

A Look at Centrino's Core: The Pentium M

Pentium M is clearly the latest and greatest version of Intel's venerable "P6" microarchitecture, on which Intel's 32-bit desktop chips from the Pentium Pro down through the PIII were all based. But Banias includes a few tricks from the P4, too, as well as some innovations that set it apart from both processors.

The present article takes a look at Banias a.k.a. the Pentium M (or PM), the processor at the heart of Intel's Centrino platform. Unlike with some of my previous articles, I won't even attempt to cover most of what a CPU geek could possibly want to know about the PM. I cover only the points that I think are the most interesting and the most important from a big-picture perspective. Of course, this means that I'll gloss over a few things, and that I'll leave other things out entirely. That being the case, I've made it a point to resume my recently lapsed practice of including a bibliography with links to sources for more information on the PM. If you want a more general overview of the processor's features, especially with regards to power-saving technologies like Enhanced Speedstep, then you should look there.

As I noted above, the PM is the latest iteration of the venerable PPro (a.k.a. "P6") architecture, probably the most commercially successful microprocessor architecture of all time. Intel's standard line about the PM is that they took what they learned from the P4 and mixed it with the PIII, and that's true in a certain sense. But the best way to look at the PM is as an evolutionary advance of the P6 microarchitecture.

In the following few sections, we'll take a look at what's known about the Pentium M's microarchitecture. Unfortunately, Intel has been close-fisted with the details, but there's enough out there to allow us to put together a general picture of the core.

