3.
The microarchitecture of Intel, AMD and VIA CPUs
An optimization guide for assembly programmers and compiler makers

By Agner Fog. Technical University of Denmark.

Contents

1 Introduction

1.1 About this manual

This is the third in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.

2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.

3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.

5. Calling conventions for different C++ compilers and operating systems.

The latest versions of these manuals are always available from www.agner.org/optimize. Copyright conditions are listed on page 217 below.

The present manual describes the details of the microarchitectures of x86 microprocessors from Intel and AMD. The Itanium processor is not covered. The purpose of this manual is to enable assembly programmers and compiler makers to optimize software for a specific microprocessor. The main focus is on details that are relevant to calculations of how much time a piece of code takes to execute, such as the latencies of different execution units and the throughputs of various parts of the pipelines. Branch prediction algorithms are also covered in detail.

This manual will also be interesting to students of microarchitecture. But it must be noted that the technical descriptions are mostly based on my own research, which is limited to what is measurable. The descriptions of the "mechanics" of the pipelines are therefore limited to what can be measured by counting clock cycles or micro-operations (μops) and what can be deduced from these measurements. Mechanistic explanations in this manual should be regarded as a model which is useful for predicting microprocessor behavior. I have no way of knowing with certainty whether it is in accordance with the actual physical structure of the microprocessors. The main purpose of providing this information is to enable programmers and compiler makers to optimize their code.

On the other hand, my method of deducing information from measurements rather than relying on information published by microprocessor vendors provides a lot of new information that cannot be found anywhere else. Technical details published by microprocessor vendors is often superficial, incomplete, selective and sometimes misleading.

My findings are sometimes in disagreement with data published by microprocessor vendors. Reasons for this discrepancy might be that such data are theoretical while my data are obtained experimentally under a particular set of testing conditions. I do not claim that all information in this manual is exact. Some timings etc. can be difficult or impossible to measure exactly, and I do not have access to the inside information on technical implementations that microprocessor vendors base their technical manuals on.

The tests are done mostly in 32-bit and 64-bit protected mode. Most timing results are independent of the processor mode. Important differences are noted where appropriate. Far jumps, far calls and interrupts have mostly been tested in 16-bit mode for older processors. Call gates etc. have not been tested. The detailed timing results are listed in manual 4: "Instruction tables".

Most of the information in this manual is based on my own research. Many people have sent me useful information and corrections, which I am very thankful for. I keep updating the manual whenever I have new important information. This manual is therefore more detailed, comprehensive and exact than other sources of information; and it contains many details not found anywhere else.

This manual is not for beginners. It is assumed that the reader has a good understanding of assembly programming and microprocessor architecture. If not, then please read some books on the subject and get some programming experience before you begin doing complicated optimizations. See the literature list in manual 2: "Optimizing subroutines in assembly language" or follow the links from www.agner.org/optimize.

The reader may skip chapters describing old microprocessor designs unless you are using these processors in embedded systems or you are interested in historical developments in microarchitecture.

Please don't send your programming questions to me, I am not gonna do your homework for you! There are various discussion forums on the Internet where you can get answers to your programming questions if you cannot find the answers in the relevant books and manuals.

1.2 Microprocessor versions covered by this manual

The following families of x86 microprocessors are discussed in this manual:

Microprocessor name	Microarchitecture code name	Abbreviation
Intel Pentium (without name suffix)	P5	P1
Intel Pentium MMX	P5	PMMX
Intel Pentium Pro	P6	PPro
Intel Pentium II	P6	P2
Intel Pentium III	P6	P3
Intel Pentium 4 (NetBurst)	Netburst	P4
Intel Pentium 4 with EM64T, Pentium D, etc.	Netburst, Prescott	P4E
Intel Pentium M, Core Solo, Core Duo	Dothan, Yonah	PM
Intel Core 2	Merom, Wolfdale	Core2
Intel Core i7	Nehalem	Nehalem
Intel 2nd generation Core	Sandy Bridge	Sandy Bridge
Intel 3rd generation Core	Ivy Bridge	Ivy Bridge
Intel 4th generation Core	Haswell	Haswell
Intel 5th generation Core	Broadwell	Broadwell
Intel 6th generation Core	Skylake	Skylake
Intel Atom 330	Diamondville	Atom
Intel Bay Trail	Silvermont	Silvermont
AMD Athlon	K7	AMD K7
AMD Athlon 64, Opteron, etc., 64-bit	K8	AMD K8
AMD Family 10h, Phenom, third generation Opteron	K10	AMD K10
AMD Family 15h, Bulldozer	Bulldozer	Bulldozer
AMD Family 15h, Piledriver	Piledriver	Piledriver
AMD Family 15h, Steamroller	Steamroller	Steamroller
AMD Bobcat	Bobcat	Bobcat
AMD Kabini, Temash, etc.	Jaguar	Jaguar
VIA Nano, 2000 series		Nano 2000
VIA Nano, 3000 series	Isaiah	Nano 3000

Table 1.1. Microprocessor families

The abbreviations here are intended to distinguish between different kernel microarchitectures, regardless of trade names. The commercial names of microprocessors often blur the distinctions between different kernel technologies. The name *Celeron* applies to P2, P3, P4 or PM with less cache than the standard versions. The name *Xeon* applies to P2, P3, P4 or Core2 with more cache than the standard versions. The names *Pentium D* and *Pentium Extreme Edition* refer to P4E with multiple cores. The brand name Pentium was originally applied to the P5 and P6 microarchitectures, but the same name has later been reapplied to some processors with later microarchitectures. The name Centrino applies to Pentium M, Core Solo and Core Duo processors. Core Solo is rather similar to Pentium M. Core Duo is similar too, but with two cores.

