Bobcat: AMD's Low-Power x86 Processor

Bobcat is an AMD processor core designed for the low-power, mobile, lower-end desktop x86 markets. This core should push current technology in many areas while balancing performance, area, and power consumption. Bobcat supports the 64-bit AMD64 ISA, various SIMD extensions, and a full virtual machine implementation. Bobcat is featured on the AMD Fusion processor family roadmap alongside vector-based parallel processing units in accelerated processing unit configurations.

Brad Burgess
Brad Cohen
Marvin Denman
Jim Dundas
David Kaplan
Jeff Rupley
Advanced Micro Devices

Bobcat, AMD's latest low-power x86 processor core, is designed to meet the special demands of netbook, thin-and-light, small-form-factor, and low-cost PC markets. Bobcat aims to reduce size and power demand while providing excellent performance. Except for a few custom memory arrays, the core is fully synthesized, so it can be ported quickly and efficiently to different process technologies.

Power, area, and performance

High-performance and small or low-power devices are often considered to be at opposite ends of the design spectrum. Finding a balance is a challenge, but several fundamental features helped Bobcat achieve this goal.

- The microarchitecture was developed around a powerful set of complex micro-operations (COPs). A single COP can read from memory, execute an arithmetic logic unit (ALU) operation on the data, and write the result back to memory. Such COPs let Bobcat designers build a two-instruction-wide decode/rename/retire machine, save significant power and area, and achieve excellent performance simultaneously.
- Bobcat's aggressive instruction fetch engine fetches up to 32 bytes per cycle, predicts up to two branches per cycle, and incorporates a highly accurate branch predictor. This enabled designers to keep the rest of the machine full of useful instructions to execute, thereby minimizing performance loss and wasted power going down wrong branch paths.
- The machine's execution portion supports full out-of-order (OoO) instruction execution, including the ability to execute loads and stores out of order. This delivers significant performance improvement and permits better utilization of execution hardware.
- AMD implemented numerous other microarchitectural techniques and tricks throughout the design that save significant power and minimally affect performance. For example, nearly all

Published by the IEEE Computer Society

the queues and register files in the design are pointer-based to minimize unnecessary data copying or movement (compared with shifting-queue or future-file structures). Careful attention to the implementation also contributed significantly to power efficiency.

The Bobcat core uses about one-third the area the earlier K8 architecture would have used if implemented in the same fabrication process.

Feature set

Even though the Bobcat core is both small and power efficient, it supports a rich architectural instruction and feature set, such as

- the 32-bit/64-bit x86 AMD64 instruction set architecture;
- Streaming SIMD Extensions, including SSE1, SSE2, and SSE3, and Supplemental Streaming SIMD Extensions 3 (SSSE3);
- SSE4A and 128-bit misaligned data type extensions introduced in AMD's Barcelona processor;
- AMD-V secure processor virtualization extensions, including rapid virtualization indexing to accelerate guest page table walks;
- instruction-based sampling for dynamic code optimization; and
- the C6 power feature, whereby processor state is saved to memory and the processor is powered down without operating system intervention.

This instruction and feature set provided software compatibility with the Barcelona class processor.

Bobcat microarchitecture

The Bobcat processor is an OoO, dual-decode, dual-issue, dual-retire machine with an advanced branch predictor, two 64-bit integer execution units, two 64-bit address generation units, two 64-bit pipelined floating-point execution stacks, and a fully OoO load/store execution unit. It has a 32-Kbyte instruction cache, a 32-Kbyte data cache, and a 512-Kbyte Level 2 (L2) cache. Figure 1 depicts Bobcat's high-level microarchitecture.

Fetch unit

The Bobcat fetch unit consists of the instruction fetch logic, instruction cache, instruction translation look-aside buffer (ITLB), branch prediction logic, and branch address logic. For optimal performance and power consumption, the fetch unit relies on highly accurate branch prediction, enabling Bobcat to speculate while using power efficiently.

Instruction fetch

The Bobcat fetch unit selects the next fetch address from among the branch correction addresses, branch prediction addresses, stall recirculation addresses, and the sequential address. The fetch address enables access to the 32-Kbyte two-way set-associative instruction cache. The ITLB and the instruction cache are accessed in parallel. The ITLB contains address translations for up to eight 2-Mbyte pages and up to 512 4-Kbyte pages. To save power, Bobcat tries to access the ITLB only when a fetch occurs to a different 4-Kbyte page than the previous fetch or if the translation might have changed since the previous access. Both the L1 instruction cache and the ITLB are parity protected to improve reliability.

