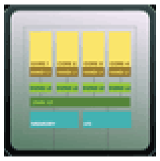


AMD K10 Micro-Architecture

First AMD Barcelona processors should be announced in a month already. They will be the first solutions to be built on AMD's new K10 micro-architecture that the company pins a lot of hopes upon. Let's take a closer look at the micro-architectural innovations that will be introduced in these new solutions.

by Yury Malich
08/17/2007 | 10:06 AM

Introduction

AMD promises to introduce its new quad-core processors with K10 micro-architecture in the end of August – beginning of September this year. The first processors on this new micro-architecture will be server Opteron chips based on a core codenamed Barcelona. Unfortunately, AMD engineers failed to hit mass production quantities with the current revision of high-frequency chips. Looks like the main obstacle on the way to higher working frequencies was the fact that four cores running at high speed consume much more power than the platform TDP actually allows. With every new revision and transition to finer production technologies the power consumption will keep lowering and the working frequencies will keep growing. So far, AMD has to immediately start selling processors in order to improve its financial situation, so the first one to start selling will be the quad-core server processor model working at 2.0GHz.

In Q4 2007 AMD promises to increase Opteron working frequencies up to 2.4-2.5GHz and release desktop processors on K10 micro-architecture:

- Phenom FX (codenamed Agena FX) – 4 cores, 2MB L3 cache, clock frequencies starting at 2.2-2.4GHz, AM2+ Socket, F+;
- Phenom X4 (codenamed Agena) – 4 cores, 2MB L3 cache, clock frequencies starting at 2.2-2.4GHz, AM2+ Socket.

Later in early 2008 AMD promises to introduce "lite" modifications of their new processors, such as:

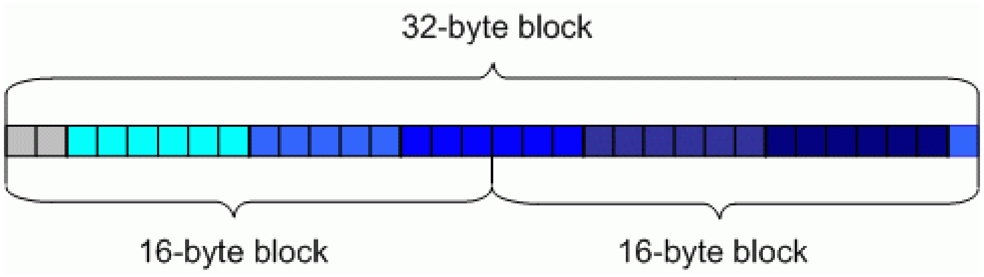
- Phenom X2 (codenamed Kuma) – 2 cores, 2MB L3 cache, clock frequencies starting at 2.2-2.6GHz, AM2+ Socket;
- Athlon X2 (codenamed Rana) – 2 cores, no L3 cache, clock frequencies start at 2.2GHz, AM2+ Socket;
- Sempron (codenamed Spica) – 1 core, clock frequencies start at 2.2-2.4GHz, AM2+ Socket.

But this is all in the future. So far let's take a look at the innovations introduced in the new AMD micro-architecture. In our today's article I am going to try and reveal all the new architecture details and see what practical value they will have for us.

Instructions Fetch

Processor starts the code processing from fetching instructions from the L1I instruction cache and their decoding. x86 instructions have variable length, which makes it harder to determine their boundaries before decoding starts. To ensure that the identification of the instructions length doesn't affect the decoding speed, K8/K10 processors decode instructions while the lines are being loaded into L1I cache. Instruction labeling info is stored in special fields of the L1I cache (3bits of predecoding info per each byte of instructions). By performing the predecoding during loading into cache the instructions boundaries can be determined beyond the decoding pipes, which allows maintaining steady decoding rate independent of instructions format and length.

Processors load blocks of instructions from the cache and then pick out instructions that need to be sent for decoding. A CPU on K10 micro-architecture fetches instructions from the L1I cache in aligned 32-byte blocks, while K8 and Core 2 processors fetch instructions in 16-byte blocks. At 16 bytes per clock the instructions are fetched fast enough for K8 and Core 2 processors to send three instructions with the average length of 5 bytes for decoding every clock cycle. However, some x86 instructions may be 16 bytes long and in some algorithms the length of a few adjacent instructions may be greater than 5 bytes. As a result, it is impossible to decode three instructions per clock in such cases. (Pic.1).



Pic 1: A few adjacent long instructions limit the decoding speed during instructions fetch 16-byte blocks.

Namely, SSE2 – a simple instruction with operands of register-register type (for example, *movapd xmm0, xmm1*) – is 4 bytes long. However, if the instruction generates addressed memory requests using the base register and offset (for example, *movapd xmm0, [eax+16]*), the instruction increases up to 6-9 bytes, depending on the offset. If additional registers are involved in 64-bit mode, there is one more single-byte REX-prefix added to the instruction code. This way, SSE2 instructions in 64-bit mode may become 7-10 bytes long. SSE1 instructions are 1 byte shorter, if it is a vector instruction (in other words, if it works on four 32-bit values). But if it is a scalar SSE1 instruction (on one operand) it can also be 7-10 bytes long in the same conditions.

Fetching maximum 16-byte blocks is not a limitation for K8 processor in this case, because it cannot decode vector instructions faster than 3 per 2 clocks anyway. However, for K10 architecture a 16-byte block could become a bottleneck, so increasing the maximum fetch block size to 32 bytes is an absolutely justified measure.