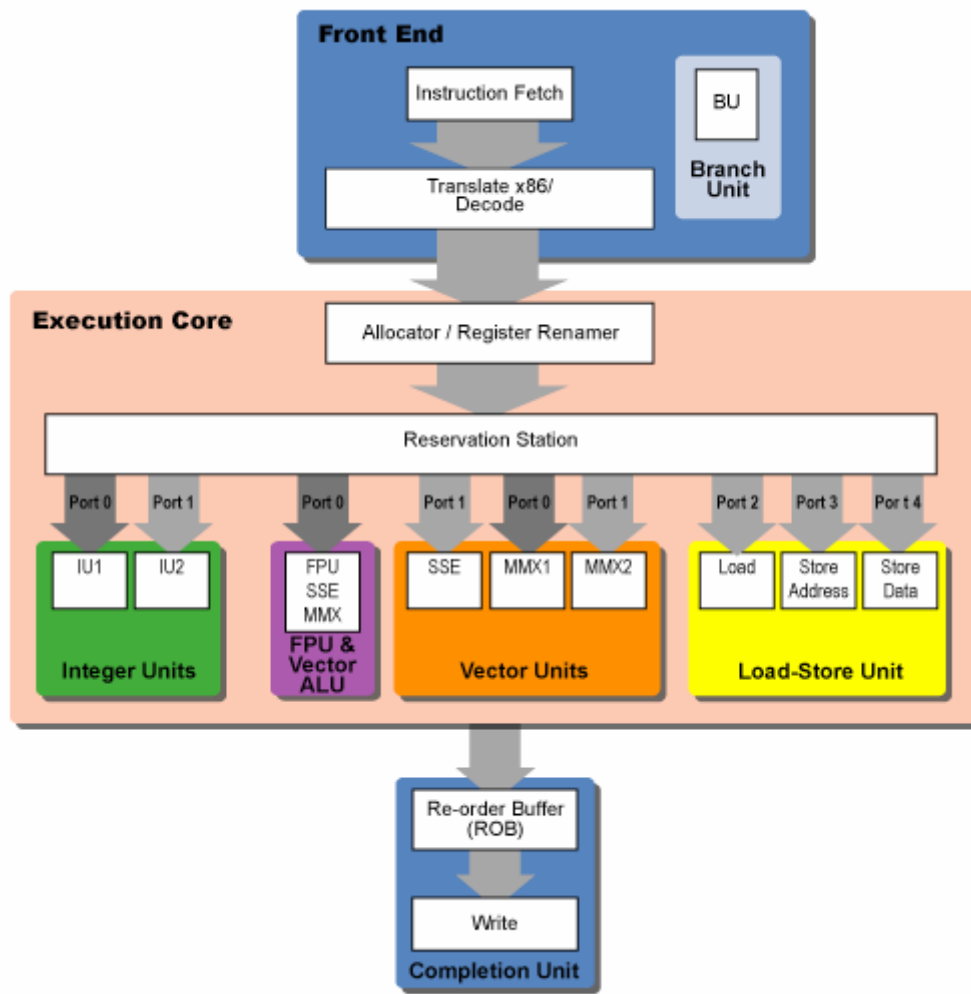
Pipeline and execution core

I normally discuss the front end first and the execution core second. Because Centrino is based on the PIII version of the P6 core, and because the execution core is where it differs the least from its predecessor, I'll go ahead and cover it first.

Not much is known about the details of the PM's pipeline--only that it's a little longer than that of the P3 yet shorter than that of the P4. I think that this slight increase in pipeline depth was done for two reasons. First, Intel wanted more headroom to ramp up the PM's clock speed, especially since they opted to tune the circuit layout for lower power consumption versus higher clockspeed. A few extra pipeline stages will help offset the hit to clockspeed headroom that their new layout gives. The second reason possibly has something to do with micro-ops fusion, about which we'll talk more in a moment.

Since I've never really covered the P6 architecture in much detail, I don't have an article to which I can refer you for the lowdown on the P6 execution core. So what follows is a

brief breakdown of the P6 execution core in its PIII incarnation; you can pretty safely take all of this to apply to the PM, as well.



The diagram above makes the core look a bit "wider" than it actually is, but note the port designations on the arrows leading from the reservation station to the execution units; there are five "dispatch ports" in the P6's reservation station through which micro-ops (a.k.a. uops) can travel to reach the execution units. The RS can dispatch up to 5 uops per cycle to the units, but the average is more around 2.5 to 3 uops per cycle.

I think it's possible that the number of entries in the PM's RS has been decreased from the P6 core's 20, and that the number of entries in the ROB has been decreased from 40. I say this because of the PM's micro-ops fusion technique, which I'll cover in the next section.

You'll notice that the P6 core's floating-point and integer capabilities are weaker than those of the 970; when compared to the P4, its floating-point capabilities don't look too bad but its integer capabilities are much weaker. Also, its SIMD capabilities aren't too shabby, either, when compared to the P4. Speaking of SIMD, I suspect that if the core

diagram above is inaccurate with respect to the PM, then it's because Intel has somehow changed the way that the FPU + vector units handle the distribution of floating-point and vector work. This is just sort of a random hunch, though.

All of this means that, clock-for-clock, the PM's execution core stands up quite well to that of the P4, but of course, we would expect exactly this from just looking at the results of the numerous PIII vs. P4 comparisons that accompanied the P4's launch. Now, at this point, I'd normally feel obligated to point out that clock-for-clock comparisons don't mean much when the pipelines of the two architectures are as different as those of the PM and the P4. But when power is an issue, as it is in this case, the P4's higher clockrates spell higher power consumption, so the PM's better clock-for-clock performance is actually a mark in its favor as a mobile platform.

Aside from a deeper pipeline and thus higher clock speeds, one of the main performance-enhancing features that the P4 boasted over the PIII was improved branch prediction. Well, the PM takes the P4's branch prediction and does it not one but two better; the next section tells how.

The front end: branch prediction, micro-op fusion, and the stack engine

The front-end of the Pentium M is where all the interesting work has been done, at least from the standpoint of microarchitecture. This is where Intel has improved the P6 core in a way that makes it not only more power-efficient but more powerful as well.