Branch prediction

Branch prediction includes branch location, direction prediction, and address prediction. Because the x86 instruction set has variable-length instructions, one challenge to branch prediction is to quickly determine the branch's location in the instruction stream. Once branch instructions are discovered, Bobcat's branch prediction logic keeps information about them in the predictor arrays.

Most cache lines contain only a few branches, but some contain many. To handle both cases efficiently, Bobcat puts information about the first two branches for a cache line in a ''sparse'' branch marker array. This array is indexed in the same manner as the instruction cache and can be considered a logical extension of the instruction cache. Information about additional branches discovered in the cache line is stored in a dense branch marker array, which can hold information about two dense branches for every 8 bytes.

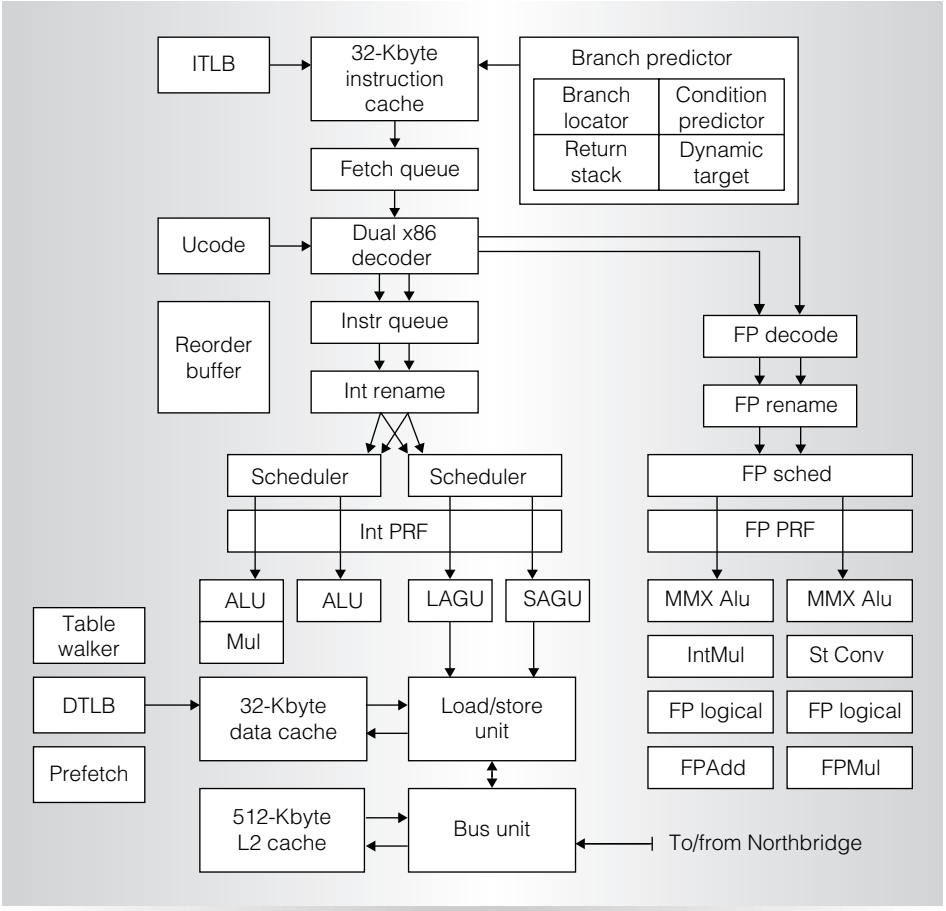

Figure 1. Bobcat block diagram. (ALU: arithmetic logic unit; ITLB: instruction translation look-aside buffer; Int PRF: integer physical register file; FP PRF: floating-point physical register file; LAGU: load address generation unit; SAGU: store address generation unit; DTLB: data translation look-aside buffer; St Conv: store convert; MMX ALU: multimedia instruction ALU.)

The two branches in the sparse marker array are predicted in parallel, and a one-cycle fetch bubble is inserted for a predicted taken branch. The sparse array contains information about which, if any, 8-byte portions of the cache line have dense branches. If the sparse branches are predicted as not taken, then any dense branches are processed one per cycle. The dense markers are allocated dynamically to a 1,024-entry structure. Both the sparse and dense predictors contain information about the branch type, branch end byte, and branch target offset. This branch information specifies which structures and logic are necessary to process the branch, and only those structures are clocked.

