6th Generation CPU Comparisons.

by Paul Hsieh

AMD K6	Intel Pentium II	Cyrix 6x86MX	Common features	Closing words	Glossary

While Agner Fog busily works out every little nuance and cycle disturbances of the Pentium II, I have written up some comparative documentation to give people a feel for the differences in the various 6th generation x86 CPUs. For this little ditty, I've chosen the Intel **P-II** (aka Klamath, P6), the AMD **K6** (aka NX686), and the Cyrix **6x86MX** (aka M2). These are all MMX capable 6th generation x86 compatible CPUs, however I am not going to discuss the MMX capabilities at all beyond saying that they all appear to have similar functionality and performance.

In what follows, I am assuming a high level of competence and knowledge on the part of the reader (basic 32 bit x86 assembly at least). For many of you, the discussion will be just slightly over your head. For those, I would recommend sitting through the 1 hour online lecture on the **Post-RISC architecture** by Charles Severence to get some more background on the state of modern processor technology. It is really an excellent lecture, that is well worth the time:

- Beyond RISC - The Post RISC Architecture (Mark Brehob, Travis Doom, Richard Enbody, William H. Moore, Sherry Q. Moore, Ron Sass, Charles Severance)

Much of the following information comes from online documentation from Cyrix, AMD and Intel. I have played a little with my Pentium and Pentium-II at work, as well as my AMD-K6 at home. I would also like to thank, **Dan Wax** from AMD who corrected me on several points about the K6, and **Andreas Kaiser** who also provided a lot of insightful information, and corrections gleened from his first hand experiences with the K6 and P-II CPUs.

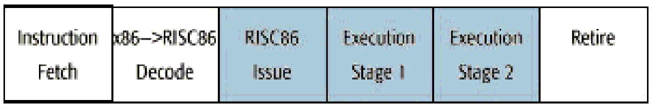
Comments welcome.

The AMD K6

The K6 architecture seems to mix some of the ideas of the P-II and 6x86MX architectures. They made trade offs, and decisions that they believed would deliver the maximal performance over all potential software. The have emphasized short latencies (like the 6x86MX) but the K6 translates their x86 instructions into RISC operations that are queued in large instruction buffers and feed many (7 in all) independent units (like the P-II.) While they don't always have the best single implementation of any specific aspect, this was the result of conscious decisions that they believe helps strike a balance that hits a good performance sweet spot. Versus the P-II, they avoid situations of really deep pipelining which has high penalties when the pipeline has to be backed out. Versus the Cyrix, the AMD is a fully POST-RISC architecture which is not as susceptible to pipeline stalls which artificially back ups other stages.

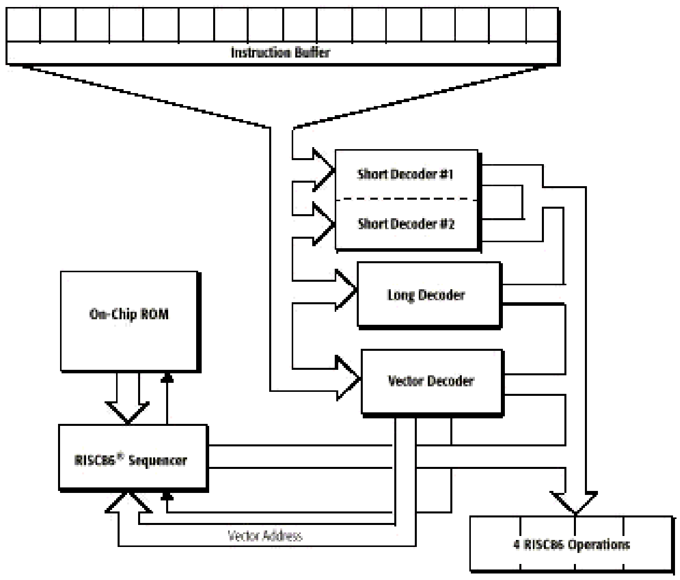
General Architecture

The K6 is an extremely short and elegant pipeline. The **AMD-K6 MMX Enhanced Processor x86 Code Optimization Application Note** contains the following diagrams:

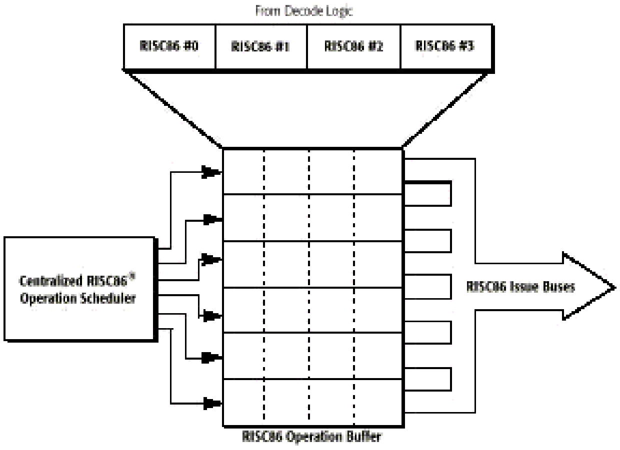


Which seems remarkably simple considering the features that are claimed for this processor. The secret, is that most of these stages do very complicated things. The light blue stages execute in an out of order fashion (and were colored by me, not AMD.)

The fetch stage, is much like a typical Pentium instruction fetcher, and is able to present 16 cache aligned bytes of data per clock. Of course this means that instructions that are not aligned will suffer an extra clock penalty before reaching the decode stage, much like they do on a Pentium.



The decode stage attempts to simultaneously decode 2 simple, 1 long, and fetch from 1 ROM x86 instruction(s). If both of the first two fail (usually only on rare instructions), the decoder is stalled for a second clock which is required to completely decode the instruction from the ROM. If the first fails but the second does not (the usual case when involving memory, or an override), then a single instruction or override is decoded. If the first succeeds (the usual case when not involving memory or overrides) then two simple instructions are decoded. Thus **the K6's execution rate is limited to a maximum of two x86 instructions per clock**. This decode stage decomposes the x86 instructions into **RISC86 ops**.



The issue stage is actually broken down into two pipelined stages. The first accepts up to 4 RISC86 instructions from the decoder and stores them in the scheduler. This scheduler can hold up to 6 clocks of instructions (or 12 dual issued x86 instructions.) And issues them at a maximum rate of 6 RISC86 instructions per clock (one per target execution unit.) The instructions are marked as issued, but not removed until retirement.

The second reads the issued instruction operands, and does so at a maximum rate of one per execution unit (the K6 has 6 execution units, and therefore can read 6 operands per clock.) This far exceeds the P-II's rate of 2 operands per clock. The main reason the K6 can do this is because the general registers support two reads per clock, at most one of the FPU and MMX units read the FPU registers which are seperate, and all possible address combinations (without immediate offsets) are computed in parallel, and are available to the load and store units on every clock! If this stage "fails" because of a data dependency, the instruction is re-entered into the scheduler, essentially moving the instruction ***backwards*** through the pipeline!

This is an ingenius design that allows the K6 to perform "late" data dependency determinations without over complicating the schedulers' issue logic.

The execution stages execute in one or two pipelined stages (with the exception of the floating point unit which is not pipelined, or complex instructions which stall those units during execution.) In theory, all six units can be executing at once. (Of course, architecturally, its impossible for the FPU and MMX units to be executing simultaneously, so really its a maximum of five units.)

