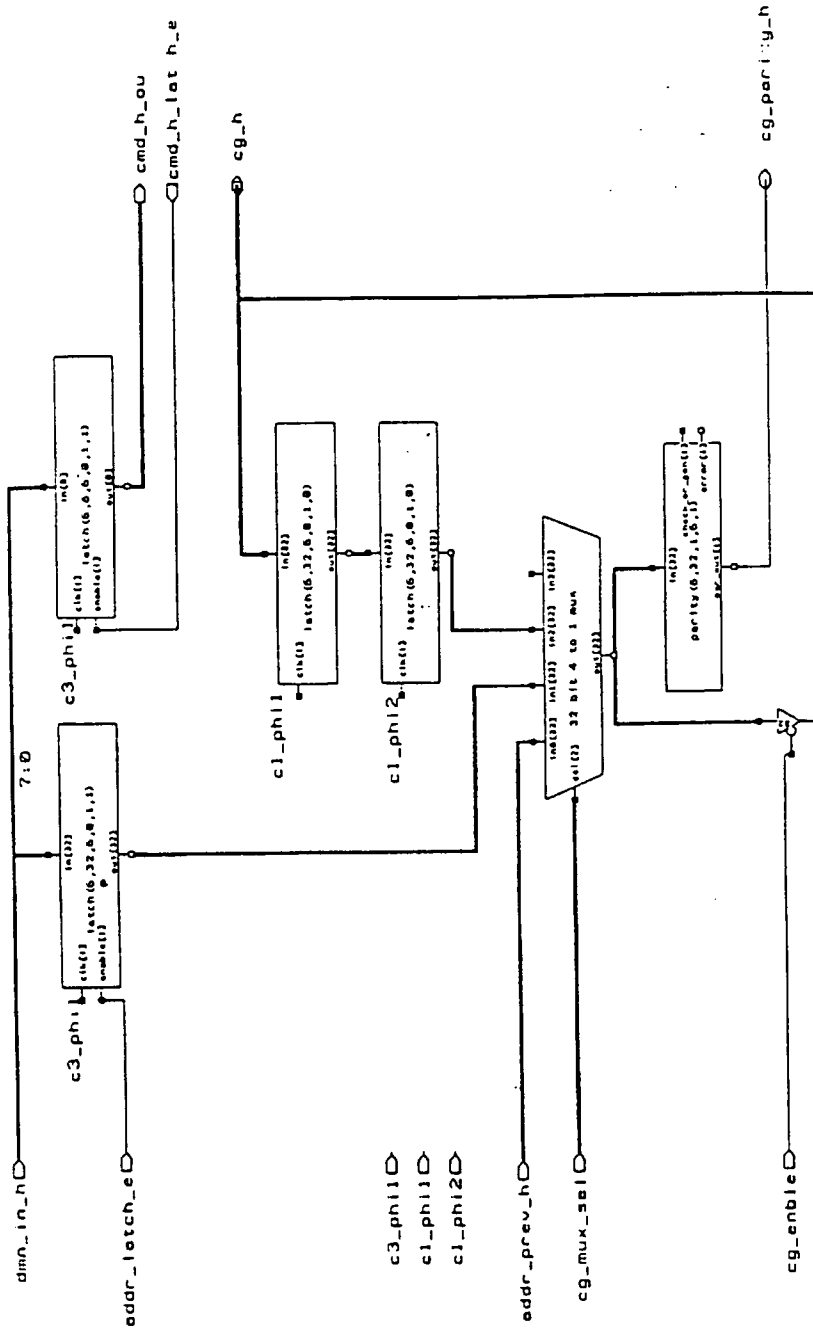
5,119,481

26

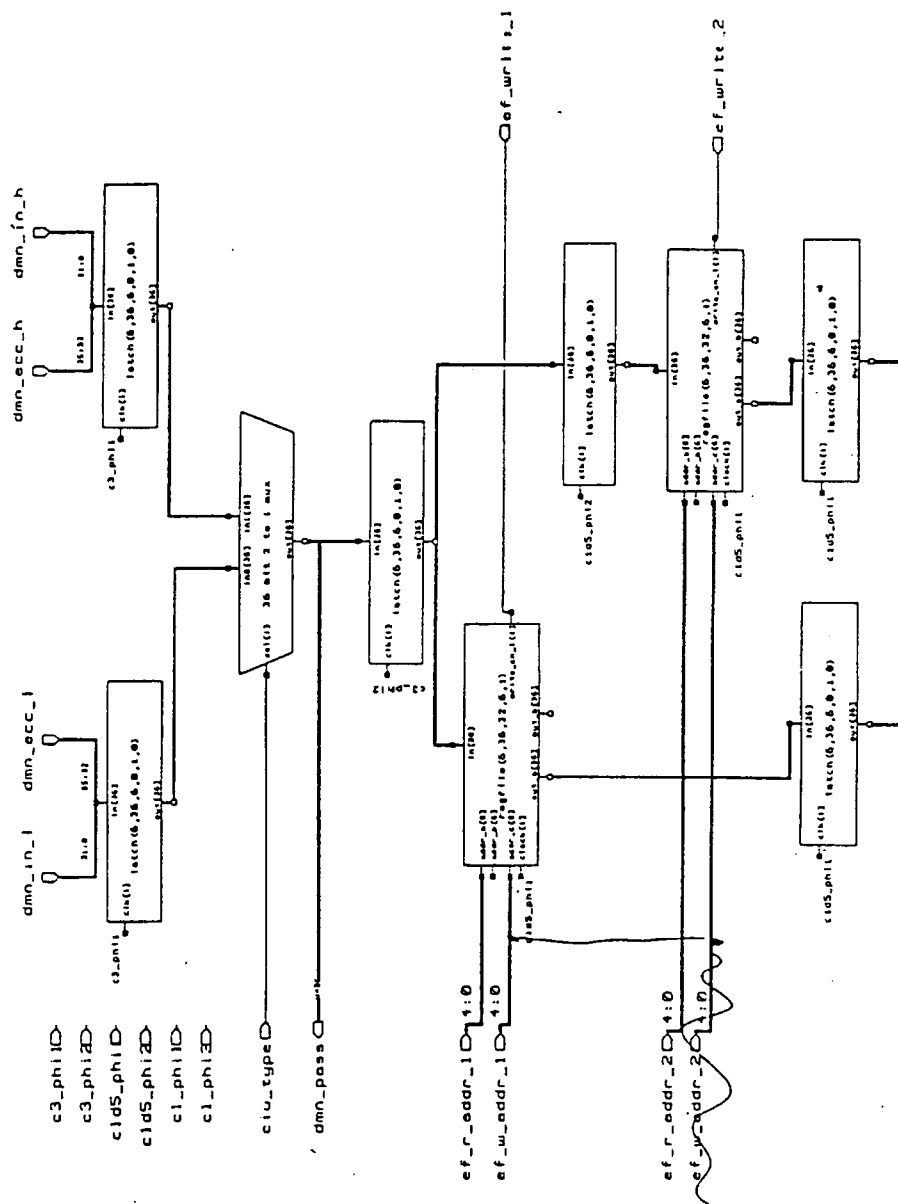
Data Path Cache Group Low Schematic



Data Path Cache Group High Schematic



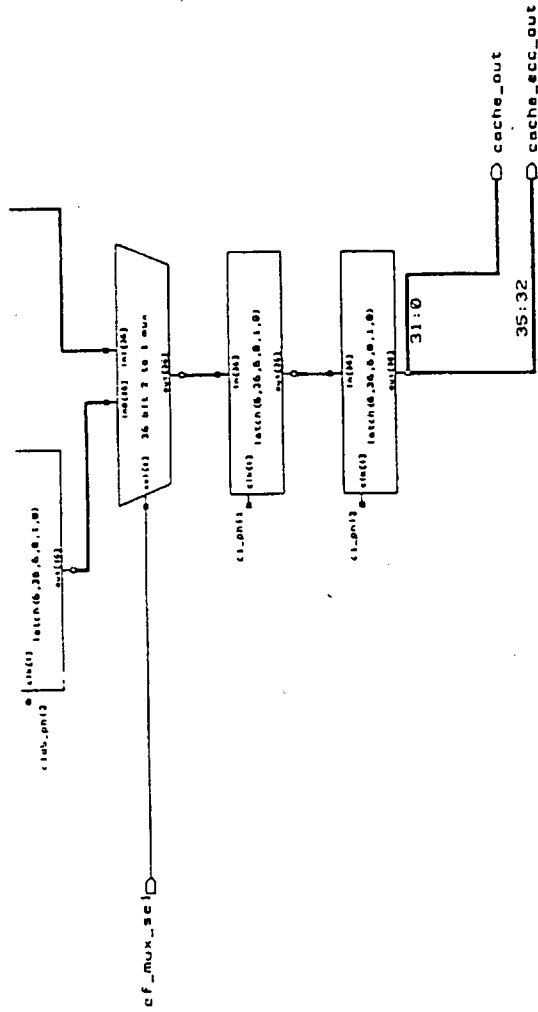
CIU Datapath Extract Fifo Schematic (0,0)



31

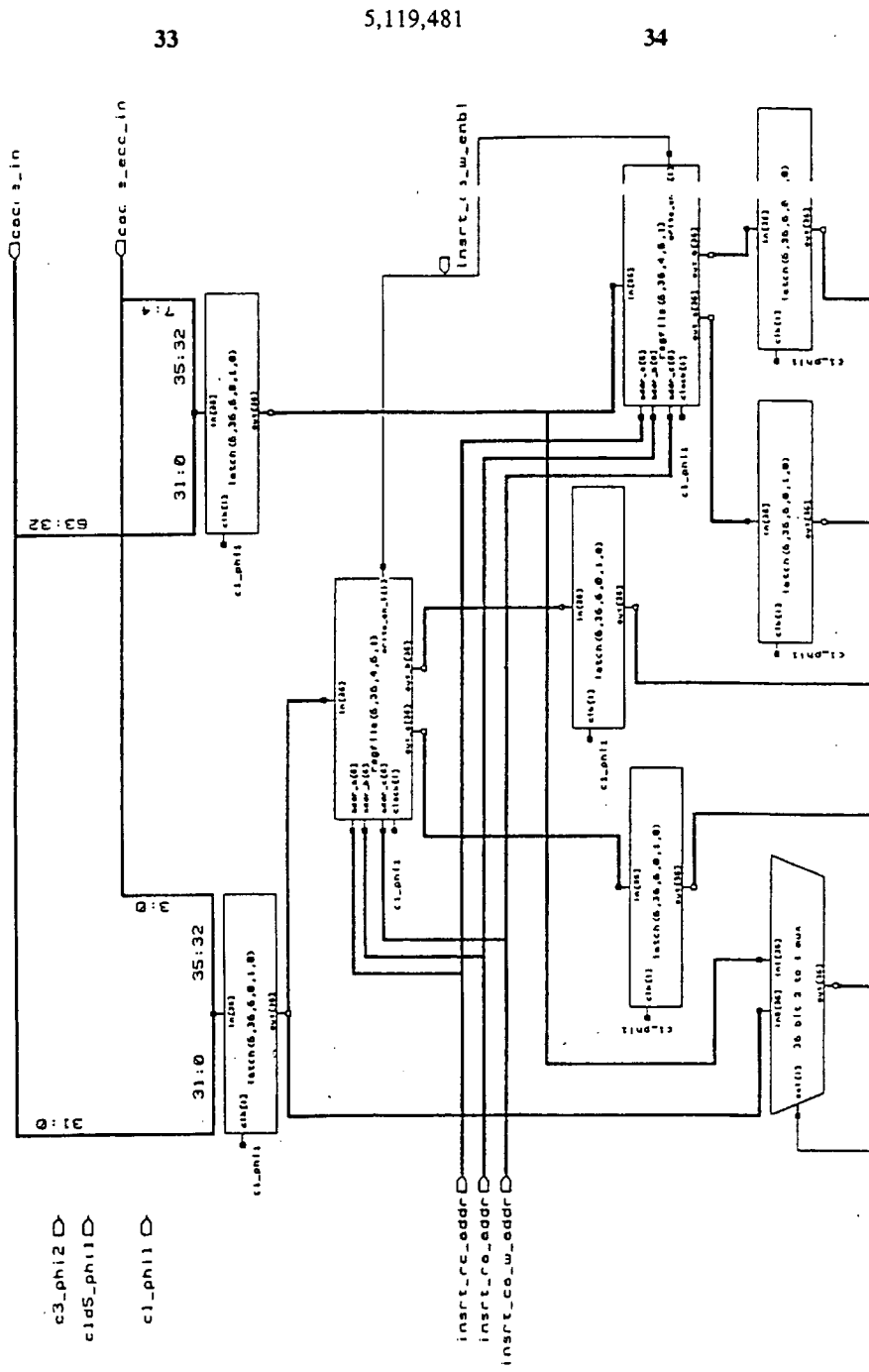
5,119,481

32



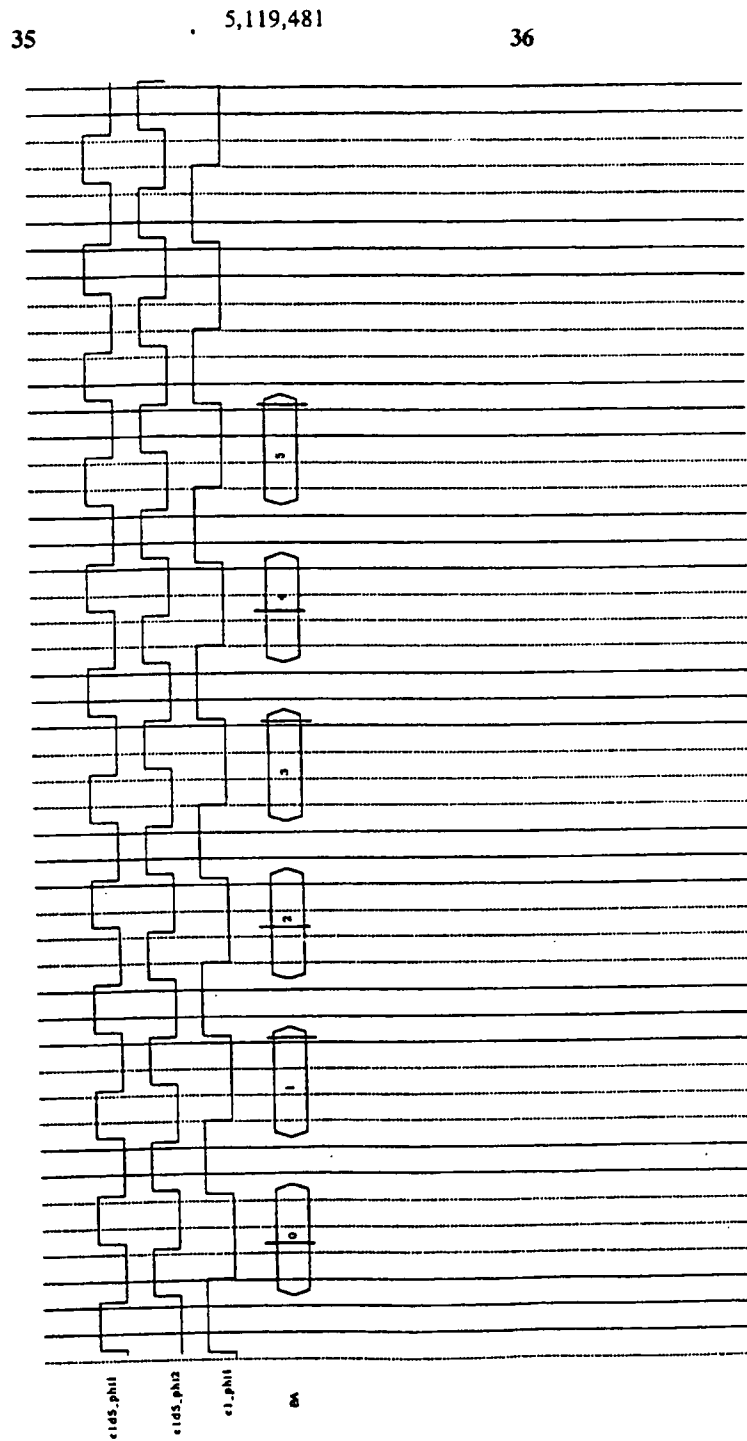
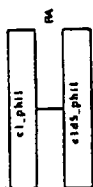
CIU Datapath Extract Fifo Schematic (0,1)

luta Path Insert Fifo (0,0)



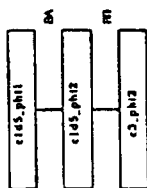
Ideal CIU Clocks	
Revision 1	10/11/07
Proprietary and confidential to Kendall Square Research. Reproduction Forbidden.	

Cache Bus to Insertion FIFO



Ideal CIU Clocks	
Revision 1	10/14/07
Proprietary and confidential information. No part of this document may be reproduced without permission.	

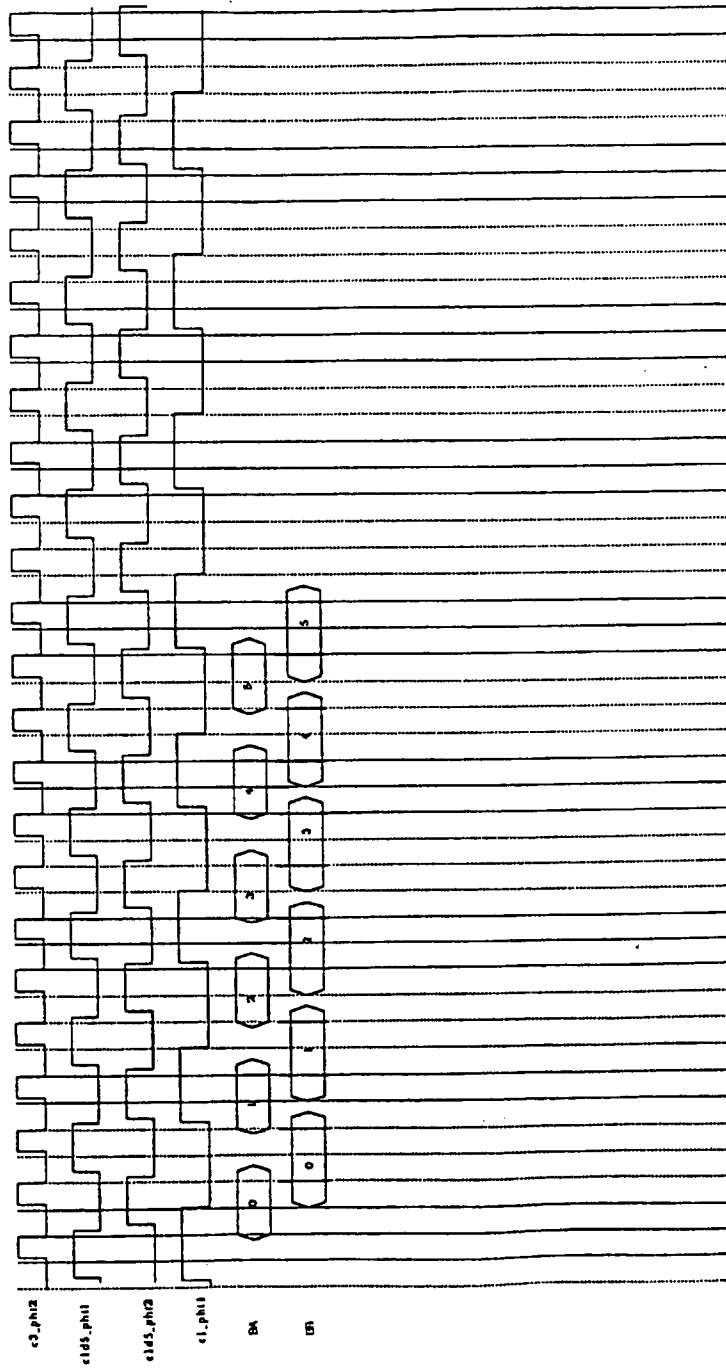
Insertion Fifo to Domain



37

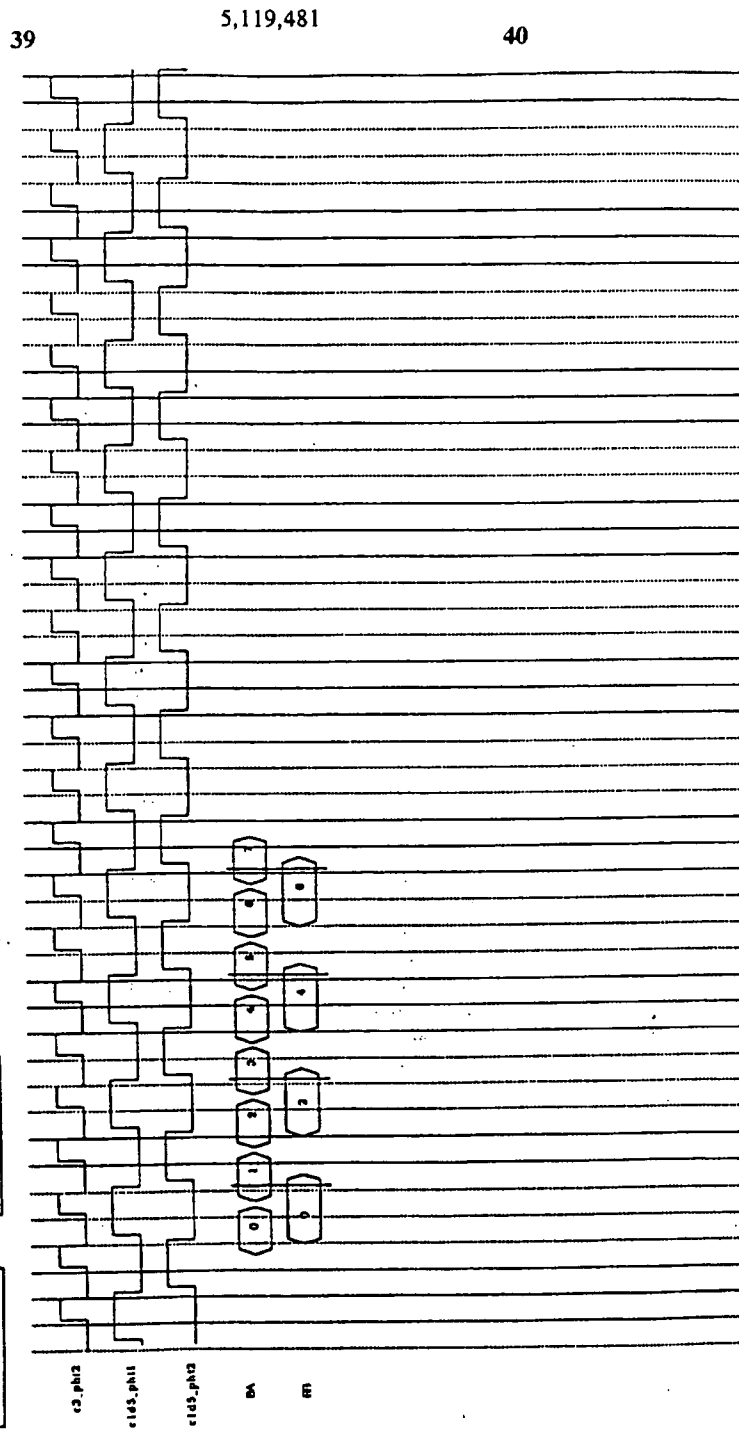
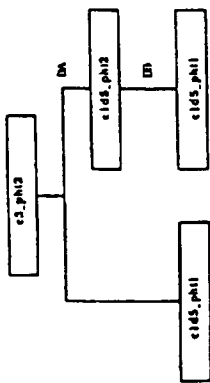
5,119,481

38



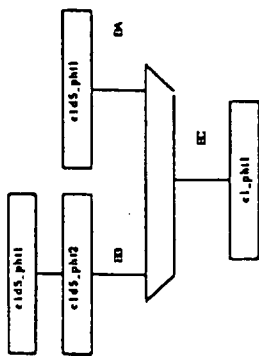
Ideal CIU Clocks	
Revision 1	10/14/07
Proprietary and Confidential Intellectual Property of Intel Reproduction Prohibited	

Domain to Extraction Fifo



Ideal CIU Clocks	
Version 1	10/14/07
Proprietary and confidential to K. S. S. Square Research. Reproduction forbidden.	

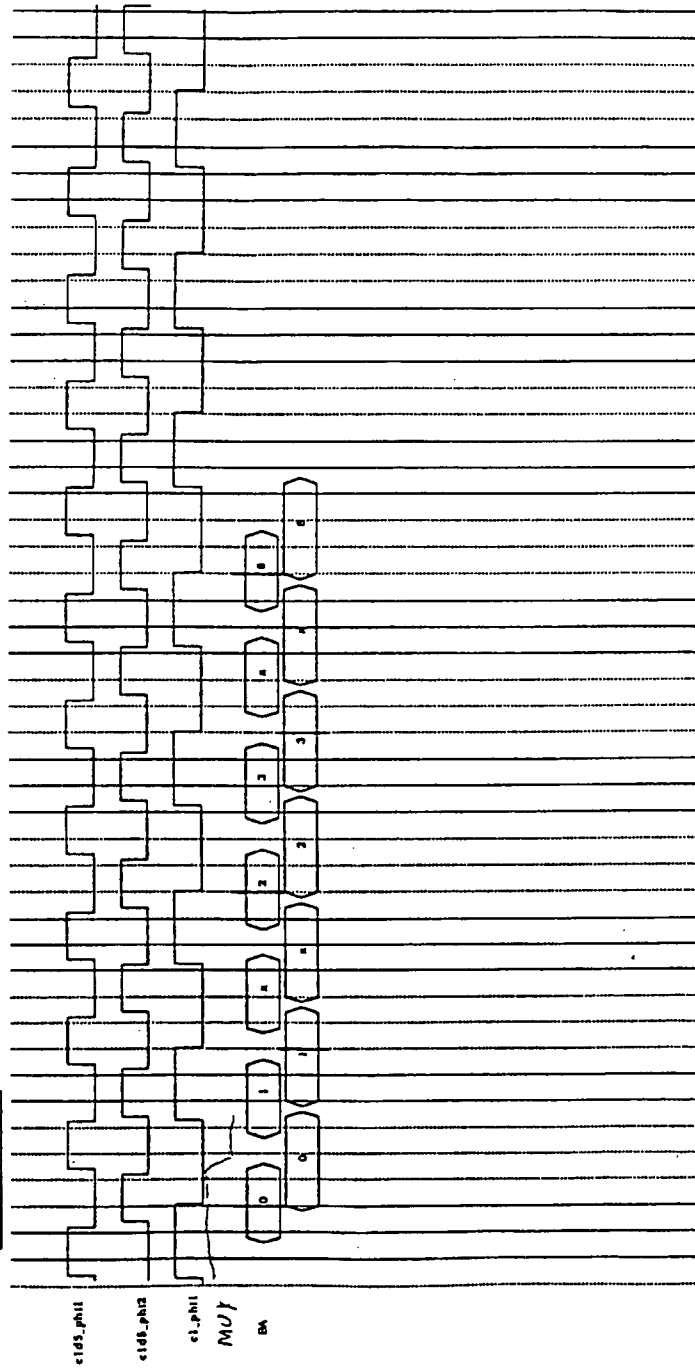
Extraction Fio to Cache Bus

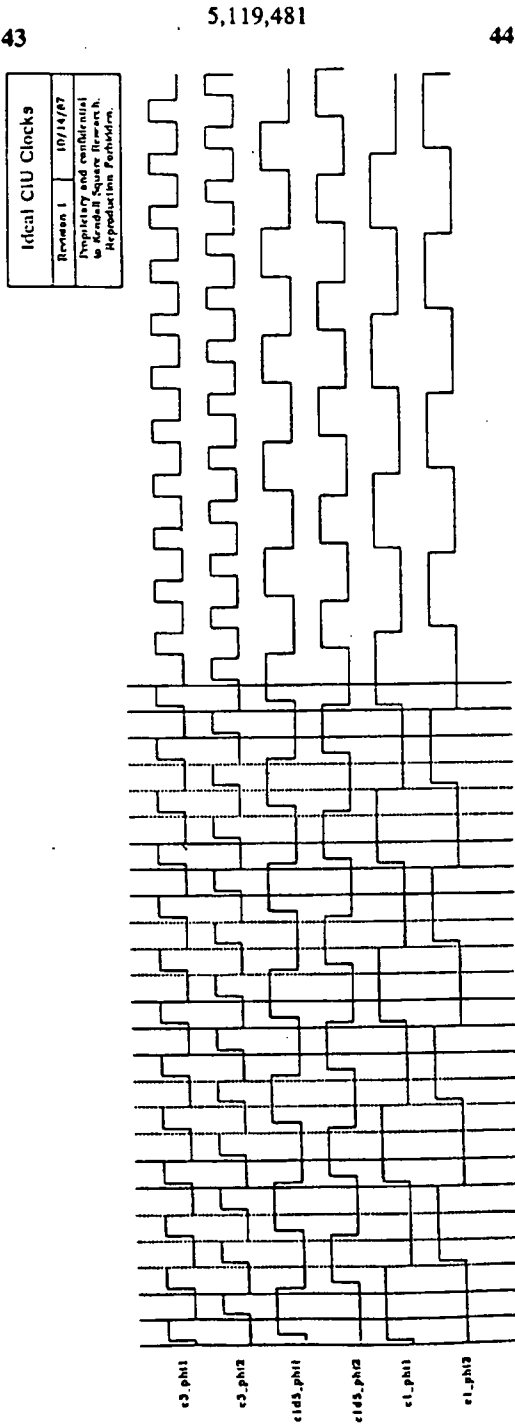


41

5,119,481

42





45

5,119,481

46

Having described the invention, what is claimed as new and secured by Letters Patent is:

1. A digital data processing apparatus comprising
 - A. bus means for transferring packets of information-representative digital signals, said bus means including shift register means comprising a plurality of digital storage and transfer stages connected in series in a ring configuration for sequentially storing and transferring said information-representative digital signals, wherein each said stage within said shift register means includes means for storing an information-representative signal of (M) bits, where (M) is greater than one,
 - B. a plurality of processing cells, connected in a ring configuration through said bus means, each processing cell being in communication with an associated subset of (N) said stages, where (N) is greater than one, at least one of said cells having associated memory means coupled thereto for storing a plurality of information-representative digital signals,
 - C. said at least one said processing cell further including cell interconnect means, connected to said associated subset of stages and said associated memory means, for selectively transferring information-representative signals between said associated subset of stages and said associated memory means,
 - D. said cell interconnect means including means for performing at least one of modifying, extracting, replicating and transferring a packet of digital information-representative signals, wherein at least a portion of said packet is stored within said associated subset of stages, based on an association, if any, between an information-representative signal identified in that packet and one or more information-representative signals of said plurality of information-representative signals stored in said associated memory means, and
 - E. said cell interconnect means including means responsive to applied digital clock cycle signals for simultaneously transferring at least a selected digital signal packet through successive stages of said associated subset of stages, at a rate responsive to said digital clock cycle rate, while performing said at least one modifying, extracting, replicating and transferring operation on that same digital signal packet.
2. A digital data processing apparatus according to claim 1, wherein said at least one processing cell includes
 - A. directory means for storing signals representative of one or more information-representative digital signals of said plurality of information-representative signals stored in said associated memory means,
 - B. means coupled with said cell interconnect means and with said directory means for generating a cell interconnect control signal based on a comparison of said information-representative signal identified in said packet with those represented in said directory means, and
 - C. said cell interconnect means includes means selectively responsive to said cell interconnect control signal for performing said at least one modifying, extracting, replicating and transferring operation on said digital signal packet.
3. A digital data processing apparatus comprising
 - A. bus means for transferring packets of information-representative digital signals, said bus means in-

- cluding shift register means comprising a plurality of digital storage and transfer stages connected in series in a ring configuration for sequentially storing and transferring said information-representative digital signals, wherein each said stage within said shift register means includes means for storing an information-representative signal of (M) bits, where (M) is greater than one.
 - B. a plurality of processing cells, connected in a ring configuration through said bus means, each processing cell being in communication with an associated subset of (N) said stages, where (N) is greater than one, at least one of said cells having associated memory means coupled thereto for storing information-representative digital signals,
 - C. said at least one said processing cell further including cell interconnect means, connected to said associated subset of stages and said associated memory means, for selectively transferring information-representative signals between said associated subset of stages and said associated memory means, and
 - D. said cell interconnect means including means responsive to applied digital clock cycle signals for simultaneously transferring at least a selected digital signal packet through successive stages of said associated subset of stages, at a rate responsive to said digital clock cycle rate, while performing at least one modifying, extracting, replicating and transferring operation on that same digital signal packet.
4. A data processing apparatus according to any of claims 1 - 3, wherein said at least one processing cell includes
 - A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet store request, and
 - B. said associated cell interconnect means includes means responsive to said packet store request signal for storing a selected information-representative signal to an associated stage of said shift register means.
 5. A data processing apparatus according to any of claims 1 - 3, wherein said at least one processing cell includes
 - A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet access request, and
 - B. said associated cell interconnect means includes means responsive to said packet access request signal for providing said processing cell access to an information-representative signal stored in an associated stage of said shift register means.
 6. A data processing apparatus according to any of claims 1 - 4, wherein said at least one processing cell includes
 - A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet extraction request, and
 - B. said associated cell interconnect means includes means responsive to said packet extraction request signal for extracting an information-representative signal stored in an associated stage of said shift

47

5,119,481

register means and for transferring that extracted information-representative signal to said associated processing cell.