The name Sempron applies to a low-end version of Athlon 64 with less cache. Turion 64 is a mobile version. Opteron is a server version with more cache. Some versions of P4E, PM, Core2 and AMD processors have multiple cores.

The P1 and PMMX processors represent the fifth generation in the Intel x86 series of microprocessors, and their processor kernels are very similar. PPro, P2 and P3 all have the sixth generation kernel (P6). These three processors are almost identical except for the fact that new instructions are added to each new model. P4 is the first processor in the seventh generation which, for obscure reasons, is not called seventh generation in Intel documents. Quite unexpectedly, the generation number returned by the CPUID instruction in the P4 is not 7 but 15. The confusion is complete when the subsequent Intel CPUs: Pentium M, Core, and later processors all report generation number 6.

The reader should be aware that different generations of microprocessors behave very differently. Also, the Intel and AMD microarchitectures are very different. What is optimal for one generation or one brand may not be optimal for the others.

2 Out-of-order execution (All processors except P1, PMMX)

The sixth generation of microprocessors, beginning with the PPro, provided an important improvement in microarchitecture design, namely out-of-order execution. The idea is that if the execution of a particular instruction is delayed because the input data for the instruction are not available yet, then the microprocessor will try to find later instructions that it can do first, if the input data for the later instructions are ready. Obviously, the microprocessor has to check if the later instructions need the outputs from the earlier instruction. If each instruction depends on the result of the preceding instruction, then we have no opportunities for out-of-order execution. This is called a dependency chain. Manual 2: "Optimizing subroutines in assembly language" gives examples of how to avoid long dependency chains.

The logic for determining input dependences and the mechanisms for doing instructions as soon as the necessary inputs are ready, gives us the further advantage that the microprocessor can do several things at the same time. If we need to do an addition and a multiplication, and neither instruction depends on the output of the other, then we can do both at the same time, because they are using two different execution units. But we cannot do two multiplications at the same time if we have only one multiplication unit.

Typically, everything in these microprocessors is highly pipelined in order to improve the throughput. If, for example, a floating point addition takes 4 clock cycles, and the execution unit is fully pipelined, then we can start one addition at time T, which will be finished at time T+4, and start another addition at time T+1, which will be finished at time T+5. The advantage of this technology is therefore highest if the code can be organized so that there are as few dependences as possible between successive instructions.

2.1 Instructions are split into µops

The microprocessors with out-of-order execution are translating all instructions into micro-operations - abbreviated µops or uops. A simple instruction such as `ADD EAX,EBX` generates only one µop, while an instruction like `ADD EAX,[MEM1]` may generate two: one for reading from memory into a temporary (unnamed) register, and one for adding the contents of the temporary register to `EAX`. The instruction `ADD [MEM1],EAX` may generate three µops: one for reading from memory, one for adding, and one for writing the result back to memory. The advantage of this is that the µops can be executed out of order. Example:

```
; Example 2.1. Out of order processing
mov eax, [mem1]
imul eax, 5
add eax, [mem2]
mov [mem3], eax
```

Here, the `ADD EAX,[MEM2]` instruction is split into two µops. The advantage of this is that the microprocessor can fetch the value of `[MEM2]` at the same time as it is doing the multiplication. If none of the data are in the cache, then the microprocessor will start to fetch `[MEM2]` immediately after starting to fetch `[MEM1]`, and long before the multiplication can start. The splitting into µops also makes the stack work more efficiently. Consider the sequence:

```
; Example 2.2. Instructions split into µops
push eax
call func
```

The PUSH EAX instruction may be split into two µops which can be represented as SUB ESP,4 and MOV [ESP],EAX. The advantage of this is that the SUB ESP,4 µop can be executed even if the value of EAX is not ready yet. The CALL operation needs the new value of ESP, so the CALL would have to wait for the value of EAX if the PUSH instruction was not split into µops. Thanks to the use of µops, the value of the stack pointer almost never causes delays in normal programs.

2.2 Register renaming

Consider the example:

```
; Example 2.3. Register renaming
mov eax, [mem1]
imul eax, 6
mov [mem2], eax
mov eax, [mem3]
add eax, 2
mov [mem4], eax
```

This piece of code is doing two things that have nothing to do with each other: multiplying [MEM1] by 6 and adding 2 to [MEM3]. If we were using a different register in the last three instructions, then the independence would be obvious. And, in fact, the microprocessor is actually smart enough to do just that. It is using different temporary registers in the last three instructions so that it can do the multiplication and the addition in parallel. The IA32 instruction set gives us only seven general-purpose 32-bit registers, and often we are using them all. So we cannot afford the luxury of using a new register for every calculation. But the microprocessor has plenty of temporal registers to use. The microprocessor can *rename* any of these temporary registers to represent a logical register such as EAX.