By the way, Core 2 processors fetch 16-byte instruction blocks, just like K8 processors, that is why they can decode efficiently 4 instructions per clock cycle if the average instruction length doesn't exceed 4 bytes. Otherwise, the decoder will not be able to process 4 or even 3 instructions per clock efficiently enough. However, Core 2 processors feature a special internal 64-byte buffer that stores the last four requested 16-byte blocks. The instructions are fetched from this buffer at the rate 32 bytes per clock speed. This buffer allows caching short cycles, removing their fetching speed limitations and save up to 1 clock cycle each time the prediction to move to the cycle beginning is made. Although the cycles shouldn't have more than 18 instructions, more than 4 conditional branches and no ret instructions in them.

Branch Prediction

If the chain of instructions branches, the CPU should try to predict further direction of the program to avoid decoding interruption and continue decoding the most probable branch. In this case branch prediction algorithms are used to fetch the next instructions block. K8 processors use two-level adaptive algorithm for branch prediction. This algorithm takes into account prediction history not only for the current instruction, but also for 8 previous instructions. The main drawback of K8 branch prediction algorithms was the inability to predict indirect branches with dynamically alternating addresses.

Indirect branches are the branches that use a pointer, which is calculated dynamically during program code execution. These indirect branches are usually inserted into switch-case constructions by the compiler. They are also used during addressed function calls and virtual function calls in object-oriented programming. K8 processor always tries to use the last branch address to grasp a block of code to be fetched. If the address has changed, the pipeline is cleared. If the branch address is alternating occasionally, the processor will make prediction mistakes all the time. The prediction of dynamically changing addresses for indirect branches was first introduced in Pentium M processor. Since there is no such algorithm in K8 CPUs, they are less efficient in object-oriented codes.

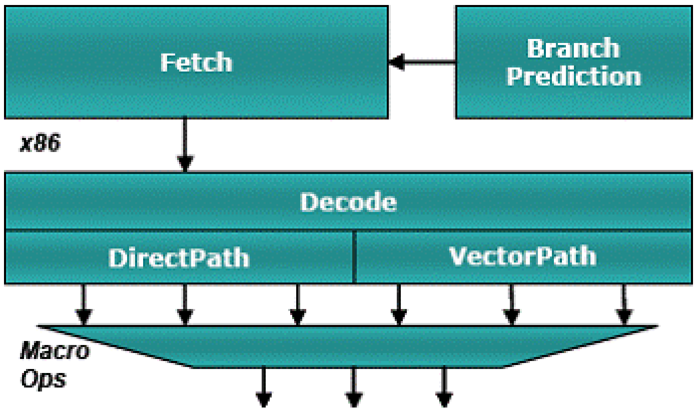
As we have expected, K10 boasts improved conditional branch prediction algorithms:

- It acquired prediction algorithms for dynamically changing indirect branches addresses. This algorithm uses a table of 512 elements.
- The global history register increased from 8 to 12 bits. It serves to determine the succession history for previous branch instructions.
- The depth of return-address stack increased from 12 to 24 positions. This stack serves to obtain the function return address quickly, so that the fetching could continue and there were no need to wait for the ret instruction to receive the stack return address.

These improvements should help K10 execute programs written in high-level object-oriented code much faster. Unfortunately, it is very hard to objectively estimate the efficiency of the K10 branch prediction unit, but according to some data, it may be lower in some cases than by Intel processors.

Decoding

The blocks received from the instructions cache are copied into the **Predecode/Pick Buffer** , where instructions are singled out from the block, their types are defined, and then they are sent to the corresponding decoder pipes. Simple instructions that can be decoded with one (Single) or two (Double) micro-operations are sent to the "simple" decoder called **DirectPath** . Complex instructions that require 3 or more micro-operations to be decoded, are sent to the micro-program decoder aka **VectorPath** .



Pic.2: Decoder

Up to 3 macro-operations (MOPs) may leave decoder pipes each clock cycle. Every clock cycle DirectPath decoder may process 3 simple single-MOP instructions, or one 2-MOP instruction and one single-MOP instruction, or 1.5 2-MOP instructions (three 2-MOP instructions in two clocks). Decoding of complex instructions may require more than 3 MOPs that is why they may take a few clocks to complete. To avoid conflicts on leaving the decoder pipes, K8 and K10 simple and complex instructions may be sent for decoding simultaneously.

MOPs consist of two micro-operations (micro-ops): one integer or floating point arithmetic operation and one memory address request. Micro-operations are singled out from the MOPs by the scheduler, which then sends them to be executed independently from one another.

MOPs leaving the decoder every clock are combined into groups of three. Sometimes the decoder may generate a group of 2 or even only 1 MOP because of the alternating DirectPath and VectorPath instructions or different delays in the selection of instructions for decoding. An incomplete group like that is filled with empty MOPs to make three, and then is sent to be executed.

Vector SSE, SSE2 and SSE3 instructions in K8 processor are split into MOP pairs that process separately the upper and lower 64-bit halves of the 128-bit SSE register in 64-bit devices. It slows down the instructions decoding by half and cuts down in half the number of instructions in the scheduler queue.

Thanks to larger 128-bit FPU units in K10 processors, there is no need to split vector SSE-instructions into 2 MOPs any more. Most SSE-instructions that K8 used to decode as DirectPath Double, are now decoded in K10 as DirectPath Single in 1 MOP. Moreover, some SSE-instructions that used to be decoded through K8 micro-program VectorPath decoder, are now decoded in K10 through simple DirectPath decoder with fewer generated MOPs: 1 or 2 depending on the operation.