Branch prediction

One of the most important changes to the PM's front end is its improved branch prediction capabilities. If you've read many of my articles in the past two years then you've been hearing a lot about the importance of branch prediction. This technique has emerged as one of the most important tools for keeping modern superpipelined architectures from wasting execution cycles on branch-related stalls. The PowerPC 970 spends massive resources on branch prediction, using two separate branch prediction schemes and history table for determining which one works the best. The P4 also spends quite a few transistors on branch prediction, but I'll say more about its branch prediction scheme in just a moment.

The PM takes the P4's scheme and builds on it by adding two new, more specialized branch predictors that work in tandem with primary, P4-based branch predictor: the **loop detector** and the **indirect branch predictor**.

More branch prediction

The loop detector

One of the most common types of branches that a processor encounters is test condition of a loop. In fact, loops are so common that the static method of branch prediction, in which all branches are assumed to be loop test conditions that evaluate to "taken", works reasonably well for processors with shallow pipelines (i.e. if a loop is set to go through 100 iterations before the loop counter runs out and the loop's test condition evaluates to "not taken," then a static branch predictor will have been 99% accurate for that portion of the code).

One problem with static branch predictors is that they *always* make a wrong prediction on the final iteration of the loop--the iteration on which the branch evaluates to "not taken"--thereby forcing a pipeline stall as the processor recovers from the erroneous prediction. And of course, static prediction works poorly for non-loop branches, like standard if-then branches. In such branches, a static prediction of "taken" is roughly the equivalent of a coin toss.

Dynamic predictors, like the P4's branch predictor, fix this by shortcoming by keeping track of the execution history of a particular branch instruction--whether it was "taken" or "not taken" the past few times it was evaluated--in order to give the processor a better idea of what its outcome on the current pass will probably be. The bigger the table used to track the branch's history, the more data the branch predictor has to work with and the more accurate its predictions can be. Now let's look at P4's branch predictor as I described it in my first PowerPC 970 article:

The P4's branch predictor uses a 4096-entry branch history table (BHT) to keep track of the branches in a program by recording whether they were taken or not taken when they executed on previous cycles. Using the BHT in combination with an undisclosed but highly accurate branch prediction algorithm, the P4 decides whether each branch that it encounters should be taken or not taken. If the branch is taken, a 4K-entry (or 4096-entry) branch target buffer (BTB) attached to the BHT helps the P4 predict the address at which it should begin speculative execution.

One of the main shortcomings of the P4's branch predictor is that, even though its BHT is relatively sizeable, it doesn't have enough space to store all the relevant execution history information on the loop branches that tend to take a very large number of iterations.

Now that we've got enough background, let's take a look at Intel's own description of the loop detector:

The Loop Detector analyzes branches to see if they have loop behavior. Loop behavior is defined as moving in one direction (taken or not-taken) a fixed number of times interspersed with a single movement in the opposite direction. When such a branch is detected, a set of counters are allocated in the predictor such that the behavior of the program can be predicted completely accurately for larger iteration counts than typically captured by global or local history predictors.

So the loop detector plugs into the PM's P4-style BHT and augments it by providing it with extra, more specialized data on the currently executing program's loops. The PM's predictor can then use this data to avoid being caught by surprise when the loop's exit condition is fulfilled.

The second type of new branch predictor that the PM introduces is the indirect predictor. Branches come in two flavors: direct and indirect. Direct branches have the branch target explicitly specified in the instruction, which means that the branch target is fixed at compile time. Indirect branches, on the other hand, have to load the branch target from a register and so can have multiple potential targets. Storing these potential targets is the function of the branch target buffer (BTB) described in the above P4 quote.

Direct branches are the easiest to predict, and can often be predicted with upwards of 97% accuracy. Indirect branches, in contrast, are notoriously difficult to predict, and at least one paper that I read on the topic puts indirect branch prediction accuracies at around 75% using the standard BTB method.

The PM's indirect branch predictor works a little like the branch history table that I described above, but instead of storing information on whether a particular branch was "taken" or "not taken" the past few times it was executed, it stores information about each indirect branch's favorite target addresses--which targets a particular branch usually likes to jump to, and under which conditions it likes to jump to them. So the PM's indirect branch predictor knows that for a particular indirect branch in the BHT with a specific set of "favorite" target addresses stored in the BTB, under this set of conditions it tends to jump to one target address, while under that set of conditions it likes to jump to another.