Register renaming works fully automatically and in a very simple way. Every time an instruction writes to or modifies a logical register, the microprocessor assigns a new temporary register to that logical register. The first instruction in the above example will assign one temporary register to EAX. The second instruction is putting a new value into EAX, so a new temporary register will be assigned here. In other words, the multiplication instruction will use two different registers, one for input and another one for output. The next example illustrates the advantage of this:

```
; Example 2.4. Register renaming
mov eax, [mem1]
mov ebx, [mem2]
add ebx, eax
imul eax, 6
mov [mem3], eax
mov [mem4], ebx
```

Assume, now, that [MEM1] is in the cache, while [MEM2] is not. This means that the multiplication can start before the addition. The advantage of using a new temporary register for the result of the multiplication is that we still have the old value of EAX, which has to be kept until EBX is ready for the addition. If we had used the same register for the input and output of the multiplication, then the multiplication would have to wait until the loading of EBX and the addition was finished.

After all the operations are finished, the value in the temporary register that represents the last value of EAX in the code sequence is written to a permanent EAX register. This process is called retirement (see e.g. page 80).

All general purpose registers, stack pointer, flags, floating point registers, vector registers, and possibly segment registers can be renamed. Many processors do not allow the control words, and the floating point status word to be renamed, and this is the reason why code that modifies these registers is slow.

3 Branch prediction (all processors)

The pipeline in a modern microprocessor contains many stages, including instruction fetch, decoding, register allocation and renaming, µop reordering, execution, and retirement. Handling instructions in a pipelined manner allows the microprocessor to do many things at the same time. While one instruction is being executed, the next instructions are being fetched and decoded. The biggest problem with pipelining is branches in the code. For example, a conditional jump allows the instruction flow to go in any of two directions. If there is only one pipeline, then the microprocessor does not know which of the two branches to feed into the pipeline until the branch instruction has been executed. The longer the pipeline, the more time does the microprocessor waste if it does not know which branch to feed into the pipeline.

The microarchitecture tries to overcome this problem by feeding the most probable branch into the pipeline and execute it *speculatively*. Speculative execution means that the instructions are decoded and executed, but the results are not *retired* into the permanent register file, and memory writes are pending until the branch instruction is finally resolved. If it turns out that the guess was wrong and the wrong branch was executed speculatively, then the pipeline is flushed, the results of the speculative execution are discarded and the other branch is fed into the pipeline. This is called a branch misprediction, and the result is that several clock cycles are wasted. The number of wasted clock cycles is approximately equal to the length of the pipeline.

The designers are inventing more and more sophisticated mechanisms for predicting which way a branch will go, in order to minimize the frequency of branch mispredictions. The history of branch behavior is stored in order to use past history for predicting future behavior. This prediction has two aspects: predicting whether a conditional jump will be taken or not, and predicting the target address that a conditional or unconditional jump goes to. A cache called Branch Target Buffer (BTB) stores the target address of all jumps. The target address is stored in the BTB the first time an unconditional jump is executed and the first time a conditional jump is taken. The second time the same jump is executed, the target address in the BTB is used for fetching the predicted target into the pipeline, even though the true target is not calculated until the jump reaches the execution stage. The predicted target is very likely to be correct for unconditional jumps, but not certain, because the BTB may not be big enough to contain all jumps in a program, so different jumps may replace each other's entries in the BTB. The risk of misprediction is much higher for conditional jumps.

3.1 Prediction methods for conditional jumps

When a conditional jump is encountered, the microprocessor has to predict not only the target address, but also whether the conditional jump is taken or not taken. If the guess is right and the right target is loaded, then the flow in the pipeline goes smoothly and fast. But if the prediction is wrong and the microprocessor has loaded the wrong target into the pipeline, then the pipeline has to be flushed, and the time that was been spent on fetching, decoding and perhaps speculatively executing instructions in the wrong branch is wasted.

Saturating counter

A relatively simple method is to store information in the BTB about what the branch has done most in the past. This can be done with a saturating counter, as shown in the state diagram in figure 3.1.

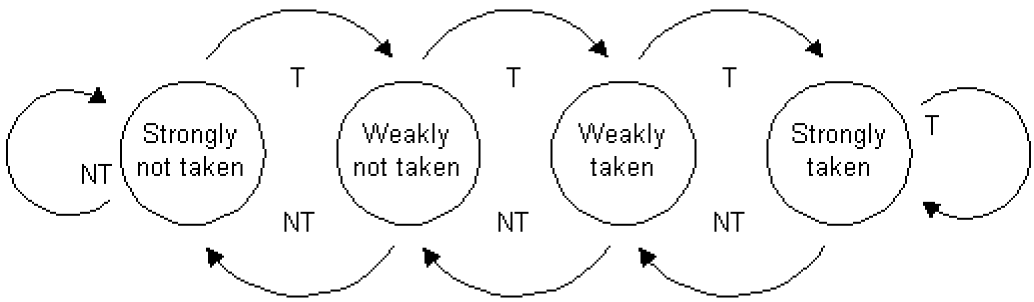

Figure 3.1. Saturating 2-bit counter

This counter has four states. Every time the branch is taken, the counter goes up to the next state, unless it already is in the highest state. Every time the branch is not taken, the counter goes down one step, unless it already is in the lowest state. When the counter is in one of the highest two states, it predicts that the branch will be taken the next time. When the counter is in one of the lowest two states, it predicts that the branch will not be taken the next time. If the branch has been not taken several times in a row, then the counter will be in the lowest state, called "strongly not taken". The branch then has to be taken twice for the prediction to change to taken. Likewise, if the branch has been taken several times in a row, it will be in state "Strongly taken". It has to be not taken twice before the prediction changes to not taken. In other words, the branch has to deviate twice from what it has done most in the past before the prediction changes.