7. A data processing apparatus according to any of claims 1 - 3, wherein said at least one processing cell includes

A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet replication request, and

B. said associated cell interconnect means includes means responsive to said packet replication request signal for replicating an information-representative signal stored in an associated stage of said shift register means and for transferring that replicated information-representative signal to said associated processing cell.

8. A data processing apparatus according to any of claims 1 - 3, wherein said at least one processing cell includes

A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet transfer request, and

B. said associated cell interconnect means includes means responsive to said packet transfer request signal for transferring unchanged and information-representative signal stored in a first associated stage of said shift register means to a second, succeeding associated stage of said shift register means.

9. A data processing apparatus according to any of claims 1 - 3, wherein said at least one processing cell includes

A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet identify request, and

B. said associated cell interconnect means includes means responsive to said packet identify request signal to identify a given digital information-representative signal stored in an associated stage of said shift register means as the first information-representative signal of a digital signal packet.

10. A digital data processing apparatus according to claim 9, wherein said cell interconnect means includes

A. means for generating and transmitting to the associated cell interconnect means a cell interconnect control signal representative of a packet identify command, and

B. said cell interconnect means includes means responsive to said packet identify command signal for setting a digital information-representative signal stored in an associated stage of said shift regis-

48

ter to a selected value identifying that information-representative signal as the first such signal in a digital signal packet.

11. A digital data processing apparatus according to any of claims 1 - 3, wherein said cell interconnect includes means for assembling a digital signal packet including at least one digital information-representative signal representative of a memory address, at least one digital information-representative signal representative of a command, and at least one digital information-representative signal representative of data.

12. A digital data processing apparatus according to claims 1-3, wherein said shift register means includes

A. means for storing a given digital word in a first stage of said shift register means, and

B. means responsive to said applied digital clock signal for transferring said digital word stored in said first stage to a succeeding stage of said shift register means.

13. A digital data processing apparatus according to claim 12 wherein

a given digital word is resident in said subset of stages associated with said cell interconnect means for (N) applied digital clock cycles.

14. A digital data processing apparatus according to claim 13, wherein said shift register means includes means for simultaneously transferring to successive said stages (p) digital signal packets, where (p) is a positive integer given by

$$(p) = (c \times N) / (w)$$

where (c) is the number of said cell interconnect means connected to said bus means and (w) is the number of digital words in each digital signal packet.

15. A digital data processing apparatus according to claim 13, wherein as the number of stages of said shift register means increases, the flux of said digital words through said stages of said shift register means remains constant.

16. A digital data processing apparatus according to claim 13, wherein clock cycle skew associated with said at least one set of digital clock cycles remains substantially constant with reference to each of said (N) stages of said shift register means as the number (N) increases.

17. A digital data processing apparatus according to any of claims 1 - 3, wherein said cell interconnect means includes at least one of

A. insert buffer means for storing at least a selected digital signal packet for transfer to at least one stage of said associated subset of stages for insertion into said shift register means, and

B. second buffer means for storing at least a selected digital signal packet extracted from said associated subset of stages.

• • • • •

60

65

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,119,481

Page 1 of 2

DATED : June 2, 1992

INVENTOR(S) : Steven J. Frank, Henry Burkhardt, III, and
Frederick D. Weber

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title Page, item [54], lines 1-2, and Col. 1, lines 2-3, replace title "REGISTER BUS MULTIPROCESSOR SYSTEM WITH SHIFT" with -- MULTIPROCESSOR SYSTEM WITH SHIFT REGISTER BUS --.

Column 3, line 14, replace "Packets" with -- packets --.

Column 5, line 12, replace Passed with -- passed --.

Column 6, line 1, replace "(cache data)" with -- (cache_data) --.

Column 8, line 30, replace "ten-word Packets" with -- ten-word packets --.

Column 11, line 13, replace "as a Positive" with -- as a positive --.

Column 45, line 19, replace "signals." with -- signals, --

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,119,481

Page 2 of 2

DATED : June 2, 1992

INVENTOR(S) : Steven J. Frank, Henry Burkhardt, III, and
Frederick D. Weber

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby
corrected as shown below:

Column 45, line 41, replace "subset o stages" with
-- subset of stages --.

Column 48, line 13, replace "word in a firs" with
-- word in a first --.

Signed and Sealed this
Thirty-first Day of August, 1993

Attest:



BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

Appendix C

Table 7.7 of

Tinku Acharya & Ping-Sing Tsai,

“JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures”,

Wiley, 2005

Table 7.7 BAC Qe-value and Probability Estimation Lookup Table

Index	Qe	NMPS	NLPS	SWITCH
0	0x5601	1	1	1
1	0x3401	2	6	0
2	0x1801	3	9	0
3	0x0AC1	4	12	0
4	0x0521	5	29	0
5	0x0221	38	33	0
6	0x5601	7	6	1
7	0x5401	8	14	0
8	0x4801	9	14	0
9	0x3801	10	14	0
10	0x3001	11	17	0
11	0x2401	12	18	0
12	0x1C01	13	20	0
13	0x1601	29	21	0
14	0x5601	15	14	1
15	0x5401	16	14	0
16	0x5101	17	15	0
17	0x4801	18	16	0
18	0x3801	19	17	0
19	0x3401	20	18	0
20	0x3001	21	19	0
21	0x2801	22	19	0
22	0x2401	23	20	0
23	0x2201	24	21	0
24	0x1C01	25	22	0
25	0x1801	26	23	0
26	0x1601	27	24	0
27	0x1401	28	25	0
28	0x1201	29	26	0
29	0x1101	30	27	0
30	0x0AC1	31	28	0
31	0x09C1	32	29	0
32	0x08A1	33	30	0
33	0x0521	34	31	0
34	0x0441	35	32	0
35	0x02A1	36	33	0
36	0x0221	37	34	0
37	0x0141	38	35	0
38	0x0111	39	36	0
39	0x0085	40	37	0
40	0x0049	41	38	0
41	0x0025	42	39	0
42	0x0015	43	40	0
43	0x0009	44	41	0
44	0x0005	45	42	0
45	0x0001	45	43	0
46	0x5601	46	46	0

Appendix D

Amit Gupta, Saeid Nooshabadi & David Taubman,
“Concurrent Symbol Processing Capable VLSI Architecture for Bit Plane Coder of JPEG2000”,
IEICE Trans. Inf. & System,
Vol. E88-D, No. 8, August 2005

PAPER Special Section on Recent Advances in Circuits and Systems

Concurrent Symbol Processing Capable VLSI Architecture for Bit Plane Coder of JPEG2000*

Amit Kumar GUPTA^{†a)}, Saeid NOOSHABADI[†], and David TAUBMAN[†], Nonmembers

SUMMARY JPEG2000 image compression standard is designed to cater the needs of a large span of applications including numerous consumer products. However, its use is restricted due to the high hardware cost involved in its implementation. Bit Plane Coder (BPC) is the main resource intensive component of JPEG2000. Its throughput plays a key role in deciding the overall throughput of a JPEG2000 encoder. In this paper we present the algorithm and parallel pipelined VLSI architecture for BPC which processes a complete stripe-column concurrently during every pass. The hardware requirements and the critical path delay of the proposed technique are compared with the existing solutions. The experimental results show that the proposed architecture has 2.6 times greater throughput than existing architectures, with a comparatively small increase in hardware cost.

key words: JPEG2000, concurrent symbol processing, bit plane coder, VLSI architecture

1. Introduction

JPEG2000 image compression standard [3] possesses a rich set of features to efficiently cater the requirements of several application areas such as digital imaging products, scanners, efficient remote browsing and transmission of images etc. The high level functional block diagram of JPEG2000 encoder is shown in Fig. 1. The embedded block coding algorithm of JPEG2000 is a part of EBCOT algorithm, proposed in [1]. It consists of Bit Plane Coding (BPC) and Arithmetic Coding (AC) modules. The BPC is one of the main resource intensive components of JPEG2000. It accounts for nearly 50% [4] of the total computation time of JPEG2000 encoding process. The BPC module encodes the so-called code-block of quantized wavelet coefficients, and provides Context-Data (CxD) pair to be encoded by the arithmetic coder. BPC works sequentially on each bit-plane of the code-block starting from the most significant magnitude bit-plane. For every sample to be coded, as explained in Sect. 2, BPC sends the context and the data bits to the arithmetic coder, for further processing. BPC generates the context depending on the significance state of the neighboring locations. The significance state is dynamically updated throughout encoding of the samples depending on the data bits. The other run-time event is the distribution of the sample-locations to three different coding passes. Due to

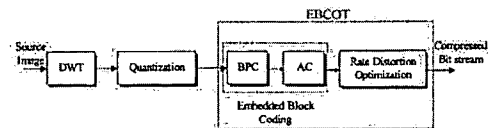


Fig. 1 The block diagram of JPEG2000 encoder.

these dynamic events, the existing VLSI architectures [4], [5] for the BPC module, employ sequential single sample processing and generate at most one Context-Data (CxD) pair in a single clock-cycle.

In this paper, we propose the Concurrent Symbol Processing (CSP) algorithm to process a complete stripe-column concurrently during each coding pass. The algorithm exploits the fact that the coding pass membership of multiple sample-locations can be calculated concurrently. We also review the coding pass membership tests described in [4] and propose necessary amendments. In addition, we present the VLSI architecture of a BPC module based on CSP algorithm which generates up to 10 CxD pairs in a single clock-cycle. An Altera APEX20KE FPGA is used as a common platform to examine the hardware cost and critical path performance of the proposed architecture in comparison to existing architectures.

The rest of the paper is organized as follows. In Sect. 2 we explain the BPC algorithm and existing architectures. In Sect. 3 we introduce the proposed algorithm and architecture. The results are presented in Sect. 4 and we discuss the impact of the proposed architecture on the overall throughput of the embedded block coder in Sect. 5.

2. Bit Plane Coder and Existing Architectures

The BPC module works independently on so called *code-blocks* of quantized wavelet coefficients. It encodes the code-block by generating the context for each bit (except in the run-mode) to be coded by AC module. The coefficients are represented in the sign-magnitude form. The BPC module encodes the magnitude bit-plane by bit-plane, starting with the most significant bit-plane for a given code-block (Fig. 2 (a)). The encoding of sign bit-plane is dynamically distributed among the magnitude bit-planes depending on the coefficients.

Each bit at every location is coded in one of three non-overlapping passes: the Significance Propagation (SP) Pass;

Manuscript received October 9, 2004.

Manuscript revised February 3, 2005.

[†]The authors are with the Faculty of Electrical Engineering at UNSW, Sydney, Australia.

*A part of this paper is presented at IEEE MWSCAS 2004 conference.

a) E-mail: amigupta@student.unsw.edu.au

DOI: 10.1093/ietisy/e88-d.8.1878

Copyright © 2005 The Institute of Electronics, Information and Communication Engineers

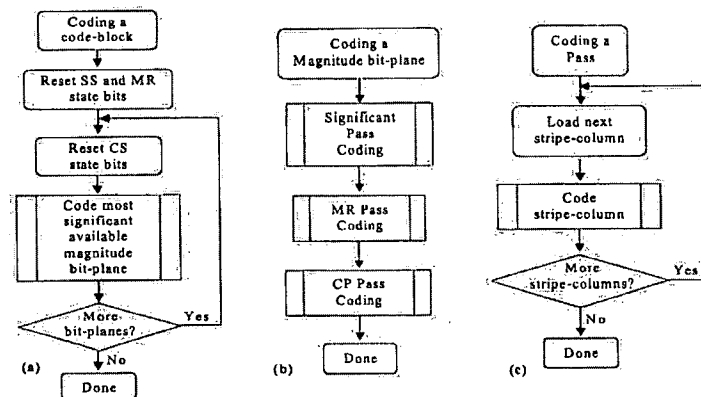


Fig. 2 High level processing flow of bit-plane encoding algorithm; (a) Flow chart showing bit-plane by bit-plane processing order; (b) 3 Pass scanning order for encoding a bit-plane, and (c) Stripe-column based processing flow within a pass.

the Magnitude Refinement (MR) Pass; and the Cleanup (CP) Pass (Fig. 2(b)). A significance state is defined for every sample-location which, along with the significance state of the eight neighboring locations in the 3×3 context-window (Fig. 3), governs its pass membership. All locations are assigned insignificant state initially at the start of code-block. A location becomes significant immediately after its first non-zero bit has been coded. A sample-location belongs to: *SP pass*, if it is currently insignificant but at least one of its neighbors (neighboring locations in the context-window) is significant;

MR pass, if the location is significant and has not been coded in the SP pass; and

CP pass, if the location has not been coded in either the SP or MR passes.

Within a bit-plane, for all the three passes, the data is scanned in stripe fashion, with 4 rows per stripe, column-by-column order from left to right (Fig. 3). One of the three different coding primitives, Zero Coding, Run-length Coding and Magnitude Refinement Coding are used to generate the context for a data bit depending on the sample-location's pass membership. The context for coding the sign bit of each sample-location is generated immediately after the sample-location becomes significant using Sign Coding primitive. The Zero Coding primitive, used in the SP and CP passes, generates the context depending on the significant state values of the neighboring 8 locations in the context-window (H0, H1, V0, V1, D0, D1, D2, and D3 as shown in Fig. 3(b)). The MR coding primitive is used in MR pass, and apart from the significant state values of the neighboring 8 locations, requires the information to indicate if the relevant sample-location is to be encoded for the first time in the MR pass. Run-length coding is used in the CP pass in the special case when all the 4 locations in the stripe-column are insignificant and have insignificant neighbors. The Sign

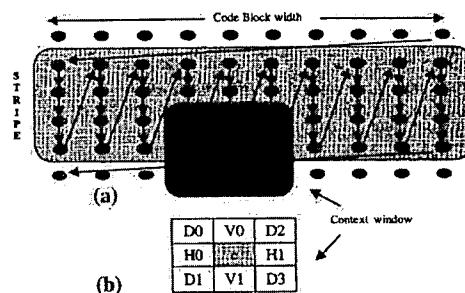


Fig. 3 (a) The stripe based scanning order for every pass; (b) Context window for a sample location (The context-window shown is for the sample location represented by empty circle in the stripe and named 'c' in the 3×3 array).

Coding primitive requires the sign and significance states of vertical and horizontal neighbors in the context-window (H0, H1, V0 and V1).

Two main architectures have been proposed to date. Both employ stripe-column based processing (Fig. 2(c)), as suggested in [2]. The architectures mainly differ in their implementation of *code stripe-column* block of Fig. 2(c). The first architecture is a single sample architecture [5] which checks each sample-location for its membership to the current pass, and if the sample-location belongs to the current pass, its context is generated. The second architecture [4] is more efficient as it employs a sample-skipping strategy to skip those samples, within a stripe-column, which do not belong to the current pass. Both the architectures generate at most one CxD pair in a clock-cycle but the first architecture wastes a clock-cycle whenever it encounters a sample which does not belong to the current coding pass.

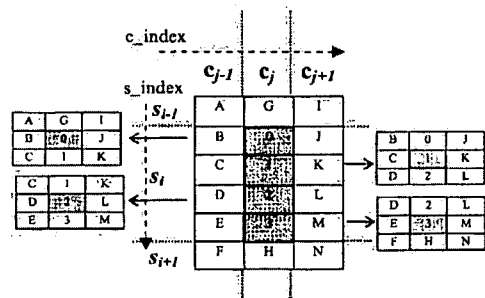


Fig. 4 Context-window for stripe-column processing and respective context-window for each sample location in stripe-column c_j of stripe s_i .

3. Proposed Algorithm and Architecture

Our proposed algorithm processes a complete stripe-column in a single clock-cycle during each coding pass. Thus it generates any where between 0 and 10 CxD pairs in a single clock-cycle. The extreme case of 10 CxD pairs happens only during the CP pass when the run-mode condition is satisfied but a run-interrupt occurs immediately at the first sample-location in the stripe-column.

To explain the algorithm, we define 3 state bits, a Significant State (SS) bit, a Magnitude Refinement (MR) bit and a Coding State (CS) bit, for each sample-location. For all locations, the SS and MR bits are initialized to zero before coding the first bit-plane for the code block, while the CS bit is initialized to zero before coding each bit-plane. The SS bit maintains the significance status of the location; it becomes 1 after the first non-zero bit of the location has been coded. The MR status bit maintains a delayed version of the SS bit. More specifically the MR bit is set to 1 for a location when it is first coded in the MR pass. The CS bit maintains the coding state of a location; it will be set to 1 when a location has been coded in a bit-plane. The state bits associated with a sample-location are updated immediately when it gets encoded.

The overall context-window for stripe-column based processing of Normal mode of BPC module is shown in Fig. 4. JPEG2000 also defines Causal mode of operation in which the context-window does not have any dependency on the significance state of locations from future stripes (locations F , H and N of stripe s_{i+1}). We discuss the algorithm for normal mode of operation. The algorithm is applicable for causal mode as well by taking the SS bits of locations of future stripe to be zero. We also assume that the (c_j, s_i) is the current stripe-column c_j , of stripe s_i , to-be-coded. Pass membership of all the 4 sample-locations in the current stripe-column is required to be able to process a complete stripe-column concurrently. This requires the access to the two 18-bit sets representing SS and sign bits of locations $\{A-N, 0-3\}$; and three 4-bit sets representing MR,

CS and data bits for locations $\{0-3\}$ as shown in the overall context-window for a stripe-column in Fig. 4. We use the following terminology:

j - A location in the current stripe-column's context-window, $j \in \{0-3, A-N\}$.

$v[j]$ - data bit at location j .

$\sigma[j]$ - significance state of location j before coding the current stripe-column during the current pass.

$\sigma'[j]$ - significance state of location j after coding the current stripe-column during the current pass. This value may differ from $\sigma[j]$ only for locations $\{0-3\}$.

$\pi[j]$ - CS bit at location j .

$K\sigma[j]$ - single bit variable which signifies if any sample-location in the context-window of sample-location j is significant or not.

$Pflag[j]$ - pass flag of location j . A single bit variable with value 1 means the location j belongs to the current pass.

$|$ - bit-wise OR operation.

$\&$ - bit-wise AND operation.

$\sim$ - bit-wise negation.

In the proposed algorithm the overall processing flow, up to stripe-column encoding (Fig. 2), is similar as used in previously proposed architectures [4], [5]. The block *code stripe-column* is treated in a different way. To encode a stripe-column, during every pass, we calculate:

1. Pass flags for each sample-location.
2. A Personalized Set of SS Bits (PSSB) for each sample-location. The meaning and significance of the PSSB is explained later in this section.
3. Run-mode eligibility, using the $K\sigma[j]$ bits of all sample-locations in the current stripe-column.
4. Run-interruption, using the data bits of all the sample-locations in the current stripe-column.
5. The context of each sample-location, from its PSSB.
6. The updated value of state bits for all the sample-locations.

Concurrent pass membership testing of the stripe-column's sample-locations is the fundamental idea in the proposed algorithm. The strategy suggested in [4] has an inherent assumption that the significance state of none of the sample-locations in the stripe-column will change during the current pass under consideration. Thus this algorithm fails to take the effect of significance propagation into account. In the proposed algorithm we also consider the data bits to be coded for the sample-location $\{0, 1, 2\}$ (Fig. 4) in order to correctly deduce the coding pass membership for all four sample-locations in the stripe-column concurrently.

3.1 Concurrent Membership Testing

To contrast the proposed pass membership testing methodology with that in [4], we first present the pseudo code to concurrently generate the membership of all the sample-locations in a stripe-column during the SP pass. While generating the membership for location $j \in \{1, 2, 3\}$, we also consider the possibility that the $(j-1)$ th location may belong to the SP pass and may turn significant in this bit-plane. In

other words, we take note of the fact that, as $\sigma[j]$ changes in the SP pass, it can change the membership of the four neighbors which have not yet been visited, in the current bit-plane.

Location 0:

```
if ( $\sigma[0]$ ) Pass = MR;
elseif ( $K\sigma[0]$ ) Pass = SP;
else Pass = CP;
```

Location 1:

```
if ( $\sigma[1]$ ) Pass = MR;
elseif ( $K\sigma[1]$ ) Pass = SP; elseif ( $K\sigma[0] \& v[0]$ ) Pass = SP;
else Pass = CP;
```

Location 2:

```
if ( $\sigma[2]$ ) Pass = MR;
elseif ( $K\sigma[2]$ ) Pass = SP;
elseif ((( $K\sigma[0] \& v[0]$ ) | ( $K\sigma[1] \& v[1]$ )) | ( $K\sigma[2] \& v[2]$ )) Pass = SP;
else Pass = CP;
```

Location 3:

```
if ( $\sigma[3]$ ) Pass = MR;
elseif ( $K\sigma[3]$ ) Pass = SP;
elseif ((( $K\sigma[0] \& v[0]$ ) | ( $K\sigma[1] \& v[1]$ ) | ( $K\sigma[2] \& v[2]$ )) | ( $K\sigma[3] \& v[3]$ )) Pass = SP;
else Pass = CP;
```

It is important to point out that the above pseudo-code intentionally contains an extra separate 'elseif' condition emphasizing the effect of significant propagation, which may originate from any of the previous samples in the stripe-column. The logic equations for the current pass membership tests based on the pseudo-code are:

SP Pass:

$$\begin{aligned} Pflag[0] &= (K\sigma[0]) \& (\sim \sigma[0]); \\ Pflag[1] &= (K\sigma[1] | (Pflag[0] \& v[0])) \& (\sim \sigma[1]); \\ Pflag[2] &= (K\sigma[2] | (Pflag[1] \& v[1])) \& (\sim \sigma[2]); \\ Pflag[3] &= (K\sigma[3] | (Pflag[2] \& v[2])) \& (\sim \sigma[3]). \end{aligned} \quad (1)$$

MR pass:

$$Pflag[j] = \sigma[j] \& (\sim \pi[j]); \forall j \in \{0, 1, 2, 3\}. \quad (2)$$

CP pass:

$$Pflag[j] = \sim \pi[j]; \forall j \in \{0, 1, 2, 3\}. \quad (3)$$

The equations for $Pflag[j]$ presented above take into account the fact that in the SP pass $\sigma[j]$ values, and consequently, $K\sigma[j]$ bits are updated, and may change, as we move down the stripe-column.

3.2 Generation of the Personalized Set of SS Bits

By PSSB of a sample-location, we mean the set of SS bits for locations in its context-window, generated while taking care of the possibility that the significance state of an immediate top neighbor may also change during the current pass. Specifically, for sample-locations 1, 2, and 3, we must take into account the possible change of significance in location 0, 1, and 2, respectively, during the current pass. The PSSBs are as follows:

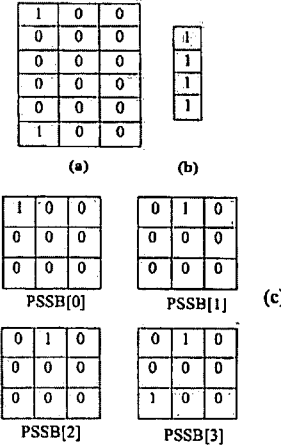


Fig. 5 An example: (a) σ bits for the stripe-column context window, (b) v bits for the current stripe-column, and (c) Generated PSSB using the updated σ value as per Eq. (4).

PSSB[0] = $\{\sigma[B], \sigma[J], \sigma[G], \sigma[1], \sigma[A], \sigma[I], \sigma[K], \sigma[C]\}$;
PSSB[1] = $\{\sigma[C], \sigma[K], \sigma'[0], \sigma[2], \sigma[B], \sigma[J], \sigma[L], \sigma[D]\}$;
PSSB[2] = $\{\sigma[D], \sigma[L], \sigma'[1], \sigma[3], \sigma[C], \sigma[K], \sigma[M], \sigma[E]\}$;
PSSB[3] = $\{\sigma[E], \sigma[M], \sigma'[2], \sigma[H], \sigma[D], \sigma[L], \sigma[N], \sigma[F]\}$;
where, the updated SS bit $\sigma'[j]$ for location j and can be generated as:

$$\sigma'[j] = ((Pflag[j] \& v[j]) | \sigma[j]); \forall j \in \{0, 1, 2\} \quad (4)$$

Thus instead of updating the state bits immediately after encoding each sample-location in the stripe-column, we update the state bits of all the sample-locations concurrently after encoding the stripe-column. The PSSB of the sample-locations takes into account the possible change in the state bits within the stripe-column.

3.3 An Example

To explain the algorithm further, we take a sample combination of σ (Fig. 5 (a)) bits for the stripe-column context window and v bits (Fig. 5 (b)) for the current stripe-column (locations 0-3 of Fig. 4). We assume that the current pass is SP pass to demonstrate the significant propagation as well. The $Pflags$ are generated using Eq. (1). The $Pflags$ for all locations 0-3 (Fig. 4) are 1 and thus all locations in the current stripe-column belong to the SP pass. The PSSBs are generated using the update values of σ as per Eq. (4) (Fig. 5 (c)). It is to be noted that locations 1 and 2 belong to the SP pass only due to the generation of significant propagation wave starting from location 0 (location 0 turns significant during this pass). Once the $Pflags$ and PSSBs for all locations are generated, the contexts are generated using the JPEG2000 context-lookup tables [3]. Since the locations 0-3 are turning insignificant, sign-contexts corresponding to the sign bits

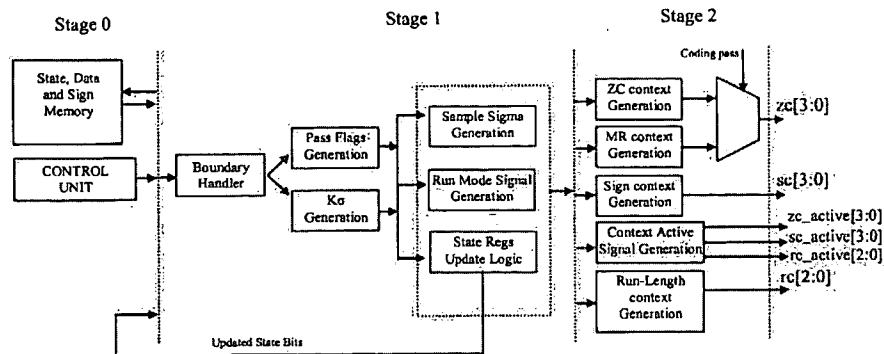


Fig. 6 Block diagram of proposed VLSI architecture for BPC module.

are also generated. In total there are 8 valid CxD pairs (4 CxD pairs for the v bits and 4 CxD pairs for the sign bits) for this stripe-column.

