Bulldozer: An Approach to Multithreaded Compute Performance

AMD's Bulldozer module represents a new direction in microarchitecture and includes a number of firsts for AMD, including AMD's multithreaded x86 processor, implementation of a shared Level 2 cache, and x86 processor to incorporate floating-point multiply-accumulate (FMAC). This article discusses the module's multithreading architecture, power-efficient microarchitecture, and subblocks, including the various microarchitectural latencies, bandwidths, and structure sizes.

Michael Butler
Leslie Barnes
Debjit Das Sarma
Bob Gelinas
Advanced Micro Devices

Advanced Micro Devices' Bulldozer module is the core building block for future AMD client and server systems on a chip (SoCs) designed for mainstream and high-performance processor markets. It combines two independent cores intended to deliver high per-thread throughput with improved area and power efficiency. A monolithic building block, the Bulldozer module can execute two threads via a combination of shared and dedicated resources. It was designed to deliver superior high-performance, general-purpose thread throughput at a given power budget. From the software point of view, the module appears as two fully capable, independent cores.

Several basic observations motivated AMD's design of the module microarchitecture. First, future SoCs would always support multiple execution threads. Even entry-level client chips were anticipated to support at least two to four threads. This assumption led AMD to explore opportunities to improve efficiency by sharing resources in the smallest possible building block. In many cases, however, sharing hardware among multiple threads influences timing and complexity of critical hardware paths. For these reasons, Bulldozer shares hardware if it's affordable and profitable (such as front-end and floating-point unit [FPU]) but replicates hardware as necessary for timing and complexity reasons (such as the integer execution core).

The second observation was that the core would always operate in a power-constrained environment. This significantly impacted the project goals and the Bulldozer module's characteristics. Power efficiency was a critical design characteristic from Bulldozer's inception. The module employs various power-reduction techniques—such as filtering, speculation reduction, and data movement minimization—to produce an inherently

Published by the IEEE Computer Society

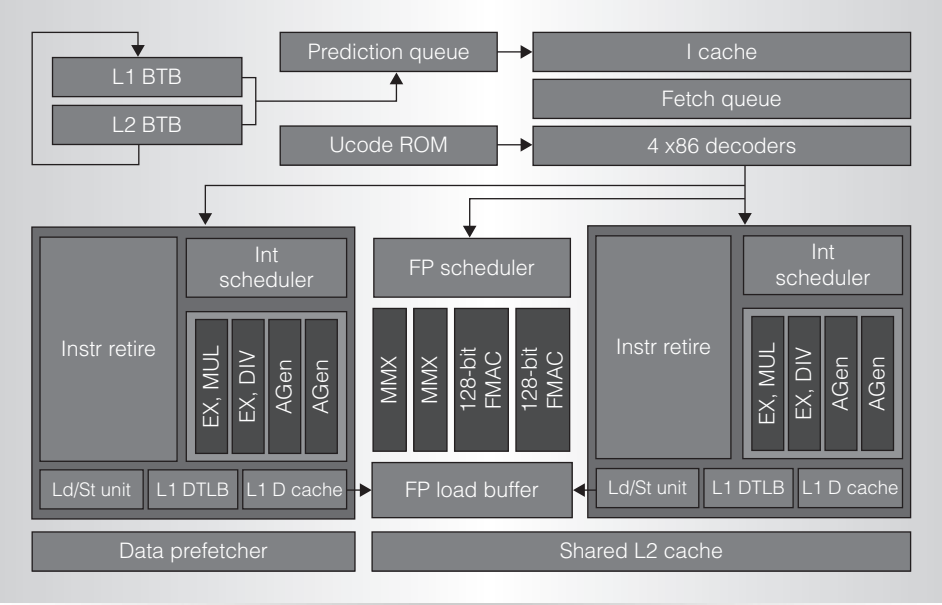

Figure 1. High-level block diagram of the Bulldozer module. Bulldozer can fetch and decode four x86 instructions per clock. (BTB: branch target buffer; I cache: instruction cache; D cache: data cache; Ucode: microcode; DTLB: data translation look-aside buffer; AGen: address generation unit; MMX: MMX and packed integer data path; FMAC: floating-point multiply-accumulate; L1: Level 1; L2: Level 2.)

power-efficient design. On this underlying efficiency, Bulldozer supports extensive dynamic power-management techniques. With aggressive peak bandwidth and throughput features, active management of peak power down to average application power levels instead of worst-case power virus levels is critical. This intentionally allows a larger spread between peak power (that is, peak performance burst) and the average power consumed by typical activity.