A highly accurate proprietary algorithm performs branch direction prediction for these marked branches. Branch target prediction depends on the type of branch instruction. For relative branches, statistical analysis of common applications shows that the vast majority of targets are on the same page as the branch. For such branches, the marker arrays simply provide a page offset for the target. For branches whose target is on a different page, the marker arrays indicate that a separate out-of-page array should be used to predict the target page's base address. Later in the fetch pipeline, the predicted target address is checked against a calculated target address. If a mismatch occurs, the target address can be corrected

without having to pay the full branch misprediction penalty.

As branches are predicted, the instructions are fetched at up to 32 bytes per cycle and sent as 16-byte memory-aligned fetch windows to a 12-entry, 16-byte-per-entry instruction buffer. This provides a queue of instructions for the decoder to operate on and decouples it from the instruction fetch pipeline.

x86 instruction decode

The x86 architecture has variable-length instructions whose length can be further modified by an indeterminate number of instruction prefixes. Instruction-length decode, for determining instruction boundaries, presents a well-known timing problem for x86 processors. AMD's high-performance x86 designs have used cached-instruction marker bits to address the timing problem of detecting three or more instructions within one clock while simultaneously pushing for maximum clock speed. This requires a secondary decoder pipeline for decoding unmarked lines, additional state in the L1 instruction cache, possibly storage in the L2 cache for marker bits (one additional bit per instruction byte), and a checking pipeline to make sure the markers are valid. (A program might, in a rare occurrence, branch into the middle of a previously seen instruction.) Bobcat's dispatch-path width reduction, compared with earlier designs, makes length decode a more tractable problem and lets us eliminate marker bits while achieving meaningful clock frequencies.

Bobcat's length decode implementation uses dedicated decoders for the 16 bytes of the oldest fetch window in the instruction buffer, and 6 bytes of the next oldest. The decode block decodes instructions two at a time until the oldest fetch window is exhausted. Internal studies have shown that having an extra 6 bytes of decoder width lets Bobcat decode the majority of two-instruction scenarios in which the pair starts in the oldest fetch window and extends into the next-oldest window. As instructions are decoded, they move into the instruction queue, which buffers the results in case of back pressure from the microengine and later pipeline stages.

Compared with AMD's K8 architecture,[1] the reduced dispatch width throughput is counterbalanced by elimination of most instruction dispatch lane restrictions. Removal of the extra lanes of instruction decoders and dispatch eliminates about a third of the circuitry used in the K8's decoder and allows simplification of some memory structures (for example, the number of ports on the instruction queue, which changes from four to two write ports and from three to two read ports). In addition, this eliminates the secondary decoder pipeline, the checker pipeline, end marker storage, and related multiple operating modes and correction mechanisms.

Microengine and microcode

Partially decoded instructions from the decode block are translated into COPs and dispatched by the microengine. COPs, x86-like instructions understood by the underlying hardware, can range from simple register moves to complex multipart operations such as load-op-stores to memory. Most commonly used instructions are decoded directly by fast-path decoders to either one or two COPs and are translated in a single cycle. Dynamic instruction execution analysis across a suite of applications shows that 89 percent of x86 instructions translate to a single COP, 10 percent translate to a pair of COPs, and fewer than 1 percent are implemented using microcode. For both microcode and fast-path COPs, the microengine checks tokens representing availability of critical downstream resources and inhibits dispatch if required resources aren't available.

As with previous AMD microprocessors, Bobcat uses microcode for x86 architecture features not easily implemented as fast-path COPs. The microcode is stored in an on-chip ROM, with each entry (or line) in the ROM specifying two COPs and sequence information. Microcode flows can be multiline, and the sequence operation in each ROM entry is provided to the microsequencer so that it can find the next line of microcode for the current flow. Microcode serves in breaking down complicated x86 instructions, interrupt and exception processing, and power management (for example, the C6 power state).

Integer execution unit

The core of the integer execution unit (EXE) consists of two ALUs and two address generation units (AGUs). On dispatch, COPs needing ALU resources are written into a 16-entry unified scheduler; from there, up to two operations per cycle can be selected for execution. Most ALU operations require only a single execution cycle. Multiplies can take three to seven cycles, depending on data size. The AGU execution units serve to generate memory addresses for loads and stores, which are scheduled using a separate eight-entry queue. One AGU is dedicated for loads, the other for stores; so, up to one load and one store can go through address generation each cycle. The pickers for both the ALUs and the AGUs can pick COPs out of order, once the COP's operands are ready. Unique to the Bobcat integer unit design is that both AGU and ALU scheduler queues are implemented using pointer-based, nonshifting queues to reduce power consumption.