The retire unit goes back to the scheduler to extract the in-order instructions at a rate of at most one issue clock's worth at a time (i.e., up to 4 RISC86 ops.) So the out of order nature of the CPU will tend to increase retire latency, but any decode penalties will reel the retire rate back in.

So, the K6's back-end will easily out perform the front end. This suggests that its scheduler and other buffers will tend to be empty or near empty most of the time. Of course, that does not mean they have gone untested, since dependencies on a data from a cache miss, will still backlog the instruction stream (this is true of the P-II as well.)

Branch Prediction

The K6 uses a very sophisiticated branch prediction mechanism which delivers better prediction and fewer stalls than the P-II. There is a 8192 table of two bit prediction entries which combine historic prediction information for any given branch with a heuristic that takes into account the results of nearby branches. Even branches that have somehow left the branch prediction table, still can have the benefit of nearby branch activity data to help their prediction. According to published papers which studied these branch prediction implementations, this allows them to achieve a 95% prediction rate versus the P-II's 90% prediction rate.

Additional stalls are avoided by using a 16 entry times 16 byte branch target cache which allows first instruction decode to occur simultaneously with instruction address computation, rather than requiring (E)IP to be known and used to direct the next fetch (as is the case with the P-II.) This removes an (E)IP calculation dependency and instruction fetch bubble. (This is a huge advantage in certain algorithms such as computing a GCD; see my examples for the code) The K6 allows up to 7 outstanding unresolved branches (which seems more than enough since the scheduler only allows up to 6 issued clocks of pending instructions in the first place.)

The K6 benefits additionally from the fact that it is a only 6 stage pipeline (as opposed to a 12 stage pipeline like the P-II) so even if a branch is incorrectly predicted it is only a 4 clock penalty as opposed to the P-II's 11-15 clock penalty.

One disadvantage pointed out to me by Andreas Kaiser is that misaligned branch targets still suffer an extra clock penalty and that attempts to align branch targets can lead to branch target cache aliasing. This is a good point, however it seems to me that you can help this along by hand aligning only your most inner loop branches.

Another (also pointed out to me by Andreas Kaiser) is that such a prediction mechanism does not work for indirect jump predictions (because the verification tables only compare a binary jump decision value, not a whole address.) The P-II's predictor however, does predict indirect jumps.

Back of envelope calculation

This all means that the average loop penalty is:

(95% * 0) + (5% * 4) = 0.2 clocks per loop

But because of the K6's limited decode bandwidth, branch instructions take up precious instruction decode bandwidth. There are no branch clocks in most situations, however, branching instructions end up taking a slot where there is essentially no calculations. In that sense K6 branches have a typical penalty of about 0.5 clocks. To combat this, the K6 executes the **LOOP** instruction in a single clock, however this instruction performs so badly on Intel CPUs, that no compiler generates it.

Floating Point

The common high demand, high performance FPU operations (FADD, FSUB, FMUL) all execute with a throughput and latency of 2 clocks (versus 1 or 2 clock throughput and 3-5 clock latency on the P-II.) Amazingly, this means that it can complete FPU operations *faster* than the P-II, however is worse on FPU code optimally *scheduled* for the P-II. Like the Pentium, in the P-II Intel has worked hard on fully pipelining the faster FPU operations which works in their favor. Despite AMD's claims to me to the contrary, the P-II's FXCH instruction is handled without introducing any execution bottlenecks just as it was on the Pentium. The K6's FXCH costs two clocks. More complex instructions such as FDIV, FSQRT and so on will stall more of the units on the P-II than on the K6. So, the K6 will allow better scheduling of non-FPU instructions with FPU instructions but has less execution bandwidth on more common FPU operations. In a test I wrote (admittedly rigged to favor Intel FPUs) the K6 measured to only perform at about 55% of the P-II's performance.

An interesting note is that FPU instructions on the K6 will retire **before** they completely execute. This is possible because it is only required that they work out whether or not they will generate and exception, and the execution state is reset on a task switch, by the OS's built-in FPU state saving mechanism.

This one is hard to choose sides on; let me explain why. For hand optimized FPU assembly code the P-II is the better approach because the goal of 1/2 clock throughput is a realizable one. On the other hand, the x86 FPU architecture has proven to be a daunting one for compiler writers to model for optimization. The problem is that compilers assume a sort of invariance of machine state from line to line of source code within a given scope, however really pushing the x86 FPU requires a multistate model (corresponding to the depth and contents of the FPU stack) of execution. Doing this is hard to weave together with branching, and subroutine calling (avoiding FPU stack overflow is yet another concern.) As such, compilers tend to use a very simple low FPU stack model (0 or 1 entry deep at a time) which tends to limit their performance by latency rather than maximal throughput. Compilers also have a hard time using FXCH effectively.

As the majority of software is written using compilers it would seem as though AMD's decision is the right one. However, the more *interesting software* (i.e., real time 3D applications such as games) tends to have a heavy dosage of hand optimized assembly in them, and some of those may have a good deal of Intel optimized FPU code. At least to some extent, longevity in the software industry tends to favor writers of *interesting software* which, if they optimize for the P-II FPU, will favor it over the K6. Another counter point is that modern day compilers are starting to use FXCH (though not always optimially even for the P-II, it will unduly penalize the K6.)

Personally, being the kind of guy that would hand optimize as a matter of course, I tend to favor the Intel approach, but the rest of the software industry doesn't see things my way. Increasingly, as compilers are getting better and better, and people like Michael Abrash are saying things like "the P-II doesn't need much hand optimizing to get good performance out of them", people are moving away (at least by percentage) from using assembly language at all. However, as compilers have been getting better, the K6 can't win here either.

Of course, the K6 documentation recommends that FPU scheduling is unnecessary. The K6 performance is not as sensitive to FPU scheduling where as the P-II still clearly derives great benefit from it. Furthermore, the FXCH instruction is just a very costly instruction for the K6 and it performs no instrinsic calculations. However, among their other more interesting recommendations is, in fact, to use the non-RISC like *load and execute instructions*. This makes perfect sense since they have independent, load/store and alu/fpu units which are better utilized by the more CISC-like x86 instructions. This also simplifies temporary result management versus the FPU stack, at the cost of scheduling advantages available only to the Intel CPUs (and, typically, the at the cost of accuracy, since results will tend to be held in DWORD or QWORD storage areas instead of 80 bit FPU stack registers.)

Cache

The K6's L1 cache is 64KB, which is twice as large as the P-II's L1 cache. But it is only 2 way set associative (as opposed to the P-II which is 4 way). This makes the replacement algorithm much simpler, but decreases its effectiveness in random data accesses. The increased size, however, more than compensates for the extra bit of associativity. For code that works with contiguous data sets, the K6 simply offers twice the working set ceiling of the P-II.

Like the P-II, the K6's cache is divided into two fixed caches for separate code and data. I am not as big a fan of split architectures (commonly referred to as the *Harvard Architecture*) because they set an artificial lower limit on your working sets. As pointed out to me by the AMD folk, this keeps them from having to worry about data accesses kicking out their instruction cache lines. But I would expect this to be dealt with by *associativity* and don't believe that it is worth the trade off of lower working set sizes.