This method is good for a branch that does the same most of the time, but not good for a branch that changes often. The P1 uses this method, though with a flaw, as explained on page 18.

Two-level adaptive predictor with local history tables

Consider the behavior of the counter in figure 3.1 for a branch that is taken every second time. If it starts in state "strongly not taken", then the counter will alternate between state "strongly not taken" and "weakly not taken". The prediction will always be "not taken", which will be right only 50% of the time. Likewise, if it starts in state "strongly taken" then it will predict "taken" all the time. The worst case is if it happens to start in state "weakly taken" and alternates between "weakly not taken" and "weakly taken". In this case, the branch will be mispredicted all the time.

A method of improving the prediction rate for branches with such a regularly recurring pattern is to remember the history of the last *n* occurrences of the branch and use one saturating counter for each of the possible 2^n history patterns. This method, which was invented by T.-Y. Yeh and Y. N. Patt, is illustrated in figure 3.2.

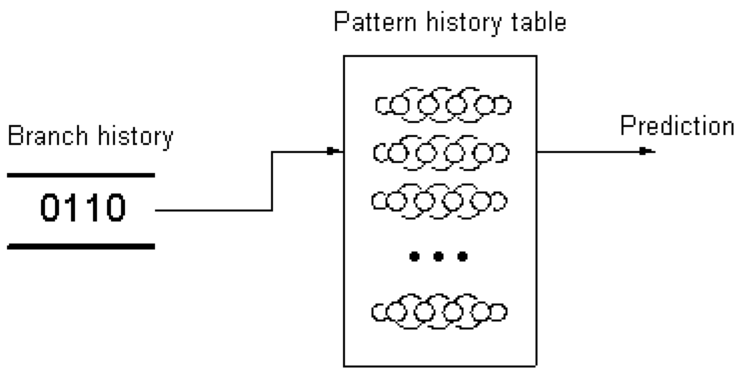

Figure 3.2. Adaptive two-level predictor

Consider the example of $n = 2$. This means that the last two occurrences of the branch are stored in a 2-bit shift register. This branch history register can have 4 different values: 00, 01, 10, and 11; where 0 means "not taken" and 1 means "taken". Now, we make a pattern history table with four entries, one for each of the possible branch histories. Each entry in the pattern history table contains a 2-bit saturating counter of the same type as in figure 3.1. The branch history register is used for choosing which of the four saturating counters to use. If the history is 00 then the first counter is used. If the history is 11 then the last of the four counters is used.

In the case of a branch that is alternately taken and not taken, the branch history register will always contain either 01 or 10. When the history is 01 we will use the counter with the binary number 01B in the pattern history table. This counter will soon learn that after 01 comes a 0. Likewise, counter number 10B will learn that after 10 comes a 1. After a short learning period, the predictor will make 100% correct predictions. Counters number 00B and 11B will not be used in this case.

A branch that is alternately taken twice and not taken twice will also be predicted 100% by this predictor. The repetitive pattern is 0011-0011-0011. Counter number 00B in the pattern history table will learn that after 00 comes a 1. Counter number 01B will learn that after a 01 comes a 1. Counter number 10B will learn that after 10 comes a 0. And counter number 11B will learn that after 11 comes a 0. But the repetitive pattern 0001-0001-0001 will not be predicted correctly all the time because 00 can be followed by either a 0 or a 1.

The mechanism in figure 3.2 is called a two-level adaptive predictor. The general rule for a two-level adaptive predictor with an n-bit branch history register is as follows:

> Any repetitive pattern with a period of n+1 or less can be predicted perfectly after a warm-up time no longer than three periods. A repetitive pattern with a period p higher than n+1 and less than or equal to 2^n can be predicted perfectly if all the p n-bit subsequences are different.

To illustrate this rule, consider the repetitive pattern 0011-0011-0011 in the above example. The 2-bit subsequences are 00, 01, 11, 10. Since these are all different, they will use different counters in the pattern history table of a two-level predictor with $n = 2$. With $n = 4$, we can predict the repetitive pattern 000011-000011-000011 with period 6, because the six 4-bit subsequences: 0000, 0001, 0011, 0110, 1100, 1000, are all different. But the pattern 000001-000001-000001, which also has period 6, cannot be predicted perfectly, because the subsequence 0000 can be followed by either a 0 or a 1.

The PMMX, PPro, P2 and P3 all use a two-level adaptive predictor with $n = 4$. This requires 36 bits of storage for each branch: two bits for each of the 16 counters in the pattern history table, and 4 bits for the branch history register.