While high-throughput performance was a primary goal for Bulldozer, AMD made a significant investment in delivering high, single-thread performance levels. A major contributor to this strategy is in scaling the core structures and an aggressive frequency goal (low gates per clock).[1] Another major component of the single-thread performance strategy is Bulldozer's investment in instruction and data prefetching.

Block diagram

Figure 1 shows a block diagram of the Bulldozer module. The module can fetch and decode up to four x86 instructions per clock. Instruction fetch, branch prediction, and decode make up the module's front-end, which the two active threads share (see the top half of Figure 1). An instruction lifetime begins with the branch prediction pipeline supplying the predicted stream of addresses to be fetched from the instruction cache. Instruction bytes are routed to four dedicated decoders that crack the x86 instructions and compute their actual length. The decoders either directly produce internal-format complex micro-operations (or "Cops") corresponding to the x86 instructions, or they identify the microcode entry point for indexing into the microcode ROM. Operations, either from the microcode engine or from the decoders, are written into the micro-operations queue, from which they can be selected for dispatch. The set of Cops that dispatch together are called a dispatch group and will always belong to a single thread (that is, they will all go to one of the two integer cores as well as the FPU if there are floating-point operations in the dispatch group).

Operations spend the remainder of their lifetime under the integer cores' control. The integer execution unit renames and

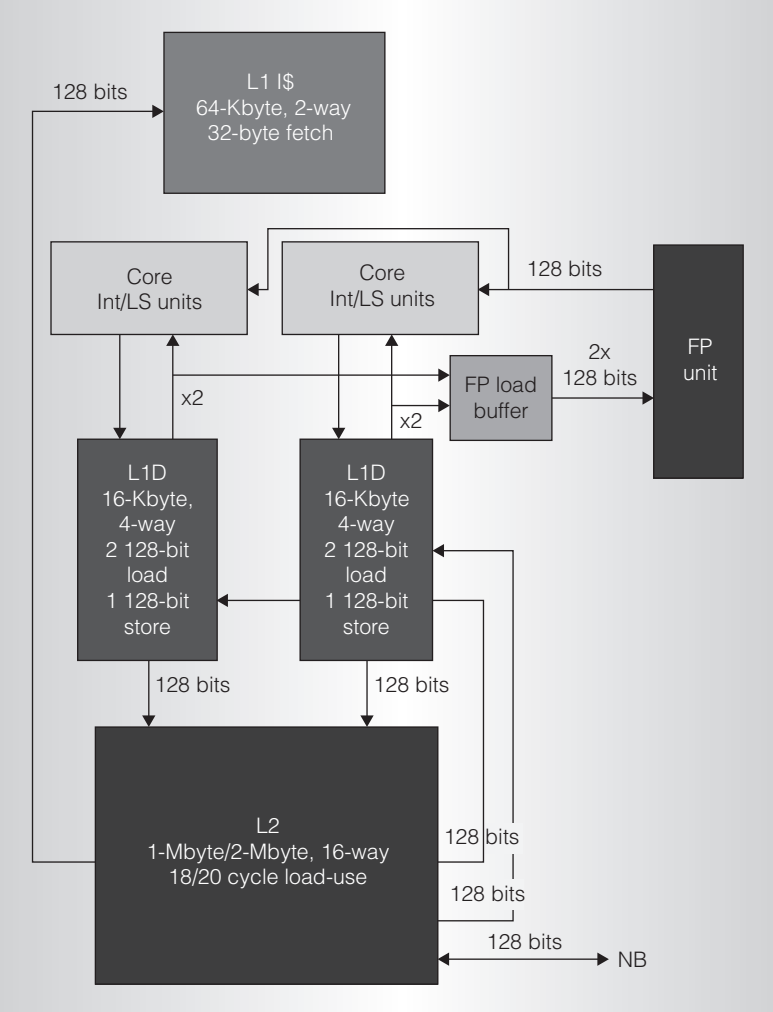

Figure 2. Cache hierarchy of the Bulldozer module. A shared 64-Kbyte L1 instruction cache provides instructions for both threads and is backed by the shared unified L2 cache. Dedicated 16-Kbyte L1 data caches provide data to their associated core or to the shared floating-point unit via the FP load buffer. (I$: instruction cache.)

executes integer operations and address-generation operations. If the dispatch group contains floating-point operations, the FPU must translate and rename the operands as well as execute these floating-point operations.

The integer cores are dedicated one per thread and consist of the integer execution units, Level 1 (L1) data cache, load/store unit, and instruction retirement logic. In a core, instruction execution is dependency driven, allowing full out-of-order scheduling and execution. One critical characteristic of the integer core is the power-efficient physical register file (PRF)-based renaming and operand delivery mechanism. The unified scheduler queue represents a significant increase in the scheduling window's size and scalability.