Decoding of stack instructions has also been simplified. Most stack operation instructions that are usually used for CALL-RET and PUSH-POP functions are now also processed by a simple decoder in a single MOP. Moreover, special **Sideband Stack Optimizer** scheme transforms these instructions into an independent chain of micro-operations that can be executed in parallel.

Sideband Stack Optimizer

Decoder schemes of the new K10 processors acquired a special block called Sideband Stack Optimizer. Its working principle is similar to that of the new Stack Pointer Tracker unit that is employed in Core processors. What do we need it for? x86 system uses CALL, RET, PUSH and POP instructions to call a function, retire a function, transfer parameters of a function and save register contents. All these instructions use, though not directly, the ESP register indicating the current stack-pointer value. When you call a function in K8 processor you can follow the execution of these instructions by representing their decoding as a succession of equivalent elementary operations modifying the stack register and load/save instructions.

Instructions	Equivalent operations
// func(x, y, z); push X push Y push Z call func	 sub esp, 4; mov [esp], X sub esp, 4; mov [esp], Y sub esp, 4; mov [esp], Z sub esp, 4; mov [esp], eip; jmp func
push esi push edi mov eax, *[esp+16]* .... pop edi, pop esi ret	sub esp, 4; mov [esp], esi sub esp, 4; mov [esp], edi mov eax, [esp+16] .... mov edi, [esp]; add esp, 4 mov esi, [esp]; add esp, 4 jmp [esp]; add esp, 4
add esp, 12	add esp, 12

As you can see from this example, when the function is called, instructions sequentially modify the ESP register, so each next instruction is implicitly dependent on the result of the previous one. Instructions in this chain cannot be reordered that is why the execution of the function body starting with mov eax, [esp+16] cannot begin unless the last PUSH instruction has been executed. Sideband Stack Optimizer unit tracks the stack status changes and modifies the instructions chain into an independent one by adjusting the stack offset for each instruction and placing sync-MOP operations (top of the stack synchronization) in front of the instructions that work directly with the stack register. This way instructions working directly with the stack can be reordered without any limitations.

Instructions	Equivalent operations	
// func(x, y, z); push X push Y push Z call func	mov [esp-4], X mov [esp-8], Y mov [esp-12], Z mov [esp-16], eip; jmp func	
push esi push edi mov eax, *[esp+16]* .... pop edi, pop esi ret	mov [esp-20], esi mov [esp-24], edi sub esp, 24 mov eax, [esp+16] .... mov esi, [esp] mov edi, [esp+4] jmp [esp+8]	sync-MOP
add esp, 12	add esp, 12 add esp, 12	sync-MOP

mov eax, [esp+16] instruction that starts the calculations in the function body in our example depends only on the sync-MOP operation. Now these operations can be performed simultaneously with other instructions before them. This way the parameters passing and register saving happen faster and the function body can start loading these parameters and working with them even before all of them have been passed successfully and before the registers saving has been complete.

So, faster stack operations decoding, Sideband Stack Optimizer unit, deeper return-address stack and successful prediction of indirect alternating branches make K10 much more efficient for processing of function-rich codes.

K10 processor decoder will not be able to decode 4 instructions per clock like Core 2 decoder in favorable conditions, but it will not become a bottleneck during programs execution. The average instructions processing speed hardly ever reaches 3 instructions per clock, therefore K10 decoder will be efficient enough for the computational units not to lack any instructions in the queues and hence not to idle.

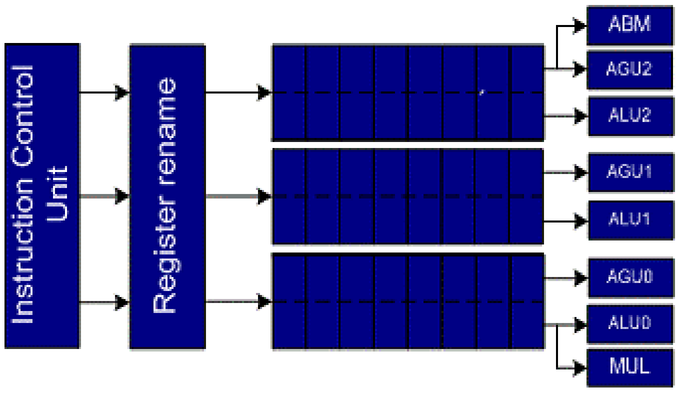
Instruction Control Unit

Decoded Mop triplets arrive to the Instruction Control Unit (ICU) that moves them to the reorder buffer (ROB). Reorder buffer consists of 24 lines three MOPs in each. Each MOP triplet is written in its own line. As a result, ROB allows the control unit to monitor the status of up to 72 MOPs until they retire.

From the reorder buffer MOPs are being dispatched to the scheduler queues of integer and floating-point units in exact same order, in which they retired from decoder. MOP triplets are stored in the ROB until all older operations are executed and retired. On retirement, the final values are written down into architectural registers and memory. The program order, in which the operations were placed to the ROB, is maintained when operations retire, their data is deleted from ROB and the final values are saved. It is necessary to make sure that the results of all further operations completed ahead of time can be canceled in case of exception or interrupt.