The powerful capability of pattern recognition has a minor drawback in the case of completely random sequences with no regularities. The following table lists the experimental fraction of mispredictions for a completely random sequence of taken and not taken:

fraction of taken/not taken	fraction of mispredictions
0.001/0.999	0.001001
0.01/0.99	0.0101
0.05/0.95	0.0525
0.10/0.90	0.110
0.15/0.85	0.171
0.20/0.80	0.235
0.25/0.75	0.300
0.30/0.70	0.362

0.35/0.65	0.417
0.40/0.60	0.462
0.45/0.55	0.490
0.50/0.50	0.500

Table 3.1. Probability of mispredicting random branch

The fraction of mispredictions is slightly higher than it would be without pattern recognition because the processor keeps trying to find repeated patterns in a sequence that has no regularities. The values in table 3.1 also apply to a predictor with a global history table as well as an agree predictor, described below.

Two-level adaptive predictor with global history table

Since the storage requirement for the two-level predictor grows exponentially with the number of history bits n, there is a practical limit to how big we can make n. One way of overcoming this limitation is to share the branch history register and the pattern history table among all the branches rather than having one set for each branch.

Imagine a two-level predictor with a global branch history register, storing the history of the last n branches, and a shared pattern history table. The prediction of a branch is made on the basis of the last n branch events. Some or all of these events may be occurrences of the same branch. If the innermost loop in a program contains m conditional jumps, then the prediction of a branch within this loop can rely on $\text{floor}(n/m)$ occurrences of the same branch in the branch history register, while the rest of the entries come from other branches. If this is enough for defining the pattern of this branch, or if it is highly correlated with the other branches, then we can expect the prediction rate to be good. Many modern processors use variants of this method with values of n from 8 to 16.

The agree predictor

The disadvantage of using global tables is that branches that behave differently may share the same entry in the global pattern history table. This problem can be reduced by storing a biasing bit for each branch. The biasing bit indicates whether the branch is mostly taken or not taken. The predictors in the pattern history table now no longer indicate whether the branch is predicted to be taken or not, but whether it is predicted to go the same way or the opposite way of the biasing bit. Since the prediction is more likely to agree than to disagree with the biasing bit, the probability of negative interference between branches that happen to use the same entry in the pattern history table is reduced, but not eliminated. My research indicates that the P4 is using one version of this method, as shown in figure 3.3.

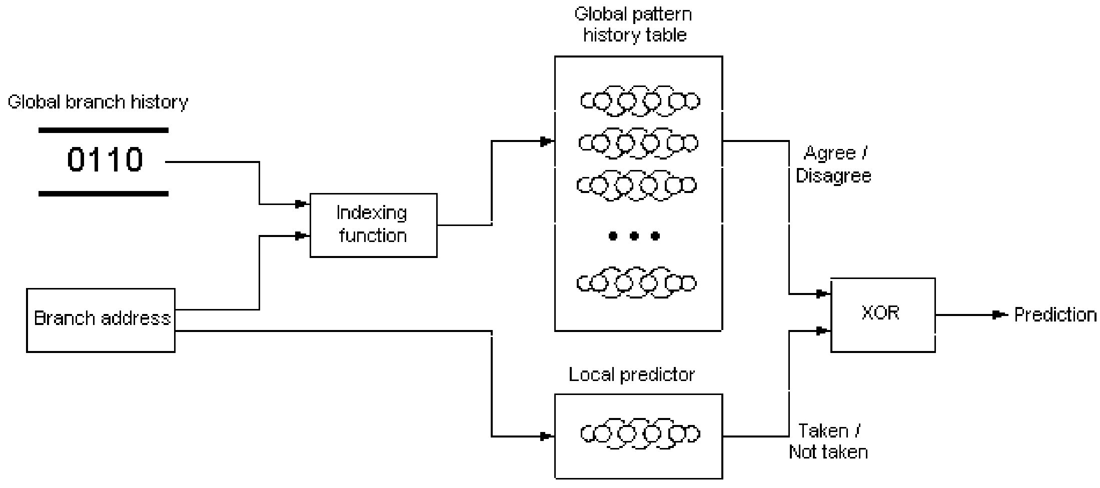

Figure 3.3. Agree predictor

Each branch has a local predictor, which is simply a saturating counter of the same type as shown in figure 3.1. The global pattern history table, which is indexed by the global branch history register, indicates whether the branch is predicted to agree or disagree with the output of the local predictor.

The global branch history register has 16 bits in the P4. Since, obviously, some of the 2^{16} different history patterns are more common than others, we have the problem that some entries in the pattern history table will be used by several branches, while many other entries will never be used at all, if the pattern history table is indexed by the branch history alone. In order to make the use of these entries more evenly distributed, and thus reduce the probability that two branches use the same entry, the pattern history table may be indexed by a combination of the global history and the branch address. The literature recommends that the index into the pattern history table is generated by an XOR combination of the history bits and the branch address. However, my experimental results do not confirm such a design. The indexing function in figure 3.3 may be a more complex hashing function of the history and the branch address, or it may involve the branch target address, BTB entry address or trace cache address.