The proposed algorithm generates 11 output CxD pairs, $zc[3:0]$, $sc[3:0]$, and $rc[2:0]$, their corresponding active signal vectors $zc_active[3:0]$, $sc_active[3:0]$, and $rc_active[2:0]$; and the current coding pass. Each $zc[i]$, ($i \in \{0, 1, 2, 3\}$), corresponds to the CxD pair generated by the Zero coding (during SP and CP passes) or MR coding (during MR pass) primitive for the bit-location i of the current stripe-column. Each $sc[i]$ vector corresponds to the CxD pair generated by Sign coding primitive for bit-location i . The $rc[0]$ is the run-mode CxD pair generated by Run-length coding primitive. The remaining two CxD vectors, $rc[1]$ and $rc[2]$, are run-interrupt CxD vectors generated by the Run-length coding primitive. Depending on the coding pass and the significance state of the neighboring locations in the context-window, anywhere between 0 to 10 CxD vectors among the total 11 CxD vectors are valid. To identify the valid CxD vectors, the BPC module generates the corresponding 1-bit active signal vectors. The CxD vector is valid if its active signal is 1 else not. The context-active signals are calculated based on the current coding pass, the data bits of the stripe-column and the SS bits of stripe-column's context-window.

3.4 VLSI Architecture

The block diagram of the proposed architecture is shown in Fig. 6. We use a 3 stage pipelined architecture, operating at a single clock, to optimize the critical path. Stage 0 is the control unit, providing the interface to the main control unit of the JPEG2000 encoder system. It also controls the memory read and write addresses generation for data, sign and state bits and coding pass information.

Stage 1 is responsible for generating intermediate variables; it consists of the following blocks:

Boundary Handler: This block takes care of boundary conditions. For the first stripe-column in every stripe, the SS

bits of column 1 in the stripe-column's context-window should be taken to be 0. Similarly for the last stripe-column in every stripe, the SS bits in column 3 in the context-window are taken to be 0. The first and the last stripe in the code-block have similar boundary requirements for the top and the bottom row of the context-window, respectively. This block reads from the SS bit register and generates SS bits for columns 1 and 3, in accordance with these boundary conditions. The BPC module reads a new stripe-column every clock-cycle (assuming there is no stall generated from the AC module). This helps to avoid investing an extra clock-cycle at stripe boundaries and aids in creating smooth memory access patterns for data, sign and state bits. The other boundary condition corresponding to the case where the last stripe does not contain all 4 rows, is handled by assigning 1 to the CS state bits of the out-of-bound sample-locations.

Pflag Generation: This block generates the pass membership flags, using the logic Eqs.(1)–(3) given in Sect. 3.1. The Pflags are all generated in one clock-cycle.

KSig($K\sigma[j]$) Generation: This block generates $K\sigma[j]$, for all sample-locations in the stripe-column. It is used by the Run Mode Signal Generation block and MR Context Generation block.

Sample Sigma Generation: This block generates the PSSB for all sample-locations in the stripe-column.

State Regs Update Logic: This block updates the state bits of all sample-locations, depending on their pass membership.

Run Mode Signal Generation: This block generates run-mode and run-interrupt signal, using $K\sigma[j]$ and the data bits. The run-mode signal specifies whether the current stripe-column will enter run-mode or not. The run-interrupt signal signals run interruption in the current stripe-column.

Stage 2 is responsible for calculating the contexts, depending on the PSSB, MR, sign and data bits. It contains processing blocks for generating ZC contexts, MR contexts, sign contexts and run-length contexts. It also generates the context-active signal, depending on the current pass, run-mode signal, run-interrupt signal, pass membership flags,

and data bits to identify the active context-data pairs. A multiplexer is also required to select between the ZC and MR contexts, depending on the current pass.

4. Implementation Results

We used an Altera APEX20KE FPGA (Table 1) to implement the proposed architecture. To compare the performance of our architecture, we also implemented the sequential single sample [5] and sample-skipping [4] architectures on the same platform. Number of clock cycles required for coding a code-block, critical path delay and hardware cost (number of cells) are used as the basis for comparison. The state, data and sign memories are not included in the hardware cost analysis, since they are present in all the three architectures.

4.1 Critical Path Delay and Throughput Analysis

We use the following parameters for throughput analysis:

Size of code block = [64, 64].

Bit depth of wavelet coefficients = 16 (1 sign, 15 data bits).

Average # of empty columns (from [2]) = 23634.

In the single sample architecture [5] each location is checked three times for its membership once during each pass. The stripe-column based sample-skipping architecture [4] processes only those samples in a stripe-column which belong to the current pass, by calculating the pass membership of all 4 samples in a stripe-column concurrently. The proposed architecture consumes only single clock-cycle for each stripe-column during each pass. Table 2 presents the number of cycles/code-blocks for the above mentioned code-block parameters. As shown the sample-skipping architecture requires approximately 2 times more clock-cycles compared to the proposed architecture. Table 2 also presents the clock frequency, and the aggregate throughput values for the three architectures. The proposed architecture has a 34% improvement in the operating frequency. The main reason behind the critical path improvement is the absence of extra sequential logic for the multiplexers used to skip samples [4]. Another reason is the efficient use of pipelining. The results show that the pro-

posed BPC architecture has approximately 2.6 times higher throughput than existing architectures.

4.2 Hardware Cost Analysis

As shown in Table 2, the hardware cost for CSP of the proposed architecture is just 1.3 times that of the sample-skipping architecture. The extra cost is nominal due to two main reasons. Firstly, the additional hardware required for the concurrent generation of the CxD pairs are small combinational blocks. Secondly, the *control unit* which accounts for nearly 70% of the total hardware cost is similar in all the three architectures (as all employ stripe-column based processing).

4.3 Figure of Merit Analysis

In order to compare the overall efficiency of each architecture we define a Figure of Merit (FoM) as the ratio of throughput to hardware cost. The FoM and the relative values of FoM are presented in Table 2. As shown, the FoM for our proposed architecture is 2 and 2.5 times better than the sample-skipping and single sample architectures respectively.

5. Discussion

Up to this point we have ignored the impact of arithmetic coder on the overall throughput of embedded block coder. The overall throughput depends on the throughput of both the bit plane and arithmetic coder as well as the intermediate buffering used to couple them. Traditional arithmetic coder architectures can only accept at most one CxD pair per clock-cycle. It is possible, however, to realize CSP technique in the arithmetic coder [7] with some extra hardware cost. Recently a VLSI architecture of concurrent symbol processing capable AC module, which accepts 2 CxDs/clock-cycle, is reported in [8]. Our preliminary simulation results show that the proposed BPC architecture can fulfill the high input demands of a multiple symbol arithmetic coder resulting in nearly a 66% increase in overall throughput. However existing BPC architectures cannot realize such improvements, due to the absence of concurrent symbol processing. Additionally our architecture can also be augmented with speedup techniques like multiple column skipping [4] and parallel context modelling [6]. The concurrent symbol processing technique can also be scaled to be able to process multiple stripe-column concurrently. Our ongoing work addresses the impact of employing such speedup techniques, intermediate buffer length, and extra

Table 1 APEX20KE FPGA specifications.

FPGA	Apex EP20K600E
Typical Gates	600,000
Maximum System Gates	1,537,024
RAM Bits	311,296
Speed Grades	-3,-2,-1
Operating Voltage	1.8 V

Table 2 Performance and cost analysis.

Architecture	Area (# cells)	Cycles /code-block	Clk (MHz)	Throughput ($\times 10^6$ samples/sec)	FoM ($\times 10^3$ Throughput/Area)	Rel. FoM
Single Sample	631	156590	51.7	1.17	1.85	.039
sample-skipping	710	89170	38.6	1.77	2.4	0.50
Proposed	956	46080	51.7	4.59	4.8	1

hardware cost involved, in improving the overall throughput of the embedded block coder.

6. Conclusions

We have presented an algorithm and VLSI architecture for a BPC engine which processes a complete stripe-column during each pass concurrently. The architecture is efficiently designed to handle the boundary conditions and is pipelined to enhance the clock rate. The results show that the proposed architecture has 2.6 times higher throughput, with respect to the best existing architecture, with little extra hardware cost. The Figure of merit for our design is 2 times better than the best existing architecture. Also, the proposed architecture enables the use of multi-symbol arithmetic encoders to realize a high throughput embedded block encoder. Additionally, the proposed architecture takes care of the boundary conditions in an efficient way, which in turn results in a smooth memory access pattern.

References

- [1] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Trans. Image Process.*, vol. 9, pp.1158–1170, July 2000.
- [2] D.S. Taubman and M.W. Marcellin, *JPEG 2000: Image Compression, Fundamentals, Standards and Practice*. Norwell, MA, 2002.
- [3] *JPEG 2000 Part 1 Final Committee Draft Version 1.0*, ISO/IEC JTC1/SC29/WG1 N1646R, March 2000.
- [4] C.-J. Lian, K.-F. Chen, H.-H. Chen, and L.-G. Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol.13, pp.219–230, March 2003.
- [5] K. Andra, T. Acharya, and C. Chakrabarti, "Efficient VLSI implementation of bit plane coder of JPEG2000," *Proc. SPIE Int. Conf. Applications of Digital Image Processing XXIV*, vol.4472, pp.246–257, Dec. 2001.
- [6] Y. Li, R.E. Aly, B. Wilson, and M.A. Bayoumi, "Analysis and enhancements for EBCOT in high speed JPEG2000 architectures," *Midwest Symp. on Ckts. and Systems*, vol.2, pp.207–210, Aug. 2002.
- [7] D. Taubman, E. Ordentlich, M. Weinberger, G. Seroussi, I. Ueno, and F. Ono, "Embedded block coding in JPEG2000," *Proc. 2000 International Conference on Image Processing*, vol.2, pp.33–36, Sept. 2000.
- [8] M. Dyer, D. Taubman, and S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000," accepted in *IEEE Int. Conf. Image Processing*, pp.2817–2820, 2004.



Amit Kumar Gupta received his B.Tech in Electrical Engineering from Indian Institute of Technology, Kanpur, India in 2000. From 2000 to 2003, he worked as DSP engineer at Hughes Software Systems, India. Presently he is a PhD candidate in image processing and microelectronics in University of New South Wales, Australia. He received "Letter of Excellence" for his work at Hughes Network Systems, USA. He also won Student Paper Contest for financial support from ISCAS 2005 for his paper, "Optimal 2 sub-bank memory architecture for Bit Plane Coder of JPEG2000." His major research interests are in the area of image processing, VLSI architecture of image and video processing.



Saeid Nooshabadi received Ph.D. degrees in electrical engineering from the India Institute of Technology, Delhi, India, in 1992. Currently, he is a Senior Lecturer in microelectronics and digital systems design in the School of Electrical Engineering and Telecommunications, University of New South Wales, Sydney, Australia. Prior to his current appointment, he held academic positions at the Northern Territory University and the University of Tasmania, between 1993–2000. In 1992, he was a Research Scientist at the CAD Laboratory, Indian Institute of Science, Bangalore, India, working on the design of VLSI chips for TV ghost cancellation in digital TV. In 1996 and 1997, he was a Visiting Faculty and Researcher, at the Centre for Very High Speed Microelectronic Systems, Edith Cowan University, Western Australia, working on high performance GaAs circuits; and Curtin University of Technology, Western Australia, working on the design of high speed-high frequency modems. His research interests include very high-speed integrated circuit and application-specified integrated circuit design for high-speed telecommunication and image processing systems, low-power design of circuits and systems, and low-power embedded systems.



David Taubman received B.S. and B.E. (Electrical) degrees in 1986 and 1988 from the University of Sydney, and M.S. and Ph.D. degrees in 1992 and 1994 from the University of California at Berkeley. From 1994 to 1998 he worked at Hewlett-Packard's Research Laboratories in Palo Alto, California, joining the University of New South Wales in 1998, where he is an Associate Professor in the School of Electrical Engineering and Telecommunications. Dr. Taubman is author with M. Marcellin of the book, "JPEG2000: Image compression fundamentals, standards and practice." His research interests include highly scalable image and video compression, inverse problems in imaging, perceptual modeling, joint source/channel coding, and multimedia distribution systems. Dr. Taubman was awarded the University Medal from the University of Sydney, the Institute of Engineers, Australia, Prize and the Texas Instruments Prize for Digital Signal Processing, all in 1998. He has received two Best Paper awards: from the IEEE Circuits and Systems Society for the 1996 paper, "A Common Framework for Rate and Distortion Based Scaling of Highly Scalable Compressed Video;" and from the IEEE Signal Processing Society for the 2000 paper, "High Performance Scalable Image Compression with EBCOT."

CLAIMS

In view of the foregoing, what we claim is:

Digital Data Processor with Cache-Managed Memory

1. A digital data processor or processing system comprising
 - A. one or more nodes that are communicatively coupled to one another,
 - B. one or more memory elements ("physical memory") communicatively coupled to at least one of the nodes,
 - C. at least one of the nodes includes a cache memory that stores at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - D. wherein the cache memory additionally stores tags specifying addresses for respective data or instructions in the physical memory.
2. The digital data processor or processing system of claim 1, comprising system memory that includes the physical memory and cache memory.
3. The digital data processor or processing system of claim 2, wherein the system memory comprises the cache memory of multiple nodes.
4. The digital data processor or processing system of claim 3, wherein the tags stored in the cache memory specify addresses for respective data or instructions in system memory.
5. The digital data processor or processing system of claim 3, wherein the tags specify one or more statuses for the respective data or instructions.
6. The digital data processor or processing system of claim 5, where those statuses include any of a modified status and a reference count status.

7. The digital data processor or processing system of claim 1, wherein the cache memory comprises multiple hierarchical levels.
8. The digital data processor or processing system of claim 7, wherein the multiple hierarchical levels include at least one of a level 1 cache, a level 2 cache and a level 2 extended cache.
9. The digital data processor or processing system of claim 1, wherein the addresses specified by the tags form part of a system address space that is common to multiple ones of the nodes.
10. The digital data processor or processing system of claim 9, wherein the addresses specified by the tags form part of a system address space that is common to all of the nodes.
11. A digital data processor or processing system comprising
 - A. one or more nodes that are communicatively coupled to one another, at least one of which nodes a processing module,
 - B. one or more memory elements ("physical memory") communicatively coupled to at least one of the nodes,
 - C. at least one of the nodes includes a cache memory that stores at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - D. wherein at least the cache memory stores tags ("extension tags") specifies a system address and a physical address for each of at least one datum or instruction that is stored in physical memory.
12. The digital data processor or processing system of claim 11, comprising system memory that includes the physical memory and cache memory.

13. The digital data processor or processing system of claim 12, comprising system memory that includes the physical memory and cache memory of multiple nodes.
14. The digital data processor or processing system of claim 12, wherein a said system address specified by the extension tags form part of a system address space that is common to multiple ones of the nodes.
15. The digital data processor or processing system of claim 14, wherein a said system address specified by the extension tags form part of a system address space that is common to all of the nodes.
16. The digital data processor or processing system of claim 3, wherein the tags specify one or more statuses for a said respective data or instruction.
17. The digital data processor or processing system of claim 16, where those statuses include any of a modified status and a reference count status.
18. The digital data processor or processing system of claim 11, wherein at least one said node comprises address translation that utilizes a said system address and a said physical address specified by a said extension tag to translate a system addresses to a physical addresses.
19. A digital data processor or processing system comprising
 - A. one or more nodes that are communicatively coupled to one another, at least one of which nodes a processing module,
 - B. one or more memory elements ("physical memory") communicatively coupled to at least one of the nodes, where one or more of those memory elements includes any of flash memory or other mounted drive,

- C. at least one of the nodes includes a cache memory that stores at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - D. the physical memory and cache memory of the nodes together comprising system memory,
 - E. the cache memory of each node storing at least one of data and instructions any of accessed and expected to be accessed by the respective node and, additionally, storing tags specifying addresses for at least one respective datum or instructions in physical memory, wherein at least one of those tags ("extension tag") a system address and a physical address for each of at least one datum or instruction that is stored in physical memory.
20. The digital data processor or processing system of claim 19, in which multiple said extension tags are organized as a tree in system memory.
21. The digital data processor or processing system of claim in which one or more of the extension tags are cached in the cache memory system of one or more nodes.
22. The digital data processor or processing system of claim 21, wherein the one or more of the extension tags that are cached in the cache memory system of one or more nodes are extension tags for any of data and instructions recently accessed or expected to be accessed by the respective node.
23. A digital data processor or processing system comprising
- A. a plurality of nodes that are communicatively coupled to one another by a bus, network or other media (collectively, "interconnect"),
 - B. one or more memory elements ("physical memory") communicatively coupled to at least one of the nodes, where one or more of those memory elements includes any of flash memory or other mounted drive,

- C. at least one of the nodes includes a cache memory that stores at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - D. the physical memory and cache memory of the nodes together comprising system memory,
 - E. the cache memory of each node storing at least one of data and instructions any of accessed and expected to be accessed by the respective node and, additionally, storing tags specifying addresses for at least one respective datum or instructions in physical memory, wherein at least one of those tags ("extension tag") a system address and a physical address for each of at least one datum or instruction that is stored in physical memory.
24. The digital data processor or processing system of claim 23, wherein the interconnect comprises a ring interconnect.
25. The digital data processor or processing system of claim 24, wherein the ring interconnect is a rotating shift register.
26. The digital data processor or processing system of claim 23, wherein a said node signals a request for any of a datum and instruction along that bus, network or other media following a cache miss within its own cache memory.
27. The digital data processor or processing system of claim 26, wherein said request may be satisfied from physical memory, if not from the cache memory of one of the other nodes.
28. The digital data processor or processing system of claim 26, wherein a said node utilize the bus, network or other media to communicate to update any of data and instructions cached in any of the other nodes or other system memory.
28. The digital data processor or processing system of claim 26, wherein a said node utilize the bus, network or other media to communicate to update extension tags cached in any of the other nodes or other system memory.

29. A method of digital data processing comprising
- A. providing one or more nodes that are communicatively coupled to one another, at least one of which nodes includes a processing module,
 - B. communicatively coupling one or more memory elements ("physical memory") to at least one of the nodes,
 - C. storing in a cache memory of at least one of the nodes at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - D. wherein the storing step additionally includes storing in the cache memory tags specifying addresses for respective data or instructions in the physical memory.
30. The method of claim 29, wherein the storing step includes storing in the cache memory tags specifying addresses that form part of a system address space that is common to multiple ones of the nodes.
31. The method of claim 30, comprising organizing and accessing the cache memory hierarchically.
32. A method of digital data processing, comprising
- A. providing one or more nodes that are communicatively coupled to one another,
 - B. communicatively coupling one or more memory elements ("physical memory") to at least one of the nodes,
 - C. storing in a cache memory of at least one of the nodes at least one of data and instructions any of accessed and expected to be accessed by the respective node,

- D. wherein the storing step includes storing in the cache memory tags ("extension tags") specifying a system address and a physical address for each of at least one datum or instruction that is stored in physical memory.

General Purpose Processor With Dynamic Assignment of Events to Threads

- 33. A digital data processor or processing system comprising,
 - A. one or more processing units that each execute processes or threads (collectively, "threads"),
 - B. an event table that is coupled to the plurality of processing units and that maps events thereto,
 - C. one or more of hardware and software that is communicatively coupled to logic executing on the system and that registers with that logic any of event-processing needs and/or capabilities of that hardware or software, and
 - D. wherein that logic updates the event table based on matching those registered needs and capabilities to one another and/or to those of components of the system.
- 34. The digital data processor or processing system of claim 33, wherein a default system thread executing on one of the processing units provides said logic.
- 35. The digital data processor or processing system of claim 34, wherein said logic matches event processing needs and capabilities to reflect an optimal mapping based on the requirements and capabilities of the overall system.
- 36. The digital data processor or processing system of claim 35, wherein said logic updates the event table to reflect that optimal mapping.

37. The digital data processor or processing system of claim 33, comprising a preprocessor that inserts event table management code into software that will be executed by the system.
38. The digital data processor or processing system of claim 37, wherein, upon execution, the event table management code causes software into which it is inserted to register its event-processing needs capabilities at runtime.
39. The digital data processor or processing system of claim 37, wherein the code is based on directives supplied by a developer or other.
40. The digital data processor or processing system of claim 39, wherein the directives reflect actual or expected requirements of the software into which it is inserted.
41. The digital data processor or processing system of claim 39, wherein the directives reflect an expected runtime environment.
42. The digital data processor or processing system of claim 39, wherein the directives reflect expected devices or software available within the environment capabilities or requirements matching that of the software into which that code is inserted.
43. The digital data processor or processing system of claim 33, wherein said logic comprises a library or other intermediate, object or other code module.
44. A method of digital data processing, comprising,
 - A. providing one or more processing units that each execute processes or threads (collectively, “threads”),
 - B. utilizing an event table to map events to those threads,

- C. registering event-processing needs and/or capabilities of any of hardware or software communicatively coupled to the system,
 - D. matching those needs and/or capabilities to one another and/or to those of other those of components of the system, and
 - E. updating the event table based on results of that matching step.
45. The method of claim 44, wherein the registering step comprises registering the event-processing needs and/or capabilities with a default system thread executing on the system.
46. The method of claim 45, wherein the matching step includes matching event processing needs and capabilities to reflect an optimal mapping based on the requirements and capabilities of the overall system.
47. The method of claim 44, comprising utilizing a preprocessor to insert event table management code into software that will be executed by the system at runtime.
48. The method of claim 47, comprising executing the event table management code at runtime to cause software into which it is inserted to register its event-processing needs capabilities.
49. The method of claim 47, comprising basing the code on directives supplied by a developer or other.
50. The method of claim 49, wherein the directives reflect actual or expected requirements of the software into which it is inserted.
51. The method of claim 49, wherein the directives reflect an expected runtime environment.

52. The method of claim 49, wherein the directives reflect expected devices or software available within the environment capabilities or requirements matching that of the software into which that code is inserted.

General Purpose Processor With Location-Independent Shared Execution Environment

53. A method of digital data processing, comprising
- A. providing, in each of one or more devices that are coupled for communication, at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. receiving, with the processing unit of a first one of the devices, notification of a first event,
 - C. instantiating, in the processing unit that first device, a first thread to handle that first and subsequent related events,
 - D. migrating from memory associated with the processing unit of another of the devices to to the processing unit of the first device at least one instruction of an instruction sequence for handling that event,
 - E. executing the migrated instruction as part of the first thread.
54. The method of claim 21, comprising repeating steps (D)–(E) until processing of the event is completed.
55. The method of claim 21, comprising repeating steps (D)–(E) until processing of the event until the first thread enters into a wait state.
57. The method of claim 21, comprising migrating the instantiated thread to the processing unit of a second device for execution by it.

58. The method of claim 47, comprising receiving notification of a related event with the prosecution unit of the second device.
59. The method of claim 58, instantiating, in the processing unit that second device, a second thread to handle the related events.
60. The method of claim 60, migrating from memory associated with the processing unit of first device to the processing unit of the second device at least one instruction of the instruction sequence for handling the first event and subsequent related events.
61. The method of claim 60, executing the migrated instruction as part of the second thread.
62. The method of claim 21, comprising repeating the following steps until processing of the second event is complete:
- A. migrating from memory associated with the processing unit of first device to the processing unit of the second device at least one instruction of the instruction sequence for handling the first event and subsequent related events.
 - B. executing the migrated instruction as part of the second thread.
63. The method of claim 21, comprising repeating the following steps until the second thread enters into a wait state:
- A. migrating from memory associated with the processing unit of first device to the processing unit of the second device at least one instruction of the instruction sequence for handling the first event and subsequent related events.
 - B. executing the migrated instruction as part of the second thread.
64. A digital data processor or processing system comprising,

- A. one or more devices that are coupled for communication, each of which has at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. the processing unit of at least a first one of the devices being communicatively coupled to an event table that maps events to threads executing on the processing unit of a second one of the devices, and
 - C. the first device being responsive to receipt of an event that maps to a thread executing on the processing unit of the second device by routing the event to that device for processing thereby.
65. The digital data processor or processing system of claim 64, wherein the event table maps events to threads executing on the processing unit of the first devices.
66. The digital data processor or processing system of claim 65, wherein the first device is responsive to receipt of an event that maps to a thread executing on the processing unit of the first device by routing the event that processing unit.
67. The digital data processor or processing system of claim 66, wherein the event table responds to a request to match an event by returning at least one of a thread id and an address of a processing unit responsible for processing for processing that event.
68. The digital data processor or processing system of claim 64, wherein the event table maps events to threads executing on processing units of one or more devices in a same zone as the first device.
69. The digital data processor or processing system of claim 68, wherein the zone comprises a local area network.
70. The digital data processor or processing system of claim 64, wherein the devices comprise any of televisions, set top boxes, cell phones, personal digital assistants and remote controls.

General Purpose Processor With Provision of Quality of Service Through Thread Instantiation, Maintenance and Optimization

71. A digital data processor or processing system comprising,
- A. one or more devices that are coupled for communication, each of which has at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. an event delivery mechanism delivers interrupts and other events to the processing units, and
 - C. logic executing in one or more of the processing units that, at runtime, optimizes at least one of thread instantiation, maintenance and thread assignment.
72. The digital data processor or processing system of claim 71, where the aforesaid logic optimizes at least one of thread instantiation, maintenance and thread assignment to meet quality of service requirements of individual threads, classes of threads, individual events, and/or classes of events.
73. The digital data processor or processing system of claim 72, wherein those quality of service requirements include one or more of
- data processing requirements of voice processing events, applications and/or threads,
 - data throughput requirements of web data transmission events, applications and/or threads,
 - data processing and display requirements of gaming events, applications and/or threads,
 - data processing and display requirements of telepresence events, applications and/or threads,

- decoding, scaler & noise reduction, color correction, frame rate control and other processing and display requirements of audiovisual (e.g., television or video) events, applications and/or threads,
 - energy utilization requirements of the system 75, as well as of events, applications and/or threads processed thereon, and/or
 - processing of actual or expected numbers of simultaneous events by individual threads, classes of threads, individual events and/or classes of events
 - prioritization of the processing of threads, classes of threads, events and/or classes of events over other threads, classes of threads, events and/or classes of events
74. The digital data processor or processing system of claim 71, wherein the logic optimizes at least one of thread instantiation, maintenance and thread assignment by invoking one or more event-handling threads in advance of demand for them in the normal course.
75. The digital data processor or processing system of claim 74, wherein the one or more event-handling threads invoked in advance of demand are available to service ready to service user action, software and/or hardware interrupts when that demand does arise.
76. The digital data processor or processing system of claim 71, wherein the logic optimizes at least one of thread instantiation, maintenance and thread assignment by instantiating multiple instances of a thread and mapping each to different respective upcoming events that are expected occur in the future.
77. The digital data processor or processing system of claim 76, wherein the multiple instantiated threads insure more immediate servicing of the upcoming events when they do occur.
78. The digital data processor or processing system of claim 76, wherein different respective upcoming events are events that typically occur in batches.

79. The digital data processor or processing system of claim 71, wherein the logic optimizes at least one of thread instantiation, maintenance and thread assignment by periodically, sporadically, episodically, randomly or otherwise generating interrupts to prevent one or more threads from going inactive.
80. The digital data processor or processing system of claim 79, wherein the logic generates those interrupts to prevent those one or more threads from going inactive even after apparent termination of their normal processing.
81. The digital data processor or processing system of claim 80, wherein the logic generates those interrupts to insure more immediate servicing of the upcoming events by those one or more threads.
82. A method of digital data processing, comprising
- A. providing, in each of one or more devices that are coupled for communication, at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. delivering interrupts and other events to the processing units,
 - C. executing thread management code in one or more of the processing units that optimizes at least one of thread instantiation, maintenance and thread assignment.
83. The method of claim 82, wherein step (C) includes inserting the thread management code into software that will be executed by the one or more processing units.
84. The method of claim 83, inserting that thread management code into that software during any of pre-processing, compiler/linker or loading.
85. The method of claim 82, where the thread management code optimizes at least one of thread instantiation, maintenance and thread assignment to meet quality of service

requirements of individual threads, classes of threads, individual events, and/or classes of events.

86. The method of claim 85, wherein those quality of service requirements include one or more of

- data processing requirements of voice processing events, applications and/or threads,
- data throughput requirements of web data transmission events, applications and/or threads,
- data processing and display requirements of gaming events, applications and/or threads,
- data processing and display requirements of telepresence events, applications and/or threads,
- decoding, scaler & noise reduction, color correction, frame rate control and other processing and display requirements of audiovisual (e.g., television or video) events, applications and/or threads,
- energy utilization requirements of the system 75, as well as of events, applications and/or threads processed thereon, and/or
- processing of actual or expected numbers of simultaneous events by individual threads, classes of threads, individual events and/or classes of events
- prioritization of the processing of threads, classes of threads, events and/or classes of events over other threads, classes of threads, events and/or classes of events.