Another power-saving feature is data-size-based clock gating. Although all integer units support 64-bit results, software doesn't always use 64-bit operations, even while in 64-bit mode. Therefore, the upper and lower halves of all result buses and forwarding logic are clocked independently, saving power during non–64-bit operations.

Physical register file

Key Bobcat power savings come from the EXE unit design using a 64-entry physical register file (PRF). The PRF design relies on pointer updates to PRF registers rather than actual data movement to implement register renaming. Using clock gaters on register writes further reduces power consumption when fewer than the full 64 bits require updating.

Reorder buffer

Bobcat's retirement logic contains a 56-entry reorder buffer (ROB). The ROB increases efficiency over AMD's K8 architecture by removing restrictions on what combinations of COPs can or must be retired together. Up to two COPs per cycle can be retired, updating the commit PRF pointers. Exceptions and other retire-time redirects are handled at this point.

Floating-point unit

Bobcat's floating-point unit (FPU) is a coprocessor model similar to other AMD designs, with code lineage from K8 and the core code-named Barcelona. Several major changes to the Bobcat floating-point microarchitecture improve its power and performance efficiency. First, like the K8, Bobcat breaks 128-bit SIMD instructions into 64-bit execution chunks. Second, the floating-point decoder bandwidth has been reduced from three COPs per cycle on prior designs to two on Bobcat. Similarly, the number of execution pipelines has been reduced from three to two. Third, where possible, the Bobcat floating-point execution units have shorter execution pipelines, consistent with the core's somewhat lower-frequency target. The control logic queues generally are reduced to better balance power versus performance and to achieve a lower-power client part. The 18-entry floating-point scheduler is implemented as nonshifting and uses fewer content addressable memories. It has two oldest-ready pickers (one for each execution pipe) and can support single-cycle-instruction operation with back-to-back scheduling.

The FPU also supports a limited amount of control word renaming to better support rapid rounding-mode changes. The floating-point retire queue, which holds FP retire-time state update information, was modified to be a per-operation queue rather than three-wide line-based, as in prior designs. The improved queue packing and ROB size of 56 entries let designers reduce the floating-point retire queue size to 40 entries at no performance cost.

The floating-point multiplier (FPM) was redesigned entirely to use a smaller multiplier tree (76 bits $\times$ 27 bits) to save area and power.[2] This trade-off allows for good overall power efficiency, trading off lower multiply double- and extended-precision (DP/EP) performance as additional latency and lower throughput results for instructions. Table 1 shows the throughput for various instructions. Other execution units were optimized to a lesser extent through pipeline stage reduction, achieved via repipelining, with

results also summarized in the FPU instruction latency and throughput table. The SSSE3 instruction set extensions were added to the existing support for SSE1 through SSE3 and SSE4a (introduced with Barcelona). To save additional area and power, the older 3DNow! instructions were removed.

For data forwarding, execution units are divided into three clusters: float, int, and store. The float cluster includes the floating-point multiply and floating-point add; the integer cluster includes the two integer ALUs and the integer multiply. Transferring data between execution clusters costs an extra cycle of bypass latency.

Table 1. Bobcat FPU instruction latency and throughput summary.

Instruction class	Latency (in cycles)	Throughput	Execution pipeline
SIMD ALU (most)	1	2 per cycle	Either
FP logical	1	2 per cycle	Either
SIMD imul*	2	1 per cycle	Pipe0
FP multiply SP	2	1 per cycle	Pipe1
FP add	3	1 per cycle	Pipe0
FP multiply DP	4	1 every two cycles	Pipe1
FP multiply EP	5	1 every three cycles	Pipe1

*imul: integer multiply; SP: single precision; DP: double precision; EP: extended precision.

Load/store unit, data cache, and table walk engine

Bobcat includes a small yet efficient OoO load/store unit (LSU). Capable of tracking up to 26 in-flight loads and 22 stores, this processor is AMD's first fully OoO LSU. It supports both loads bypassing older loads and loads bypassing older nonconflicting stores. The Bobcat LSU can perform an 8-byte load and an 8-byte store on every cycle, and it supports a three-cycle load-to-use pipeline. The data cache supports up to eight outstanding cache misses, as well as hits beneath misses. Critical-word forwarding is used on cache misses to reduce effective cache miss latency.