Intel claims that the combination of the loop detector and indirect branch predictor gives Centrino a 20% increase in overall branch prediction accuracy, resulting in a 7% real performance increase. Of course, the usual caveats apply to these statistics, i.e. the increase in branch prediction accuracy and that increase's effect on real-world performance depends heavily on the type of code being run.

Improved branch prediction gives the PM a leg up not only in terms of performance but in terms of power efficiency as well. Because of its improved branch prediction capabilities, the PM wastes less energy speculatively executing code that it will then have to throw away once it learns that it mispredicted a branch.

Instruction decoding and micro-op fusion

The PM's "micro-op" fusion feature is one of the processor's most fascinating innovations for this reason: it functions remarkably like the PowerPC 970's instruction grouping scheme. The analogy isn't perfect, but it is striking. Let me explain.

Most of the folks who've gotten this far into one of my articles are probably aware that the P6 architecture beaks down x86 ISA instructions into smaller instructions, called micro-ops or uops. The majority of x86 instructions translate into a single uop, while a

small minority translate into two or three uops and a very small, very exotic and rarely used minority (string handling instructions, mainly) translate into sequences of many uops.

Because uops are what the P6's execution core dynamically executes, they're what the reorder buffer (ROB) and reservation station (RS) must keep track of so that they can be put back in program order after being executed out-of-order. In order to track a large number of in-flight uops the P6 core needs a large number of entries in its ROB and RS, entries that take up transistors and hence consume power. The PM cuts down on the number of ROB and RS entries needed by fusing together certain types of related uops and assigning them to a single ROB and RS entry. The fused uops still execute separately, as if they were normal, unfused uops, but they're tracked as a group.

If you read my [970 coverage](#), then what I just said should sound very familiar. The 970 does essentially the same thing--first it breaks down PPC architected instructions into multiple smaller internal operations (iops) for processing by the execution core, and then it groups them together for tracking in the group completion table; the iops still execute out-of-order, but they're tracked as groups to cut down on the number of GCT entries, thereby saving power and allowing the 970 to track more instructions with less logic and less overhead.

Now, you'll recall that the 970 had to introduce a number of architectural features to make this grouping scheme work; I'm talking specifically about the elaborate issue queue structure. Indeed, it's apparent that the 970 was designed from the ground up with such grouping in mind. The PM, on the other hand, is based on the P6 core, and as such the designers were more constrained in the kinds of things they could do to make micro-op fusion work. Specifically, the P6 core's reservation station and issue port structure would have to be pretty heavily rethought for a pervasively used micro-op fusion scheme to be feasible, which is why the PM's designers appear to have limited the technique to two very specific types of instructions: the load-op instruction type and the store.

It's not completely clear to me whether or not the two "examples" of micro-op fusion cited in Intel's literature are indeed the only kinds of micro-op fusion that goes on. I've been led astray by examples before, so I proceed with some caution here. I do, however, have good reasons to suspect that load-op and store instructions are the only two types that are fused, reasons which I'll outline near the end of the present discussion.

Store instructions on the P6 are broken down into two uops: a store-address uop and a store-data uop. The store-address uop is the command that calculates the address where the data is to be stored, and it's sent to the address generation unit in the P6's store-address unit for execution. The store-data uop is the command that writes the data to be stored into the outgoing store data buffer, from which the data will be written out to memory when the store instruction retires; this command is executed by the P6's store-data unit. Because the two operations are inherently parallel and are performed by two separate execution units on two separate issue ports, these two uops can be executed in

parallel--the data can be written to the store buffer at the same time that the store address is being calculated.