The integer execution units comprise four execution pipelines: two conventional arithmetic logic units (ALUs, denoted by "EX" in Figure 1) and two address-generation units ("AGen" in Figure 1) capable of performing address-generation and simple increment and decrement operations.

All loads and stores for a given thread (including floating-point loads and stores) are performed in the associated core to its private L1 data cache. Each core can service two loads per cycle, in any combination of 64-bit integer or floating-point loads or 128-bit floating-point loads. In the case of floating-point operations, this peak of four loads is multiplexed down to two loads that are written into the floating-point register file per cycle. A floating-point load buffer smoothes out the bandwidth mismatch and applies back-pressure to the sourcing load/store units. This sustained bandwidth of two 128-bit loads per cycle matches the dispatch load bandwidth. The L1 data cache is write-through and is backed by a unified L2 write-back cache. This L2 cache is coupled tightly to the module and shared between the two execution cores. This block also serves as the interface unit for all chip-level communication. Figure 2 shows the complete cache hierarchy, and Figure 3 shows the instruction and data translation lookaside buffers (TLB) hierarchies.

The TLB hierarchy is well provisioned for large-footprint server workloads. On the instruction side, the L1 instruction TLB (ITLB) uses fully associative structures, including a 24-entry table for large, 2-Mbyte or 1-Gbyte pages. The L2 ITLB is a low-latency structure for 4-Kbyte pages only. The ITLB structures are shared competitively between threads.

On the data side, the L1 data TLB is a 32-entry fully associated structure that can hold 4-Kbyte, 2-Mbyte, or 1-Gbyte pages and is thread dedicated. The L2 data TLB is a large, 1,024-entry, eight-way structure that can hold 4-Kbyte, 2-Mbyte, or 1-Gbyte pages and is shared competitively

between the threads. The L2 data TLB is highly optimized for virtualized workloads and can effectively cache various intermediate translations found in nested-paging scenarios.

Key features and motivation

In this section, we describe several key features of the Bulldozer module and the rationale behind the direction AMD adopted.

Multithreading microarchitecture

Each Bulldozer module supports two execution threads to deliver compelling throughput/watt and throughput/area relative to both chip multiprocessing (CMP) and simultaneous multithreading (SMT). Through the appropriate use of replication and shared hardware, the design balances high throughput and attractive area and power. In addition, Bulldozer is designed to achieve robust performance from a software viewpoint—threads receive dedicated integer execution resources and data cache. This was done to avoid the negative performance interference that can occur when threads share execution resources and the data cache.

From the thread's viewpoint, it runs on a complete core even though some parts of the machine are shared. Three main types of support exist for multithreading: none (that is, single threaded with replicated hardware), vertical multithreading,[2,3] and SMT.[4-7]

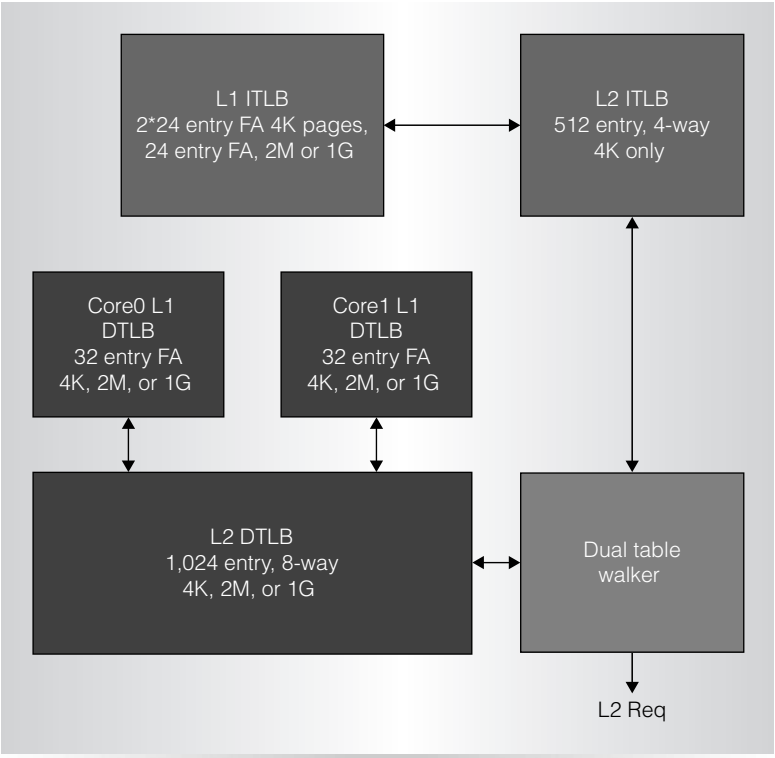

Figure 3. Instruction and data translation look-aside buffer (TLB) hierarchy. The instruction TLB (ITLB) hierarchy consists of a fully associative L1 backed by large L2 TLB, both shared between the two threads. Dedicated L1 data TLBs are backed by a large shared L2 DTLB and dual-table walk state machines.