Among the design benefits they do derive from a split architecture is that they can add pre-decode bits to just the instruction cache. On the K6, the predecode bits are used for determining instruction length boundaries. Their address tags (which appears to work out to 9 bits) point to a sector which contains two 32 byte long cache lines, which (I assume) are selected by standard associativity rules. Each cache line has a standard set of *dirty bits* to indicate accessibility state.

The K6 cache has some interesting features in its cache design that improve overall access to and from it. First, it allows up to one read and one write in the same cycle, which appears to be the similar to the P-II implementation. But more interestingly, cache misses and cache line fetches do not stall the entire cache interface, but rather only stalls the incident cache line. That is to say, access to other cache lines are still available even when a *fault* has occurred on another line. (These are claims by AMD folks, that at least one person, Andreas Kaiser, was not able to verify.) This is an improvement over the P-II and AMD denotes this under the term *non-blocking cache*. Such a mechanism can allow an astute programmer to write linear data access loops with a *data pre-fetch* mechanism. This lets the programmer get more work out of the K6's cache.

In addition, like the 6x86MX, the store unit of the K6 actually is buffered by a store Queue. This store queue can be up to 7(!) entries deep and does not incurr any stalls until it overflows. This, in conjunction with their non-blocking features makes for a more effective L1 cache than the P-II.

But perhaps more fundamentally, as AMD have said themselves, bigger is better, and at twice the P-II's size, I'll have to give the nod to AMD (though a bigger nod to the 6x86MX; see below.)

Other

- The K6 has a 2/3 (32 bits ready/64 bits ready) clock integer multiply, which is good counter to the P-II's 1/4 (through-put/latency) clock integer multiply. Programmers usually only use the base 32 bit LSB result of the multiply, and so are likely to achieve realistic 2 clock through-puts. On the other hand the P-II's 4 clock "hand's off" rule is unlikely to be so easily scheduled, since no contentions in 4 clocks is unlikely. In the real world, I would be very surprised if the P-II actually achieves a 1 or even 2 clock throughput.

- The K6 does not suffer the same kind of partial register stalls that the P-II does. Register contention is accurate down to the byte sub-register as required. Special clearing of the register is unnecessary. However 16 bit partial register instructions will have instruction decode overrides which will cost an extra clock.

- The K6 seems to prefer [esi+0] to [esi] memory addressing (for faster pre-decoding.) It appears to be a strange exception that doesn't have a real sensible explanation. Perhaps they just ran out of chip space.

- The K6 strikes a balance in decode, execute and retire bandwidth, by allowing up to 6 RISC86 operations to issue, 6 units to execute and 4 retires per clock. While resource contentions and other coding practices will tend to hamper this idealistic setup, this does appear to be better than the P-II. The P-II can issue 6 micro-ops, execute in 5 units, but only retire 3 micro-ops per clock. AMD's contention is that they alleviate a large bottleneck that the P-II has, namely the relatively low retire throughput. I'm not sure about this, since register output should not be directly affected by retirement, though memory write back scheduling is.

- The K6 has a fast (1 cycle) LOOP instruction. It looks like the Intel CPUs may be the lone wolves with their slow LOOP instruction. If you ask me, this is the most ideal instruction to use for loops.

- Of course, the K6 has more a sophisticated instruction decode implementation in the sense that they can decode two 7 byte instructions or one 11 byte instruction in a single clock. Like the 6x86MX (though, with an entirely different mechanism) it can only decode a maximum of two instructions per clock versus the P-II's maximum rate of 3 instructions per clock. However, the P-II's decoding is overly optimistic since it balks on any instructions more than 7 bytes long and is also limited by micro-op decode restrictions.

  Anyhow, this design is very much in line with AMD's recommendation of using complicated load and execute instructions which tend to be longer and would favor the K6 over the P-II. In fact, the AMD just seems better suited overall for the CISCy nature of the x86 ISA. For example, the K6 can issue 2 push reg instructions per clock, versus the P-II's 1 push reg per clock.

  According to AMD, the typical 32 bit decode bandwidth is about the same for both the K6 and the P-II, but 16 bit decode is about 20% faster for the K6. Unfortunately for AMD, if software developers and compiler writers heed the P-II optimization rules with the same vigor that they did with the Pentium, the *typical* decode bandwidth will change over time to favor the P-II.

- The K6's *issue to execute* scheduling is pretty cool. They use complete logical comparisons between pipeline stages to always find the best path to propagate from issue to operand read to execute. This is particularly effective to divide the work between the two integer units. The scheduler will actually push stalled instructions backwards through the pipeline to simplify and avoid over speculation in multi-clock stalls situations. This also allows other instructions to slip through rather than being caught behind a stalled instruction. This is an effective alternative to the P-II's reservation station which is an optional extra pipeline stage that serves a similar purpose.

  6x86MX seems to just let their pipelines accumulate with work moving only in a forward direction which makes them more susceptible to being backed up, but they do allow their X and Y pipes to swap contents at one stage.

- The K6 does not support the new P6 ISA instructions, specifically, the conditional move instructions. It also does not appear to support any like the Intel specific performance monitoring MSRs (besides the ever important TSC register.) So from a programmer's architecture point of view, the K6 is more like a Pentium than a Pentium-II. Its not clear that this is a real big issue since all the modern compilers still target the 80386 ISA.

- In their unveiling of the K6 they made a big deal about the modular design of the K6. Indeed, I got the feeling that this would enable them to tweak their design more easily, and more importantly debug higher clock rate limiting factors with relatively less effort than Intel designs or earlier AMD designs. However, fruit of this "benefit" has yet to bear. They introduced them at 233Mhz in May 1997 and it took them until Jan 1998 before releasing a 266Mhz version. Their moving to a 0.25 micron process seems to have paid off big for them since they were able to release a 300Mhz part a few months later.

As I have been contemplating the K6 design, it has really grown on me. Fundamentally, the big problem with x86 processors versus RISC chips is that they have too few registers and are inherently limited in instruction decode bandwidth due to natural instruction complexity. The K6 addresses both of these by maximizing performance of memory based in-cache data accesses to make up for the lack of registers, and by streamlining CISC instruction issue to be optimally broken down into RISC like sub-instructions.

It is unfortunate, that compilers are favoring Intel style optimizations. Basically there are several instructions and instruction mixes that compilers avoid due to their poor performance on Intel CPUs, even though the K6 executes them just fine. As an x86 assembly nut, it is not hard to see why I favor the K6 design over the Intel design.

Brass Tacks

The K6 is cheap, supports socket 7, has won some key OEM business (Digital, IBM and ACER are the big ones as of this writing), and has a promising future growth path. They are very confident in their process, and have made public the knowledge that their 0.25 micron shrink should yield a core die size of 68 square millimeters. (This is in comparison to the P-II which measures 150 square millimeters at that same process.) This gives AMD lee way to challenge low power CPUs, really squeeze on price, or perhaps add more functionality such as a larger cache, or perhaps even an integrated full clock rate L2 cache. Fundamentally, though, since their clock utilization is more in line with P-II speeds than 6x86MX speeds moving to higher clock rates will be crucial for them to remain competitive with Intel.

In the past few months it has come out that AMD is having difficulty keeping up with demand. This is good for demand, but bad that they cannot manufacture enough to make a real profit. Vinod Dahm recently resigned, and it is rumored that he is the scape goat for AMD’s manufacturing difficulties. At the same time there has been a is a surprising up swing in the affordable computer market. Again this means good things for both AMD (and Cyrix) but only if they have chips to sell people.