The 32-Kbyte eight-way set-associative data cache is parity protected. The L1 DTLB (data translation look-aside buffer) supports 40 4-Kbyte pages and eight 2-Mbyte pages, all fully associative. The L2 DTLB supports 512 4-Kbyte pages and 64 2-Mbyte pages, each four-way set associative. The table walk engine (TWE) handles table walks for TLB misses. The 1-Gbyte architected page size is supported, but the TWE divides it into 2-Mbyte pages. The TWE includes a page directory cache to accelerate walks, and it supports rapid virtualization indexing (also known as nested page tables). This simplifies the host management of guest page tables and significantly improves virtualized workloads' performance.

The load-to-use pipeline appears in Figure 2. The AGU calculates a linear address, which is sent to both the data cache tags (DTags) and L1 DTLB in cycle DC1 (data cache 1). The L1 DTLB translates the linear address into a physical address, which is then compared against eight physical tag addresses read from the DTags. On a match or hit, only a single matching way of the cache is read. The data is aligned and, if necessary, extended; then it's sent to the register file. This design of the DTag/DTLB/cache pipeline saves power. The pipeline is designed so that a dependent operation can be scheduled to consume the data on the cycle after DC2. Most load/store operations will hit in the TLB and cache, and can be completed in three cycles. Operations unable to complete remain in the LSU and replay later.

In addition to including an OoO picker, the Bobcat LSU includes several performance optimizations to increase throughput and instructions per cycle. Store-to-load forwarding is supported from all completed and retired stores to all incoming loads, potentially allowing for loads to complete long before the cache is written. And, like AMD's Barcelona processor, Bobcat supports 16-byte internal misalignment, so that loads or stores contained within a 16-byte boundary require only one cache access. Loads or stores that cross the 16-byte boundary require two back-to-back accesses. However, unlike previous AMD processors, Bobcat explicitly allows misaligned SSE instructions (for example, MOVUPD) to be executed as quickly for aligned memory accesses as their explicitly aligned counterparts (for example, MOVAPD). This gives software the flexibility to use the unaligned forms of these