Figure 4 shows how the Bulldozer core uses these different mechanisms. Each block in Figure 4 represents a change in multithreading support or thread domain.

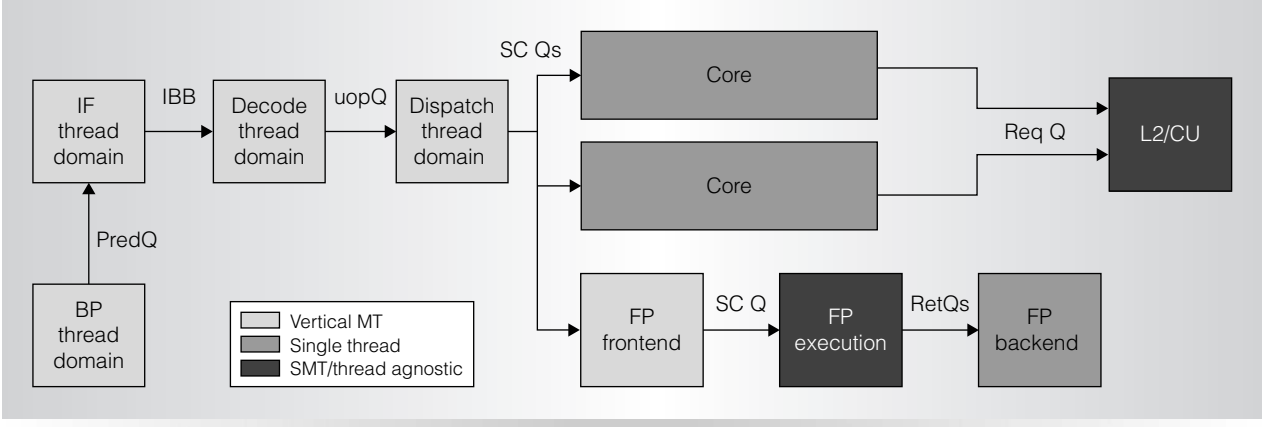

Figure 4. Multithreading model that shows how the Bulldozer core uses different mechanisms. Each block represents a change in multithreading support or thread domain. (PredQ: prediction queue; IBB: fetch queue; uopQ: micro-operations queue; SC Q: scheduler queue; RetQ: retire queue; ReqQ: request queue.)

The front-end supports multiple threads via vertical multithreading. The main advantage to sharing instruction cache and branch prediction structures is that when a single thread is running, it has access to larger structures than it otherwise would have if a given area and power budget was hard-partitioned to threads. In addition, excess fetch bandwidth often exists in a single-threaded front-end (because of back-end stalls, for example), which implies that the total fetch bandwidth can be more efficiently used by two threads.

Each block indicates a local pipeline that forms a different thread switch domain. Once a thread switch decision has been made at the start of the local pipeline, the decision propagates the pipeline's length. A thread switch can occur as often as every cycle, so multiple threads can be in flight in the pipelines, but never in the same pipestage. Decoupling queues, which exist for the normal pipelining of the front-end, serve as the different domains' boundaries. Conceptually, the FPU coprocessor's front end is an extension of the dispatch thread domain.

Integer cores are replicated, so we can consider them to be single-threaded. The primary motivation is to avoid introducing complexity, scalability, and timing problems into the integer execution engine. Each core is unaware that the other core or thread exists. The core hardware can stay lean by supporting execution resources and bandwidth for a single thread, instead of scaling up to cover SMT throughput. As a result, the core remains small and enables a higher-frequency design.

Because the floating-point execution units are so large, Bulldozer shares them between the two threads via SMT. The back-end FPU employs replication to handle retirement queuing and the interface to the independent cores.

The L2 cache is shared between the cores and thus deals with both threads simultaneously, although it's almost entirely thread-agnostic. Sharing the L2 cache can be advantageous from an area or power perspective or for workloads with shared instruction and data images. The L2 cache's large capacity and high associativity makes destructive interference between the two threads less likely.

The software view of a core is that of a stand-alone processor, as in CMP systems. There are no separate single-threaded or multithreaded operation modes. Both threads are enabled, and one thread's idleness dynamically lets the other thread access the full bandwidth of shared hardware, leading to excellent latency-hiding and frequency-scaling properties for the whole module.