According to Intel, the PM's instruction decoder not only decodes the store operation into two separate uops but it also fuses them together. I suspect that there has been an extra stage added to the decode pipe to handle this fusion. The instructions remain fused until they're issued (or "dispatched," in Intel's language) through the issue port to the actual store unit, at which point they're treated separately by the execution core. When both uops are completed they're treated as fused by the core's retirement unit.

A load-and-op, or "read-modify" instruction is what it sounds like: an instruction that loads data from memory into a register, and then performs an operation on that data. Such instructions are broken down into two uops: a load uop that's issued to the load unit and is responsible for loading the needed data, and a second instruction that performs some type of operation on the loaded data and is executed by the appropriate execution unit.

Load-and-op instructions are treated in much the same way as store instructions with respect to decoding, fusion and execution. The main difference in practice is that load-and-op instructions are inherently serial, so they must be executed in sequence.

Now let's take a look at what Intel claims for the results of this scheme.

We have found that the fused micro-ops mechanism reduces the number of micro-ops handled by the out-of-order logic by more than 10%. The reduced number of micro-ops increases performance by effectively widening the issue, rename and retire pipeline. The biggest boost is obtained during a burst of memory operations, where micro-op fusion allows all decoders, rather than the one complex decoder, to process incoming instructions. This practically widens the processor decode, allocation, and retirement bandwidth by a factor of three.

It'll probably help if I parse the above quote for you. The P6 decoding hardware has three decoders: two simple/fast decoders, which are capable of translating into uops only those x86 instructions that translate into a single uop, and one complex/slower decoder, which translates all multi-uop x86 instructions. Before the advent of uop fusion, store and load-and-op instructions, being multi-uop instructions, had to pass through the complex decoder. So in situations where there was a burst of memory operations, the complex decoder would be backed up with work while the other two decoders sat idle. With uop fusion, however, the two simple/fast decoders are now capable of decoding stores and load-and-op instructions, because they can now produce a single, fused uop for these two types of instructions.

Now that that's clear, let's take a look at the next paragraph, on the performance increases afforded by uop fusion:

The typical performance increase of the micro-op fusion is 5% for integer code and 9% for Floating Point (FP) code. The store fusion contributes most of the performance

increase for integer code. The two types of fused micro-ops contribute about equally to the performance increase of FP code.

This is about what we would expect, given that floating-point code is usually more memory-intensive than integer code.

The quoted paragraphs above also lend credence to my claim that only store and load-and-op instructions are fused by Centrino's decoders. This seems to be the plain sense of the statements, and it also fits with the aforementioned fact that if more types of instructions were fused then Intel might've had to do something weird with Centrino's reservation station and issue ports (called "dispatch ports" in Intel-speak). Because the load and store units each have their own dedicated ports (the load unit is on port 2 and the store units are on ports 3 and 4), it's likely that the modifications to the RS necessary to accommodate uop fusion were kept to a minimum. Finally, there's also the fact that few arithmetic instructions break up into multiple uops, so the performance benefits to rewiring the RS to accommodate fused arithmetic instructions would be offset by increases in the complexity, and hence the power consumption, of the RS. It's worth noting, however, that a hypothetical future iteration of Centrino with a higher power budget, like for use on the desktop or in a blade server, might be under fewer restrictions and so it might use uop-fusion more extensively.

The stack execution unit

The stack execution unit is the hardest to understand of the PM's innovations, because it performs such a specialized function. I'll give my best shot at an explanation, so that you can have a general idea of what it does and what role it plays in saving power.

x86 includes stack-manipulation instructions like POP, PUSH, RET, and CALL, for use in passing parameters to functions in function calls. During the course of their execution, these instructions update x86's dedicated stack pointer register, ESP. In the PIII's and P4's cores, this update was carried out by a special uop, which was generated by the decoder and was tasked with using the IEUs (integer execution units) to update ESP by adding to it or subtracting from it as necessary.