Update: As of 10/14/97 it has become known that AMD has sampled 266 parts to some vendors and magazines and tests have shown that it performs just a shade slower than the P-II at the same clock rate. (This is also true of the 233 part.) At the Microprocessor Forum, AMD announced an enhancement to their design called the **K6-2** which will support 24 new packed 32 bit floating point instructions for improving 3D geometry performance, an integrated on chip L2 cache and a 100 Mhz system bus. This chip is expected to be available in 1998. (They also announced the K7, which will use a P-II like slot architecture and use some Dec Alpha bus technology which they plan to ship in 1999.)

Further update: AMD, Cyrix, Centaur and Microsoft have gotten together to establish a standard set of packed floating point instruction extensions that all of these alternative x86 CPU vendors will incorporate into their next generation CPUs. (Microsoft is just there to crack the whip, while the CPU technology companies hammer out what instructions they want to go with.) Personally, I think the whole idea is a very good gamble as it intends to upset Intel's MMX-2 plans, as well as undermine Intel's current FPU advantage on specific applications such as Direct 3D's geometry pipeline which is very important in the growing 3D games market.

AMD very recently announced that 266Mhz versions of their K6 would find their way into notebooks shipping before the end of 1997. (I'm going to hold out for the K6-2 @ 300, from which I am expecting good things.)

Update: Compaq, the number 1 PC vendor has been added to the list of converts which now uses Cyrix, AMD, and Intel CPUs in its various brands of CPUs. On Jan 6, AMD did release 266 and 233 Mhz versions of their K6 processor using a 0.25 process. (Reading more reports and press releases has lead me to be desirous of the K6-3 at 300 Mhz as an upgrade to my K6-233 system.)

Update: AMD's Austin plant is apparently having severe difficulty yielding chips at the 0.25 size. Furthermore, initial yields of the K6-2 processor show its sweet spot to be at 250 Mhz which is somewhat lower than what AMD desires and needs to remain competitive with Intel. 1998 looks to be the make it or break it year for AMD. They need to solve these problems to return to profitability. (This report is unconfirmed, and may be nothing more than a rumor. My AMD contacts have indicated that the K6-2 is sampling at higher than 250.)

Update: AMD has released their specification for the 3DNOW instruction set. The upshot: packed 32bit floating point pairs executing with a peak throughput of 4 floating point operations per clock. For low precision floating point applications (i.e., 3D games) the K6-2 should be a monster. (Intel's current 80 bit FPU can only a achieve a maximum throughput of 1 floating point instruction per clock, and rarely achieves this peak.)

As a side effect of AMD's new design, MMX instructions can also now issue at a maximum rate of two per clock (that's twice the rate of any Intel CPU.) The K6-2 ought to do really well using the MMX PhotoShop filters. They've also added a faster FPU<->MMX unit switch command that does not clear the MMX/FPU registers.

Finally, they've added a **PREFETCH** command. Something we've been waiting for for a long time. The instruction basically will pre-ready a cacheline containing the source memory operand. With the right scheduling this instruction will allow better parallel memory access with execution.

Update: 04/07/98 AMD has released a 300Mhz K6 processor, to be used in IBM Aptiva PCs. (This came as a total surprise to me, I thought they were going to wait for the K6-2 to go to 300Mhz.)

AMD performance documentation links

The release of their x86 Optimization guide is what triggered me to write this page. With it, I had documentation for all three of these 6th generation x86 CPUs. Unfortunately, they often elect to go with terse explanations that assume the reader is very familiar with CPU architecture and terminologies. This lead me to some misunderstandings from my initial reading of the documentation (I'm just a software guy.) On the other hand, the examples they give really help clarify the inner workings of the K6.

Update: AMD has since updated their documentation with a few diagrams, and added some important documentation regarding their cache from previous documentation. This rev made the internal workings of the K6 crystal clear, and appears even to help clear up some of the clouds left by Intel's documentation (specifically description of the K6 scheduler has convinced me that the P-II scheduler works in a similary way.)

- AMD K6-2 specifications
- AMD-K6 Optimization Guide
- AMD-K6 Processor Data Sheet

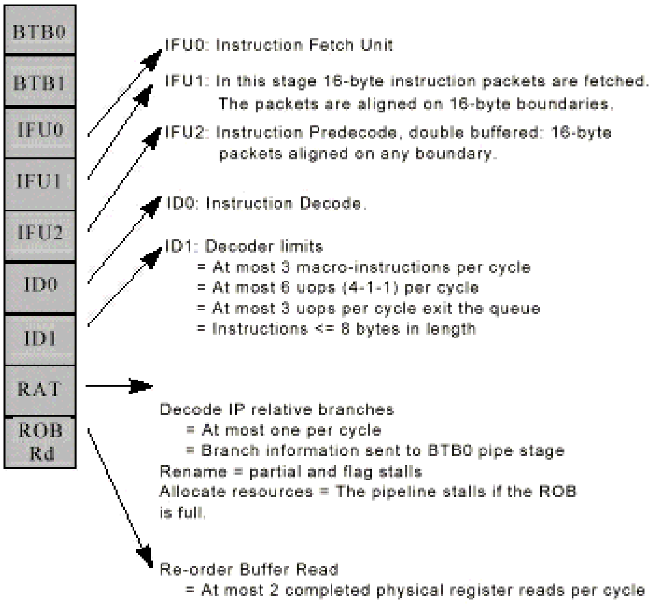
The Intel P-II

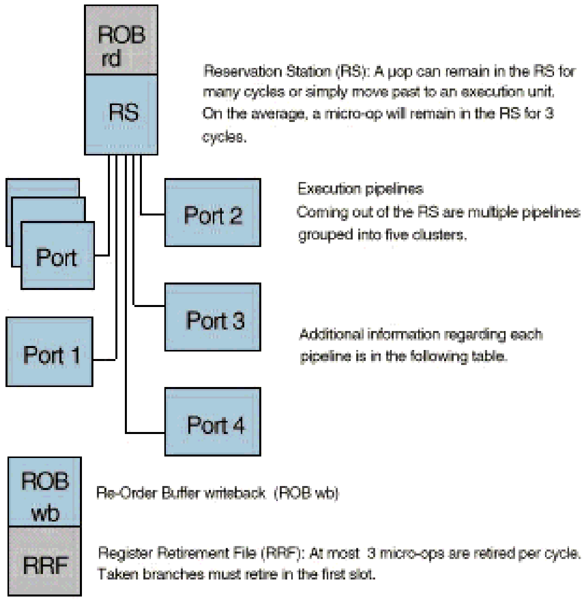
This was the first processor (I knew of) to have completely documented post-RISC features such as dynamic execution, out of order execution and retirement. (PA-RISC predated it as far as implementing the technology, however; I am suspicious that HP told Intel to either work with them on Merced, or be sued up the wazoo.) This stuff really blew me away when I first read it. The goal is to allow longer latencies in exchange for high through put (single cycle whenever possible.) The various stages would attempt to issue/start instructions with the highest possible probability as often as possible, working out dependencies, register renaming requirements, forwarding, resource contentions, as later parts of the instruction pipe by means of speculative and out of order execution.