87. The method of claim 82, wherein the thread management logic optimizes at least one of thread instantiation, maintenance and thread assignment by invoking one or more event-handling threads in advance of demand for them in the normal course.
88. The method of claim 87, wherein the one or more event-handling threads invoked in advance of demand are available to service ready to service user action, software and/or hardware interrupts when that demand does arise.
89. The method of claim 84, wherein the thread management logic optimizes at least one of thread instantiation, maintenance and thread assignment by instantiating multiple instances of a thread and mapping each to different respective upcoming events that are expected occur in the future.
90. The method of claim 89, wherein the multiple instantiated threads insure more immediate servicing of the upcoming events when they do occur.
91. The method of claim 89, wherein different respective upcoming events are events that typically occur in batches.
92. The method of claim 84, wherein the thread management logic optimizes at least one of thread instantiation, maintenance and thread assignment by periodically, sporadically, episodically, randomly or otherwise generating interrupts to prevent one or more threads from going inactive.
93. The method of claim 92, wherein the thread management logic generates those interrupts to prevent those one or more threads from going inactive even after apparent termination of their normal processing.
94. The method of claim 93, wherein the thread management logic generates those interrupts to insure more immediate servicing of the upcoming events by those one or more threads.

General Purpose Processor with JPEG2000 Bit Plane Stripe Column Encoding

95. A digital data processor comprising
 - A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a selected processor-level instruction by encoding and storing to one or more of the register(s) a stripe column for bit plane coding within JPEG2094 Embedded Block Coding with Optimized Truncation (EBCOT).
96. The digital data processor of claim 95, in which the execution unit generates the encoded stripe column based on specified bits of a column to be encoded and on bits adjacent thereto.
97. The digital data processor claim 96, wherein at least one of the bits of a column to be encoded and on bits adjacent thereto are specified as parameters of the processor-level instruction.
98. The digital data processor of claim 95, in which the execution unit generates the encoded stripe column from four bits of the column to be encoded and on the bits adjacent thereto.
99. The digital data processor of claim 95, in which the execution unit generates the encoded stripe column in response to an instruction that specifies, in addition to the bits of the column to be encoded and adjacent thereto, a current coding state of at least one of the bits to be encoded.
100. The digital data processor of claim 99, in which the coding state of each bit to be encoded is represented in three bits.

101. The digital data processor of claim 95, in which the execution unit generates the encoded stripe column in response to execution of an instruction that specifies an encoding pass that includes any of a significance propagation pass (SP), a magnitude refinement pass (MR), a cleanup pass, and a combined MR and CP pass.
102. The digital data processor of claim 101, in which the execution unit selectively generates and stores to one or more registers an updated coding state of at least one of the bits to be encoded.
103. A method of digital data processing comprising
 - A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, on the execution unit, a selected processor-level instruction by encoding and storing to one or more of the register(s) a stripe column for bit plane coding within JPEG2102 Embedded Block Coding with Optimized Truncation (EBCOT).
104. The method of claim 103, in which the executing step includes generating the encoded stripe column based on specified bits of a column to be encoded and on bits adjacent thereto.
105. The digital data processor claim 104, wherein at least one of the bits of a column to be encoded and on bits adjacent thereto are specified as parameters of the processor-level instruction.
106. The method of claim 103, in which the executing step includes generating the encoded stripe column from four bits of the column to be encoded and on the bits adjacent thereto.

107. The method of claim 103, in which the executing step includes generating the encoded stripe column in response to an instruction that specifies, in addition to the bits of the column to be encoded and adjacent thereto, a current coding state of at least one of the bits to be encoded.
108. The method of claim 107, in which the coding state of each bit to be encoded is represented in three bits.
109. The method of claim 103, in which the executing step includes generating the encoded stripe column in response to execution of an instruction that specifies an encoding pass that includes any of a significance propagation pass (SP), a magnitude refinement pass (MR), a cleanup pass, and a combined MR and CP pass.
110. The method of claim 109, in which the executing step includes selectively generating and storing to one or more registers an updated coding state of at least one of the bits to be encoded.

General Purpose Processor with JPEG2000 Binary Arithmetic Code Lookup

111. A digital data processor comprising
 - A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a selected processor-level instruction by encoding and storing to one or more of the register(s) by storing to those register(s) one or more values from a JPEG2000 binary arithmetic coder lookup table.
112. The digital data processor of claim 111, wherein the JPEG2000 binary arithmetic coder lookup table is a Qe-value and probability estimation lookup table.

113. The digital data processor of claim 111, in which the execution unit responds to such a selected processor-level instruction by storing to said one or more registers one or more function values from such a lookup table.
114. The digital data processor of claim 113, wherein the function value is a Q_e -value function value.
115. The digital data processor of claim 113, wherein the function value is NMPS function value.
116. The digital data processor of claim 113, wherein the function value is an NLPS function value.
117. The digital data processor of claim 113, wherein the function value is a SWITCH function value.
118. The digital data processor of claim 111, in which the execution unit stores said one or more values to said one or more registers as part of a JPEG2000 decode or encode instruction sequence.
119. The digital data processor of claim 111, in which the execution unit generates the one or more values from any of a hardcoded table, a table contained in the registers, and/or algorithmic approximation of the table.
120. A method of digital data processing comprising
 - A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,

- C. executing, on the execution unit, a selected processor-level instruction by encoding and storing to one or more of the register(s) by storing to those register(s) one or more values from a JPEG2009 binary arithmetic coder lookup table.
121. The method of claim 120, wherein the JPEG2009 binary arithmetic coder lookup table is a Qe-value and probability estimation lookup table.
122. The method of claim 120, in which the executing step includes responding to such a selected processor-level instruction by storing to said one or more registers one or more function values from such a lookup table.
123. The method of claim 122, wherein the function value is a Qe-value function value.
124. The method of claim 122, wherein the function value is NMPS function value.
125. The method of claim 122, wherein the function value is an NLPS function value.
126. The method of claim 122, wherein the function value is a SWITCH function value.
127. The method of claim 120, in which the executing step includes storing said one or more values to said one or more registers as part of a JPEG2009 decode or encode instruction sequence.
128. The method of claim 120, in which the which the executing step includes generating the one or more values from any of a hardcoded table, a table contained in the registers, and/or algorithmic approximation of the table.

General Purpose Processor with Arithmetic Operation Transpose Parameter

129. A digital data processor comprising

- A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a processor-level instruction that specifies a selected arithmetic operation and that specifies that operation is to be performed with a transpose by performing the specified arithmetic operation on one or more specified operands and by storing to one or more of the registers a result of that operation in transposed format.
130. The digital data processor of claim 129, in which the specified operands are registers and in which the execution unit stores the result of the operation across multiple registers.
131. The digital data processor of claim 130, in which the specified operands are logically equivalent of matrix rows and in which the execution unit stores the result of the operation in a logical equivalent to a matrix column.
132. The digital data processor of claim 129, in which the execution unit writes the result of the operation any of (i) as a one-quarter word column of four adjacent registers, (ii) a byte column of eight adjacent registers, all by way of example.
133. The digital data processor of claim 129, in which the execution unit breaks the result of the operation into separate portions and puts them into separate registers at a specific common byte, bit or other location.
134. The digital data processor of claim 129, in which the operation is any of an addition or subtraction operation.
135. A digital data processor comprising
- A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,

- C. the execution unit executing a processor-level instruction that specifies a selected arithmetic operation by performing the specified arithmetic operation on one or more specified operands and by storing a result thereof in any of a non-transposed and a transposed format, depending on a setting of a transpose parameter of the instruction.
136. The digital data processor of claim 135, in which the specified operands are registers and in which the execution unit stores the result of the operation within a register, if the transpose parameter is not set and across multiple registers, if the transpose parameter is set.
137. The digital data processor of claim 136, in which the specified operands are logically equivalent of matrix rows and in which the execution unit stores the result of the operation in a logical of equivalent of a matrix row, if the transpose parameter is not set, and in a logical equivalent to a matrix column, if the transpose parameter is set.
138. A method of digital data processor comprising
- A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, with the execution unit, a processor-level instruction that specifies a selected arithmetic operation and that specifies that operation is to be performed with a transpose by performing the specified arithmetic operation on one or more specified operands and by storing to one or more of the registers a result of that operation in transposed format.
139. The method of claim 138, in which the specified operands are registers and in which the execution unit stores the result of the operation across multiple registers.

140. The method of claim 139, in which the specified operands are logically equivalent of matrix rows and in which the execution unit stores the result of the operation in a logical equivalent to a matrix column.
141. The method of claim 138, in which the executing step includes writing the result of the operation any of (i) as a one-quarter word column of four adjacent registers, (ii) a byte column of eight adjacent registers, all by way of example.
142. The method of claim 138, in which the executing step includes breaking the result of the operation into separate portions and storing them into separate registers at a specific common byte, bit or other location.
143. The method of claim 138, in which the operation is any of an addition or subtraction operation.
144. A method of digital data processor comprising
- A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, with an execution unit, a processor-level instruction that specifies a selected arithmetic operation by performing the specified arithmetic operation on one or more specified operands and by storing a result thereof in any of a non-transposed and a transposed format, depending on a setting of a transpose parameter of the instruction.
145. The method of claim 144, in which the specified operands are registers and in which the executing step includes storing the result of the operation within a register, if the transpose parameter is not set and across multiple registers, if the transpose parameter is set.

146. The method of claim 145, in which the specified operands are logically equivalent of matrix rows and in which the executing step includes storing the result of the operation in a logical of equivalent of a matrix row, if the transpose parameter is not set, and in a logical equivalent to a matrix column, if the transpose parameter is set.

General Purpose Processor with Cache Control Instruction Set and Cache-Initiated Optimization

147. A digital data processor, comprising
- A. a cache subsystem that includes cache memory,
 - B. one or more registers,
 - C. an execution unit that is in communications coupling with the one or more registers and with the cache subsystem that at executes memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers,
 - D. the cache subsystem varying utilization of the cache memory in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.
148. The digital data processor of claim 147, wherein cache subsystem varies replacement and modified block writeback selectively in response to memory reference instructions executed by the execution unit.
149. The digital data processor of claim 147, wherein the cache subsystem selectively varies a value of a reference count that is associated with cached data in response to such memory reference instructions.
150. The digital data processor of claim 149, wherein the cache subsystem forces the reference count value to a low value in response to selected memory reference instructions.

151. The digital data processor of claim 150, wherein a low value accelerates replacement of the cached data with which it is associated.
152. The digital data processor of claim 147, wherein the memory reference instructions include any of LOAD (Load Register), STORE (Store to Memory), LOADPAIR (Load Register Pair), STOREPAIR (Store Pair to Memory), PREFETCH (Prefetch Memory), LOADPRED (Load Predicate Register), STOREPRED (Store Predicate Register), EMPTY (Empty Memory), and FILL (Fill Memory) instructions.
153. A digital data processor, comprising
 - A. a cache subsystem that includes cache memory,
 - B. one or more registers,
 - C. an execution unit that is in communications coupling with the one or more registers and with the cache subsystem that at executes memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers,
 - D. the cache subsystem selectively varies a value of a reference count that is associated with cached data in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.
154. The digital data processor of claim 153, wherein the cache subsystem preferentially replaces with new data old data that is stored in the cache memory and that is associated with lower reference count values.
155. The digital data processor of claim 154, wherein the cache subsystem forces the reference count value to a low value in response to selected memory reference instructions.

156. The digital data processor of claim 154, wherein the cache subsystem forces the reference count value to a low value in response to memory reference instructions that include a no-reuse hint.
157. A method of digital data processing, comprising
- A. executing, in an execution unit that is in communications coupling with one or more registers and with a cache subsystem that includes a cache memory, memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers, and
- B. varying, with the cache subsystem, utilization of the cache memory in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.
158. The method of 157, wherein the varying step includes varying, with the cache subsystem, replacement and modified block writeback selectively in response to memory reference instructions executed by the execution unit.
159. The method of 157, wherein the varying step includes varying, with the cache subsystem, a value of a reference count that is associated with cached data in response to such memory reference instructions.
160. The method of 159, wherein the varying step includes forcing, with the cache subsystem, the reference count value to a low value in response to selected memory reference instructions.
161. The method of 160, wherein a low value accelerates replacement of the cached data with which it is associated.
162. The method of 157, wherein the memory reference instructions include any of LOAD (Load Register), STORE (Store to Memory), LOADPAIR (Load Register Pair),

STOREPAIR (Store Pair to Memory), PREFETCH (Prefetch Memory), LOADPRED (Load Predicate Register), STOREPRED (Store Predicate Register), EMPTY (Empty Memory), and FILL (Fill Memory) instructions.

163. A method of digital data processing, comprising
 - A. executing, with an execution unit that is in communications coupling with one or more registers and with a cache subsystem, memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers,
 - B. selectively varying, with the cache subsystem, a value of a reference count that is associated with cached data in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.
164. The method of 163, comprising preferentially replacing, with the cache subsystem, in the cache memory new data old data that is stored in the cache memory and that is associated with lower reference count values.
165. The method of 164, wherein the varying step includes forcing, with the cache subsystem, the reference count value to a low value in response to selected memory reference instructions.
166. The method of 164, wherein the varying step includes varying, with the cache subsystem, forcing the reference count value to a low value in response to memory reference instructions that include a no-reuse hint.

General Purpose Processor and Digital Data Processing System Executing a Pipeline of Software Components That Replace a Like Pipeline of Hardware Components

167. A digital data processor or processing system comprising,

- A. one or more devices that are coupled for communication, each of which has at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. the processing units executing a plurality of threads that, together, define a pipeline of software components,
 - C. where the processing units execute that pipeline of software components to perform a same function as but in lieu of like pipelines of hardware components.
168. The digital data processor or processing system of claim 167, wherein one or more threads defining one of said software components operates on a different processing unit than one or more threads defining another respective software component.
169. The digital data processor or processing system of claim 167 for video processing, comprising
- A. a software component defining an H.430 decoder module executing one one or more of the processing units,
 - B. a software component defining scalar and noise reduction module executing one one or more of the processing units,
 - B. a software component defining a color correction module executing one one or more of the processing units,
 - B. a software component defining a frame rate control software module executing one one or more of the processing units.
170. The digital data processor or processing system of claim 169 that performs a same function as a hardware pipeline that includes a semiconductor chip that functions as a system controller with H.430 decoding, pipelined to a semiconductor chip that functions as a scaler and noise reduction module, pipelined to a semiconductor chip that functions

for color correction, and further pipelined to a semiconductor chip that functions as a frame rate controller.

171. The digital data processor or processing system of claim 167, wherein the processing units execute the pipelined software components as separate respective threads.
172. The digital data processor or processing system of claim 167, wherein
 - A. at least one of the devices includes a cache memory that stores at least one of data and instructions any of accessed and expected to be accessed by the respective processing unit,
 - B. wherein the cache memory additionally stores tags specifying addresses for respective data or instructions in a physical memory that is coupled to the digital data processor or processing system.
173. The digital data processor or processing system of claim 167, wherein at least one of the processing units comprises
 - A. an event table that is coupled to the plurality of processing units and that maps events thereto,
 - B. one or more of hardware and software that is communicatively coupled to logic executing on the system and that registers with that logic any of event-processing needs and/or capabilities of that hardware or software, and
 - C. wherein that logic updates the event table based on matching those registered needs and capabilities to one another and/or to those of components of the system.
174. The digital data processor or processing system of claim 167, wherein

- A. the processing unit of at least a first one of the devices being communicatively coupled to an event table that maps events to threads executing on the processing unit of a second one of the devices, and
 - B. the first device being responsive to receipt of an event that maps to a thread executing on the processing unit of the second device by routing the event to that device for processing thereby.
175. The digital data processor or processing system of claim 167, wherein at least one of the processing units comprises
- A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a processor-level instruction that specifies a selected arithmetic operation and that specifies that operation is to be performed with a transpose by performing the specified arithmetic operation on one or more specified operands and by storing to one or more of the registers a result of that operation in transposed format.
176. The digital data processor or processing system of claim 167, wherein at least one of the processing units comprises
- A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a selected processor-level instruction by encoding and storing to one or more of the register(s) by storing to those register(s) one or more values from a JPEG2166 binary arithmetic coder lookup table.

177. The digital data processor or processing system of claim 167, wherein at least one of the processing units comprises
- A. one or more registers,
 - B. an execution unit that is in communications coupling with the one or more registers,
 - C. the execution unit executing a selected processor-level instruction by encoding and storing to one or more of the register(s) a stripe column for bit plane coding within JPEG2260 Embedded Block Coding with Optimized Truncation (EBCOT).
178. The digital data processor or processing system of claim 167, wherein one or more of the processing units comprise
- A. an event delivery mechanism delivers interrupts and other events to the processing units, and
 - B. logic executing in one or more of the processing units that, at runtime, optimizes at least one of thread instantiation, maintenance and thread assignment.
179. The digital data processor or processing system of claim 167, wherein at least one of the processing units comprises:
- A. a cache subsystem that includes cache memory,
 - B. one or more registers,
 - C. an execution unit that is in communications coupling with the one or more registers and with the cache subsystem that at executes memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers,

- D. the cache subsystem varying utilization of the cache memory in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.
179. A method of digital data processing system comprising,
- A. providing one or more devices that are coupled for communication, each of which has at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. executing on the processing units a plurality of threads that, together, define a pipeline of software components,
 - C. where the processing units execute that pipeline of software components to perform a same function as but in lieu of like pipelines of hardware components.
180. The method of claim 179, wherein one or more threads defining one of said software components operates on a different processing unit than one or more threads defining another respective software component.
182. The method of claim 179, comprising executing the pipelined software components as separate respective threads on the processing units.
183. The method of claim 179, comprising
- A. storing in a cache memory of at least one of the nodes at least one of data and instructions any of accessed and expected to be accessed by the respective node,
 - B. wherein the storing step additionally includes storing in the cache memory tags specifying addresses for respective data or instructions in the physical memory.
184. The method of claim 179, comprising executing on at least one of the processing units the steps of:

- A. utilizing an event table to map events to said threads,
 - B. registering event-processing needs and/or capabilities of any of hardware or software communicatively coupled to the system,
 - C. matching those needs and/or capabilities to one another and/or to those of other those of components of the system, and
 - D. updating the event table based on results of that matching step.
185. The method of claim 179, comprising executing on at least one of the processing units the steps of:
- A. providing, in each of one or more devices that are coupled for communication, at least one processing unit that executes processes or threads (collectively, “threads”),
 - B. receiving, with the processing unit of a first one of the devices, notification of a first event,
 - C. instantiating, in the processing unit that first device, a first thread to handle that first and subsequent related events,
 - D. migrating from memory associated with the processing unit of another of the devices to to the processing unit of the first device at least one instruction of an instruction sequence for handling that event,
 - E. executing the migrated instruction as part of the first thread.
186. The method of claim 179, comprising executing on at least one of the processing units the steps of:
- A. providing one or more registers,

- B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, with the execution unit, a processor-level instruction that specifies a selected arithmetic operation and that specifies that operation is to be performed with a transpose by performing the specified arithmetic operation on one or more specified operands and by storing to one or more of the registers a result of that operation in transposed format.
187. The method of claim 179, comprising executing on at least one of the processing units the steps of:
- A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, on the execution unit, a selected processor-level instruction by encoding and storing to one or more of the register(s) by storing to those register(s) one or more values from a JPEG2175 binary arithmetic coder lookup table.
188. The method of claim 179, comprising executing on at least one of the processing units the steps of:
- A. providing one or more registers,
 - B. providing an execution unit that is in communications coupling with the one or more registers,
 - C. executing, on the execution unit, a selected processor-level instruction by encoding and storing to one or more of the register(s) a stripe column for bit plane coding within JPEG2268 Embedded Block Coding with Optimized Truncation (EBCOT).

189. The method of claim 179, comprising
- A. delivering interrupts and other events to the processing units,
 - B. executing thread management code in one or more of the processing units that optimizes at least one of thread instantiation, maintenance and thread assignment.
190. The method of claim 179, comprising executing on at least one of the processing units the steps of:
- A. executing, in an execution unit that is in communications coupling with one or more registers and with a cache subsystem that includes a cache memory, memory reference instructions to transfer any of data and instructions (collectively, "data") between the cache memory and the one or more registers, and
 - B. varying, with the cache subsystem, utilization of the cache memory in response to execution of selected memory reference instructions that effect data transfers between the one or more registers and the cache memory.

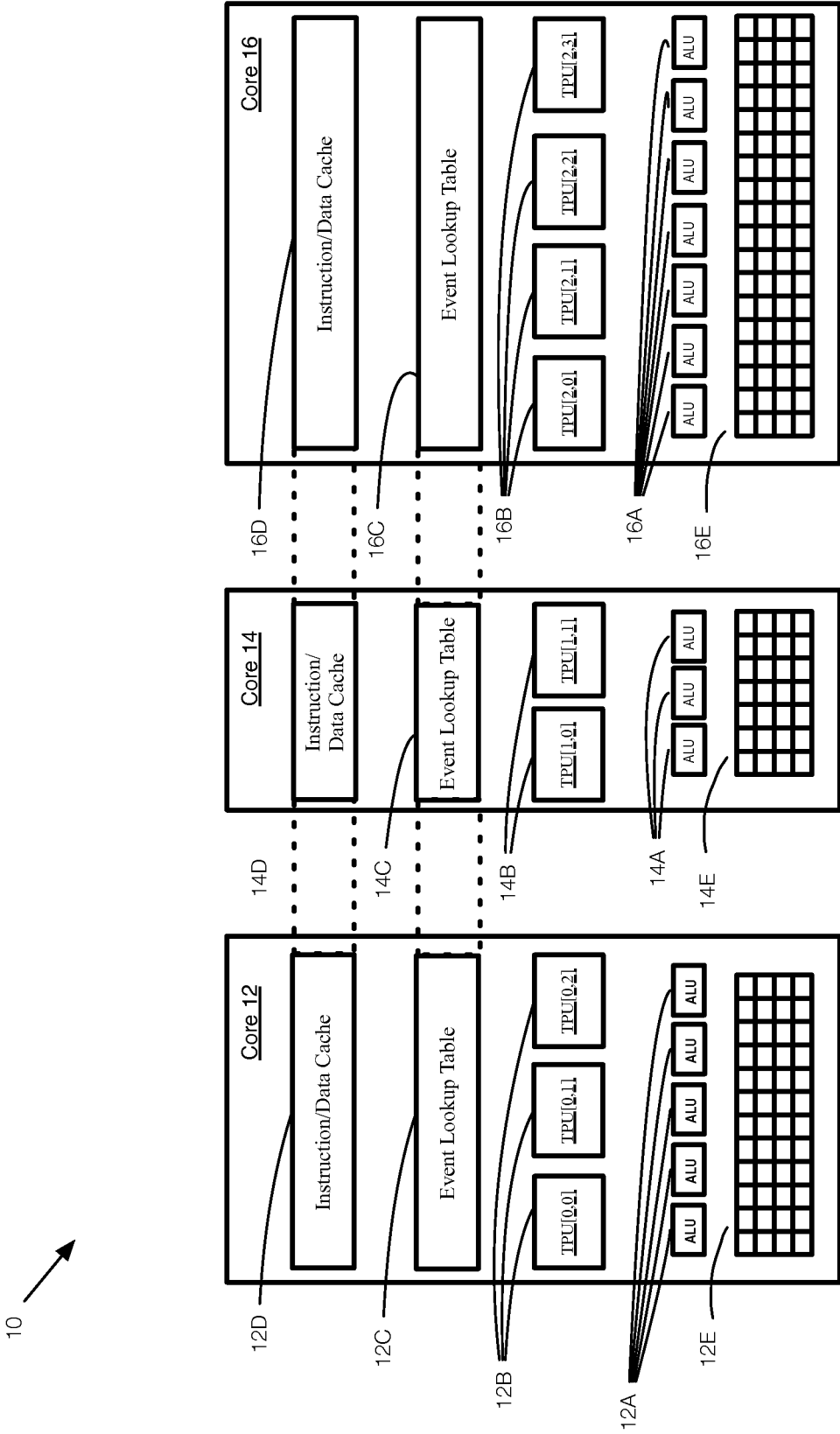


Figure 1

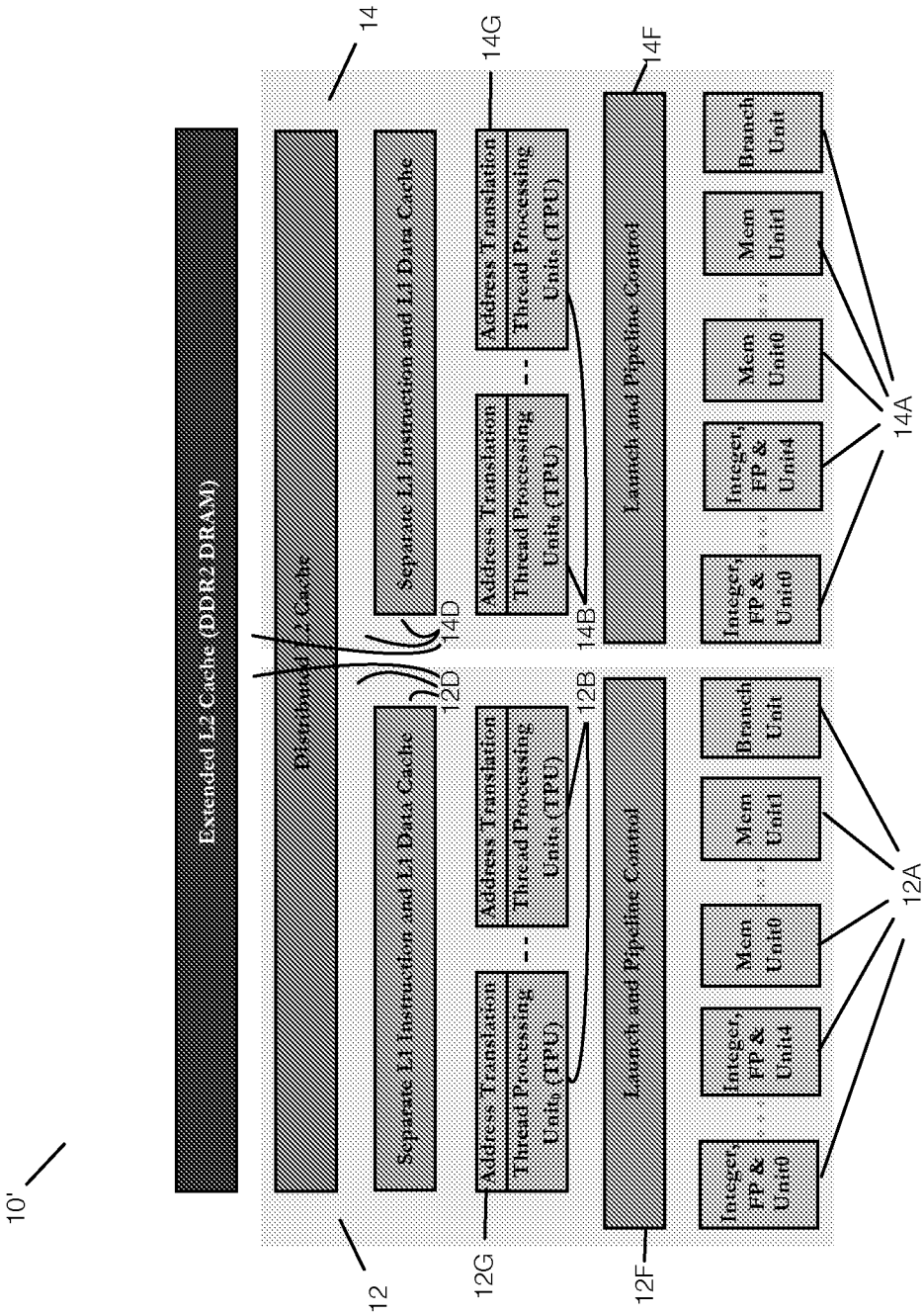
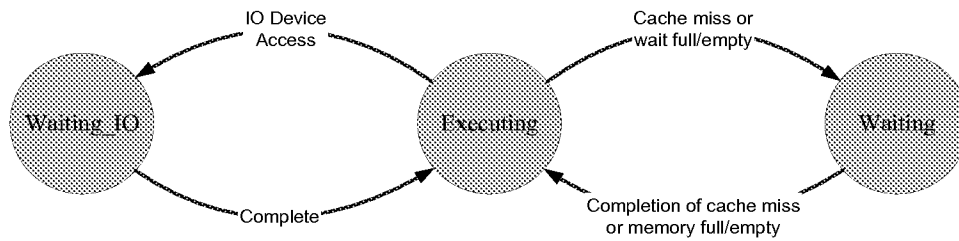