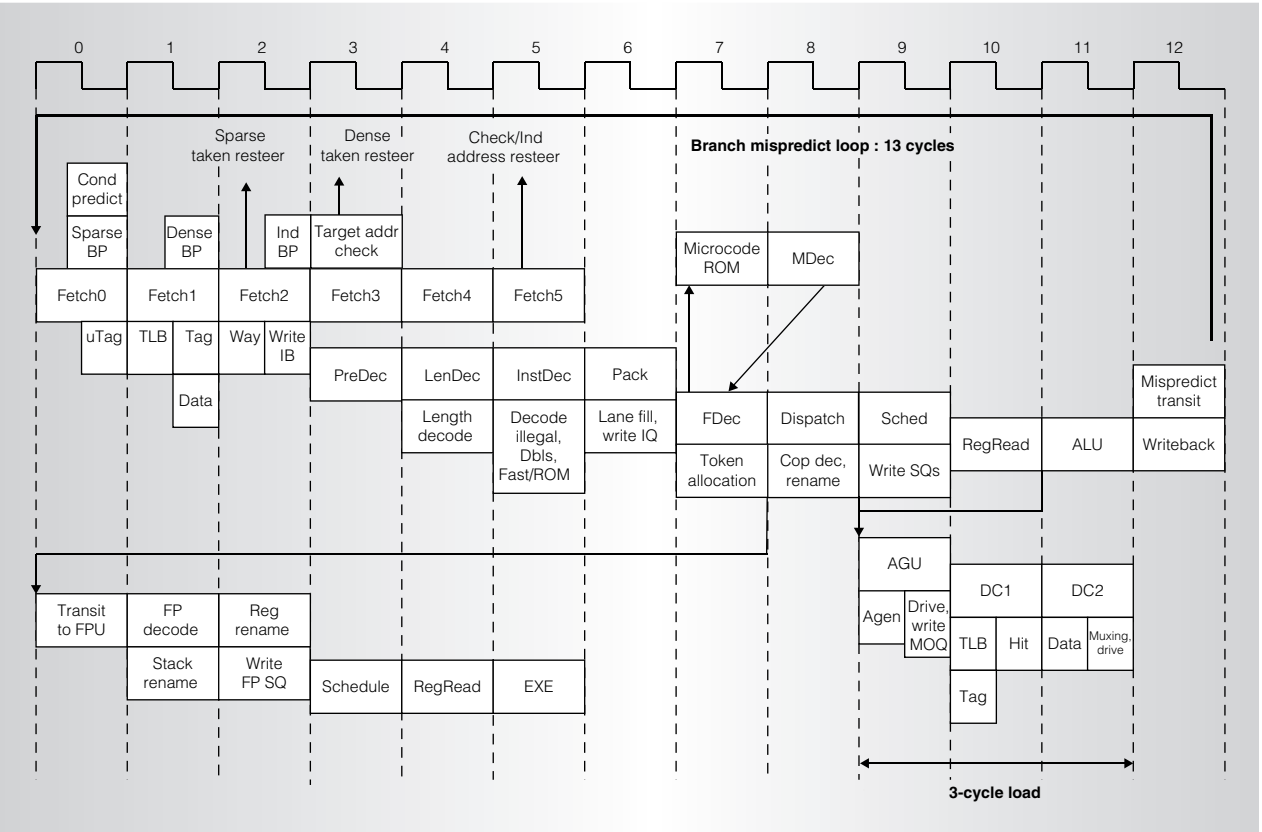

Figure 2. The Bobcat consists of six fetch stages, which partially overlap the four decode stages, followed by the microengine stages and the integer schedule, register read, and execute and write-back stages. The lower left shows the basic floating-point pipeline, and the lower right shows the basic load pipeline. There's a three-cycle load use latency, and for mispredicted branches there's a 13-cycle correction penalty.

instructions without penalty. An additional component of the data cache unit, the prefetcher, can track and prefetch up to eight separate data streams. Data cache prefetches are initiated on second miss (to confirm direction) and can run one to four misses ahead.

Bobcat supports all x86 segmentation features. Segment base addition, if needed, can rely on an additional SegAdd pipeline stage inserted between the AGU cycle and the DC1 cycle. Because most modern software rarely uses segmentation, Bobcat optimizes the ''segment base equals zero'' case and bypasses this SegAdd stage whenever possible.

Finally, like all units in the Bobcat pipeline, the LSU was designed for low-power operation. All major queues in the block are nonshifting, reducing the dynamic power needed for moving queue entries. Fine-grained clock gaters save power when areas of logic or queue entries are invalid. Also, to reduce power consumption, operation replays are done relatively conservatively, with operations selected only when they have a high chance of completing. These features help the Bobcat LSU balance high performance with low power consumption.

Bus interface unit and L2 cache

The bus unit handles memory and I/O transactions and controls the L2 cache. It receives instruction fetch requests and attribute evictions from the instruction cache; load, stores, and dirty evictions from the data cache; and combinable writes from the LSU. It manages system probes from the north bridge. It can support up to eight outstanding load/store requests and two outstanding instruction fetch requests.

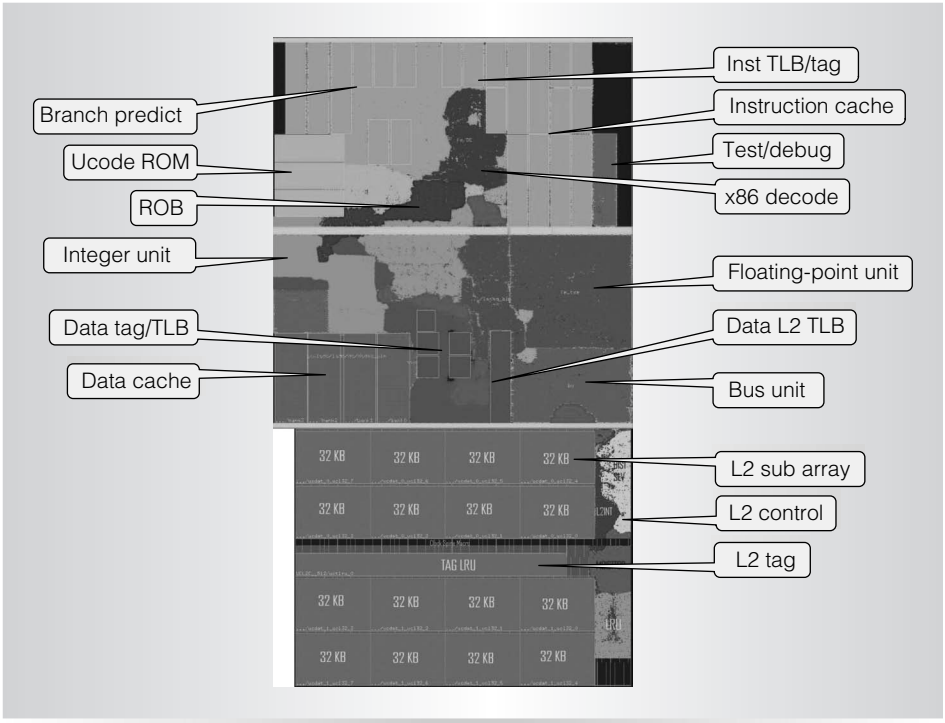

Figure 3. Physical design of the Bobcat core, showing the various functional blocks, the malleable logic blocks which were synthesized and automatically placed, and the few rectilinear custom memory arrays.