Intel has enjoyed the status of "defacto standard" in the x86 world for some time. Their P6/P-II architecture, while not delivering the same performance boost of previous generational increments, solidifies their position. Its is the fastest, but it is also the most expensive of the lot.

General Architecture

The P-II is a highly pipelined architecture with an out of order execution engine in the middle. The Intel Architecture Optimization Manual lists the following two diagrams:





The two sections shown are essentially concatenated, showing 10 stages of in-order processing (since retirement must also be in-order) with 3 stages of out of order execution (RS, the Ports, and ROB write back colored in light blue by me, not Intel.)

Intel's basic idea was to break down the problem of execution into as many units as possible and to peel away every possible stall that was incurred by their previous Pentium architecture as each instructions marches forward down their assembly line. In particular, Intel invests 5 pipelined clocks to go from the instruction cache to a set of ready to execute micro-ops. (RISC architectures have no need for these 5 stages, since their fixed width instructions are generally already specified to make this translation immediate. It is these 5 stages that truly separate the x86 from ordinary RISC architectures, and Intel has essentially solved it with a brute force approach which costs them dearly in chip area.)

Each of Intel's "Ports" is an execution pipeline, each of which corresponds to an instructions type (integer, floating point, MMX, load, or store; branches are not resolved here.) As a note interest, Intel divides the execution and write back stages into two seperate stages (the K6 does not, and there is really no compelling reason for the P6's method that I can see.)

Although it is not as well described, I believe that Intel's re-order buffer serves substantially the same purpose as the K6's scheduler, and similarly the retire unit acts on instruction clusters in exactly the same way as they were issued. Thus the micro-op throughput is limited to 3 per clock (compared with 4 RISC86 ops for the K6.)

So when everything is working well, the P-II can take 3 simple x86 instructions and turn them into 3 micro-ops on every clock. But, as can be plainly seen in their comments, they have a bizzare problem: ***they can only read two physical input register operands per clock*** (rename registers are not constrained by this condition.) This means scheduling becomes very complicated. Registers to be read for multiple purposes will not cost very much, and data dependencies don't suffer from any more clocks than expected, however the very typical trick of spreading calculations over several registers (used especially in loop unrolling) will upper bound the pipeline to two micro-ops per clock because of a physical register read bottleneck.

In any event, the decoders (which can decode up to 6 micro-ops per clock) are clearly out-stripping the later pipeline stages which are bottlenecked both by the 3 micro-op issue and two physical register read operand limit. The front end easily outperforms the back end.

Branch Prediction

Central to facilitating the P-II’s aggressive fire and forget execution strategy is full branch prediction. The P-II has fewer slots, and contains far less data in their branch predict tables than the K6 or 6x86MX. Their functionality has been documented by Agner Fog, and can track very complex patterns of branching. They have advertised a prediction rate of about 90% (based on academic work using the same implementation.) This prediction mechanism was also incorporated into the Pentium MMX CPUs. Unlike the K6, the branch target buffer contains target addresses, not instructions and predictions only for the current branch. This means an extra clock is required for taken branches to be able to decode their branch target. Branches not in the branch target buffer are predicted statically (backward jumps taken, forward jumps not.) However, this "extra clock" is generally overlapped with execution clocks, and hence is not a factor except in short loops, or code loops with poorly translated code sequences (like compiled sprites.)

In order to do this in a sound manner, subsequent instructions must be executed "speculatively" under the proviso that any work done by them may have to be undone if the prediction turns out to be wrong. This is handled in part by renaming target write-back registers to shadow registers in a hidden set of extra registers. The K6 and 6x86MX have similar rename and speculation mechanisms, but with its longer latencies, it is a more important feature for the P-II. ***The trade off is that the process of undoing a mispredicted branch is slow (since the pipelines must completely flush) costing as much as 15 clocks (and no less than 11.) These clocks are non-overlappable with execution, of course, since the execution stream cannot be correctly known until the mispredict is completely processed.*** This huge penalty slightly offsets the performance of the P-II, especially in code in which no P6/P-II optimizations considerations have been made.

For programmers this means one major thing: Avoid mispredicted branches in your inner loops at all costs. Using tables or conditional move instructions are common methods, however since the predictor is used even in indirect jumps, there are situations with branching situations where you have no choice but to suffer from branch prediction penalties.

Floating Point

In keeping with their post-RISC architecture, the P-II’s have in some cases increased the latency of some of the FPU instructions over the Pentium for sake of simplicity and with idea that it hopefully will not matter if the code is properly scheduled. Intel says that FXCH requires no execution cycles, but does not explicitly state whether or not throughput bubbles are introduced. AMD claims its effectively impossible to implement FXCH in an out of order CPU without introducing bubbles, but my own benchmarking shows that the P-II FPU is as fast, if not faster on some hand crafted loops with plenty of FXCH instructions. I suspect that Intel has kept the FPU very Pentium-like in the Pentium II, with some special handling of the FXCH instruction (since their is no longer a "V-pipe".)

The Intel floating point design has traditionally beat the Cyrix and AMD CPUs on floating point performance and this still appears to hold true as tests with Quake and 3D Studio have confirmed. (But the K6 is closing in on the gap.)

Cache

The P-II's cache is 32KB divided into two fixed 16KB caches for separate code and data. These caches are 4-way set associative which decreases thrashing versus the K6. But relatively speaking, this is quite small and inflexible when compared with the 6x86MX's unified cache. I am not a big fan of the P-II's smaller, less flexible L1 cache, and it appears as though they have done little to justify it being half the size of their competitors' L1 caches. (The Celeron cache is 64K, however, its lack of L2 cache is a more overridingly negative performance factor.)

The greater associativity helps programs that are written indifferently with respect to data locality, but has no effect on code mindful of data locality (i.e., keeping their working set sequential and no larger than the L1 cache size.)

Other

- The P-II has a partial register stall which is very costly (7 clocks?) This occurs when writing to a sub-register within 7(?) clocks of writing to a 32 bit register. That is to say, writing to a ?l or ?h 8 bit register will cause a partial register stall when next reading the corresponding ?x or e?x register. The same is true of writing to a ?x register then reading the corresponding e?x register.

  My best reckoning as to why this is, is that the CPU always targets a new 32 bit wide temporary register for any register written to, including partial registers. So when the larger version of the register which requires all parts of the register is read, it requires that the register be first recombined from its "partial registers". This gives a complete explanation for the partial register stall behavior I have so far observed as well as an explanation as to why Intel couldn't "fix" this problem when moving from the Pentium Pro to the P-II.

  Intel recommends using XOR reg,reg or SUB reg,reg which can remove the stall at the cost of an extra instruction. But obviously this can be inappropriate if you need other parts of the register, to be non-zero.

  These clocks overlap with other execution clocks, so in some rare cases, no work arounds for the partial register stall condition are necessary.

- The P-II, like the P6, performs worse on 16 bit code per clock rate than the Pentium. (Significantly worse than the Cyrix 6x86MX, and somewhat worse than the K6.) However, the P-II is not as bad as the P6. In particular, it uses a small 16 bit segment/selector cache which the P6 does not.

- The P-II's data access actually require an additional address unit for stores. What this means is that memory writes must be broken down into "address store" and "data store" micro-ops. This increases data throughput latency (versus the K6.)