Dynamic power management

A major component of Bulldozer's strategy for achieving high performance at a given thermal design power (TDP) is the use of dynamic power management to manage application power. Actively monitoring and throttling power enables average application power to be closer to TDP, with a corresponding performance increase. Rather than reserving a large power margin to cover the difference between average and peak power, active management lets the core exploit its power budget. A key feature of this event-based monitoring and response mechanism is its deterministic behavior and highly tunable design (via programmable weights and thresholds).

This mechanism is similar to power management mechanisms used in previous-generation AMD products.[1]

Decoupled branch-prediction and instruction-fetch pipelines

Decoupling the branch-prediction pipeline from the instruction-fetch pipeline[8] achieves several significant benefits (see Figure 5). The first advantage is the relaxation of several difficult timing paths from the next-fetch generation logic in the branch predictor and instruction fetch. Because the front end is vertically multithreaded, a second benefit of decoupling is the overlap of structural or timing-induced bubbles in a single-thread fetch stream with productive fetches for the other thread. The final and most significant benefit is the enablement of instruction prefetch using the prediction queue.

Because the control-flow footprint captured by the multilevel branch target buffers (BTBs) significantly exceeds the instruction footprint that the L1 instruction cache can capture, multiple instruction-cache misses

can be overlapped or completely hidden. Thread-dedicated queues separate the branch-prediction and instruction-fetch activities, allowing discovery of future instruction fetch addresses while hiding the actual branch-prediction latency. The only back-pressure on branch prediction is a full prediction queue, so branch prediction can run ahead of actual instruction fetching. This provides a series of fetch targets that the instruction cache controller can inspect ahead of the demand requests. In the shadow of a cache miss associated with a demand fetch, the cache examines any following entries in the prediction queue and can initiate a target's prefetch if it's not already present in the instruction cache. In this manner, several instruction-cache misses can be hidden in the shadow of the first miss.

Register renaming and operand delivery

The primary motivation for adopting a PRF-based renaming microarchitecture is power efficiency. The single biggest power consumer in the integer execution unit is the scheduler and operand-handling mechanism. PRF-based renaming improves power efficiency in two ways. First, it eliminates data replication by storing all operands in the PRF instead of distributed reservation stations. Second, it can physically separate dependency tracking (wake up) from data storage, easing timing pressure and allowing better scaling to larger scheduler queue sizes.

FMAC and media extensions

Bulldozer implements a significant extension to the x86 architecture that introduces a set of three source-operand, nondestructive instructions including floating-point multiply-accumulate (FMAC)[9] as well as other media instructions.[10] In support of these ISA extensions, Bulldozer has made a sizable investment in dual 128-byte FMAC units. These execution units deliver significant peak execution bandwidth while amortizing the area cost across two threads.

Function block highlights

The Bulldozer module is organized in various functional blocks, each designed to deliver key capabilities and features.

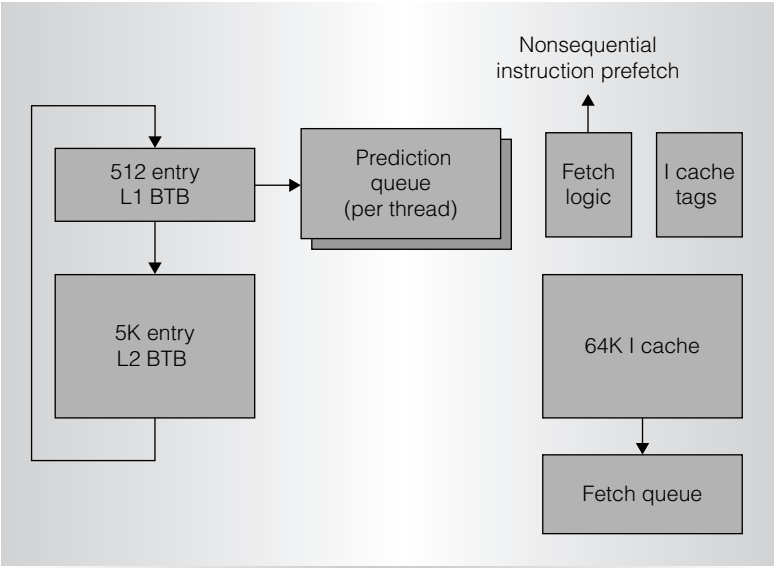

Figure 5. Prediction-directed instruction prefetch. The branch prediction pipeline is allowed to run ahead of the instruction fetch pipeline during fetch misses or other stalls. Following an instruction cache miss, future fetch addresses are looked up in the tags and subsequent miss requests are overlapped with the initial request providing an effective instruction prefetch mechanism.

Table 1. Major branch prediction structures.