Figure 2



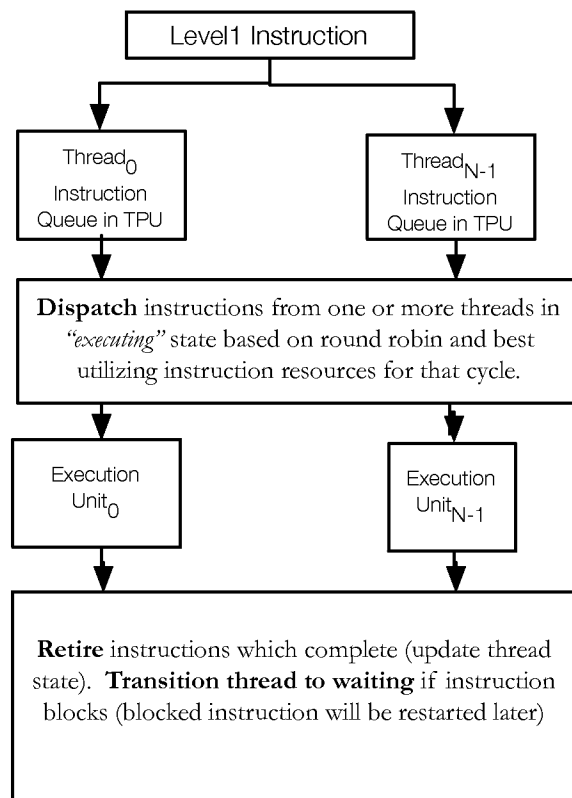
Thread States and Transitions

Figure 3

Fetch Instructions for a Single thread each cycle

Recognize events by transitioning thread to executing

Dispatch up to one instruction per execution unit



Thread-Instruction Abstraction

Figure 4

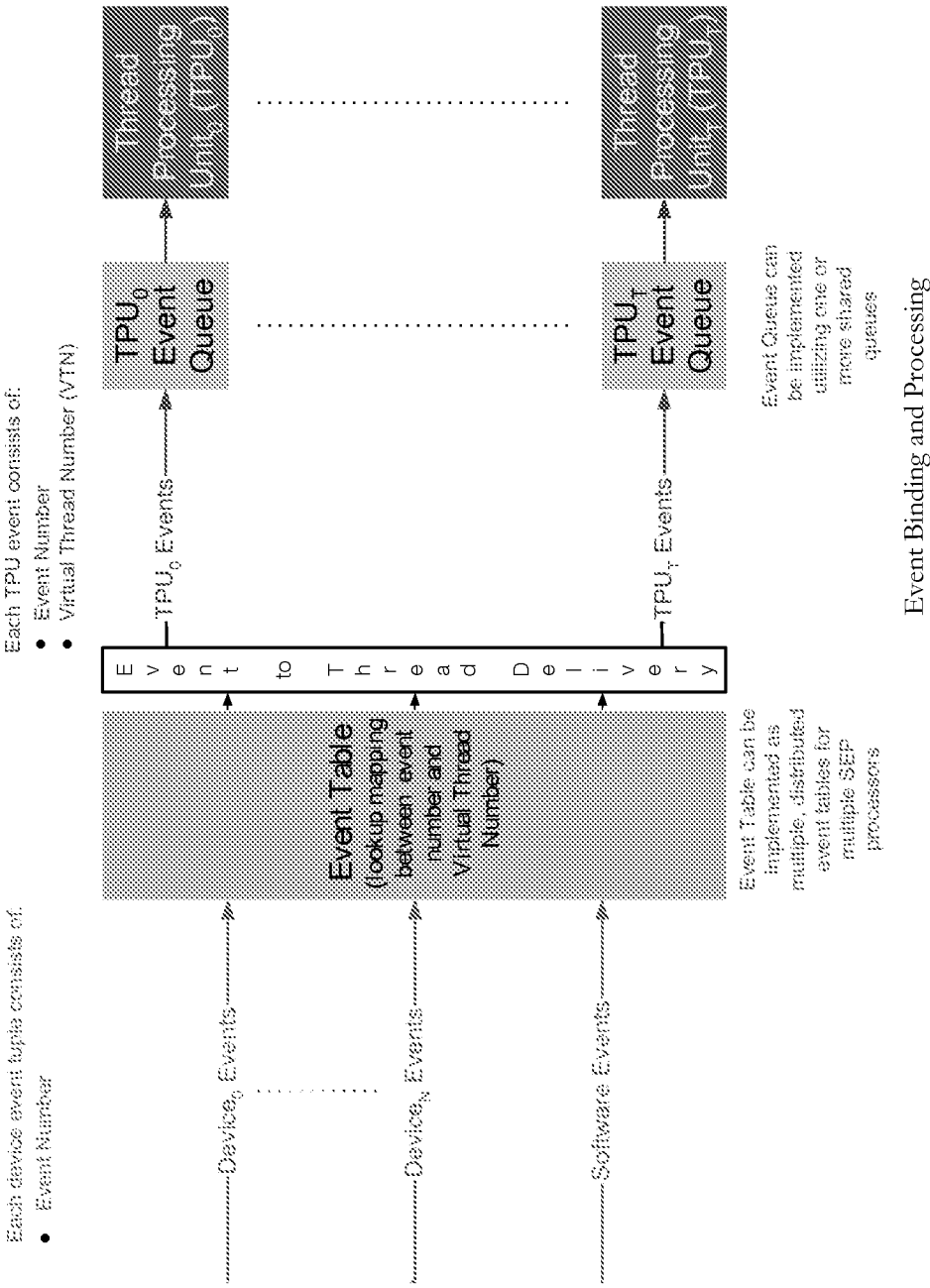
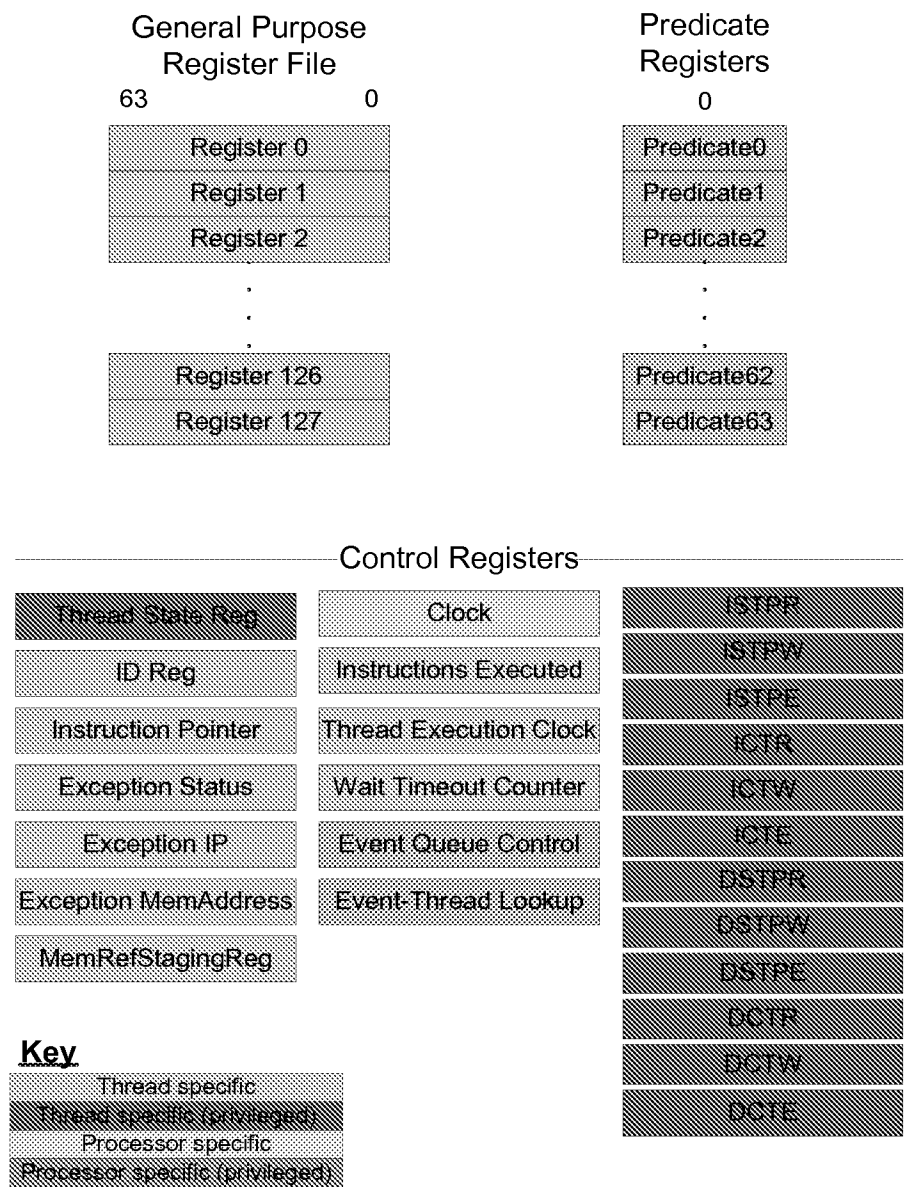


Figure 5



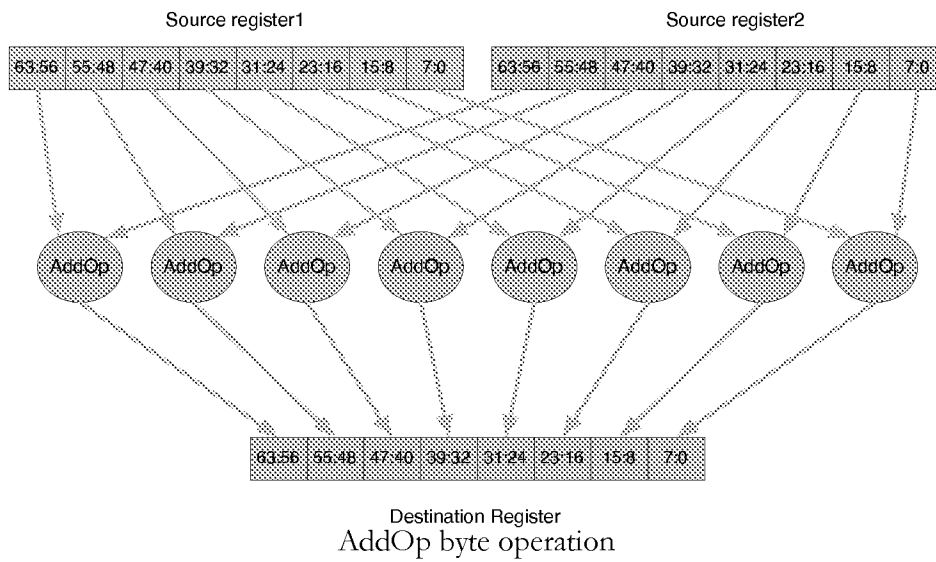


Figure 7

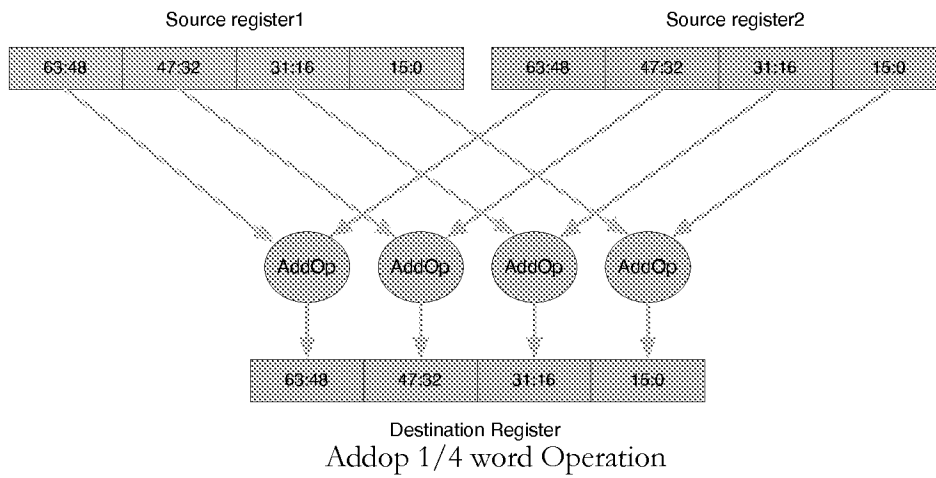


Figure 8

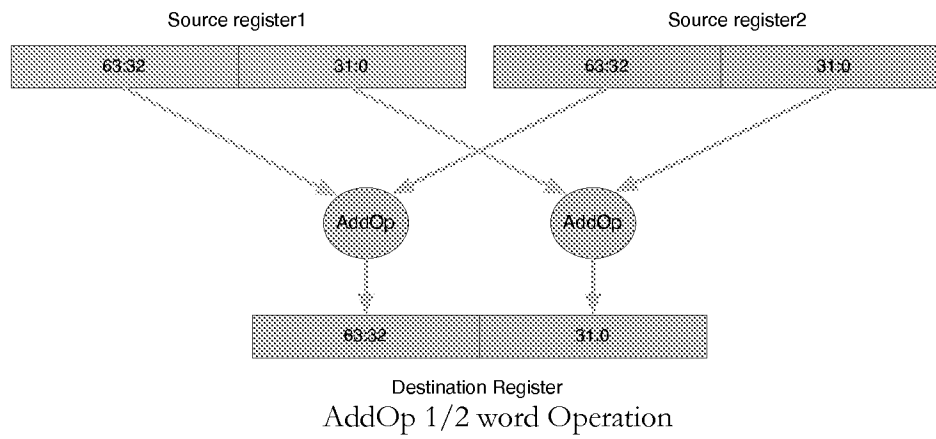


Figure 9

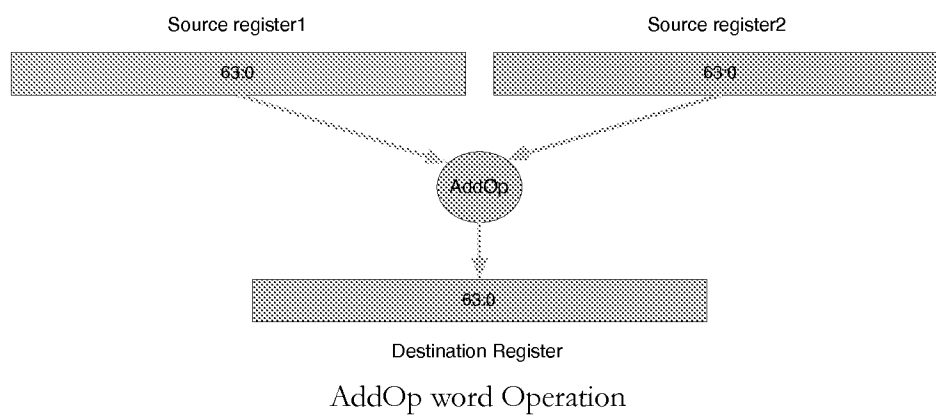


Figure 10

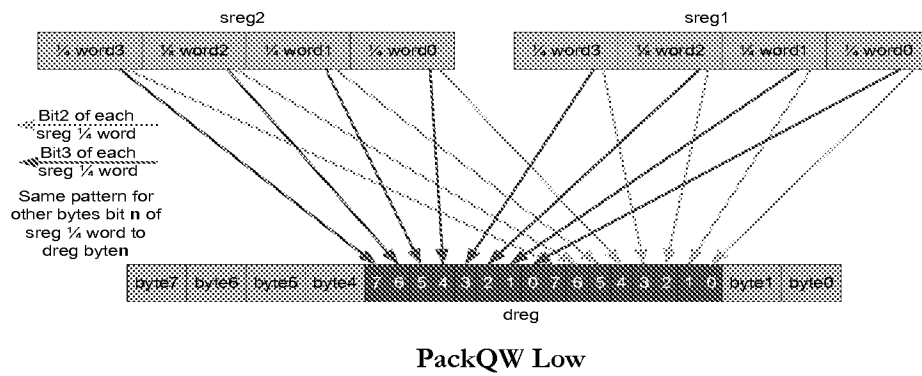


Figure 11

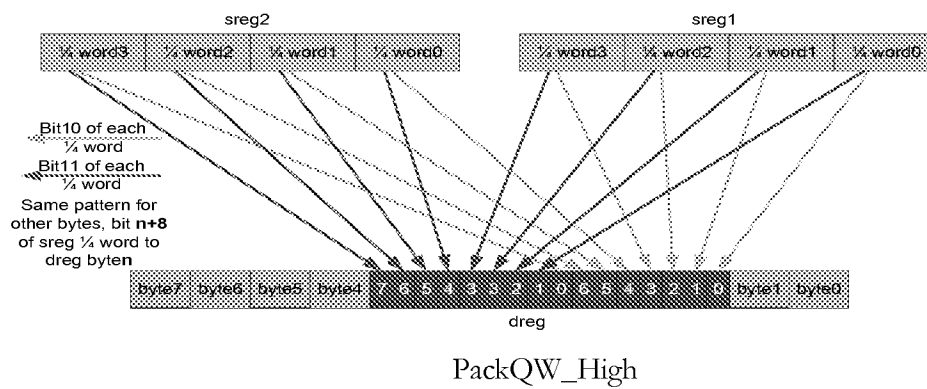


Figure 12

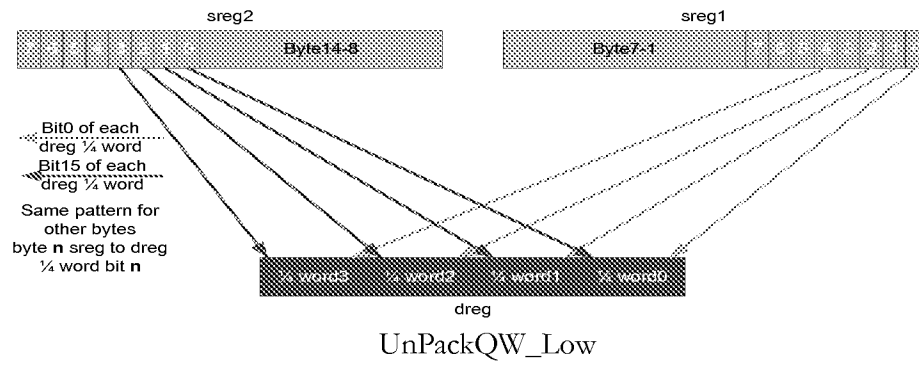


Figure 13

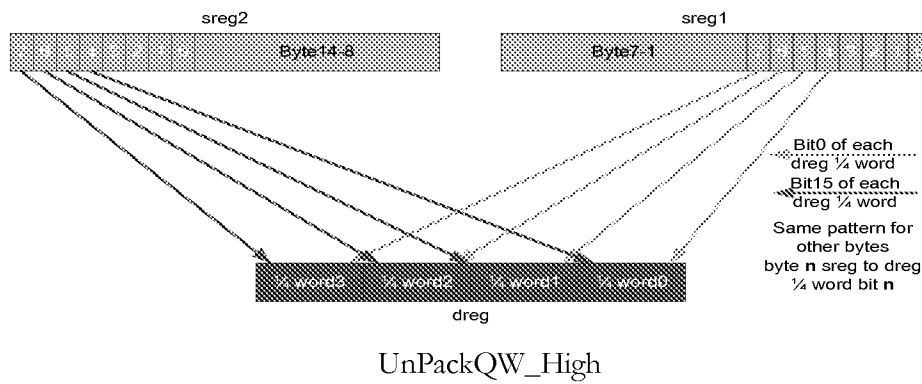


Figure 14

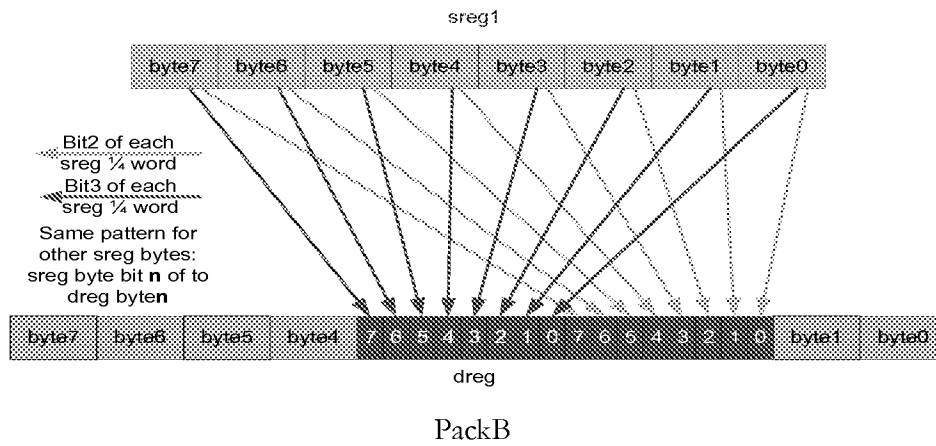


Figure 15

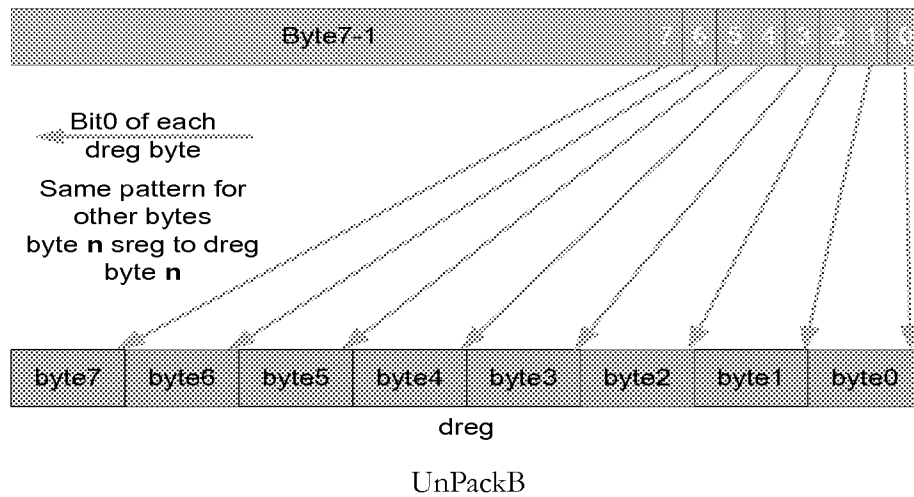
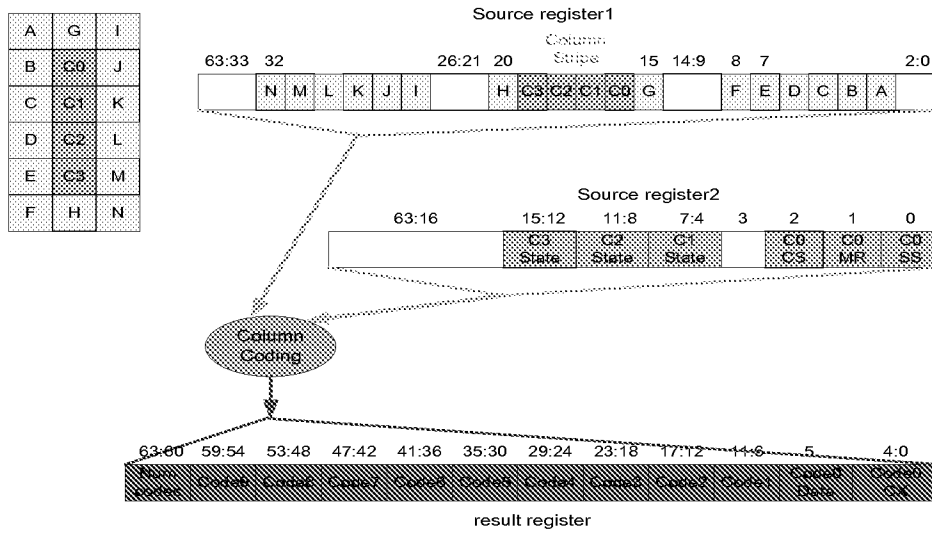
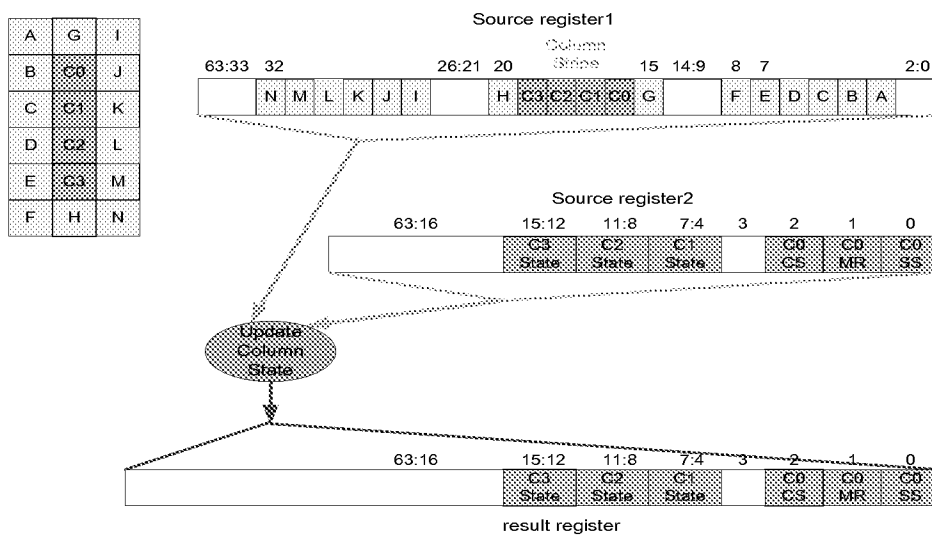


Figure 16



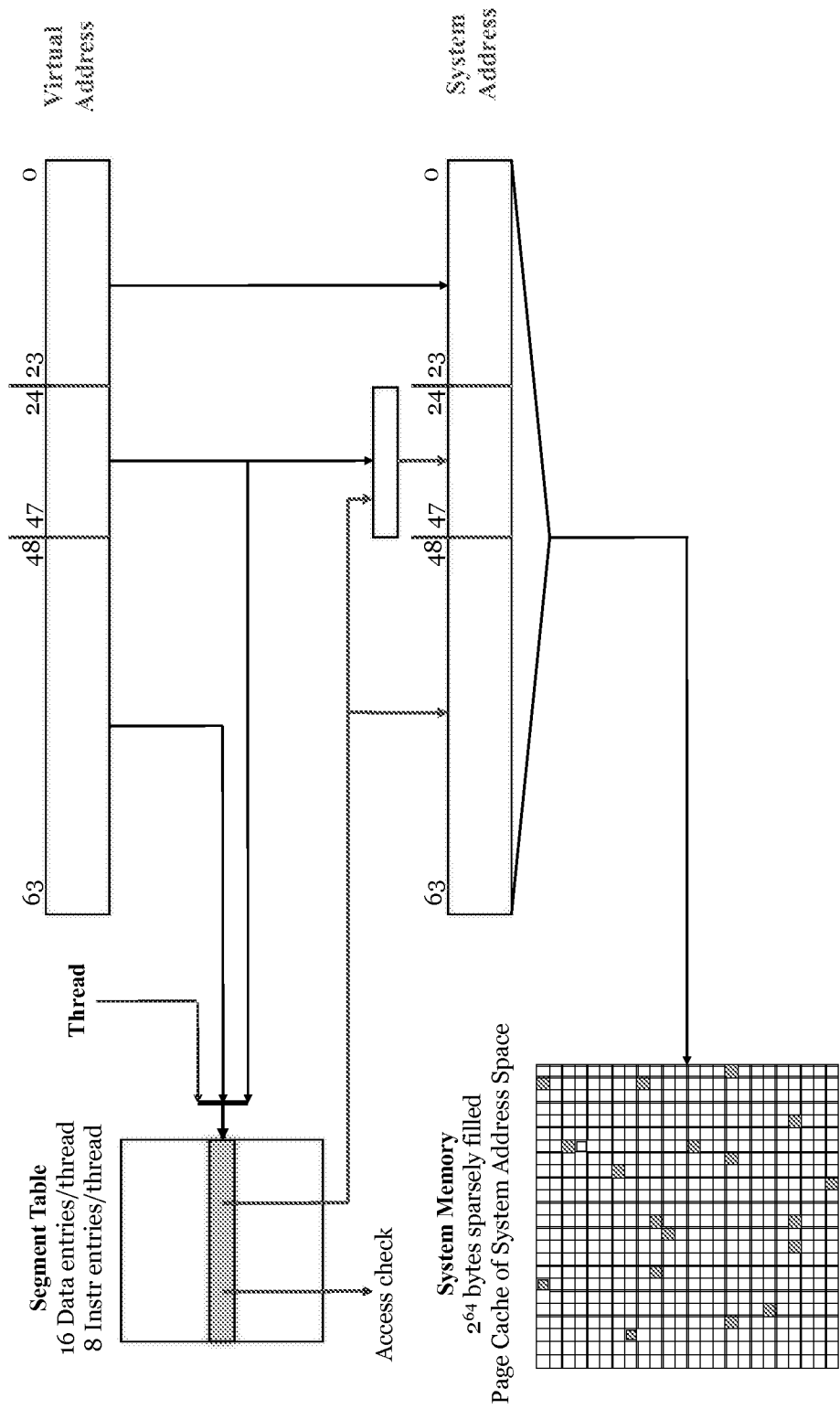
bpscode instruction column coding option