- The P-II can decode instructions to many, many micro-ops, but really only decodes optimally when 2 out of every 3 instructions are decoded to a single micro-op and in a specific "4-1-1" sequence (that is for three instructions to decode in parallel the first must decode to no more than 4 micro-ops, and the second and third in no more than 1 micro-op). Instructions must also be 8 bytes or less to allow other instructions to be decoded in the same clock. According to MicroProcessor Report, only one load or store memory operation can be decoded in the first of the at most 3 instructions. If this is true, it almost completely nullifies the "one load or store operations per clock" claim Intel makes (of course the second of the two store microops might execute as the same time as a load.)

Only under these circumstances can the P-II achieve its maximum rate of decoding 3 instructions per cycle. However, such scenarios don't seem terribly likely to me in ordinary code since all store instructions have to be broken down into an address micro-op, then a fetch or store micro-op (i.e., at least 2 micro-ops.)

Update: I recently tried to hand optimize some code, and found that it is actually not all that difficult to achieve the 3 instruction issue per clock, but that certainly no compiler I know of is up to the task. It turns out, though, that such activities are almost certainly a red herring since dependency bubbles, and the P-II's terribly limited retire bandwidth will end up throttling your performance anyways. My recommendation is to interleave parallel calculations when you can, and use **lea** whenever possible since it uses the ordinarily under taxed store unit instead of the over taxed integer alu units.

Update: A review of the Intel documentation as shown above has revealed to me that the P-II can be limited by the two register read operand limit in their front end.

- Stores are pipelined, but not queued like the 6x86MX or K6. This means cache misses necessarily stalls subsequent store micro-op *execution*. It is not totally clear to me if this stalls the load unit, but I am guessing not since the cache has been claimed to be **non-blocking**.

- Intel has leveraged its dominance in the market, advanced process and a daring approach to L2 cache usage to introduce their slot 1 cartridge interface to motherboards. The upshot of all of this is that they are able to use a larger heat sink and have better control over a more reliably yielded L2 cache running at reasonable clock rate (half the processor speed.)

  At the same clock rate, this is its biggest advantages over the current K6 whose L2 cache is tied to the chipset speed of 66Mhz.

Brass Tacks

When it comes right down to brass tax though, the biggest advantage of their CPU is the higher clock rates that they have achieved. As of this writing P-II 266's are definitely available and P-II 300's are just beginning to be deployed. (Update: Intel has announced and begun deploying P-II 333's) The chip also enjoys the benefit of the branding *Intel Inside*. Intel has spent a ton of money in brand name recognition to help lock its success over competitors. Like the Pentium, the P-II still requires a lot of specific coding practices to wring the best performance out of them, and its no doubt that many programmers will do this, and Intel has gone to some great lengths to write tutorials that explain how to do this.

Very recently, what consumers have been begging for for years, a transition to super cheap PCs has taken place. This is sometimes called the sub-$1000 PC market segment. Intel's P-II CPUs are simply too expensive (costing up to $800 alone) for manufacturers to build compelling sub-$1000 systems with them. As such, Intel has watched AMD and Cyrix pick up unprecedented market share. In a very reactionary move, Intel did announce that they would build a L2 cacheless version of the P-II to better compete in this fast growing market segment, (and that they were entertaining the idea of a redesigned motherboard for further cost reductions,) but I am expecting this to be a very poor performer.

It is becoming clearer to me that Intel has tied itself in knots with its P6/P-II design. Its too complicated, with too many strange rules and exceptions to compete toe to toe with the far more elegant K6 design; their performance lead being held only by their local L2 cache and high prices. They don't have a good way to drop the price, and they don't have a good strategy for future x86 designs (hence Merced). While the present looks rosey for Intel, I look at their future with a big question mark. Sooner or later they will run out of room trying to push their clock rate, at which point we will have the option of AMD's K7 or Cyrix's Cayenne (both of which are expected to have lots of functional improvements and design elegance) versus an over- clocked P-II/MMX2, or a Merced (which might as well have the "es" on the end looking at the price tag.)

My personal belief is that missing the sub-$1000 PC market segment is going to cost Intel dearly. And with the sub-$500 PC market just around the corner, they can expect a bleak future ahead of them. They simply have no good solution, and I don't see how they are going to be able to solve this conundrum they've gotten themselves into.

Update: Intel has released the "Celeron" (a 266Mhz P6 based part with no L2 cache) which is dirt cheap as well as a pair of high end P-II's (350Mhz and 400Mhz) based on a 100Mhz slot-architecture bus with are out of consumer ranges price-wise. The performance of the Celeron is so bad, that numerous PC vendors have come out and said that they definately will not make systems based around them. Packard Bell is the only company to actually endorse the Celeron.

Intel also has announced the intention to integrate an L2 cache on chip in a future product called "Mendocino". For some reason this product is being branded under the Celeron family of products, but this makes no sense since Mendocino will probably perform very well.

Intel performance documentation links

While Intel does have good documentation on their web site, they quite simply do not sit still with their URLs. It is impossible to keep track of these URLs, and I suspect Intel keeps changing their URLs based on some ulterior motive. All I can suggest is, slog through their links starting at the top:

- www.intel.com
- PC World report on Intel's new Celeron CPU

The Cyrix 6x86MX

Originally codenamed M2, this chip was very eagerly awaited. Cyrix had been making grand claims about their superior architecture and this was echoed by Cyrix's small but loyal following. Indeed it did come out, and indeed it is very interesting architecture, but to say the least, it was very late. Both AMD and Intel had announced their 6th generation MMX processors and enjoyed a couple months of extra revenue before Cyrix appeared on the scene. But as always, Cyrix's have hit the street at a substantially lower cost than its rivals, although recent price cuts from AMD may be marginalizing the 6x86MX at the high end.

The primary microarchitecture difference of the 6x86MX CPU versus the K6 and P-II CPUs is that it still does native x86 execution rather than translation to another instruction set.

Branch Prediction

The Cyrix CPU uses a 512 entry 2 bit predictor and this does not have a prediction rate that rivals either the P-II or K6 designs. However, both sides of the branch will have first instruction decoded simultaneously in the same clock. In this way, the Cyrix hedges its bets so that it doesn't pay such a severe performance penalty when its prediction goes wrong. Beyond this, it appears as though Cyrix has gone full **Post-RISC architecture** and supports a branch predict and speculative execution model. This fits nicely with their aggressive register renaming, and data forwarding model from the original 6x86 design. Because of potential FPU exceptions, all FPU instructions are treated the same way as branch prediction. I would expect the same to be true of the P-II, but Intel has not documented this, whereas Cyrix has.

According to their documentation, rather than using a general coloring scheme for speculation tracking (as described by Charles Severence) they have a fixed scheme of 4 levels of speculation, that are simply increased for every new speculative instruction issued (this is somewhat lower than the P-II and K6 which can have 20 or 24 *live* instructions at any one given time.) I believe this to equivalent to instruction coloring, but I have not reasoned it out fully to be sure.

The 6x86MX architecture is more lock stepped than the K6, and as such their issue follows their latency timings more closely. Specifically their decode, issue and address generation stages are executed in lock step, with any stalls from resource contentions, complex decoding etc, backing up the entire instruction fetch stages. However their design makes it clear that they do everything possible to reduce these resource contentions as early as possible. This is to be contrasted with the K6 design which is not lock step at all, but due to its late resource contention resolution, is often in the situation of re-issuing instructions after potentially wasting an extra clock that it didn't need to in its operand fetch stage. Another advantage of this for the 6x86MX is that backing out of a bad speculation is generally much faster than either the K6 or P-II.