The bus unit stores active transactions in a set of queues. It performs arbitration among its active transactions for various resources: the L2 cache, requests to the north bridge, data cache probes, and instruction cache probes. The bus unit detects hazards among its active transactions by comparing each new transaction's address against all other active transactions. It remembers any matches and uses the information to enforce ordering and coherence rules.

The L2 cache is 512 Kbytes, 16-way set associative. To save power, it operates at half the frequency of the core, with best-case pipeline latency for load-miss, L2 cache hit of 17 cycles. L2 misses are loaded into both the L2 and the primary requester, leading to a statistically inclusive L2 cache—although inclusion is not guaranteed formally. The L2 data and tag arrays are protected by error-correcting code.

Physical design

Bobcat implementation leverages automated flows, including synthesis auto place and route (SAPR). The core and L2 cache were built as two separate tiles. The core includes seven custom macro arrays as well as a few other custom circuits, including clock spines. The macro array elements build up microarchitectural elements such as the data cache, instruction cache, tags, TLBs, branch predictors, and microcode ROMs. Some elements originally planned to be macros were converted to guided flip-flop arrays on the basis of implementation feedback. Arrays consume approximately 40 percent of the core area. The remaining 60 percent is built using standard cell synthesis and automated placement. As Figure 3 shows, the core is built as a single monolithic tile, which leads to irregular amoeba-like blocks that can intermingle to facilitate timing optimization.

The broad use of SAPR tools allowed for a smaller team and faster design iteration than would full-custom design methodologies. The SAPR timing team beat the initial project timing targets and met the higher goals that management set for 1.6-GHz silicon.