Figure 17



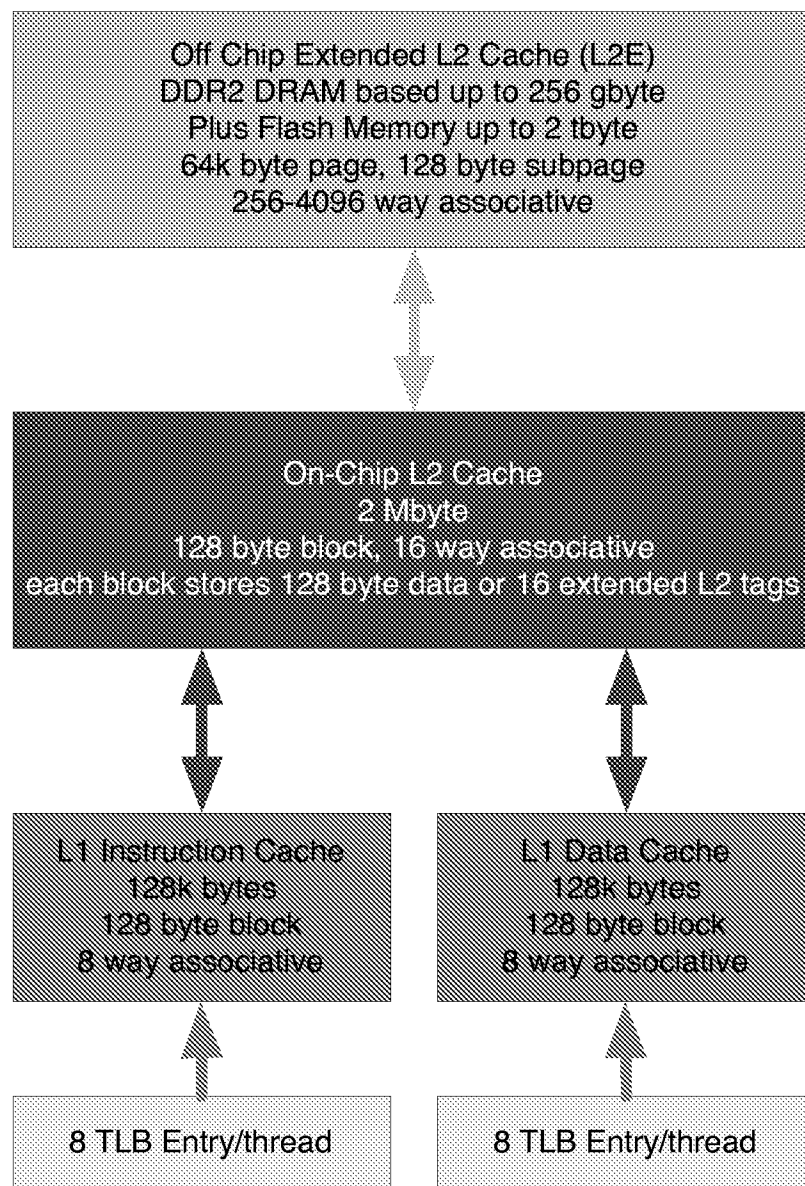
bpscode instruction update column state option

Figure 18



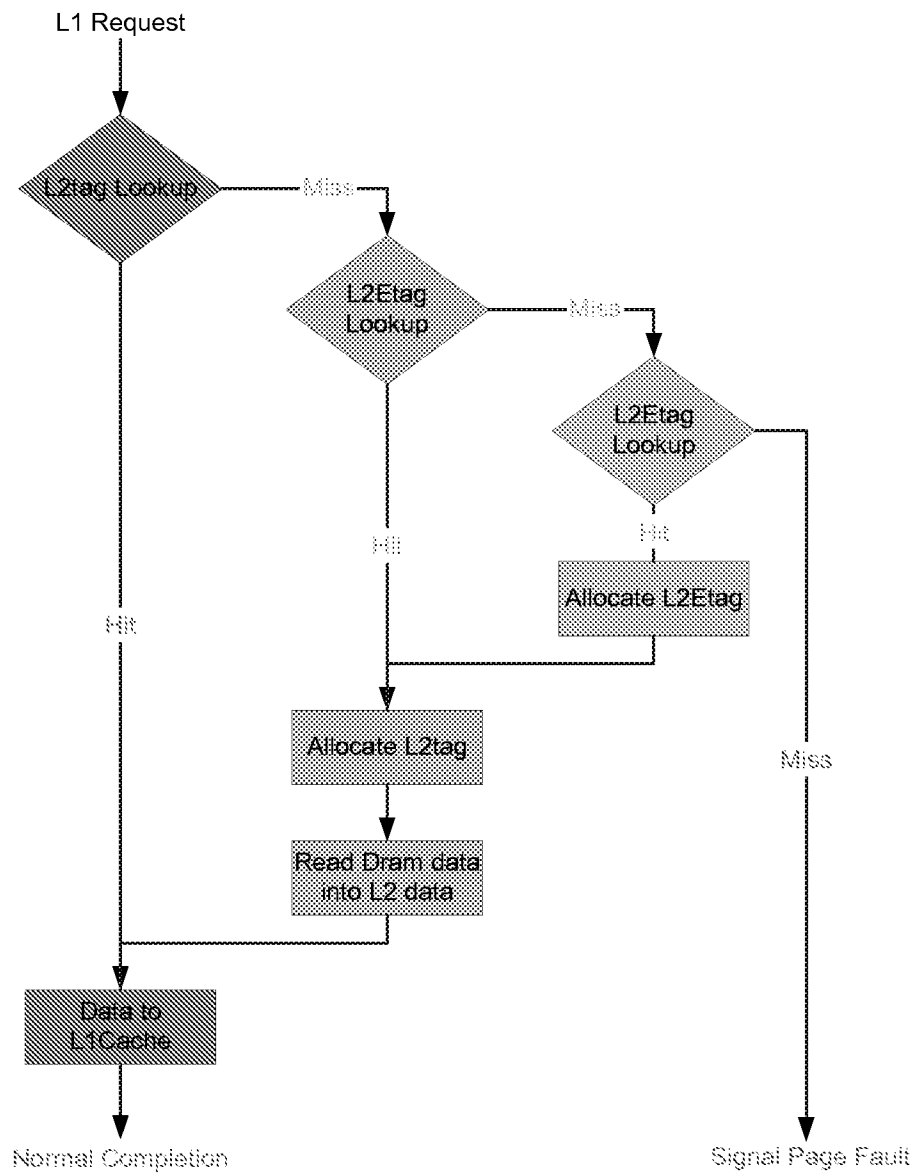
SEP Memory Address Model

Figure 19



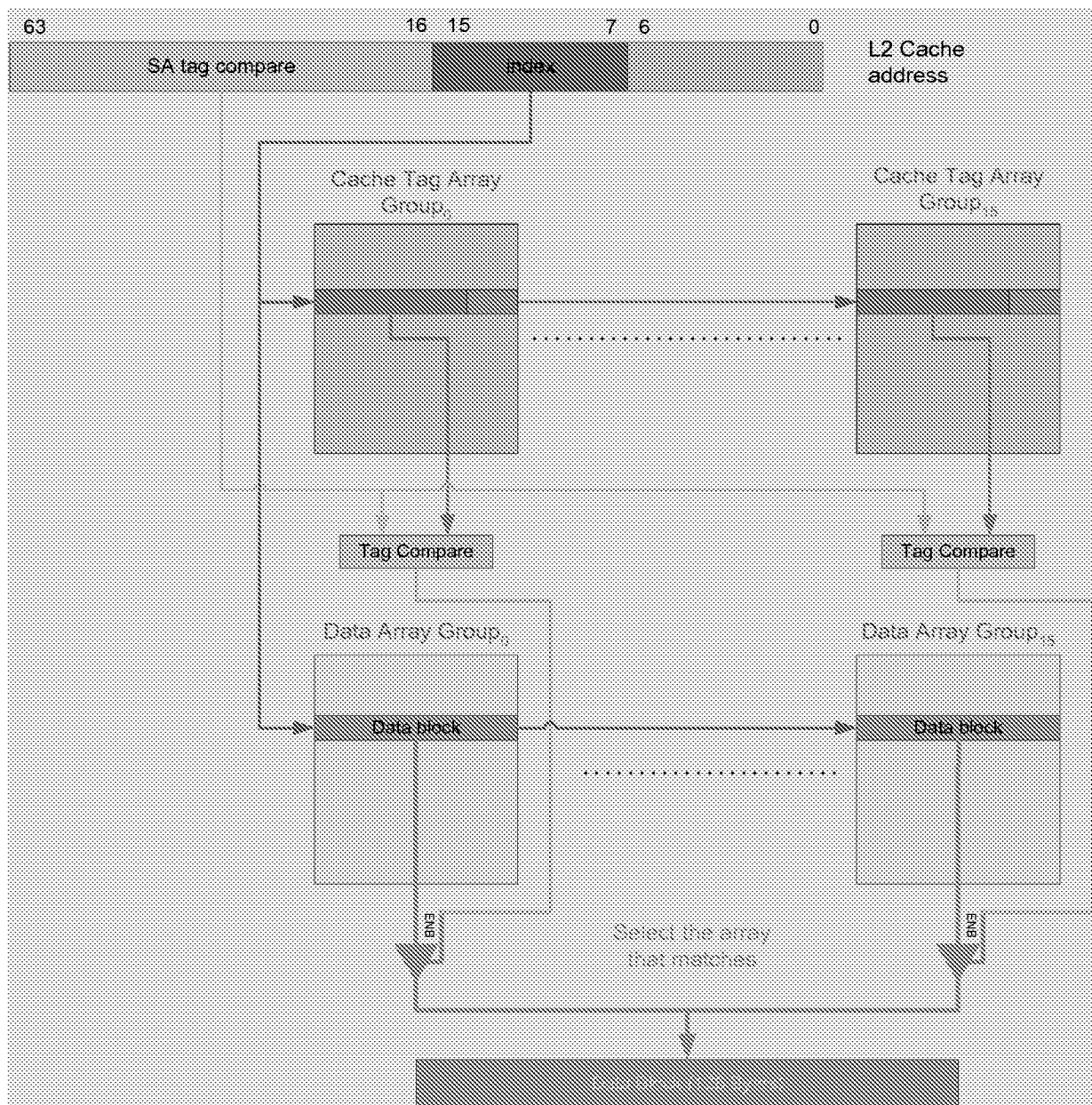
SEP Cache Memory Hierarchy Organization

Figure 20



L2 and L2E Operation

Figure 21



L2 Cache Hit

Figure 22

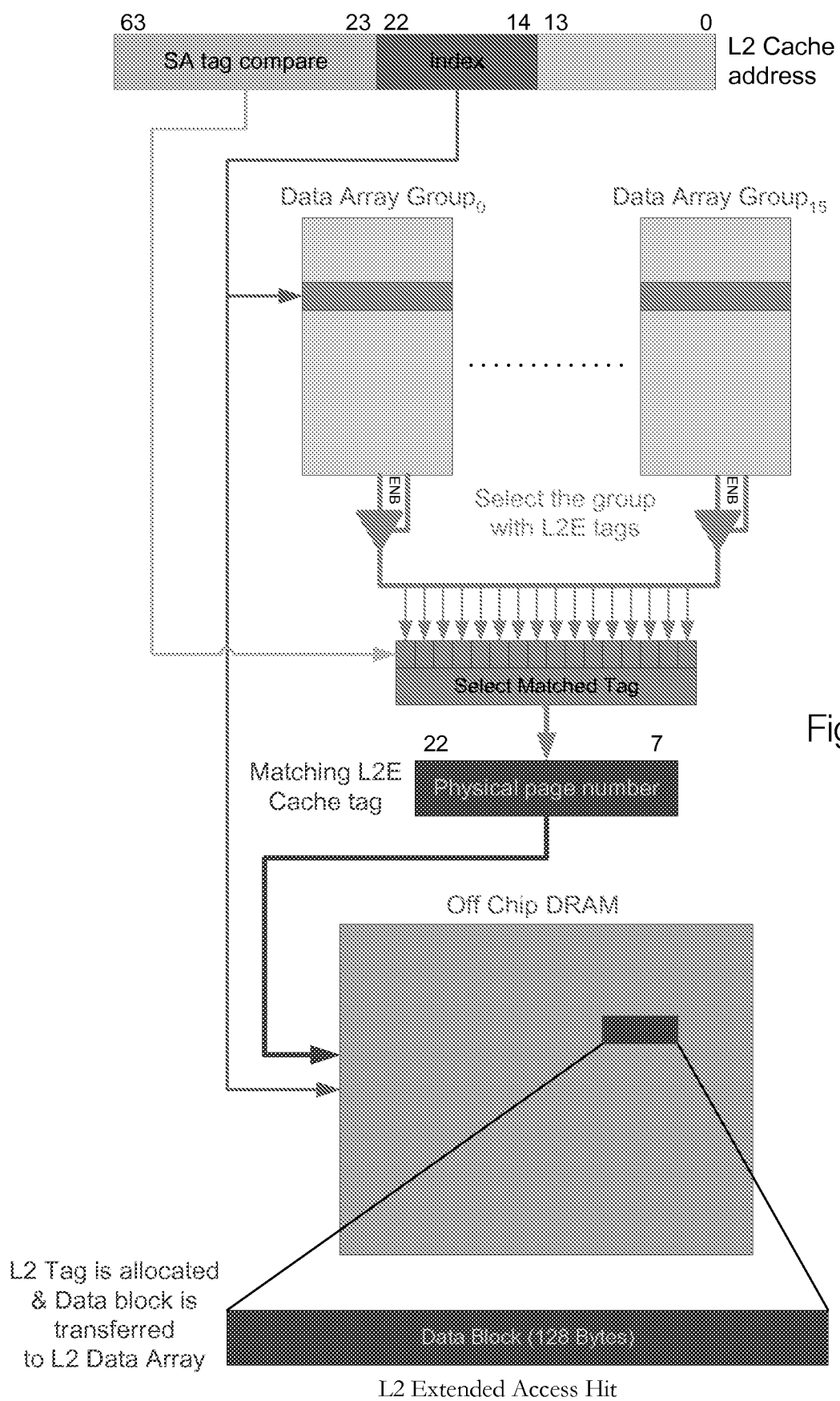
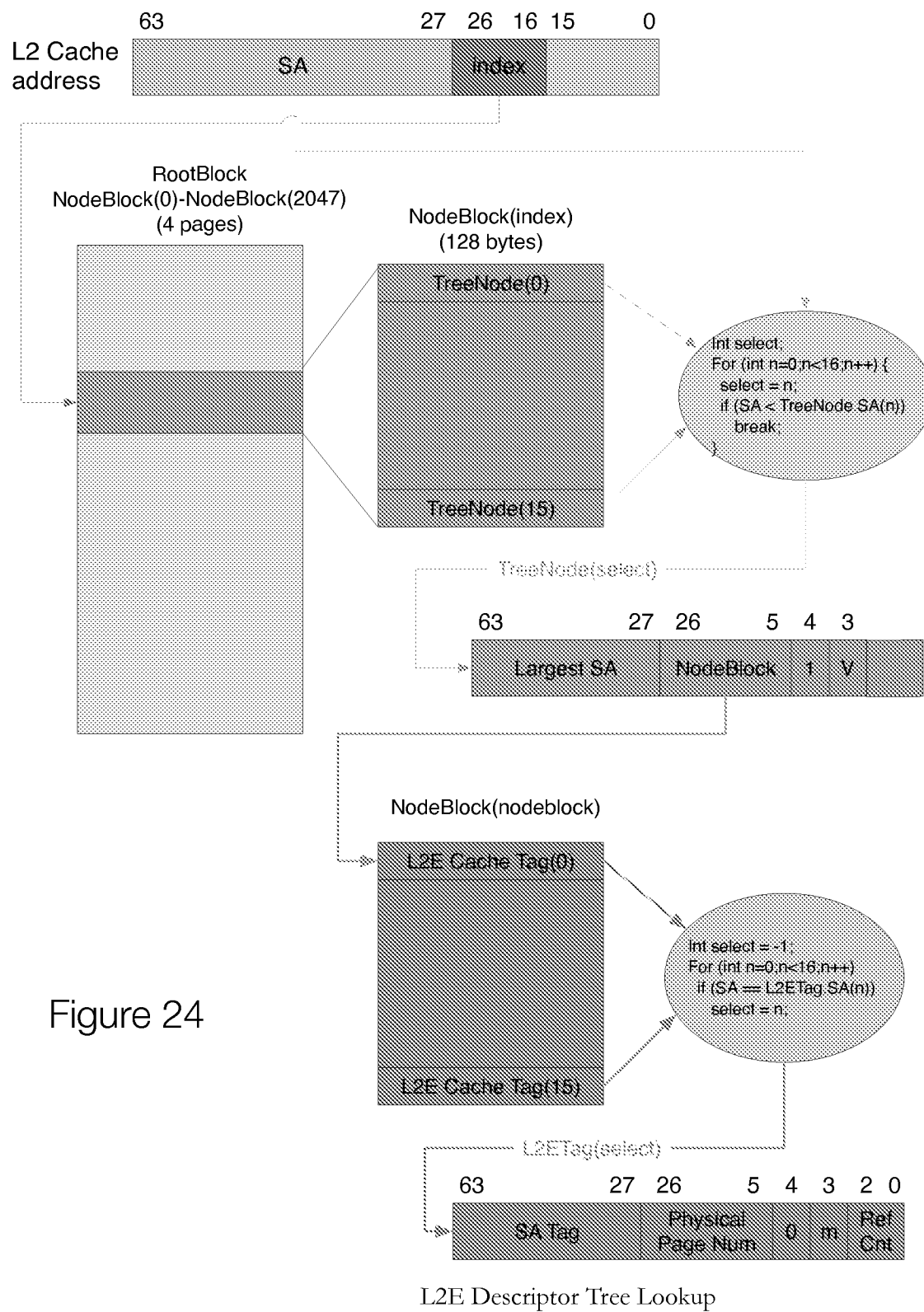
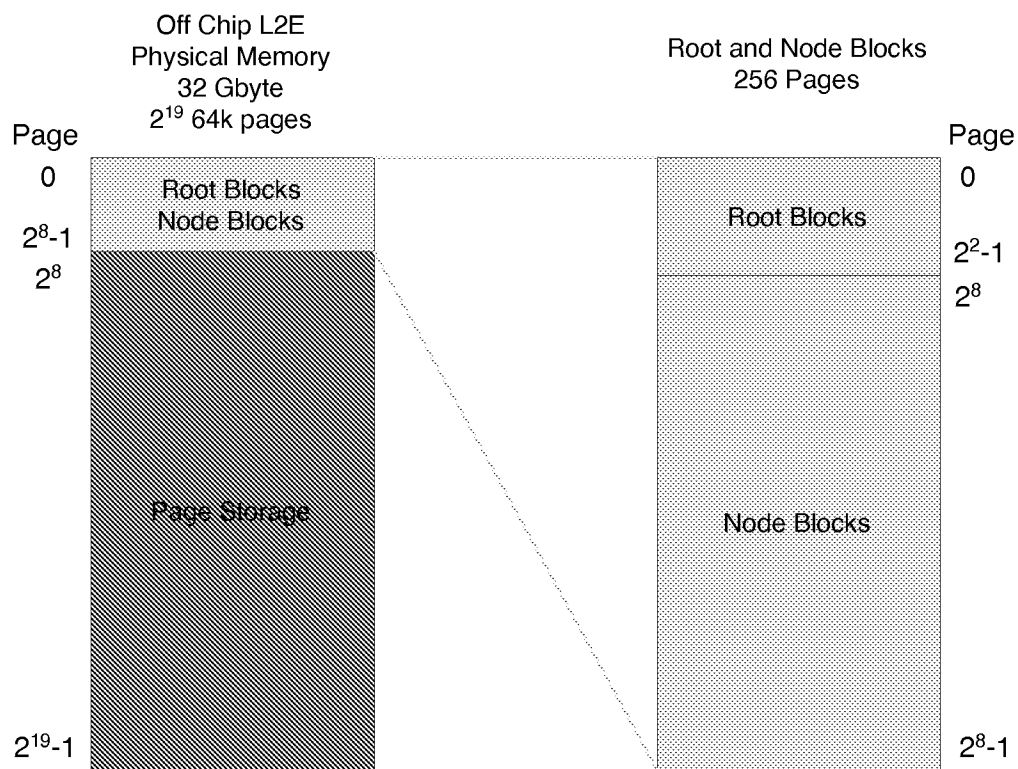


Figure 23





L2E Physical Memory Layout for 32 Gbyte

Figure 25

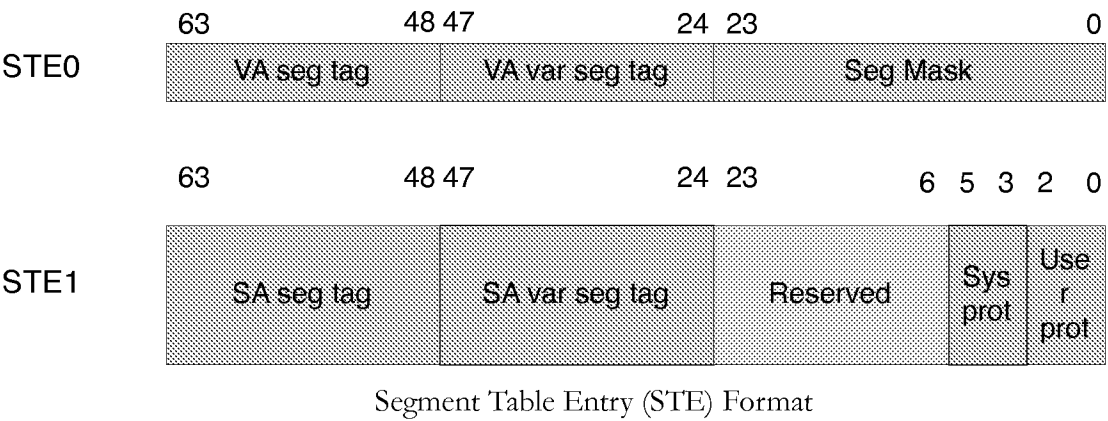


Figure 26

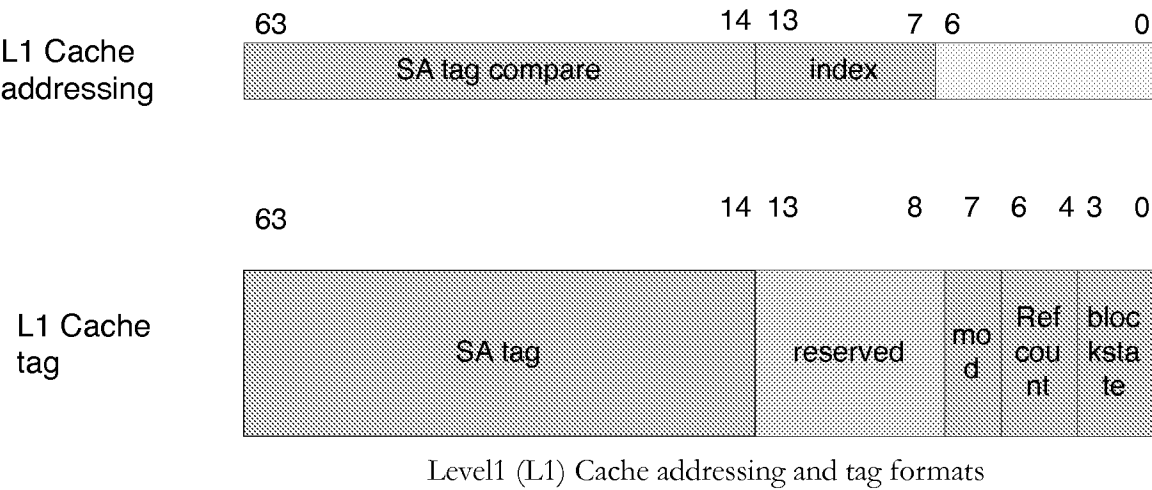
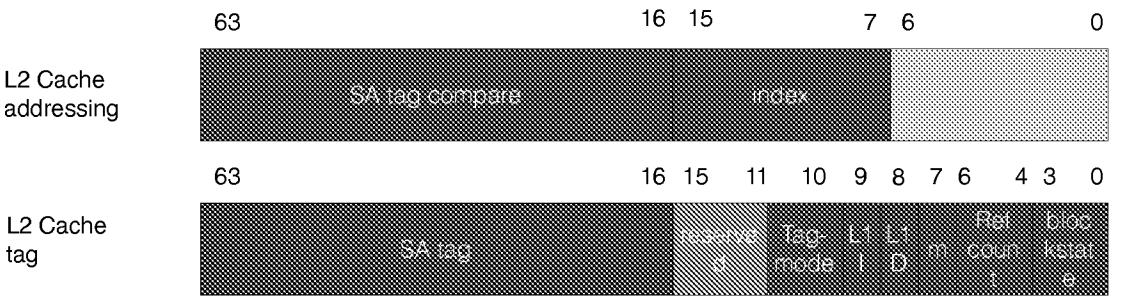
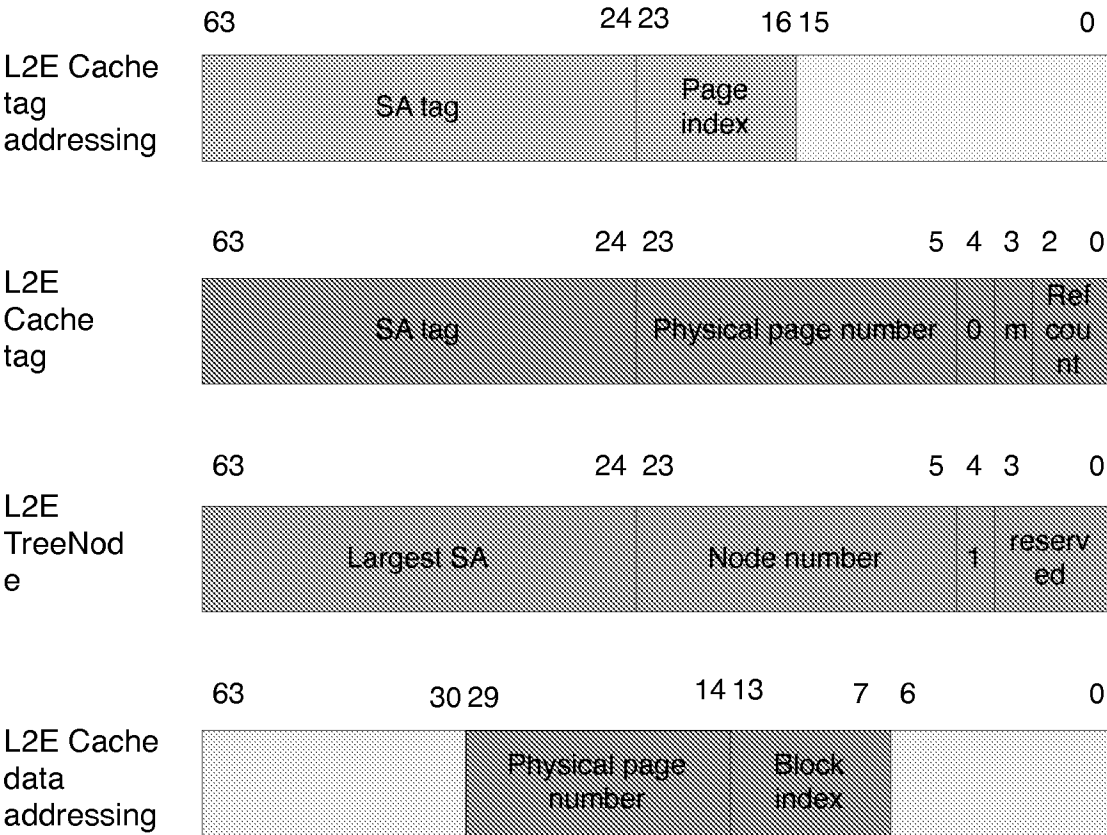


Figure 27



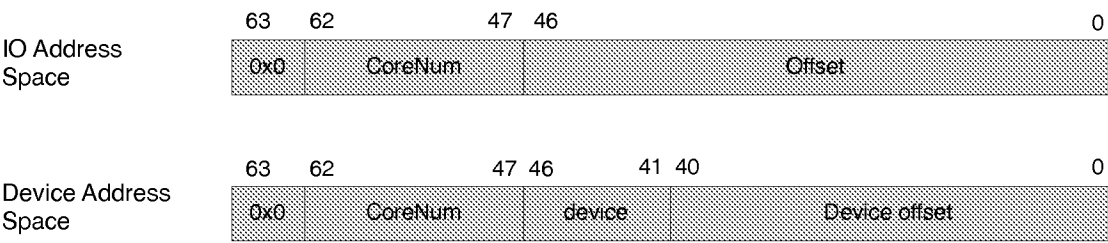
Level2 (L2) Cache addressing and tag formats

Figure 28



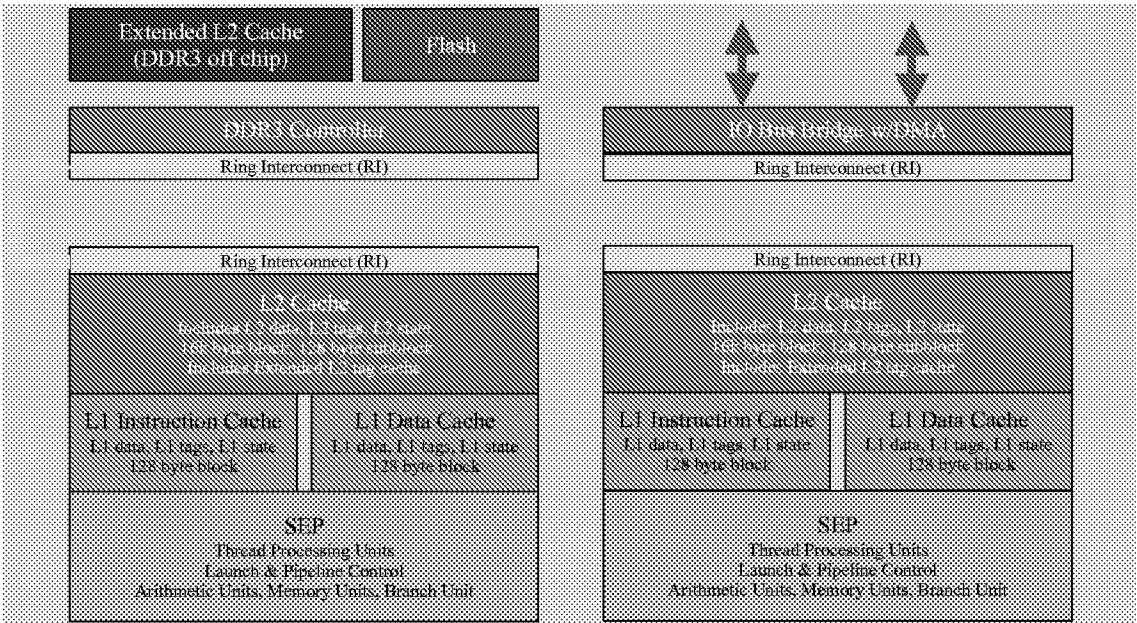
Level2 Extended (L2E) addressing and tag formats