Since the indexing function is not known, it is impossible to predict whether two branches will use the same entry in the pattern history table. For the same reason, I have not been able to measure the size of the pattern history table, but must rely on rumors in the literature.

(Literature: E. Sprangle, et. al.: The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. Proceedings of the 24th International Symposium on Computer Architecture, Denver, June 1997. J. Stokes: The Pentium 4 and the G4e: an Architectural Comparison: Part I. arstechnica.com, Mar. 2001).

Loop counter

The branch that controls a loop will typically go *n*-1 times one way and then one time the other way, where *n* is the period. For example the loop `for (i=0; i<6; i++)` will produce the branch pattern 000001-000001 or 111110-111110, depending on whether the branch instruction is at the top or the bottom of the loop code. A loop with a high period and several branches inside the loop body would require a very long history table and many entries in the pattern history table for making good predictions in a two-level predictor. The best solution to this problem is to use a different prediction method for loops, called a loop counter or switch counter. A counter counts the period *n* the first time the loop is executed. On subsequent executions, the repetition count is compared with *n* and the loop is predicted to exit when the count equals the period. The information that must be stored in the BTB for a loop counter includes: whether the branch has loop behavior or not, whether the branch is taken or not taken at loop exit, the period, the current repetition count, and the branch target.

The PM and Core2 have a loop counter with 6 bits, allowing loops with a maximum period of 64 to be predicted perfectly.

(Literature: US Patent 5909573).

Indirect jump prediction

An indirect jump or call is a control transfer instruction that has more than two possible targets. A C++ program can generate an indirect jump or indirect call with a `switch` statement, a function pointer, or a `virtual` function. An indirect jump or call is generated in assembly by specifying a register or a memory variable or an indexed array as the destination of a jump or call instruction. Older processors make only one BTB entry for an indirect jump or call. This means that it will always be predicted to go to the same target as it did last time.

As object oriented programming with polymorphous classes has become more common, there is a growing need for predicting indirect calls with multiple targets. This can be done by assigning a new BTB entry for every new jump target that is encountered. The history buffer and pattern history table must have space for more than one bit of information for each jump incident in order to distinguish more than two possible targets.

The PM is the first x86 processor to implement this method. The prediction rule on p. 14 still applies with the modification that the theoretical maximum period that can be predicted perfectly is m^n, where m is the number of different targets per indirect jump, because there are m^n different possible n-length subsequences. However, this theoretical maximum cannot be reached if it exceeds the size of the BTB or the pattern history table.

(Literature: Karel Driesen and Urs Hölzle: Accurate Indirect Branch Prediction. Technical Report TRCS97-19, 1998. Department of Computer Science, University of California).

Subroutine return prediction

A subroutine return is an indirect jump that goes to wherever the subroutine was called from. The prediction of subroutine returns is described on page 35.

Hybrid predictors

A hybrid predictor is an implementation of more than one branch prediction mechanism. A meta-predictor predicts which of the prediction mechanisms is likely to give the best predictions. The meta-predictor can be as simple as a two-bit saturating counter which remembers which of two prediction schemes has worked best in the past for a particular branch instruction.

The PM uses a hybrid predictor consisting of a two level adaptive predictor with a global history table of length eight, combined with a loop counter.

Future branch prediction methods

It is likely that branch prediction methods will be further improved in the future as pipelines get longer. According to the technical literature and the patent literature, the following developments are likely:

- Hybrid predictor. A hybrid predictor with a loop counter and a two-level predictor will probably be implemented in more processors in the future.

- Alloyed predictor. The two-level predictor can be improved by using a combination of local and global history bits as index into the pattern history table. This eliminates the need for the agree predictor and improves the prediction of branches that are not correlated with any preceding branch.

- Keeping unimportant branches out of global history register. In typical programs, a large proportion of the branches always go the same way. Such branches may be kept out of the global history register in order to increase its information contents.

- Decoding both branches. Part or all of the pipeline may be duplicated so that both branches can be decoded and speculatively executed simultaneously. It may decode both branches whenever possible, or only if the prediction is uncertain.

- Neural networks. The storage requirement for the two-level predictor grows exponentially with n, and the warm-up time may also grow exponentially with n. This limits the performance that can be achieved with the two-level predictor. Other methods with less storage requirements are likely to be implemented. Such new methods may use the principles of neural networks.

- Reducing the effect of context switches. The information that the predictors have collected is often lost due to task switches and other context switches. As more advanced prediction methods require longer warm-up time, it is likely that new methods will be implemented to reduce the loss during context switches.

3.2 Branch prediction in P1

The branch prediction mechanism for the P1 is very different from the other processors. Information found in Intel documents and elsewhere on this subject is misleading, and following the advices given in such documents is likely to lead to sub-optimal code.

The P1 has a branch target buffer (BTB), which can hold information for up to 256 jump instructions. The BTB is organized as a 4-way set-associative cache with 64 entries per way. This means that the BTB can hold no more than 4 entries with the same set value. Unlike the data cache, the BTB uses a pseudo random replacement algorithm, which means that a new entry will not necessarily displace the least recently used entry of the same set-value.