Integer Execution Unit

The **Integer Execution Unit** of K8 and K10 processors consists of three symmetrical integer pipes. Each of these pipes has its own scheduler with an 8-MOP queue, an identical set of integer arithmetic and logical units (ALU), address generation units (AGU) and a branch prediction unit. Moreover, there is a multiplication unit connected to pipe 0, and pipe 2 is tied to the execution unit for new operations: LZCNT and POPCNT, which we are going to discuss in detail later in this article.



Pic. 3: Integer Execution Unit

The queue choice for each MOP is determined by the static location of the instruction in the triplet. Each macro-operation from the triplet is dispatched from the reorder buffer to be executed in its turn. On the one hand it simplifies instructions control, but on the other – may result in ill-balanced load on the queues in case a chain of dependent operations is not very favorably placed in the program code (in reality this occurs very rarely and hardly affects the actual performance). The decoder places multiplication and extended bit operations in the corresponding triplet slots, so that they could fall into the proper pipe.

As we have already said before, MOPs are split into integer operations and addressed memory requests in scheduler queues of the integer pipes. Upon the availability of data, the scheduler may issue one integer operation to ALU and one address operation to AGU from each queue. There can be maximum two simultaneous memory requests. So, up to 3 integer operations and 2 memory operations (64-bit read/write in any combination) may be issue for execution per clock. Micro-operations from various arithmetic MOPs are issued for execution from their queues in an out-of-order manner, depending on the readiness of the data. As soon as the arithmetic and address MOP micro-operations are executed, this MOP is removed from the scheduler queue giving room for new operations.

K8 processor selects memory request address micro-operations on a program level. The memory requests that occur later in the program code cannot be executed ahead of the earlier ones. As a result, if the address for an earlier operation cannot be calculated, all following address operations get blocks, even if the operands for them are already ready.

For example:

add ebx, ecx
mov eax, [ebx+10h] – quick address calculation
mov ecx, [eax+ebx] – address depends on the result of the previous instruction
mov edx, [ebx+24h] – this instruction will not be sent for execution until the addresses for all previous instructions have been calculated.

This may cause performance losses and is one of grave K8 processor bottlenecks. As a result, although K8 processor can process two read instructions per clock, in some codes it may execute memory requests less efficiently than Core 2 processor, which launches one read instruction per clock, but applies speculative out-of-order instructions execution and can jump over preceding reads and writes.

CPUs with K10 micro-architecture do not suffer from this bottleneck any more. K10 processors now can not only process out-of-order reads, but even process the writes before reads if the CPU is certain that no address conflict between these reads and writes exists. By launching writes ahead of reads, the processor can significantly speed up the processing of some types of code, such as read cycles beginning with another data read from the memory and finishing with writing the calculations result into the memory.

L1:
mov eax, [esi] // data loading
.....// data processing
mov [edi] , eax // storing result
cmp
jnz L1

in situations like that the processor that cannot process the read before the write, cannot begin processing next clock cycle iteration before the result of the current one has been written into the memory completely. CPUs supporting read reordering, can start loading new data for the next iteration without waiting for the current one to be completed.

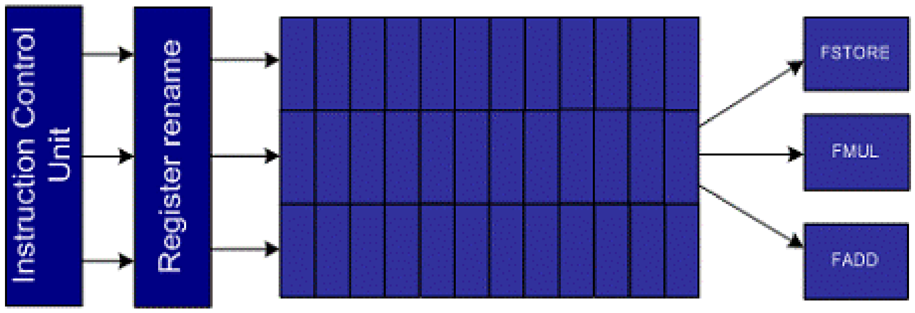
Unfortunately, K10 processor cannot perform speculative loading ahead of writing completion if the address is unknown yet, like Core 2 processor do. Although these speculations may sometimes result into penalties, these are very rare occurrences in real program code (only about 5% cases) that is why speculative loading is absolutely justified from the performance increase prospective.

Another improvement of the K10 integer unit is the optimization of the integer division algorithm. Now fastness of integer division operation depends on the most significant bits of the dividend and divider. For example, if the dividend equals 0, division takes almost half the time. Actually, integer division is a very rare operation. Since it is usually pretty slow it is being carefully avoided in real program codes most of the time. They usually replace it with multiplication by the reciprocal, with shifts or other means, that is why this optimization is very unlikely to have any significant effect on the overall applications performance.

All in all, K10 integer unit will be pretty efficient. Once they added out-of-order memory requests processing, there are no evident bottlenecks in it any more. Although K10 doesn't have the queues as deep as Core 2 processors, it is free from limitations on reading from its register file as well as some other scheduling limitations that do not let Core 2 processors to always perform the operations at maximum speed.