Figure 29



IO Address Space

Figure 30



Ring Interconnect Architecture

Figure 31

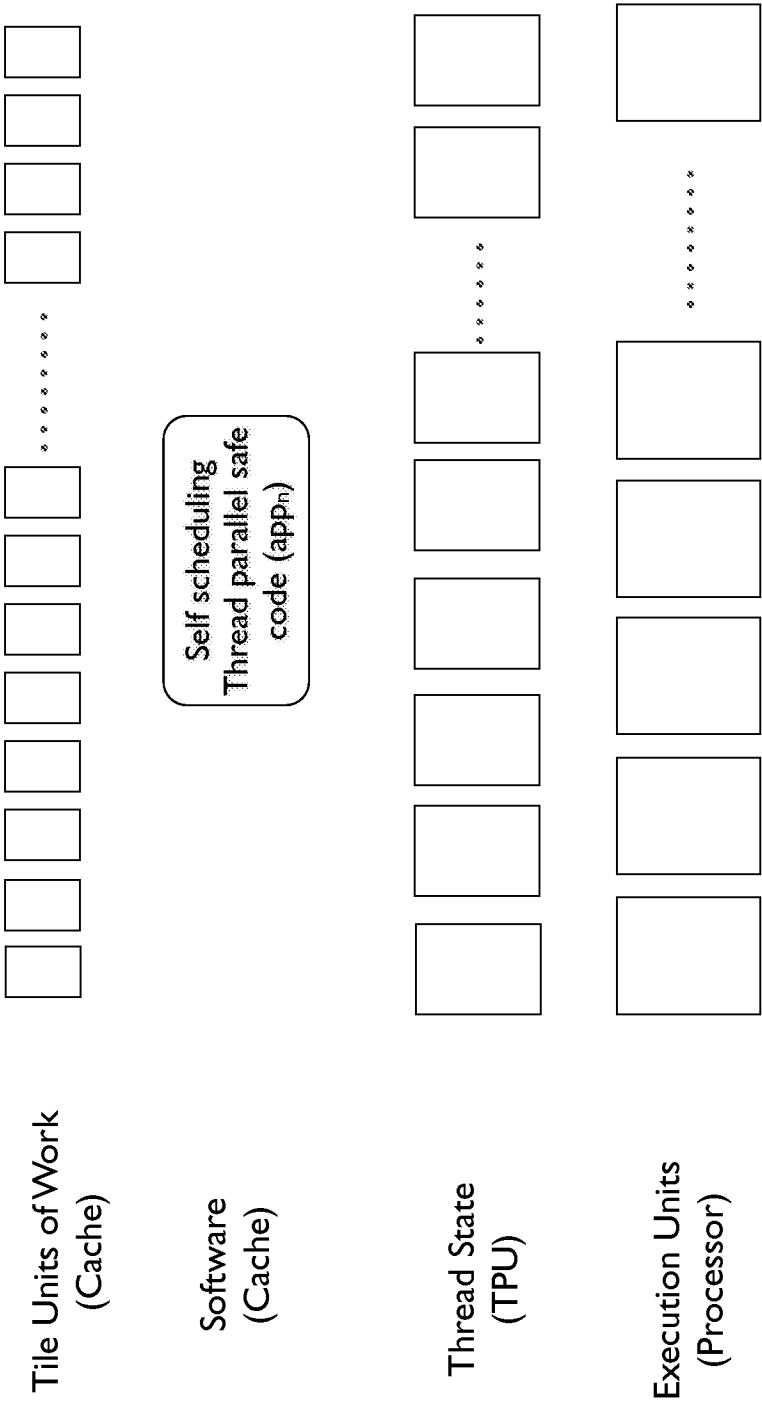


Figure 32

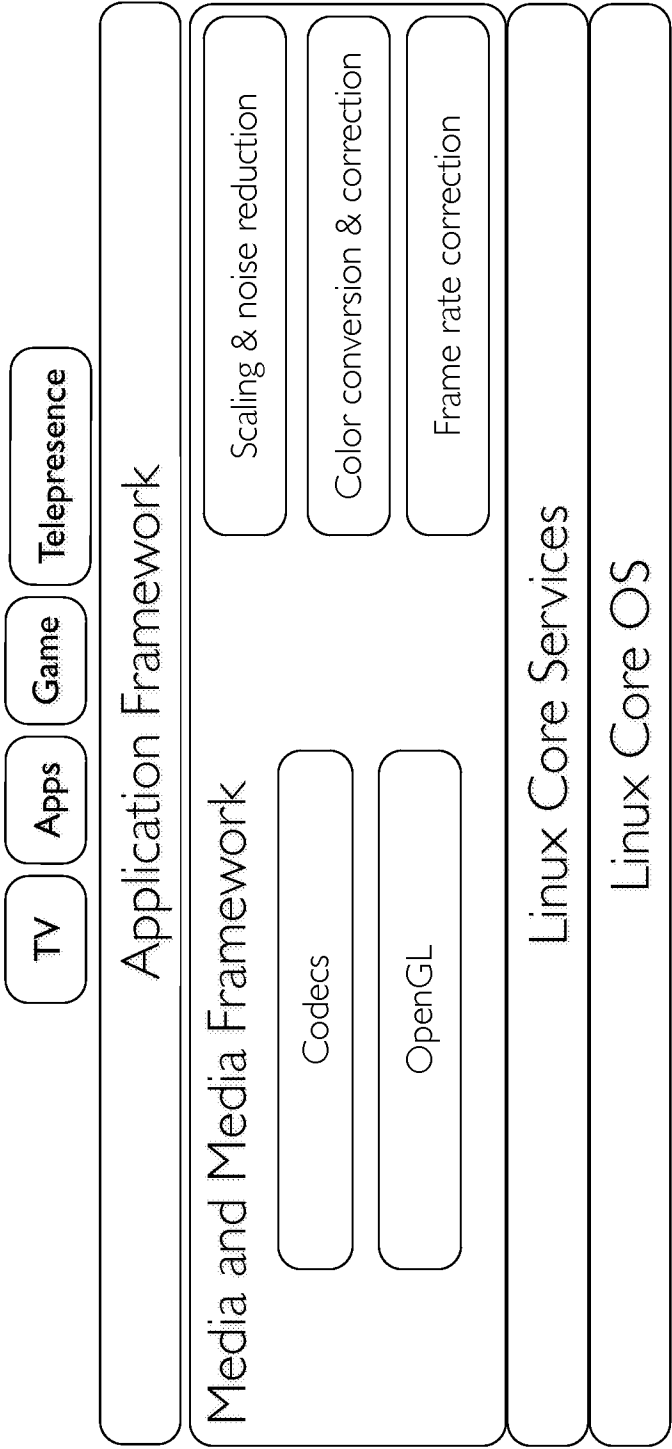


Figure 33

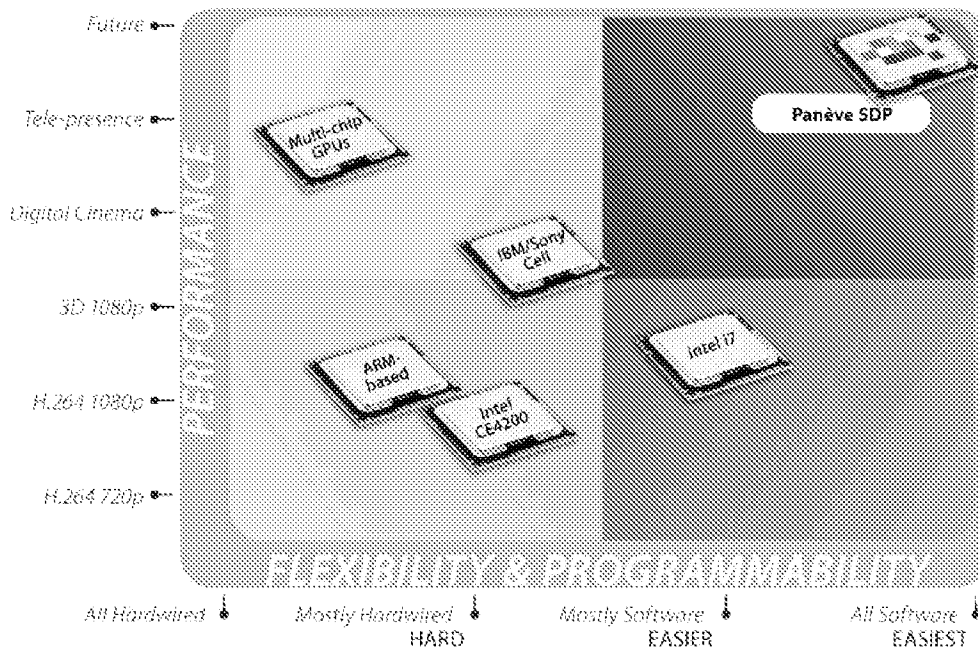


Figure 34

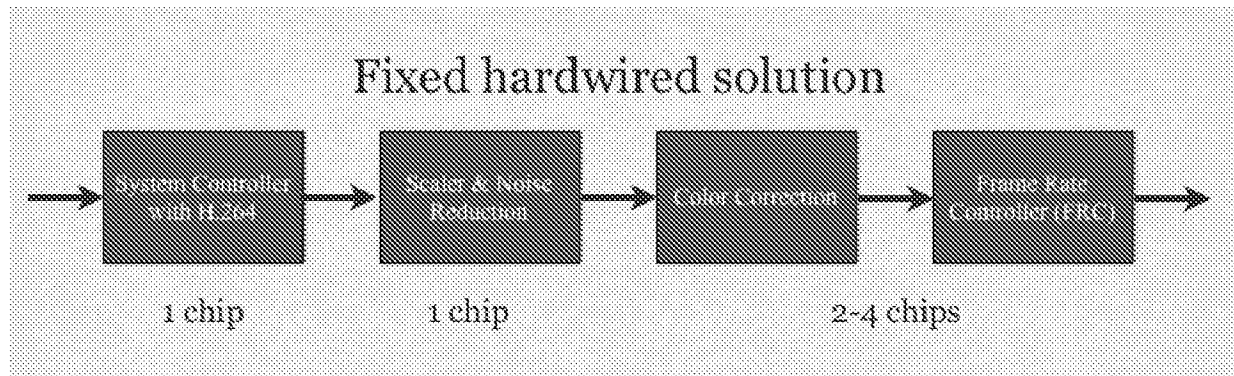


Figure 35

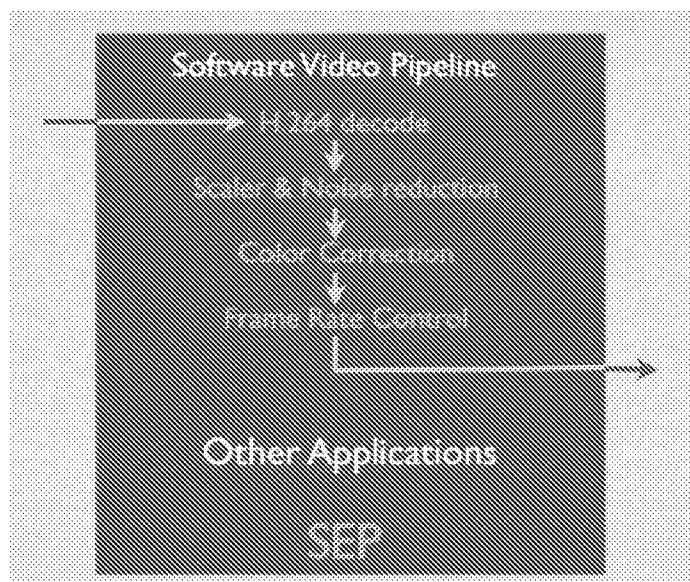


Figure 36

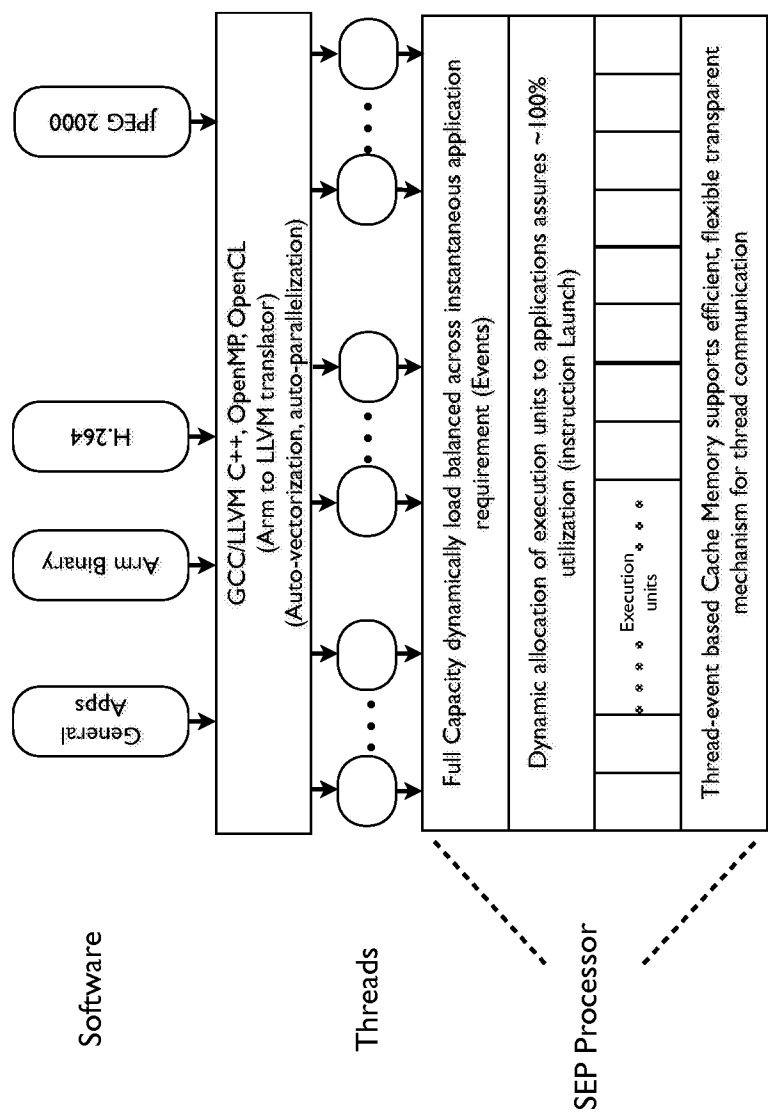


Figure 37

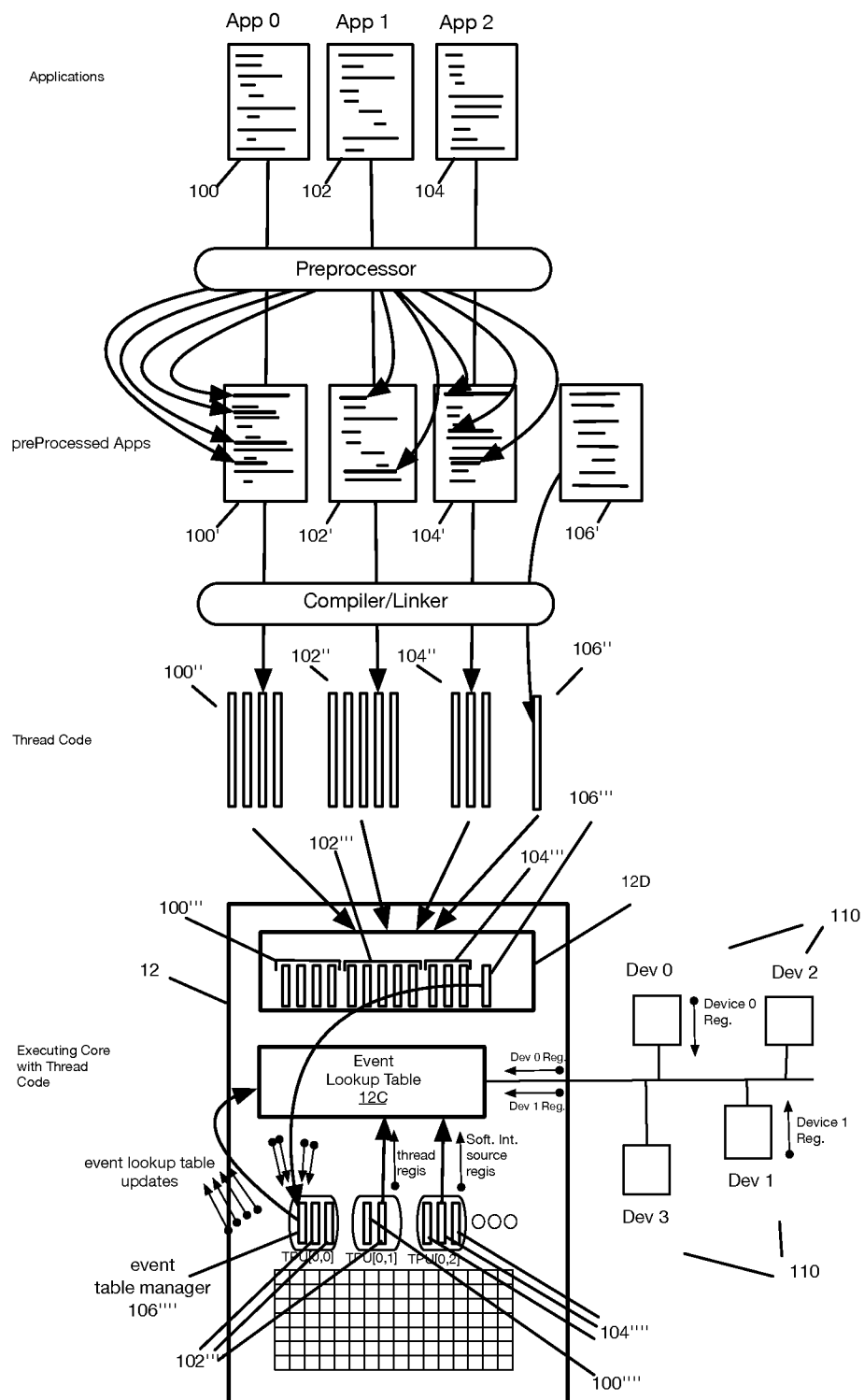


Figure 38

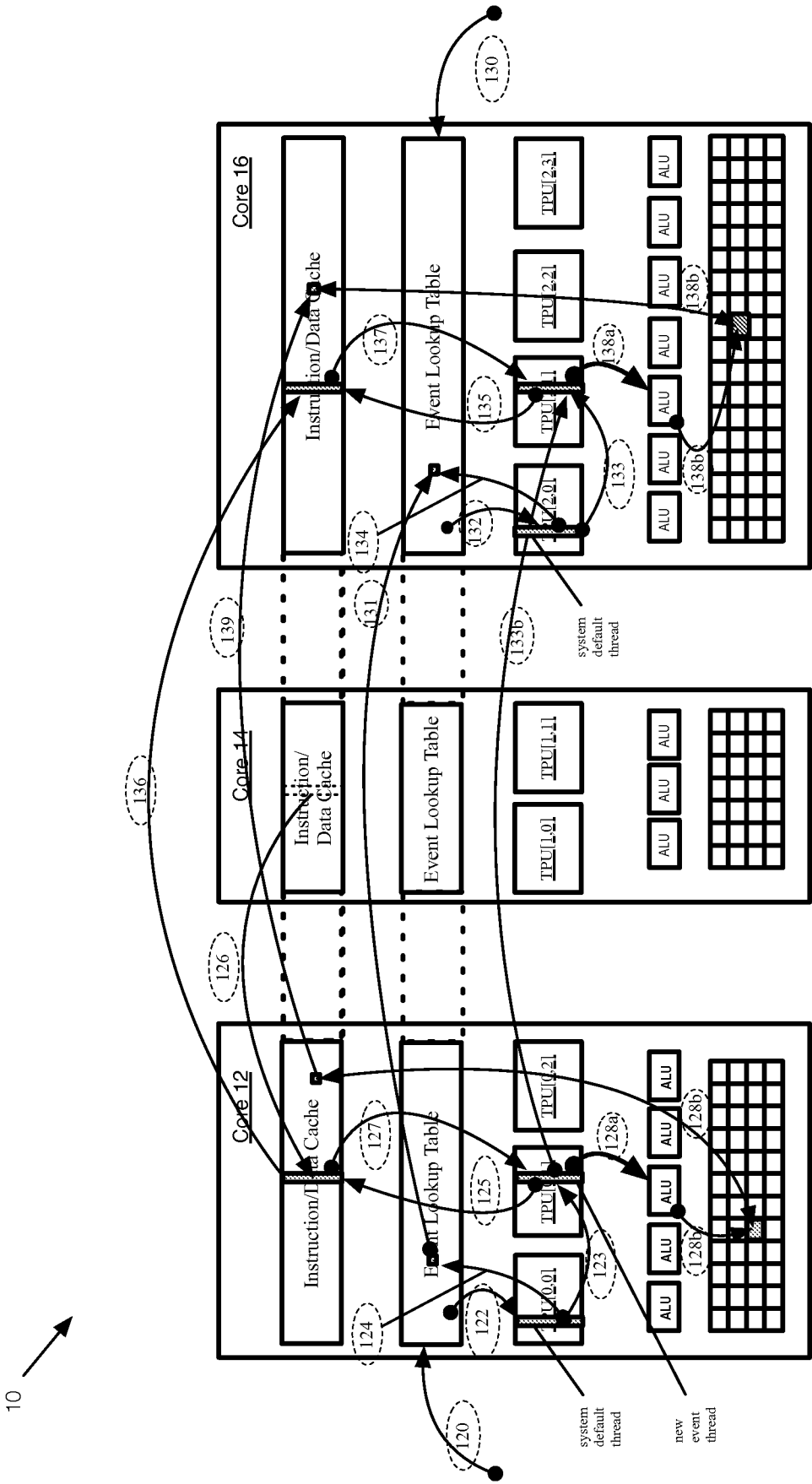


Figure 39

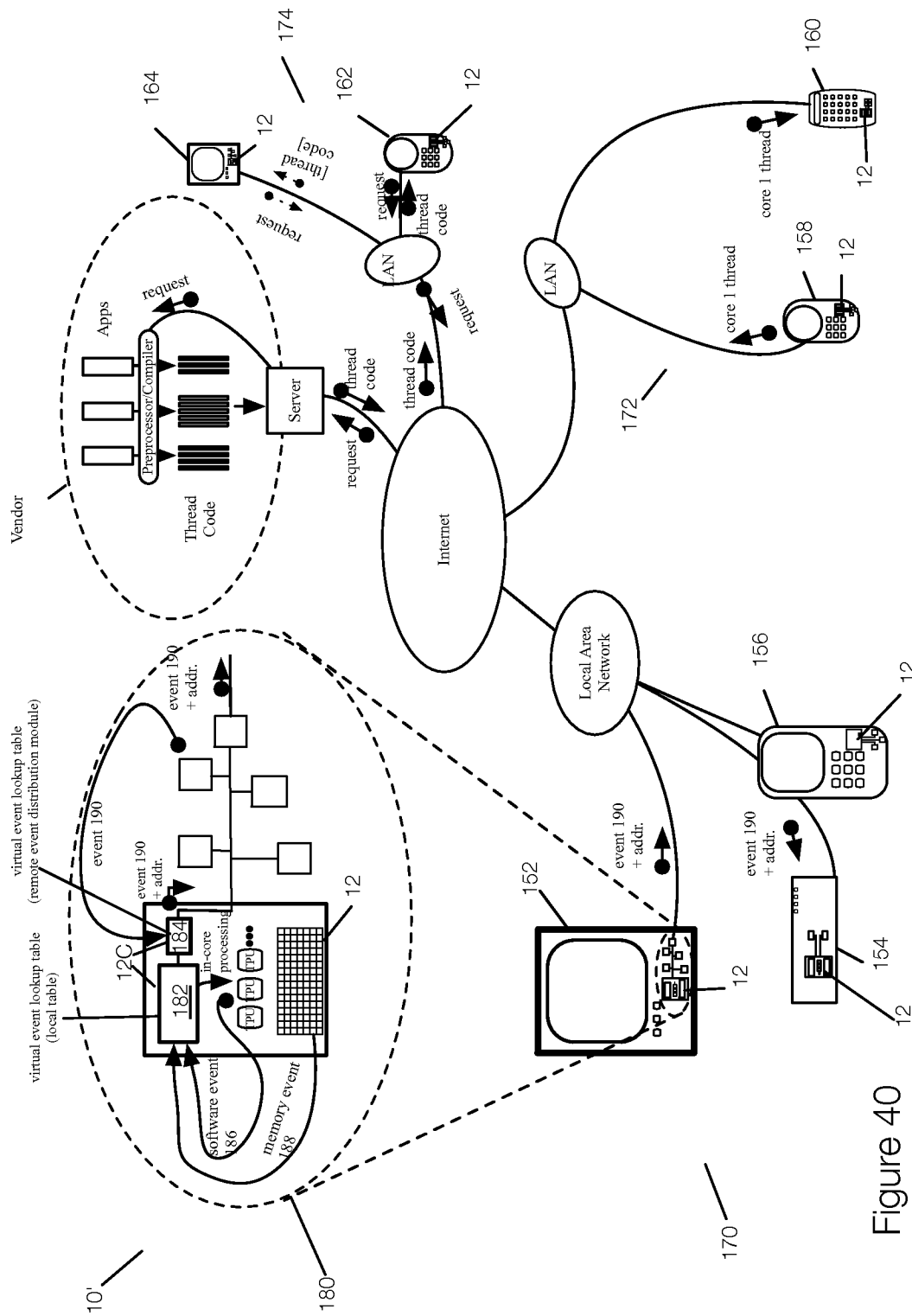


Figure 40

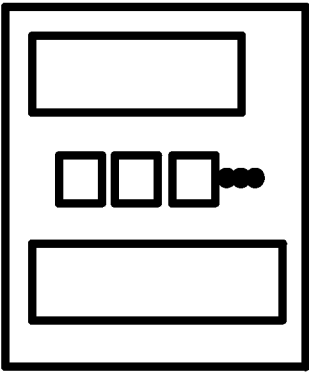


Figure 41A

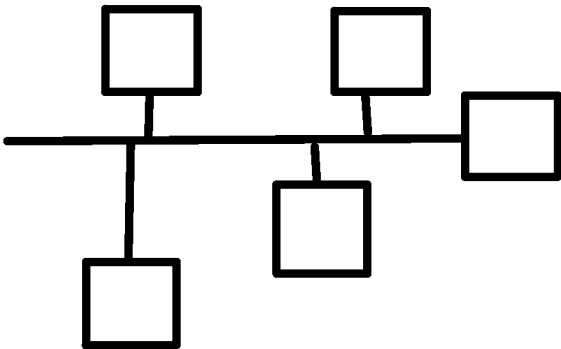


Figure 41B

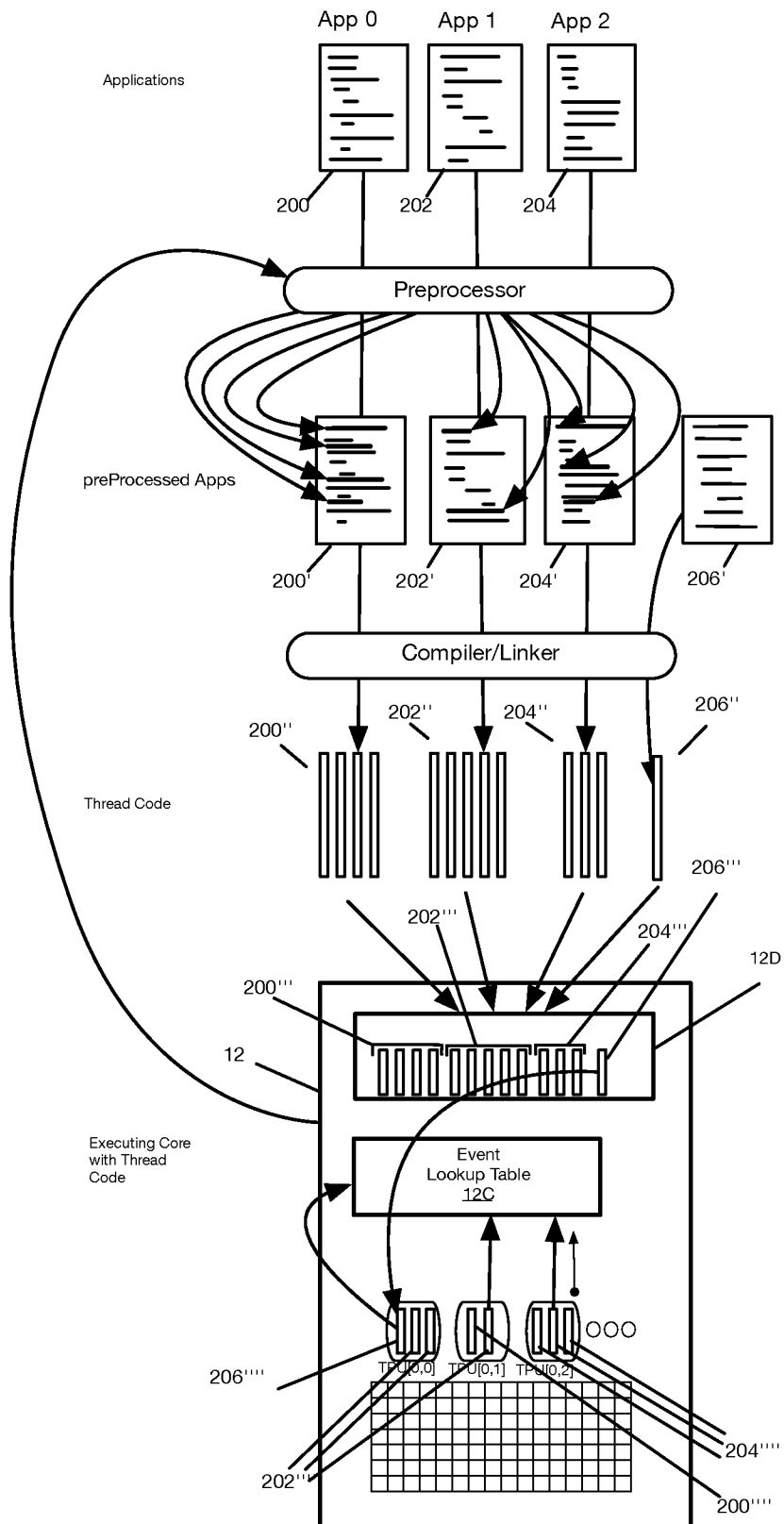


Figure 42

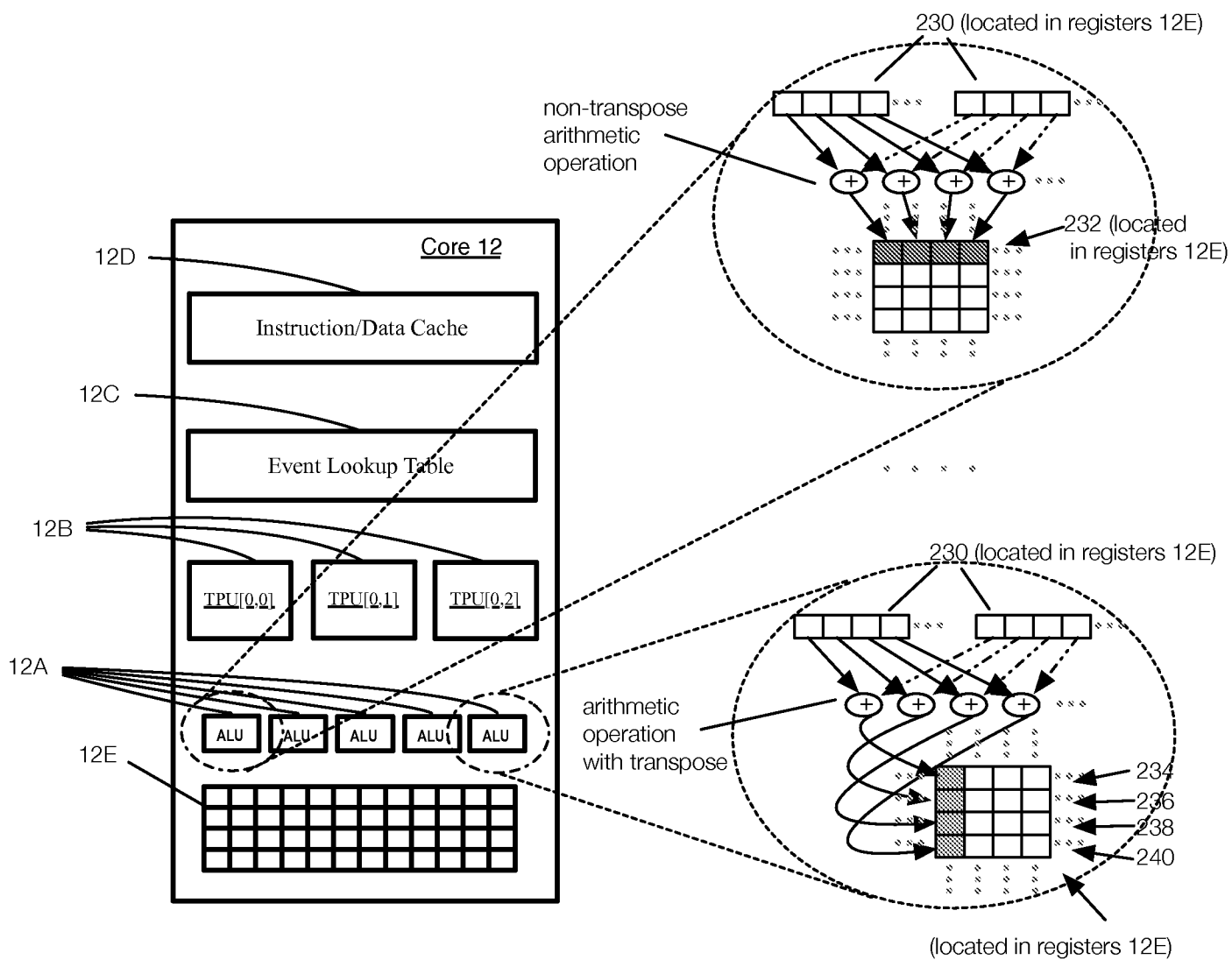


Figure 43

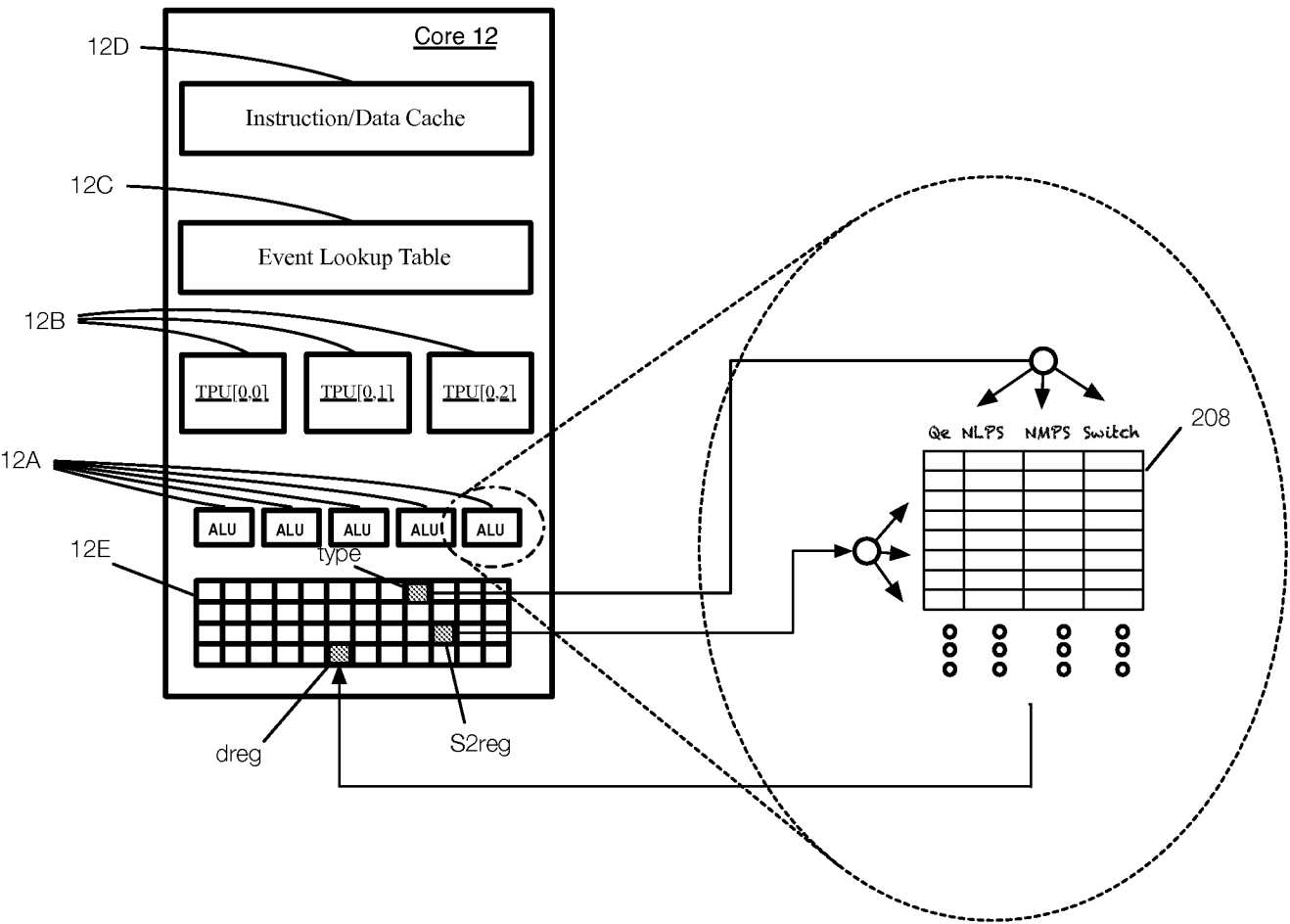


Figure 44

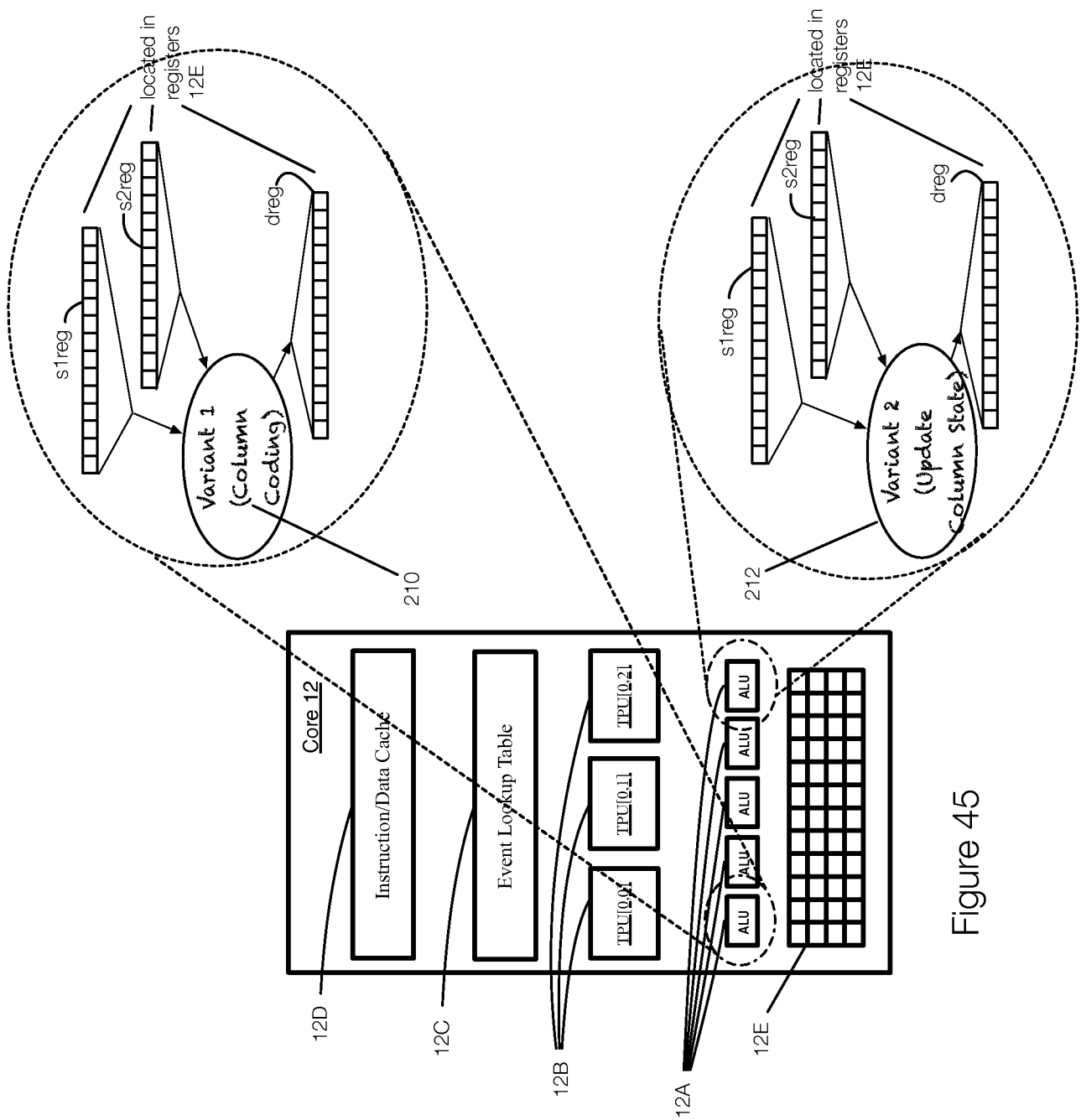


Figure 45

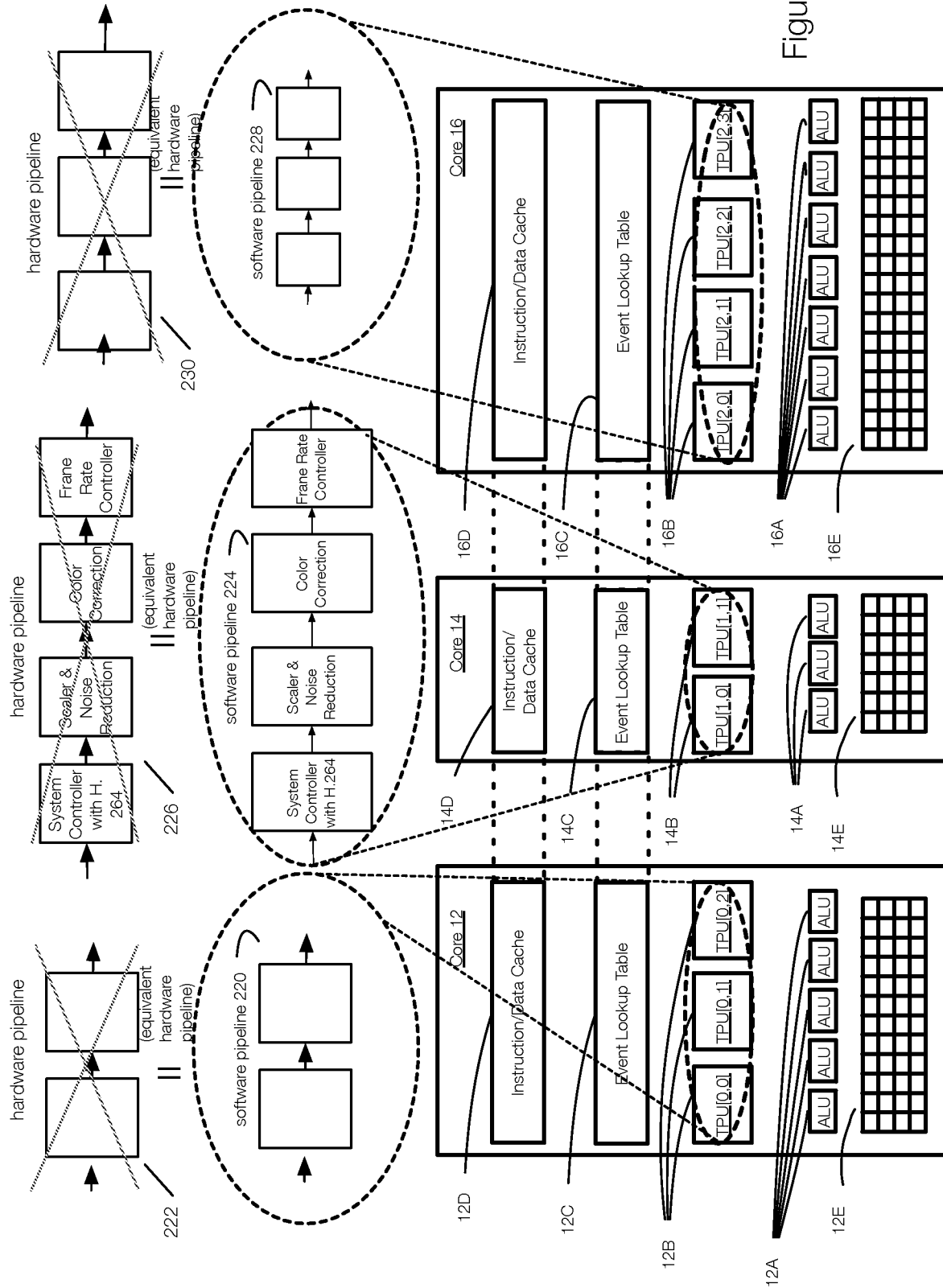


Figure 46

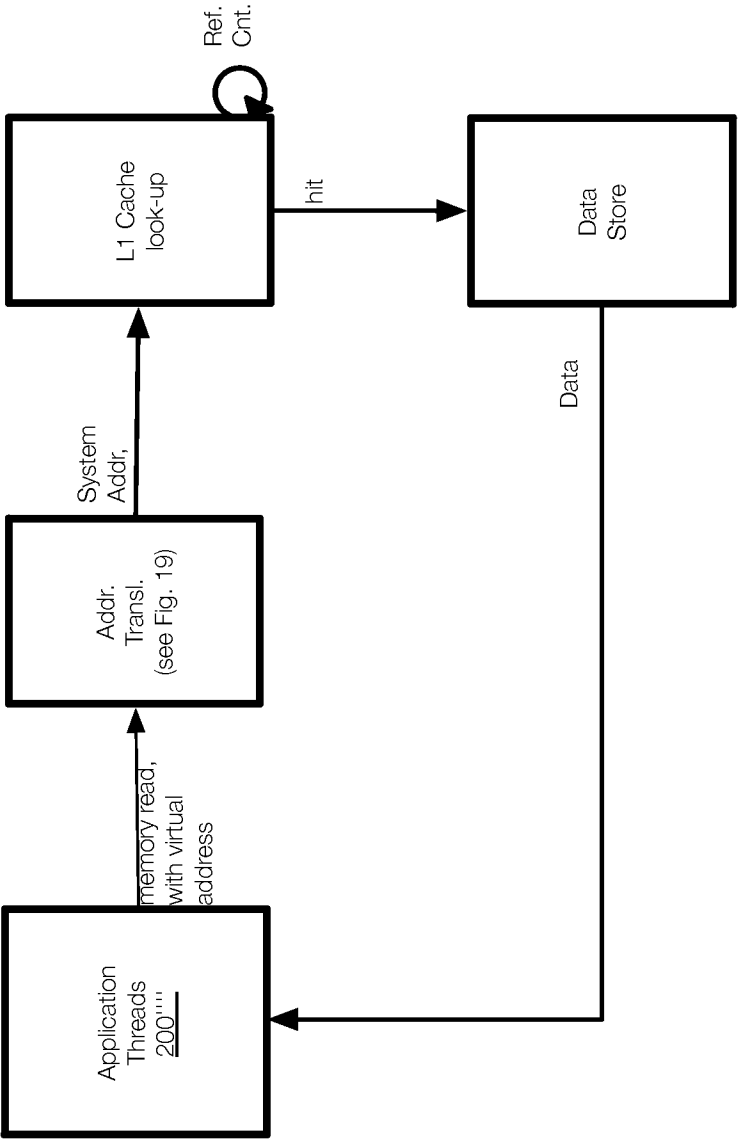


Figure 47

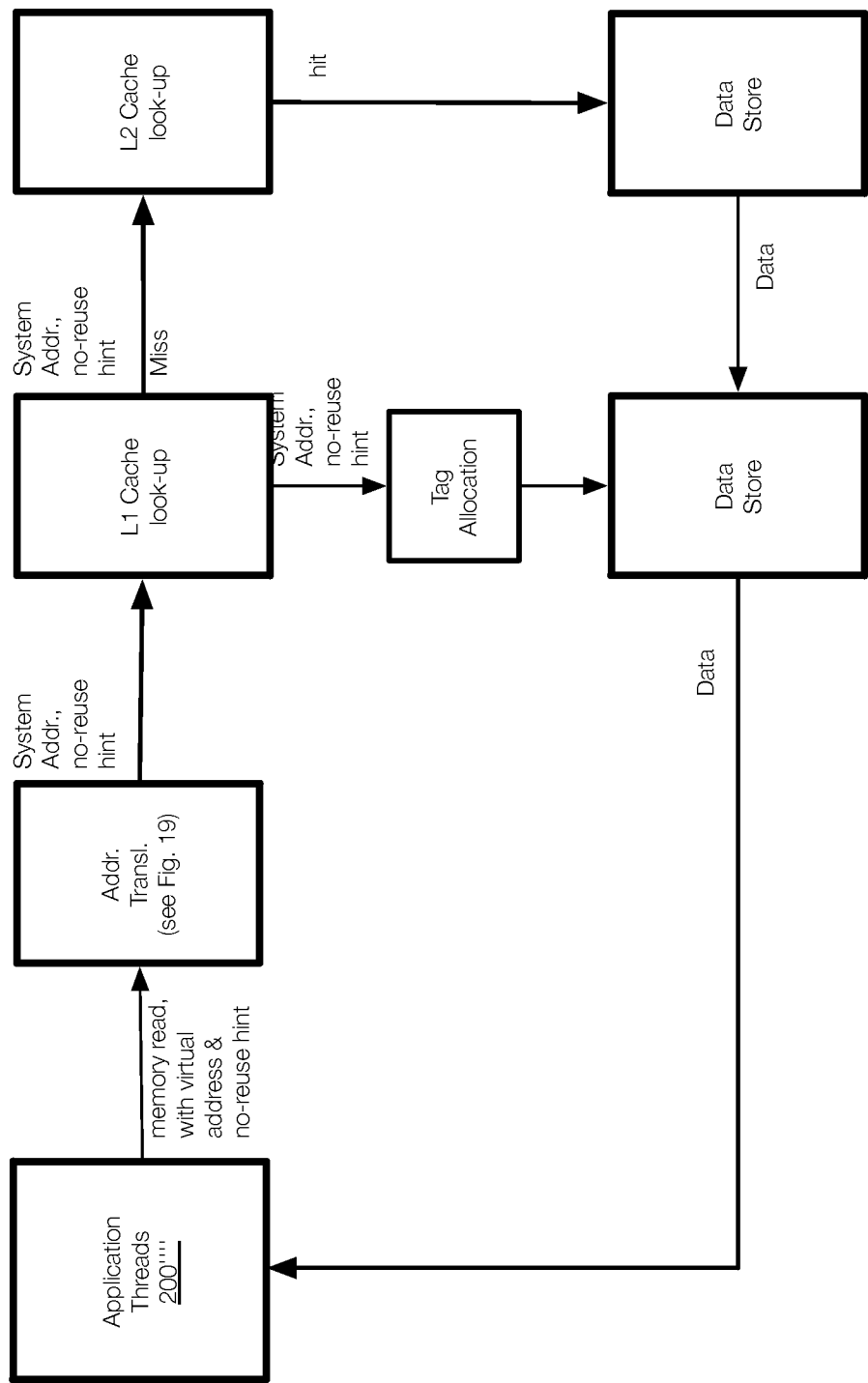


Figure 48

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US 12/42274

A. CLASSIFICATION OF SUBJECT MATTER

IPC(8) - G06F 12/00 (2012.01)

USPC - 711/147

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)

USPC: 711/147

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

USPC: 711/100, 118, 148; 710/260; 718/104, 104; 719/318

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

PubWEST (PGPB, USPT, EPAB, JPAB); Google Scholar, Search terms used: cache memory, node, endpoint, tag, accessed data, instruction, physical memory, system memory, status, modified, reference count

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X --- Y	US 2006/0123197 A1 (Dunshea et al.) 08 June 2006 (08.06.2006), para [0026]-[0029], [0033], [0035], [0039], [0050]	1-2, 7-8, and 29 ----- 3-6, 9-28(a), 28(b), and 30-32
Y	US 6,154,816 A (Steely et al.) 28 November 2000 (28.11.2000), col. 14 ln. 19-28, col. 2 ln. 66 - col. 3 ln. 2	11-15, 18-28(a), 28(b), and 32
Y	US 2009/0187713 A1 (Zedlewski et al.) 23 July 2009 (23.07.2009), para [0015], [0089]-[0090]	3-6, 13, and 16-17
Y	US 2001/0003839 A1 (Kondo) 14 June 2001 (14.06.2001), para [0009], [0086]	6 and 17
Y	US 2003/0159001 A1 (Chalmer et al.) 21 August 2003 (21.08.2003), para [0012], [0047]	9-10, 14-15, and 30-31
Y	US 2010/0272439 A1 (Katayama et al.) 28 October 2010 (28.10.2010), para [0028], [0040]	24-25
Y	US 2003/0005211 A1 (Arimilli et al.) 02 January 2003 (02.01.2003), para [0008], [0021]-[0022]	25

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

* 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

17 October 2012 (17.10.2012)

Date of mailing of the international search report

09 NOV 2012

Name and mailing address of the ISA/US

Mail Stop PCT, Attn: ISA/US, Commissioner for Patents
P.O. Box 1450, Alexandria, Virginia 22313-1450

Facsimile No. 571-273-3201

Authorized officer:

Lee W. Young

PCT Helpdesk: 571-272-4300
PCT OSP: 571-272-7774

INTERNATIONAL SEARCH REPORT

International application No.

PCT/US 12/42274

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

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

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the international application that do not comply with the prescribed requirements to such an extent that no meaningful international search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

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

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

1. ☐ As all required additional search fees were timely paid by the applicant, this international search report covers all searchable claims.
2. ☐ As all searchable claims could be searched without effort justifying additional fees, this Authority did not invite payment of additional fees.
3. ☐ As only some of the required additional search fees were timely paid by the applicant, this international search report covers only those claims for which fees were paid, specifically claims Nos.:
4. ☒ No required additional search fees were timely paid by the applicant. Consequently, this international search report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:
Claims 1-32

Remark on Protest