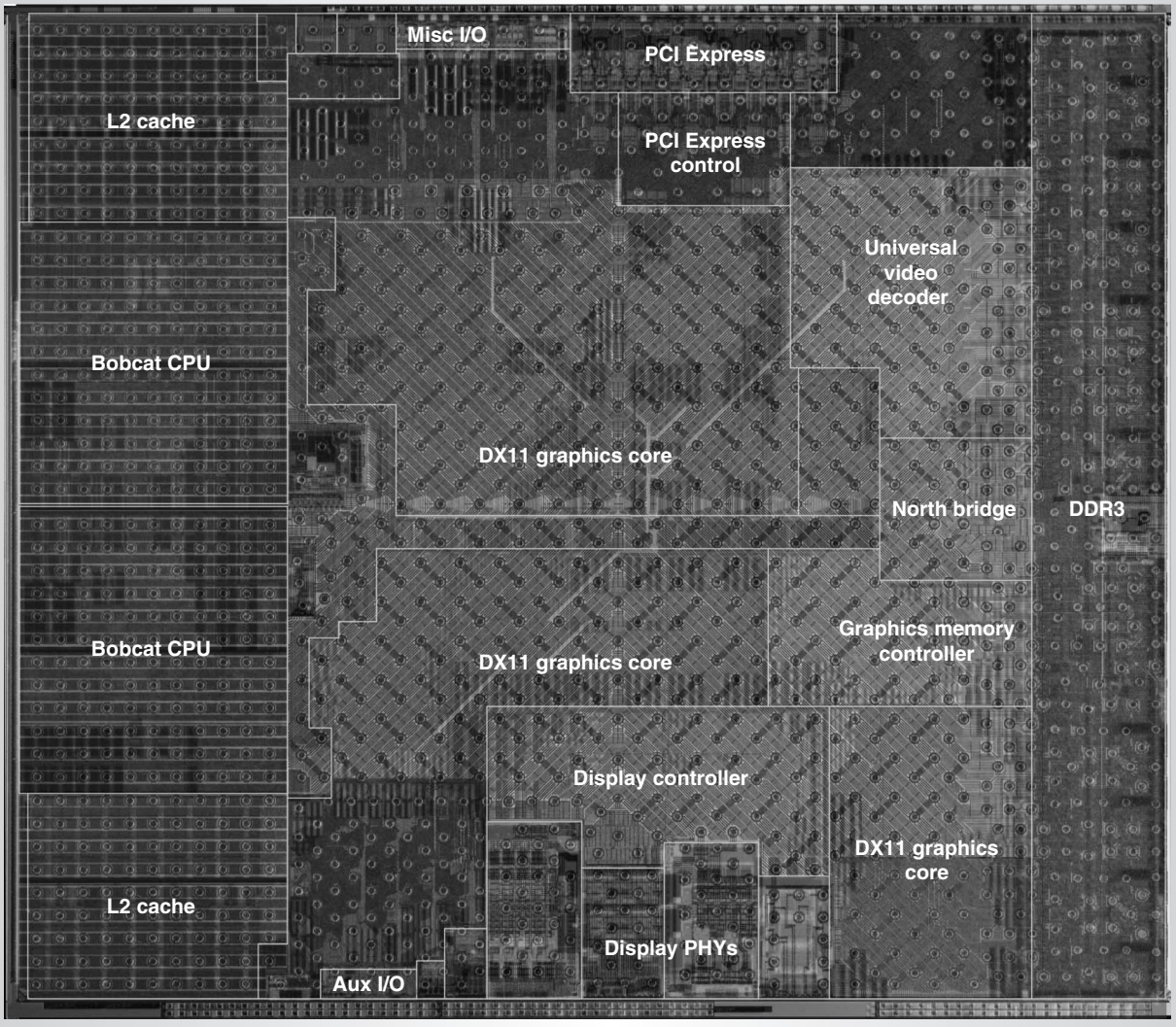

Figure 4. Photo of the Ontario die with two Bobcat cores (left), a DX11 graphics processor (distributed), double-data-rate three (DDR3) interface (right), north bridge, display controller and associated video I/O, universal video decoder, and PCI Express controller and I/O.

The combination of microarchitecture optimization and SAPR tools led to a 4.9-mm^2 core in a 40-nm process, about one-third the area that a K8 would have consumed in the same process.

The Bobcat core, shown in Figure 4, will be introduced first in the Ontario/Zacate accelerated processing unit. This system on a chip consists of two Bobcat processor cores with L2 caches, a DX11 graphics unit, a universal video decoder, a 64-bit DDR3 (double-data-rate three) memory interface, and a configurable PCI Express I/O unit. Most of the units on die can be power-gated to save power when idle. AMD Fusion architecture permits certain applications, such as video transcode and image processing, to use the multiple SIMD processing elements of the graphics unit to improve performance. On the basis of AMD modeling using benchmark simulations, the smaller, lower-power Bobcat core achieves an estimated 90 percent of the K8's performance. MICRO

References

1. C.N. Keltcher et al., ''The AMD Opteron Processor for Multiprocessor Servers,'' *IEEE Micro,* vol. 23, no. 2, 2003, pp. 66-76.
2. D. Tan, C.E. Lemonds, and M.J. Schulte, ''Low-Power Multiple-Precision Iterative Floating-Point Multiplier with SIMD Support,'' *IEEE Trans. Computers,* vol. 58, no. 2, 2009, pp. 175-187.

Brad Burgess is a fellow at Advanced Micro Devices and chief architect for the Cat family of processor cores. Burgess has an MS in electrical engineering from Texas A&M University. He is a member of IEEE.

Brad Cohen is a principal member of the technical staff at Advanced Micro Devices. He leads the Bobcat family architecture/RTL team. Cohen has a BS in electrical engineering from the Georgia Institute of Technology.

Marvin Denman is a principal member of the technical staff at Advanced Micro Devices, working on Cat family cores. He is responsible for RTL development for a future CAT family core and leads a portion of the RTL team for a next-generation CAT family core. Denman has a BS in computer science from Texas A&M University.

Jim Dundas is a senior member of the technical staff at Advanced Micro Devices. He is the branch prediction architect for Bobcat and the branch prediction and instruction cache architect for the AMD Cat family of processors. Dundas has a PhD in electrical engineering from the University of Michigan, Ann Arbor.

David Kaplan is a member of the technical staff at Advanced Micro Devices. He is part of the Bobcat architecture team and helped design the microcode and load/store unit. Kaplan has a BS in computer engineering from the University of Illinois at Urbana-Champaign.

Jeff Rupley is a principal member of the technical staff at Advanced Micro Devices, working in low-power core design for Bobcat, and chief architect on a follow-on core. Rupley has an MS in electrical engineering from Purdue University. He is a member of IEEE and ACM.

Direct questions and comments about this article to Brad Burgess, Advanced Micro Devices, MS B400.2A, 7171 Southwest Parkway, Austin, TX 78735; brad.burgess@amd.com.