Floating Point Unit

The floating point unit (FPU) scheduler of K8 and K10 processors is separated from the integer unit scheduler and is designed in a slightly different way. The scheduler buffer can accommodate up to 12 groups with 3 MOPs each (theoretically - 36 floating point operations). Unlike the integer unit with symmetrical pipes, the FPU consists of three different units: FADD for floating-point addition, FMUL for floating-point multiplication and FMISC (also known as FSTORE) for operations of saving in the memory and auxiliary operations. Therefore, the scheduler buffer doesn't assign each specific MOP in a group to a particular unit (Pic.4):



Pic.4: Floating Point Unit

Each clock cycle K8 and K10 may issue one operation to each floating point unit for execution. K8 processor features 80-bit floating-point units. At a decoding stage vector 128-bit SSE-instructions are split into two MOPs that process 64-bit halves of a 128-bit operand and are executed successively in different clocks. It not only slows down vector instructions processing, but cuts down in half the effective size of the FPU scheduler buffer and as a result reduces the depth of out-of-order instructions execution.

K10 processor has the floating-point units width increased to 128 bit. K10 processes vector 128-bit operands in a single operation, which doubles the theoretical processing speed for vector SSE-instructions compared with K8. Moreover, since there are twice fewer MOPs now, the effective length of the scheduler queue increases, which allows for deeper out-or-order execution.

K8 processor performed loading SSE-instructions using FSTORE unit. On the one hand, it doesn't allow any other instructions requiring this unit to be executed at the same time, and on the other – limits the number of simultaneously launched load instructions to one only. K8 can perform two parallel read from the memory only if one of the instructions combines a memory request and a data operation (the so-called Load-Execute instruction), for example, *ADDPS xmm1, [esi]*.

K10 processor boasts improved mechanism for loading SSE-instructions.

Firstly, data load instructions no longer use FPU resources. This way FSTORE port is free now and available for other instructions. Load instructions can now be executed two per clock.

Secondly, if the data in the memory is aligned along 16-byte boundary, the unaligned data loading (MOVU**) works as efficient as aligned data loading (MOVA**). So, the using MOVA** doesn't bring any advantages for K10 processors any more.

Thirdly, K10 processors can now use unaligned loading even for Load-Execute instructions that combine loading with the data operations. If it is unclear whether the data in memory is aligned, the compiler (or programmer) usually uses MOVU** instruction to read the data into registers for further processing. By using unaligned loading together with the Load-Execute instructions, they can reduce the number of individual load instructions in the program code and hence increase the performance. Compilers should have the support of this feature integrated. Actually, SSE specification developed by Intel suggests that a request from Load-Execute instruction issued

to an address that hasn't been aligned along 16-byte boundary should lead to exception. To retain compatibility with the spec, the unaligned loads with Load-Execute instructions should be allowed by a special flag in the program code designed and compiled taking into account new processor features.

Fourthly, two buses for data reading from the L1 cache of the K10 processor were expanded to 128 bit. As a result the CPU can read two 128-bit data blocks each clock. This is a very important architectural peculiarity, because 4 operands are required for 2 instructions to be executed in parallel at the same time (2 per instruction), and in some algorithms of streaming data processing two of four operands are usually read from RAM. On the contrary, two buses fort data writing in the K10 processor remained 64 bits wide and 128-bit result is split into two 64-bit packets when written to memory. So, every clock the CPU can only make one 128-bit write or two 128-bit reads, or one 128-bit read and one 64-bit write. However, since the number of reads is usually at least twice as large as the number of writes, writing limitations shouldn't really affect the processor efficiency during 128-bit data processing.

Fifthly, 128-bit data copying, MOV*** register-register, can now be performed in any of the three FPU units and not only in FADD and FMUL. As a result it also frees FADD and FMUL units for dedicated operations.

As we see, the FPU of K10 processor became much more flexible. It acquired some unique features that Intel processors don't have yet, namely, efficient unaligned loading, including Load-Execute instructions, and two 128-bit reads per clock cycle. Unlike Core 2, floating-point and integer schedulers use separate queues. Separate queues eliminate operations conflicts caused by use of the same execution ports. However, K10 still shares the FSTORE unit for SSE save operations with some data transformation instructions, which may sometimes affect their processing speed.

All in all, the K10 FPU promises to be pretty efficient and more advanced than the FPU of Core 2 (for example, thanks to two 128-bit reads per clock and effective unaligned loading).

Memory Subsystem

Load/Store Unit

When the memory request addresses have been calculated in the AGU of K8 processor, all load and store operations are sent to LSU (Load/Store Unit). LSU contains two queues: LS1 and LS2. At first, load and store operations get into LS1 queue 12 elements deep. At two operations per clock speed, LS1 queue issues requests to L1 cache memory in order determined by the program code. In case of a cache-miss, operations are placed into the LS2 queue 32 elements deep. This is where the requests to L2 cache memory and RAM come from.

The LSU of the K10 processor has been modified. Now LS1 queue receives only load operations, while store operations are sent to LS2 queue. Load operations from LS1 can be executed in an out-of-order manner taking into account addresses of store operations in LS2. As we have already mentioned above, K10 processes 128-bit store operations as two 64-bit ones that is why they take two positions each in the LS2 queue.

L1 Cache

L1 cache in K8 and K10 processors is separated: 64KB for instructions (L1I) and data (L1D). Each cache is 2-way set associative; the line length is 64 bytes. This low associativity may result into frequent conflicts between the lines aiming at the same sets, which in its turn may increase the number of cache-misses and negatively affect the performance. Low associativity is often compensated by the rather large size of L1 cache. A significant advantage of L1D is the two ports: it can process two read and/or write instructions per clock in any combination.