Structure	Size and organization
L1 branch target buffer	512-entry, four-way set associative
L2 branch target buffer	5,120-entry, five-way set associative

Branch prediction

The Bulldozer branch prediction unit implements several innovative features. The large prediction structures are shared between two threads when both threads are running, but can be used fully by a single thread when only one thread is running.

Bulldozer supports several server-oriented features (such as incorporation of an indirect target array) and the use of multilevel BTBs, which better capture large footprint-control flow changes. Table 1 summarizes the prediction structure sizes.

Instruction cache

Bulldozer's instruction cache is a 64-Kbyte, two-way set-associative cache shared between both threads. Fetch addresses are read from the prediction queue and applied

Table 2. Execution unit result buses and function units.

Result bus	Execution units	Load/store interaction
EX0	ALU0, Shifter0, DIV	Store data bus to load/store
EX1	ALU1, Shifter1, Branch, MUL	Store data bus to load/store
AGLU0	AGLU0 (Address generation + simple INC/DEC Cops)	Address-generation bus to load/store Load data bus from load/store
AGLU1	AGLU1 (Address generation + simple INC/DEC Cops)	Address-generation bus to load/store Load data bus from load/store

* EX: execution; AGLU: address-generation logic unit; ALU: arithmetic logic unit; INC: increment; DEC: decrement; Cops: complex micro-operations.

to the instruction cache, fetching 32 bytes of instruction data plus associated pre-decode bits per cycle. This high fetch bandwidth supplies the needed single-threaded instruction bandwidth in higher IPC situations with long (6 or 7 bytes) instructions found, for example, in advanced vector extension (AVX) workloads.

The front-end uses multilevel structures such as L1 and L2 TLBs or BTBs to balance cycle time and capacity requirements. Various trade-offs are associated with the use of multilevel structures to optimize for the competing goals of latency and power efficiency.

Decode

The decode unit continues support for multithreading in the Bulldozer module. Fetch lines from the instruction cache are queued in an instruction byte buffer that feeds a vertically multithreaded decode pipeline. The decode unit supports many new instructions and formats, including SSE4.1, SSE4.2, and the new AVX and advanced encryption standard (AES) instruction sets. In addition, a set of AMD-defined four-operand FMAC instructions are provided, along with other instruction set extensions such as AMD Lightweight Profiling for low-overhead, user-level program profiling.

Bulldozer's decode unit extracts and decodes up to four x86 instructions per cycle from raw instruction bytes. The decode pipeline converts x86 instructions into Cops that can directly execute on the functional units. Typically, there's a one-to-one mapping from x86 instructions into internal Cops (fastpath single), but some instructions require two Cops (fastpath double), and more complex instructions map into long sequences of Cops. The dispatch interface to the cores and FPU is four Cops wide.

While supporting many new instructions, the Bulldozer core also introduces macro-instruction fusing capabilities to AMD processors. Conditional branches are often preceded immediately by the compare instructions that produce their source flags. Bulldozer detects and merges this sequence of macro-instructions into a single Cop that performs the comparison and evaluates the branch direction. The single Cop occupies a single scheduler queue entry and executes with lower latency than a sequence of dependent Cops.

Decoded instructions can dispatch to one integer core simultaneously with the FPU. Only one thread dispatches at a time from separate Cop queues. The queues aid in latency hiding by letting younger instructions be fetched and decoded as older instructions are being processed.

Write-port limitations in various queues throughout the module result in static restrictions on which operations can be combined into a dispatch group. Also, certain resource availability requirements can dynamically impose dispatch restrictions.

Integer scheduler and execution

The integer scheduler and execution unit control the out-of-order integer execution and coordinate retirement of all instructions. Operand handling is accomplished via a PRF, and PRF indices are used to encode register and flag dependencies in the wake-up logic. The integer data path comprises four result buses driven by four primary execution units (see Table 2).[11]

Table 3 indicates the size of several major execution structures. The scheduler picks and schedules four Cops per cycle to the execution units out of order. Additionally, it handles branch misprediction in a fully out-of-order fashion. The scheduler is sized to cover the active out-of-order window from dispatch through execution. In contrast, the Retire queue holds operations from dispatch through in-order instruction retirement and thus needs greater capacity.[12]

Table 3. Major execution unit structures.

Structure (per thread)	Size
Scheduler	40-entry
Physical register file	96-entry
Retire queue	128-entry

Table 4. Major load/store structures.

Structure	Size and organization
Data cache	16-Kbyte, four-way set associative
L1 data TLB	32-entry, fully associative, any page size
Load queue	40-entry
Store queue	24-entry

Table 5. Floating-point result buses and function units.