Each entry contains a saturation counter of the type shown in figure 3.1. Apparently, the designers couldn't afford to use an extra bit for indicating whether the BTB entry is used or not. Instead, they have equated state "strongly not taken" with "entry unused". This makes sense because a branch with no BTB entry is predicted to be not taken anyway, in the P1. A branch doesn't get a BTB entry until the first time it is taken. Unfortunately, the designers have decided that a branch that is taken the first time it is seen should go to state "strongly taken". This makes the state diagram for the predictor look like this:

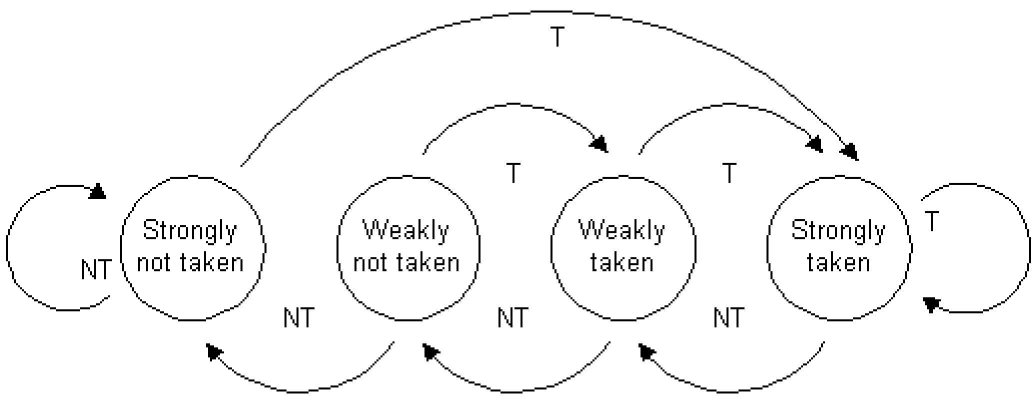

Figure 3.4. Branch predictor in P1.

This is of course a sub-optimal design, and I have strong indications that it is a design flaw. In a tight loop with no more than four instruction pairs, where the loop control branch is seen again before the BTB has had the time to update, the output of the saturation counter is forwarded directly to the prefetcher. In this case the state can go from "strongly not taken" to "weakly not taken". This indicates that the originally intended behavior is as in figure 3.1. Intel engineers have been unaware of this flaw until I published my findings in an earlier version of this manual.

The consequence of this flaw is that a branch instruction which falls through most of the time will have up to three times as many mispredictions as a branch instruction which is taken most of the time. You may take this asymmetry into account by organizing branches so that they are taken more often than not.

BTB is looking ahead (P1)

The BTB mechanism in the P1 is counting instruction pairs, rather than single instructions, so you have to know how instructions are pairing (see page 38) in order to analyze where a

BTB entry is stored. The BTB entry for any control transfer instruction is attached to the address of the U-pipe instruction in the preceding instruction pair. (An unpaired instruction counts as one pair). Example:

```
; Example 3.1. Pentium 1 BTB mechanism
shr eax,1
mov ebx,[esi]
cmp eax,ebx
jb  L
```

Here `SHR` pairs with `MOV`, and `CMP` pairs with `JB`. The BTB entry for `JB L` is thus attached to the address of the `SHR EAX,1` instruction. When this BTB entry is met, and if it predicts the branch to be taken, then the P1 will read the target address from the BTB entry, and load the instructions following `L` into the pipeline. This happens before the branch instruction has been decoded, so the Pentium relies solely on the information in the BTB when doing this.

Instructions are seldom pairing the first time they are executed (see page 38). If the instructions above are not pairing, then the BTB entry should be attached to the address of the `CMP` instruction, and this entry would be wrong on the next execution, when instructions are pairing. However, in most cases the P1 is smart enough to not make a BTB entry when there is an unused pairing opportunity, so you don't get a BTB entry until the second execution, and hence you won't get a prediction until the third execution. (In the rare case, where every second instruction is a single-byte instruction, you may get a BTB entry on the first execution which becomes invalid in the second execution, but since the instruction it is attached to will then go to the V-pipe, it is ignored and gives no penalty. A BTB entry is only read if it is attached to the address of a U-pipe instruction).

A BTB entry is identified by its set-value which is equal to bits 0-5 of the address it is attached to. Bits 6-31 are then stored in the BTB as a tag. Addresses which are spaced a multiple of 64 bytes apart will have the same set-value. You can have no more than four BTB entries with the same set-value.

Consecutive branches

When a jump is mispredicted, then the pipeline gets flushed. If the next instruction pair executed also contains a control transfer instruction, then the P1 will not load its target because it cannot load a new target while the pipeline is being flushed. The result is that the second jump instruction is predicted to fall through regardless of the state of its BTB entry. Therefore, if the second jump is also taken, then you will get another penalty. The state of the BTB entry for the second jump instruction does get correctly updated, though. If you have a long chain of control transfer instructions, and the first jump in the chain is mispredicted, then the pipeline will get flushed all the time, and you will get nothing but mispredictions until you meet an instruction pair which does not jump. The most extreme case of this is a loop which jumps to itself: It will get a misprediction penalty for each iteration.