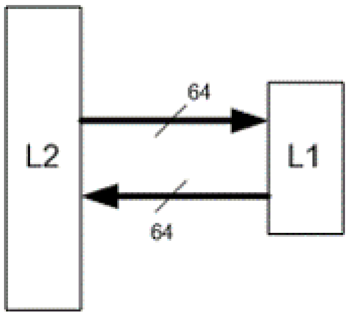
Unfortunately, L1 cache of K10 processor still has the same size and associativity. The only noticeable improvement is the read bus width increase. As we have said in the previous chapter, now the CPU can perform two 128-bit reads every clock cycle, which makes it much more efficient during SSE-data processing in local memory.

L2 Cache

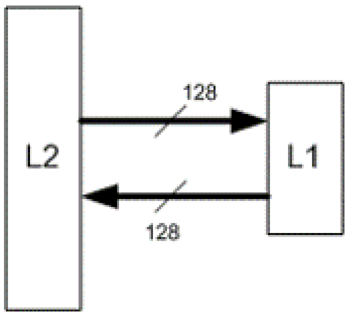
Each core of the dual-core and quad-core K8 and K10 processors has its own individual L2 cache. The L2 cache in K10 remained the same: 512KB per core with associativity of 16. Exclusive L2 caches have their pros and cons compared with the shared L2 cache in Core 2 CPUs. Among the advantages, certainly are the absence of conflicts and competition for the cache when several processor cores are heavily loaded at the same time. As for the drawbacks, there is less cache available for each core when there is only one applications running full throttle.

L2 cache is exclusive: the data stored in L1 and L2 caches do not duplicate. L1 and L2 caches exchange data along two unidirectional buses: one serves to receive data and another one – to send data. In K8 processor each bus is 64bit (8 bytes) wide (Pic.5a). This organization provides the data delivery rate to L2 cache at the modest 8 bytes/clock speed. In other words, it will take 8 clock cycles to transfer a 64-bit line, so the data delivery to the core will be delayed noticeably, especially if two or more lines of the L2 cache are addressed at the same time.

Although it hasn't been confirmed yet, the send and receive buses in K10 processor will become twice as wide, i.e. 128bit each (Pic.5b). It should reduce the cache access latency significantly when two or more lines are requested at the same time.



Pic.5a: Data transfer bus between L1 and L2 caches in K8 processors.

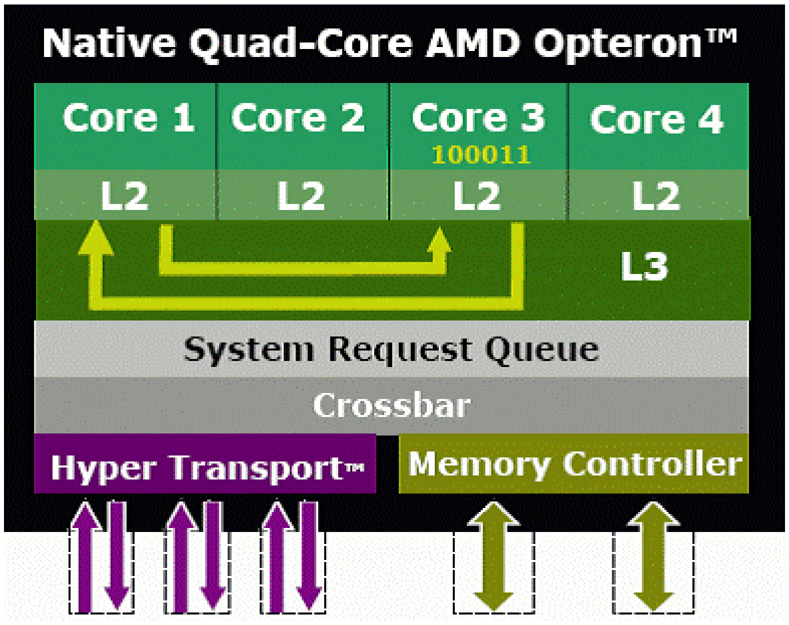


Pic.5b: Data transfer bus between L1 and L2 caches in K10 processors.

L3 Cache

To make up for the relatively small L2 cache, K10 acquired a shared between all cores 2MB L3 cache with associativity of 32. L3 cache is adaptive and exclusive: it stores all data evicted from L2 caches of all cores as well as the data shared by several cores. When the core issues a line read request, a special check is performed. If the line is only used by one core, it is removed from L3 freeing room for the line that is evicted from L2 cache of the requesting core. If the requested line is also used by another core, it remains in the cache. However, in order to accommodate the line evicted from L2 cache, another – older – line will be removed in this case.

L3 cache should help speed up the data transfer rate between the cores. As we have already found out, contemporary Athlon 64 processors exchange data between the cores via the memory bus. As a result, access to shared modified data occurs much slower. According to AMD's materials, quad-core K10 processors may exchange data via L3 cache. Once the request from one of the cores arrives, the core that has the modified data copies them to L3 cache, where the requesting core can read them from. The access time to modified data in the other core's cache should become much shorter. When we get a chance, we will certainly check it out.



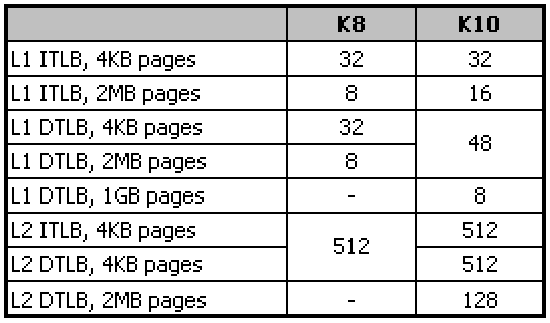
Pic.6: Data transfer between the cores in K10 processors.