The dedicated stack engine eliminates these special ESP-updating uops by monitoring the decoder's instruction stream for incoming stack instructions and keeping track itself of those instructions' changes to ESP; updates to the ESP are handled by a dedicated adder attached to the stack engine. So because the PM's front end has dedicated hardware for tracking the state of ESP and keeping it updated, there's no need to issue those extra ESP-related uops to the IEUs.

This technique has a few benefits. The obvious benefit is that it reduces the number of in-flight uops, which means fewer uops per task and less power consumed per task. Then, because there are fewer IEU uops in the core, the IEUs are free to process other instructions because they don't have to deal with the stack-related ESP updates.

Conclusions

To recap a bit from a previous section, initial benchmarks pitting the PIII against the P4 showed the PIII comparing quite favorably to its younger sibling. In fact, even with a slight clockspeed advantage the P4 still couldn't trounce its predecessor in some benchmarks. Inevitably, as the P4 started to pull away from the PIII in clockspeed, it began to open up a performance gap, as well. Aided by its superior branch prediction capabilities, trace cache, and deeper pipeline, the P4's Netburst architecture eventually outran the older P6 architecture and took its place as Intel's flagship performance processor. This victory came at a cost, though: the P4's high clockspeed made for high power requirements, making the architecture less than ideal for mobile applications.

This brings us to the Pentium M. The PM takes one of the P4's strengths--its branch prediction capabilities--and improves on it, adding its advantages to the strengths of the P6 architecture. The PM also deepens the P6's pipeline a bit, allowing for better clockspeed scaling, but without making clockspeed the central factor driving performance. In short, the PM looks like what the P4 might have been, had Intel not been so obsessed with the MHz race--it's a kind of alternate past, but one that may provide a glimpse of Intel's future.

With an increased power budget and, say, [IA-32e support](#), the Pentium M would be a serious desktop contender. With a massive 1MB L2 cache and a hefty 64K L1 cache, it already has twice the cache of Intel's P4. It may not scale to the clockspeed heights of the Netburst architecture, but with the power consumption of Prescott spiraling out into the stratosphere, that might not be such a bad thing. AMD, after all, has gotten off the MHz train and is marketing based on performance. Centrino has Intel doing the same with regard to the mobile space.

Consider [this report](#) on a talk by Intel CT Patrick Gelsinger:

Gelsinger, giving the final speech of an Intel technology forum, showed the audience a slide of the impossibly high power needs of computer processors as a way of arguing that chip designers must radically change chip architectures, and that Intel would be the company to do just that.

"We need a fresh approach," Gelsinger said. "We need an architectural paradigm shift."

That "fresh approach" might look something like a future version of the Pentium M, in a system that gangs together multiple low-power processors and relies heavily on multi-threaded software.

Not coincidentally, this looks something like what I've elsewhere described as IBM's plan for the PowerPC 970 and its offspring. In fact, let's take a look at how the 970 stacks up against the PM from a power consumption point of view:

Processor	Process	Size	Transistors	Core	Power
-----------	---------	------	-------------	------	-------

				voltage	
1.2GHz PM lv	0.13um	84 mm2	77 million	1.18v	12 Watts
1.2GHz 970	0.13um	121 mm2	58 million	1.1v	19 Watts

As you can see, the Pentium M low voltage and the PowerPC 970 are comparable in terms of power requirements for the same speed on the same process. The Pentium M will never be the floating-point and SIMD monster that the 970 is, but for most desktop and server workloads it could provide a competitive alternative, especially with some of the aforementioned tweaks.

In sum, the Pentium M's future looks bright. There are lots of places that Intel could go with this architecture, and with power consumption having fully arrived as a problem with Prescott the Pentium M's architecture just might have a brighter future than Netburst. I look for Intel begin rolling out non-mobile-oriented variants of the Pentium M across a variety of market niches--from small form factor computers to blade servers. A Pentium M derivative with IA-32e? It could certainly happen.

At the very least, the Pentium M represents a solid Plan B in case Prescott goes up in a puff of smoke at some point, and a solid Plan B is what you'd expect from a company whose motto is "only the paranoid survive."