Result bus	Execution units
Pipe0	FMAC, IMAC, CVT, AES
Pipe1	FMAC, XBAR
Pipe2	Packed integer ALU
Pipe3	Packed integer ALU, Store

* FMAC: floating-point multiply-accumulate; IMAC: integer multiply-accumulate; CVT: convert; AES: advanced encryption standard; XBAR: 128-bit permute/shift.

Load/store

The load/store unit completes all memory accesses once address generation has occurred in the execution. The L1 data cache is a 16-Kbyte, way-predicted, write-through cache designed to support up to two 128-byte loads per cycle. In addition, one 128-byte store can be committed to the cache. Load-use latency is four cycles, and loads proceed in a fully out-of-order fashion.

The load/store unit also contains conventional stride-prefetch logic, installing strided data into the L1 data cache in a timely fashion. Special care is taken to assure only useful data is prefetched to avoid evicting needed demand lines from the data cache. Table 4 details the sizes of various load/store structures.

Floating point

AMD designed the Bulldozer FPU to deliver industry-leading performance on HPC, multimedia, and gaming applications. The primary means of achieving such performance is a four-wide, two-way, multithreaded, fully out-of-order FPU, combined with two 128-bit FMAC units supported by a 128-bit high-bandwidth load/store subsystem. The FPU supports the 64-bit AMD64 ISA in addition to bringing a number of advanced extensions to the AMD roadmap (along with those we've already discussed, XOP, including four-operand FMAC, integer multiply-accumulate [IMAC], and Permute instructions).

The FPU is a coprocessor model shared between two integer cores via two-way multithreading. The FPU has its own out-of-order engine along with the execution units and register file, and interfaces with the DE to receive Cops, the load/store unit to receive and send load/store data, and the integer cores' retire unit to handshake on completion and retire.

The major execution units are two 128-bit FMAC units, two 128-bit packed integer (MMX ALU) units, one 128-bit IMAC unit, and one 128-bit permute/shift (XBAR) unit. Table 5 details the floating-point result buses and function units.

The FMAC unit implements FMAC operations with one rounding stage in a fully pipelined implementation. The FMAC unit also executes pure FADD (floating-point add) and FMUL (floating-point multiply) operations. FMAC-based divide and square-root operations are implemented with a state machine in the FMAC unit and aren't fully pipelined, but are nonblocking.

Table 6. Major floating-point structures.

Structure (per thread)	Size
Scheduler	60-entry
Physical register file	160-entry
Retire queue	128-entry (per thread)

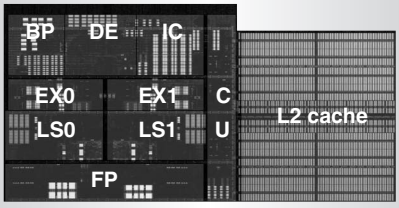

Figure 6. Bulldozer module with L2 cache die photo. (BP: branch prediction; DE: decode; EX: execution; IC: instruction cache; CU: cache unit; LS: load/store unit.) The shared front-end (BP, DE, IC) is shown along the top of the module with replicated cores (EX + LS) below. The shared floating-point (FP) unit resides at the bottom of the module with the cache unit and shared 2-Mbyte L2 cache on the right.

The FPU employs various microarchitectural and arithmetic techniques to achieve high performance and power efficiency. Some of these techniques include FMAC unrounded bypass in mixed precisions, sharing the double-precision FMAC hardware to execute pair-wise single-precision FMAC, and using power-efficient methods for executing FADD and FMUL in the FMAC unit. In addition, the FPU implements FMAC-based divide, square root, and transcendental instructions, novel FMAC-based rounding techniques for divide and square root, fast conversion algorithms between floating-point and integers, fast-packed IMAC algorithms, and hardware-assisted fast encryption support. Table 6 indicates the size for several major floating-point structures.

L2 cache

The two cores share the unified L2 cache and associated logic. The cache forms the point of global visibility for the memory references from the associated cores. It's mostly inclusive with the L1 data caches and has an 18-cycle load-use latency for the 1-Mbyte variant and 20 cycles for the 2-Mbyte variant. This unit also serves as the interface to the NorthBridge and other chip-level functions.

A major performance-enhancing feature of the unified L2 cache is hardware data prefetch. It implements two data prefetchers: strided and nonstrided. The nonstrided prefetcher detects correlated accesses within regions of addresses and has proven extremely effective for difficult-to-prefetch address streams found, for example, in server workloads. The more conventional stride prefetcher is built as an extension of the nonstrided prefetcher and can operate concurrently. The unified L2 cache-based stride prefetcher works cooperatively with the L1-based stride prefetcher to achieve higher performance. The unified L2 cache-based prefetchers are designed for robust performance when there's a surplus of memory bandwidth, but they contain a back-off mechanism for when demand traffic is heavy.