L3 cache latency will evidently be higher than L2 cache latency. However, AMD materials suggest that the latency will vary adaptively depending on the workload. If the workload isn't too heavy, latency will improve, and under heavy workload the bandwidth will rise. We still have to check what really stands behind this.

TLB

Besides cache-memory for instructions and data, processors have one more type of cache-memory: translation-lookaside buffers (TLB). These buffers are used to store the connection between virtual and physical page addresses obtained from the page

tables. The number of TLB entries determines how many memory pages can be involved without additional costly page table walks. This is especially critical for applications that process memory data randomly, when they constantly request the data on different pages. K10 processor has much more translation buffers. For your convenience they are all given in the table below:



	K8	K10
L1 ITLB, 4KB pages	32	32
L1 ITLB, 2MB pages	8	16
L1 DTLB, 4KB pages	32	48
L1 DTLB, 2MB pages	8	
L1 DTLB, 1GB pages	-	8
L2 ITLB, 4KB pages	512	512
L2 DTLB, 4KB pages		512
L2 DTLB, 2MB pages	-	128

Table 1: TLB capacity of K8 and K10 processors.

As you see from the table, there are much more buffers for translation of 2MB pages. There also appeared support of large 1GB pages that may be very useful for servers processing large volumes of data. With appropriate support from OS, applications using large 2MB and 1GB pages should run considerably faster.

Memory Controller

When the requested data isn't found in any of the caches, the request is issued to the memory controller integrated onto the processor die. On-die location of the memory controller reduces the latencies during work with the memory, but at the same time it ties up the processor to a specific memory type, increases the die size and complicates the die selection process thus affecting the production yields. The on-die memory controller was one of the advantages of the K8 processors, however, sometimes it wasn't efficient enough. The memory controller of K10 processors will be improved significantly.

Firstly, it now can transfer data not only along one 128-bit channel, but also along two independent 64-bit channels. As a result, two or more processor cores can work more efficiently with the memory at the same time.

Secondly, the scheduling and reordering algorithms in the memory controller have been optimized. The memory controller groups reads and writes so that the memory bus could be utilized with maximum efficiency. Read operations have an advantage over writes. The data to be written is stored in the buffer of still unknown size (it is assumed to accommodate between 16 and 30 64-byte lines). By handling requested lines in groups we can avoid switching the memory bus from reading to writing and back all the time and thus save resources. It is allows to significantly improve performance during alternating read and write requests.

Thirdly, the memory controller can analyze requests sequences and perform prefetch.

Prefetch

Prefetch is a definite advantage of K8 processors. Integrated memory controller with low latency has let AMD processors to demonstrate excellent performance with the memory subsystem for a long time. However, K8 processors failed to prove as efficient with new DDR2 memory, unlike Core 2 with powerful prefetch mechanism. K8 processors have two prefetch units: one for the code and another one for the data. The data prefetch unit fetches data into the L2 cache basing on simplified successions.

K10 has improved prefetch mechanism.

First of all, k10 processors prefetch data directly into the L1 cache, which allows hiding the L2 cache latency when requesting data. Although it increases the probability of L1 cache pollution with unnecessary data, especially taking into account low cache associativity, AMD claims that it is a justified measure that pays off well.

Secondly, they implemented adaptive prefetch mechanism that changes the prefetch distance dynamically, so that the data could arrive in time and so that the cache wouldn't get loaded with data that is not needed yet. Prefetch unit became more flexible: now it can trains on memory requests at any addresses, and not only the addresses that fall into adjacent lines. Moreover, prefetch unit now trains on software prefetch requests.

Thirdly, a separate prefetch unit was added directly into the memory controller. The memory controller analyzes request successions from cores and loads the data into the write buffer utilizing the memory bus in the most optimal way. Saving prefetch lines in the write buffer helps keep cache-memory clean and reduce the data access latency significantly.

As a result, we see that the memory subsystem of K10 processors has undergone some positive improvements. But we still have to say that it still potentially yields to the memory subsystem in Intel processors in some characteristics. Among these features are: the absence of speculative loading at unknown address past the write operations, lower L1D cache associativity, narrower bus between L1 and L2 caches (in terms of data transfer rate), smaller L2 cache and simpler prefetch. Despite all the improvements, Core 2 prefetch is potentially more powerful than K10 prefetch. For example, K10 has no prefetch at instruction addresses so that we could keeps track of individual instructions, as well as no prefetch from L2 to L1 that could hide L2 latency efficiently enough. These factors can have different effects on various applications, but in most cases they will determine higher performance of Intel processors.

Let's take a quick look at other innovations introduced in K10 micro-architecture.