Floating Point

The 6x86MX has significantly slower floating point. Just looking at their clock counts, the P-II ought to be at least twice as fast for floating point performance. The 6x86MX (and even older 6x86) tried to make up for it by having an FPU instruction FIFO. This means that most of the FPU clocks can be overlapped with integer clocks, and that a handful of FPU operations can be *in flight* at the same time, but in general requires hand scheduling and relatively light use of the FPU to maximally leverage it. Oddly enough, their FDIV and FSQRT performance is about as good if not better than the P-II implementation. This seems like an odd design decision, as optimizing FADD, FLD, FXCH and FMUL, are clearly of much higher importance.

Like AMD, Cyrix designed their 6x86MX floating point around the weak code that x86 compilers generate on FPU code. But, personally, I think Cyrix has gone way too far in ignoring FPU performance. Compilers only need to get a tiny bit better for the difference between the Cyrix and Pentium II to be very noticeable on FPU code.

Cache

The Cyrix's cache design is *unifed* at 64KB with a separate extra 256 byte instruction buffer. I prefer this design to the K6's and P-II's split code and data architecture, since it better takes into account the different dynamics of code usage versus data usage that you would expect in varied software designs. As an example, p-code interpreters, or just interpreted languages in general, you would expect to benefit more from a larger data cache. The same would apply to multimedia algorithms which would tend to apply simple transformations on large amounts of data (though in truth, your system benefits more from coprocessors for this purpose.) As another example, highly complex (compiled) applications that weave together the resources of many code paths (web browsers, office suite packages, and pre-emptive multitasking OSes in general) would prefer to have larger instruction caches. At both extremes, the Cyrix has twice the cache ceiling.

Thus the L1 cache becomes a sort of L2 cache for the 256 byte instruction line buffer, which allows the Cyrix design to complicate a much smaller cache structure with predecode bits and so on, and use the unified L1 cache more efficiently as described above. Although I don't know details, the prefetch units could try to see a cache miss comming and pre-load the instruction line cache in parallel with ordinary execution; this would compensate for the instruction cache's unusually small size, I would expect to the point of making it a mute point.

Other

- In moving to separate unit architectures, the K6 and P-II have sacrificed certain aspects of the Pentium-like fixed architecture. For example, the Pentium is capable of two loads and two stores per cycle to addresses with different cache line offsets. I would expect that the Cyrix has retained this as a side effect of staying with a fixed dual pipeline architecture.

- Faster selector/segment manipulation instructions than the Pentium. Apparently, the Cyrix has many more TLBs (transfer look aside buffers) that aids this significantly. This helps their 16 bit code significantly over the P-II, and even K6 which explains why their WinStone scores are so much better under Windows 95 then Windows NT.

- One clock LOOP instructions! This follows the PowerPC design choice of making a high throughput count down branch instruction. Like the PowerPC, they could in fact have implemented this with 100% accurate branch target prediction, however they did not document whether or not they have done this. Unfortunately, programmers have been using these instructions less, and less since starting with the Pentium Intel has been making this instruction slower.

- Two barrel shifters (one for each pipe) allowing greater parallelism with shift instructions. This is an advantage over both the P-II and K6 which each have only one unit that can handle shifts.

- There are no partial register stalls or smaller operand restrictions that I could find documented. Cyrix is clearly committed to retaining high performance of older 16 bit code. This is important for Windows 95, however less so for Windows NT.

- The Cyrix has a very interesting extension to their general architecture that allows them to use part of the L1 cache as a *scratch pad*. This presents a very interesting alternative for programmers who have complained about the x86's lack of registers. It is not clear that programmers would be willing to special case the Cyrix to use this feature, but you can bet that the drivers Cyrix writes for their GX platforms uses this feature.

Brass Tacks

To say the least, the 6x86MX design is radically different from either the P-II or K6 design. And when it comes to brass tax for them, they have a very interesting story to tell. Their CPU is very complicated compared to the capabilities of their fabrication process, and as such have only been able to reach clock rates of 188Mhz (Update: since being bought by National Semiconductor, they have hit 300Mhz on a 0.25 micron process.) But traditionally, the Cyrix designs have delivered more bang per clock than the comparable Intel designs, and surprising (at least to me) they reach performance levels just slightly off of a 233 Mhz P-II with their 188Mhz chips! Now, given that I've read all the public specifications of these various CPUs, something just didn't seem to jive with me. Although I agree that there seems to be numerous places with the Cyrix design has performance advantages, I am at loss to explain how they can achieve more than 25% better clock utilization than the Intel or AMD designs. When I first read the P6 design it seemed to me that Intel had subsumed nearly all of the architectural advantages of the 6x86 over the Pentium while adding speculative execution. The 6x86MX documentation doesn't point to enough to account for this 25% (in my opinion). My best justification is that one or more of the above mentioned differences has more of a performance impact than I realize (I would bet on the cache design) or there is something Cyrix isn't telling us.

Update: I can't believe I am such a retard! The Cyrix's clock rates are based on a **75Mhz** memory bus (as opposed to 66Mhz used by both AMD and Intel)! This alone is sufficient to explain the unusually good performance of the Cyrix CPUs relative to their clock rate. However, it does require that peripherals (not just the CPU) be able to run acceptably, at 75Mhz. While I don't have the global word, I think nearly everything will run in this situation just fine (otherwise why whould Cyrix release such a CPU?) but there is not generally accepted peripheral standard for 75Mhz buses. All this will be moot later on this year when (at least) AMD and Intel unveil 100Mhz bus standards.

Cyrix has recently been acquired by National Semiconductor. I'm not sure what this means to their process technology, but it ought to help their stability as a company. If Cyrix can somehow manage to get their clock rates up, I think the price/performance ratio of the Cyrix would give them a much larger portion of the market, and seriously threaten Intel.

Update: As it turns out National Semiconducter has a cross licensing patent agreement with Intel which includes access to CPU and Slot 1 technology. Although Cyrix has made non-committal comments so far, it would be a no-brainer for Cyrix to exercise this interesting legal position. Certainly it will provide protection for them from Intel lawsuits.

Update: Cyrix has announced their next generation CPU core codenamed Cayene. Its main features appear to be a dual FPU/MMX unit capable of executing 4 floating point operations per clock, as well as certain complex floating point instruction accelerations to give up to a 5 fold advantage over the P-II for lighting calculations. They also claim a die size reduction as a result of using a flipchip 0.25 micron processes that should yield PR300 and PR400 performance ratings. They also claimed this CPU will ship in 2H98, but recent agreements with AMD regarding "3D instruction extensions" may affect this.

Update: Cyrix has begun shipping a 300Mhz version of the 6x86MX based on 100Mhz socket 7 motherboards. I don't know further details.

Cyrix performance documentation links

- 6x86MX Processor Data Book

Common Features

All of these processors require *in-order* memory output and retirement. This can be reasoned out simply because of x86 architectural constraints. Specifically, in-order execution is required to do proper resource contention. Retirement is an artificial state **after** write-back, and as such shouldn't usually affect performance in any way. It is required for sound speculation resolution.