The unified L2 cache also contains the L2 data TLB and associated table-walker logic. Multiple table walks can be initiated in parallel to effectively support the multithreaded architecture.

Bulldozer-based SoC

The initial AMD products built with the Bulldozer module will be desktop and server SoCs. These SoCs are drop-in replacements for AMD's existing SoCs and deliver a significant performance improvement in the same power envelope as the company's existing products. Figure 6 shows a die photo of the Bulldozer module.

The Bulldozer module is intended to form the basis for AMD's family of future high-performance mainstream client and server processor offerings. This new microarchitecture represents the first generation in a planned roadmap of core improvements designed to support many generations of client and server processors including upcoming fusion APUs.

Extensibility and process scalability were key design goals for enabling this future product roadmap. MICRO

Acknowledgments

Bulldozer represents the combined effort of many talented AMD engineers across multiple locations during several years, including Sunnyvale, Calif.; Boston; Fort Collins, Colo.; Austin, Texas; and Bangalore, India.

References

1. T Fischer et al., ''Design Solutions for the Bulldozer 32-nm SOI 2-Core Processor Module in an 8-Core CPU,'' *IEEE Int'l Solid State Circuits Conf.,* IEEE Press, 2011.
2. P. Kongetira, K. Aingaran, and K. Olukotun, ''Niagara: A 32-Way Multithreaded SPARC Processor,'' *IEEE Micro,* vol. 25, no. 2, 2005, pp. 21-29.
3. J.D. Davis, J. Laudon, and K. Olukotun, ''Maximizing CMP Throughput with Mediocre Cores,'' *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques,* IEEE CS Press, 2005, pp. 51-62.
4. ''Hyper-Threading Technology,'' *Intel Tech. J.,* vol. 6, no. 1, 2002, pp. 4-15.
5. H.M. Mathis et al., ''Characterization of Simultaneous Multithreading (SMT) Efficiency in Power5,'' *IBM J. Research and Development,* Jul.-Sep. 2005, pp. 555-564.
6. J. Emer, ''Simultaneous Multithreading: Multiplying Alpha Performance,'' *Proc. Microprocessor Forum,* Linley Group, 1999.
7. D. Tullsen, S. Eggers, and H. Levy, ''Simultaneous Multithreading: Maximizing On-Chip Parallelism,'' *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* ACM Press, 1995, pp. 392-403.
8. G. Reinman, B. Calder, and T. Austin, ''Optimizations Enabled by a Decoupled Front-End Architecture,'' *IEEE Trans. Computers,* vol. 50, no. 4, 2001, pp. 338-355.
9. R. Jotwani et al., ''An x86, 64-Core Implemented in 32-nm SOI CMOS,'' *IEEE Int'l Solid State Circuits Conf.* IEEE Press, 2010, pp. 106-107.
10. *AMD64 Architecture Programmers Manual Volume 6: 128-Bit and 256-Bit XOP and FMA4 Instructions,* AMD, 2009.
11. M. Golden et al., ''40-Entry Unified Out-of-Order Integer Execution Unit for the AMD Bulldozer x86-64 Core,'' *IEEE Int'l Solid State Circuits Conf.,* IEEE Press, 2011, pp. 80-81.
12. R.K. Montoye, E. Hokenek, and S.L. Runyon, ''Design of the IBM RISC System/6000 Floating Point Execution Unit,'' *IBM J. Research and Development,* vol. 34, 1990, pp. 59-70.

Michael Butler is a fellow and chief architect of the Bulldozer core at Advanced Micro Devices. His research interests include high-performance CPU microarchitectures. Butler has a PhD in computer science and engineering from the University of Michigan, Ann Arbor.

Leslie Barnes is a fellow in the performance, architecture, and modeling group at Advanced Micro Devices. His work has included performance analysis and modeling of the Bulldozer microprocessor. Barnes has a PhD in theoretical chemistry from the University of Western Australia.

Debjit Das Sarma is a fellow and lead architect at Advanced Micro Devices. His work has included microprocessor architecture and design and computer arithmetic. Das Sarma has a PhD in computer science and engineering from Southern Methodist University.

Bob Gelinas is a member of the Bulldozer RTL development team at Advanced Micro Devices. His research interests include cache hierarchy and bus architectures for symmetric multiprocessing and cache-coherent nonuniform memory access systems. Gelinas has a BS in electrical engineering from the University of Maine.

Direct questions and comments about this article to Michael Butler, 1 AMD Pl., Sunnyvale, CA 94088, Mail Stop 363; mike.butler@amd.com.