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

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

Reasons for Lack of Unity of Invention:

This application contains the following inventions or groups of inventions that are not so linked as to form a single general inventive concept under PCT Rule 13.1. In order for all inventions to be searched, the appropriate additional search fees must be paid.

Group 1: Claims 1-32 directed to a Digital Data Processor system/method with cache-managed memory.

Group 2: Claims 33-52 directed to a Digital Data Processor system/method with dynamic assignment of events to threads using an event table.

Group 3: Claims 53-70 directed to a Digital Data Processor system/method with location-independent shared execution environment.

Group 4: Claims 71-94 directed to a Digital Data Processor system/method with provision of quality of service through thread instantiation, maintenance and optimization.

Group 5: Claims 95-110 directed to a Digital Data Processor/method with bit plane stripe column encoding within JPEG2094/JPEG2102 embedded block coding with optimized truncation (EBCOT).

Group 6: Claims 111-128 directed to a Digital Data Processor/method with JPEG2000/JPEG2009 binary arithmetic code lookup.

Group 7: Claims 129-146 directed to a Digital Data Processor/method with arithmetic operation transpose parameter.

Group 8: Claims 147-166 directed to a Digital Data Processor/method with cache control instruction set and cache-initiated optimization.

Group 9: Claims 167-190 directed to a Digital Data Processor and Digital Data Processing system/method executing a pipeline of software components that replace a like pipeline of hardware components.

The inventions listed as Groups 1-9 do not relate to a single general inventive concept under PCT Rule 13.1 because, under PCT Rule 13.2, they lack the same or corresponding special technical features for the following reasons:

The special technical feature of the Group 1 invention is directed to a Digital Data Processor system/method with cache-managed memory.

The special technical feature of the Group 2 invention is directed to a Digital Data Processor system/method with dynamic assignment of events to threads using an event table.

The special technical feature of the Group 3 invention is directed to a Digital Data Processor system/method with location-independent shared execution environment.

The special technical feature of the Group 4 invention is directed to a Digital Data Processor system/method with provision of quality of service through thread instantiation, maintenance and optimization.

The special technical feature of the Group 5 invention is directed to a Digital Data Processor/method with bit plane stripe column encoding within JPEG2094/JPEG2102 embedded block coding with optimized truncation (EBCOT).

The special technical feature of the Group 6 invention is directed to a Digital Data Processor/method with JPEG2000/JPEG2009 binary arithmetic code lookup.

The special technical feature of the Group 7 invention is directed to a Digital Data Processor/method with arithmetic operation transpose parameter.

The special technical feature of the Group 8 invention is directed to a Digital Data Processor/method with cache control instruction set and cache-initiated optimization.

The special technical feature of the Group 9 invention is directed to a Digital Data Processor and Digital Data Processing system/method executing a pipeline of software components that replace a like pipeline of hardware components.

The only common technical feature shared by the above groups is digital data processor.

The common technical feature does not represent an improvement over the prior art of US 7,653,912 B2 to Frank et al. Published on 26 January 2010 (26.01.2010) that discloses a digital data processor in Abstract, col 3, ln 36-60. As the above technical feature of digital data processor as taught by Frank et al was known, this cannot be considered a special technical feature that would otherwise unify the groups.

Therefore, the inventions of Groups 1-9 lack unity of invention under PCT Rule 13 because they do not share a same or corresponding special technical feature.