New Instructions

K10 processor acquired a few new instructions, expanding its functionality:

1. Extended bit operations on general purpose registers:

- LZCNT – Count Leading Zeros – counts the number of leading zero bits in operands;
- POPCNT – Bit Population Count – counts the number of bits having a value of 1 in operands;

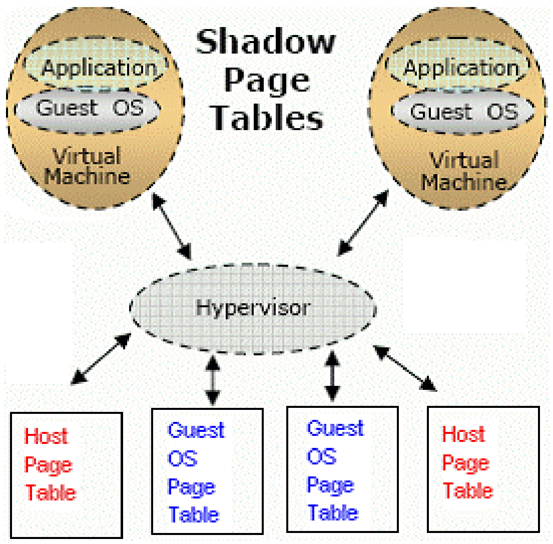
2. SSE registers processing aka SSE4a instructions:

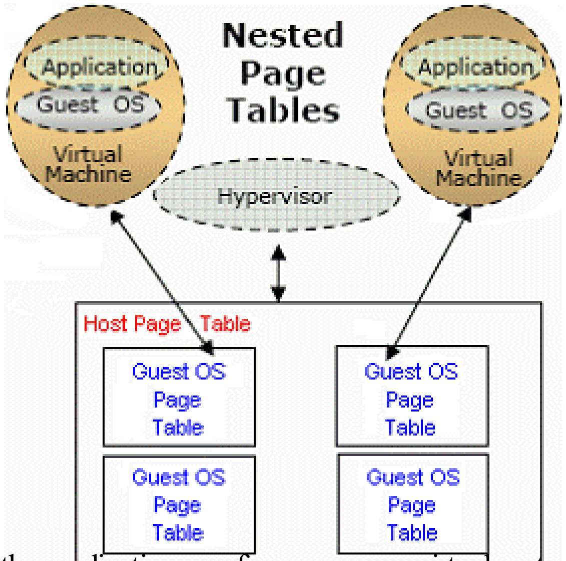
- EXTRQ – extracts the specified number of bits from the given position in the lower 64-bit part of SSE-register;
- INSERTQ – inserts the specified number of bits into the given position in the lower 64-bit part of SSE-register;
- MOVNTSS, MOVNTSD – streaming store (without involving cache-memory) of scalar floating-point values.

SSE4a instructions extension doesn't intersect in any way with the new Intel SSE4.1 and SSE4.2 instructions.

Virtualization

AMD continued to improve their virtualization technology that serves to launch several operating systems on a single PC. One of the most significant virtualization improvements is the use of Nested Paging. In this system the virtual machines pages are nested in the global hypervisor page table. If there is no link to the page in the TLB, the CPU performs all table transformations automatically, unlike Shadow Paging that requires a lot of resources to manage the table transformations of the virtual machines.





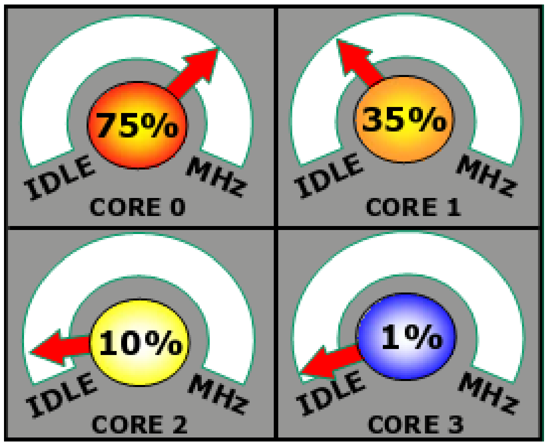
Pic.7a: Shadow Paging mode: when switching between virtual systems hypervisor switches between page tables clearing the TLB at the same time.

Pic.7b: Nested Paging mode: when switching between virtual systems hypervisor doesn't need to get involved to switch between page tables. TLB is not cleared at the same time.

Some data suggest that the use of Nested Paging increases the applications performance on a virtual system by 40% compared with the performance when Shadow paging mode is used.

Power and Frequency Management

New K10 processors will have new power and frequency management system. Each core will now work independently of the other, at its own frequency that may change dynamically depending on the load on each of the cores.



Pic.8: Independent core frequency management in K10 processors.

However, it is not clear yet how the performance of the shared L3 cache will be adjusted in this case. The core voltage is the same on all cores and is determined by the core under maximum workload. The memory controller manages its voltage independently of the cores and may lower the voltage in case of lower load.

Conclusion

All the information on the new AMD processors hasn't been released yet that is why we may still experience some surprises. However, we have enough knowledge to make the main conclusions about the new micro-architecture. Due to numerous core improvements new AMD processor promises to deliver a significant performance boost compared with the predecessor, especially in applications with intensive floating-point calculations. This CPU can compete successfully with Intel processors working at the same clock speed in a large number of applications and win the competition. New applications designed taking into account unique processor features, such as effective unaligned load and support of large 1GB pages, may also get an additional performance improvement.

However, the new CPU also has some weak spots compared with the Intel competitors. Here I have to point out caching and prefetch systems that may have a negative effect on performance in some applications. But the greatest issue in the fight for highest performance will most probably be the working frequency that is expected to be not high enough at first.

We would like to wish AMD to conquer new frequencies very soon and will continue keeping an eye on the competition between the two companies aimed at winning our, computer users', hearts.

The author would like to thank Maria Malich and Sergey Romanov aka GReY for editorial help.