The out of orderedness is limited to execution and register write-back. The benefits of this are mostly in mixing multi-clock instructions with low-clock instructions. If too many multi-clock instructions are used, then all the benefits of dynamic scheduling of these processors go out the window, when their internal FIFO's start overflowing and they run out of resources. If too few multi-clock instructions are used, then the dynamic scheduling has too little to do as there is not much execution overlapping (ie parallelism) and there is not much benefit over older processor technology.

Contrary to what many might think, the kind of self-scheduling of code these CPUs perform does not preclude the benefits of hand scheduling. For example, in the cases of using high clock count instructions such as FDIV in a short loop, the CPU will quickly run into a situation where it cannot issue the FDIV, or other FPU instructions to the CPU because it has fallen too far behind and hence is overflowing whatever its speculative instruction FIFO mechanism. If the loop has all sorts of other integer instructions that it performs then if poorly scheduled, they will have to wait until the CPU can resolve the issue of the stalling FDIV before the other integer instructions can be issued, and hence less parallelism occurs. So in a loop where slow instruction over-running is expected, it is better to schedule faster instructions before the slower instructions.

Along these lines, avoiding successive high-clock instructions in a loop is obviously also a good thing, as over-running the speculative executions FIFOs is clearly bad. This means that bringing lots of low clock instructions together with these high clock count instructions into the same loops will tend to get more work done per clock.

However, this requires more accurate clock count measurements in ways that were not as well exposed in previous architectures. In the past, penalties for cache misses, instruction misalignment and other hidden side-effects were sort of ignored. This is because on older architectures, they hurt you no matter what, with no opportunity for instruction overlap, so the rule of avoiding them as much as possible was more important than knowing the precise penalty. With these arhitectures its important to know how much code can be parallelized with these cache misses. Issues such as PCI bus, chip set and memory performance will have to be more closely watched by programmers.

The K6's documentation was the clearest about its cache design, and indeed it does appear to have a lot of good features. Their predecode bits are used in a very logical manner (which appears to buy the same thing that the Cyrix's instruction buffer buys them) and they have stuck with the simple to implement

2-way set associativity. A per-cache line status is kept allowing asynchronous access to separate lines.

One thing I noticed, at least from the documentation, is that none of these CPUs take multimedia considerations into account in their cache design. What I mean is that multimedia algorithms tend to shuffle large sequential streams of data around without ever re-using previously fetched data. What this means is that in these multimedia loops the L1 caches will unnecessarily end up kicking out pretty much all their contents to accommodate the data streams which are never reused. It would seem to me, that it would be much better to somehow detect such data streams and either restrict them to a very small subset of the cache or at least mark them as least recently read. (Perhaps Cyrix does this, but are not documenting it ... that might explain their unusually high cpi rate.)

Final words

With *out of order execution* all these processors appear to promise the programmer freedom from complicated scheduling and optimization rules of previous generation CPUs. Just write your code in whatever manner pleases you and the CPU will take care of making sure it all executes optimally for you. And depending on who you believe, you can easily be lead to think this.

While these architectures are impressive, I don't believe that programmers can take such a relaxed attitude. There are still simple rules of coding that you have to watch out for (partial register stalls, 32 bit coding, for example) and there are other hardware limitations (at most 4 levels of speculation, a 4 deep FPU FIFO etc.) that still will require care on the part of the programmer in search of the highest levels of performance. I also hope that the argument that *what these processors are doing is too complicated for programmers to model* dies down as these processors are better understood.

Some programmers may mistakenly believe that the K6 and 6x86MX processors will fade away due to market dominance by Intel. I really don't think this is the case, as my sources tell me that AMD and Cyrix are selling every CPU they make, as fast as they can make them. The demand is definately there. 3Q97 PC purchases indicated an unusually strong sales for PCs at $1000 or less (dominated by Compaq machines powered by the Cyrix CPU), making up about 40% of the market.

The astute reader may notice that there are numerous features that I did not discuss at all. While its possible that it is an oversight, I have also intentionally left out discussion of features that are common to all these processors (data forwarding, register renaming, call-return prediction stacks, and out of order execution for example.) If you are pretty sure I am missing something that should be told, don't hesitate to send me feedback.

Update: Centaur, a subsidiary of IDT, has introduced a CPU called the **WinChip C6**. A brief reading of the documentation on their web site indicates that its basically a single pipe 486 with a 64K split cache **dual** MMX units, some 3D instruction extensions and generally more RISCified instructions. From a performance point of view their angle seems to be that the simplicity of the CPU will allow quick ramp up in clock rate. Their chip has been introduced at 225 and 240 MHz initially (available in Nov 97) with an intended ramp up to 266 and 300 Mhz by the second half of 1998. They are also targeting low power consumption, and small die size with an obvious eye towards the laptop market.

Unfortunately, even in the test chosen by them (WinStone; which is limited by memory, graphics and hard disk speed as much as CPU), they appear to score only marginally better than the now obsolete Pentium with MMX, and worse than all other CPUs at the same clock rate. These guys will have to pick their markets carefully and rely on good process technology to deliver the clock rates they are planning for.

Glossary of terms

- **Branch prediction** - a mechanism by which the processor guesses the results of a condition decision and thus assumes whether or not a conditional branch is taken.
- **Data forwarding** - the process of copying the contents of a write back value to an input value for another pipeline stage in the same clock.
- **(Instruction) coloring** - a technique for marking speculatively executed instructions to put them into equivalence classes of speculative resolution. The idea is that once a speculative condition has been resolved the corresponding instructions of that color are all deal with in the same way as being either retired or undone.
- **(Instruction) issue** - the first stage of a CPU pipeline where the instruction is first recognized and decoded.
- **Latency** - the total number of clocks required to completely execute an instruction. In maximal resource contention situations, this is usually the maximum number of clocks an instruction can take. (Often manufacturers will abuse the precise definition in their documentation by ignoring clocks that are assumed to (almost) always overlap. For example, most instruction on all fully pipelined processors really take at least 5 clocks from issue to retirement, however under normal circumstances most of those clocks are consistently overlapped by stages of other instructions, and hence are documented to take that many fewer clocks.) The goal of the Post-RISC architecture is to hide latencies to the maximal degree possible via parallelism.
- **Out of order execution** - a feature of the Post-RISC architecture whereby instructions may actual complete their calculation steps in an order different from that in which they were issued in the original program.
- **Post-RISC architecture** - a term coined by Charles Severance referring to the modern trend of CPUs to use techniques not found on traditional RISC processors such as speculative execution and register renaming in conjunction with instruction retirement.
- **Register contention** - a condition where an instruction is trying to use a register whose last write back or read has not yet completed.
- **Register renaming** - retargetting the output of an instruction to an arbitrary internal register that is virtually renamed to be the value of the architectural register.
- **Resource contention** - A condition where a register, alu or pipeline stage is required for an instruction but is currently in use, or scheduled to be used, by a previously unretired instruction.
- **Retirement** - The process by which the CPU knows that an instruction has really completed (usually occurs right after, or along with write back and branch target resolution.)
- **Speculative execution** - a processor state in which execution proceeds even if it is not yet known whether such an execution path will actually be taken. Usually occurs after a branch instruction is issued but before it is resolved.
- **Throughput** - the minimal number of clocks that an instruction needs during the flow of a program. In ideal situations this is just the time it takes to issue the instruction, assuming there are no resource contentions with other subsequent or previous instructions.

Updated 05/14/98