This is not the only problem with consecutive control transfer instructions. Another problem is that you can have another branch instruction between a BTB entry and the control transfer instruction it belongs to. If the first branch instruction jumps to somewhere else, then strange things may happen. Consider this example:

```
        ; Example 3.2. P1 consecutive branches
        shr eax,1
        mov ebx,[esi]
        cmp eax,ebx
        jb  L1
        jmp L2
L1:     mov eax,ebx
        inc ebx
```

When JB L1 falls through, then we will get a BTB entry for JMP L2 attached to the address of CMP EAX,EBX. But what will happen when JB L1 later is taken? At the time when the BTB entry for JMP L2 is read, the processor doesn't know that the next instruction pair does not contain a jump instruction, so it will actually predict the instruction pair MOV EAX,EBX / INC EBX to jump to L2. The penalty for predicting non-jump instructions to jump is 3 clock cycles. The BTB entry for JMP L2 will get its state decremented, because it is applied to something that doesn't jump. If we keep going to L1, then the BTB entry for JMP L2 will be decremented to state 1 and 0, so that the problem will disappear until next time JMP L2 is executed.

The penalty for predicting the non-jumping instructions to jump only occurs when the jump to L1 is predicted. In the case that JB L1 is mispredictedly jumping, then the pipeline gets flushed and we won't get the false L2 target loaded, so in this case we will not see the penalty of predicting the non-jumping instructions to jump, but we do get the BTB entry for JMP L2 decremented.

Suppose, now, that we replace the INC EBX instruction above with another jump instruction. This third jump instruction will then use the same BTB entry as JMP L2 with the possible penalty of predicting a wrong target.

To summarize, consecutive jumps can lead to the following problems in the P1:

- Failure to load a jump target when the pipeline is being flushed by a preceding mispredicted jump.
- A BTB entry being misapplied to non-jumping instructions and predicting them to jump.
- A second consequence of the above is that a misapplied BTB entry will get its state decremented, possibly leading to a later misprediction of the jump it belongs to. Even unconditional jumps can be predicted to fall through for this reason.
- Two jump instructions may share the same BTB entry, leading to the prediction of a wrong target.

All this mess may give you a lot of penalties, so you should definitely avoid having an instruction pair containing a jump immediately after another poorly predictable control transfer instruction or its target in the P1. It is time for another illustrative example:

```
        ; Example 3.3a. P1 consecutive branches
        call P
        test eax,eax
        jz   L2
L1:     mov  [edi],ebx
        add  edi,4
        dec  eax
        jnz  L1
L2:     call P
```

First, we may note that the function P is called alternately from two different locations. This means that the target for the return from P will be changing all the time. Consequently, the return from P will always be mispredicted.

Assume, now, that EAX is zero. The jump to L2 will not have its target loaded because the mispredicted return caused a pipeline flush. Next, the second CALL P will also fail to have its target loaded because JZ L2 caused a pipeline flush. Here we have the situation where a chain of consecutive jumps makes the pipeline flush repeatedly because the first jump was mispredicted. The BTB entry for `JZ L2` is stored at the address of `P`'s return instruction. This BTB entry will now be misapplied to whatever comes after the second `CALL P`, but that doesn't give a penalty because the pipeline is flushed by the mispredicted second return.

Now, let's see what happens if `EAX` has a nonzero value the next time: `JZ L2` is always predicted to fall through because of the flush. The second `CALL P` has a BTB entry at the address of `TEST EAX,EAX`. This entry will be misapplied to the `MOV`/`ADD` pair, predicting it to jump to `P`. This causes a flush which prevents `JNZ L1` from loading its target. If we have been here before, then the second `CALL P` will have another BTB entry at the address of `DEC EAX`. On the second and third iteration of the loop, this entry will also be misapplied to the `MOV`/`ADD` pair, until it has had its state decremented to 1 or 0. This will not cause a penalty on the second iteration because the flush from `JNZ L1` prevents it from loading its false target, but on the third iteration it will. The subsequent iterations of the loop have no penalties, but when it exits, `JNZ L1` is mispredicted. The flush would now prevent `CALL P` from loading its target, were it not for the fact that the BTB entry for `CALL P` has already been destroyed by being misapplied several times. We can improve this code by putting in some `NOP`'s to separate all consecutive jumps:

```
          ; Example 3.3b. P1 consecutive branches
          call P
          test eax,eax
          nop
          jz   L2
L1:       mov  [edi],ebx
          add  edi,4
          dec  eax
          jnz  L1
L2:       nop
          nop
          call P
```

The extra `NOP`'s cost 2 clock cycles, but they save much more. Furthermore, `JZ L2` is now moved to the U-pipe which reduces its penalty from 4 to 3 when mispredicted. The only problem that remains is that the returns from `P` are always mispredicted. This problem can only be solved by replacing the call to `P` by an inline macro.

3.3 Branch prediction in PMMX, PPro, P2, and P3

<u>BTB organization</u>

The branch target buffer (BTB) of the PMMX has 256 entries organized as 16 ways * 16 sets. Each entry is identified by bits 2-31 of the address of the last byte of the control transfer instruction it belongs to. Bits 2-5 define the set, and bits 6-31 are stored in the BTB as a tag. Control transfer instructions which are spaced 64 bytes apart have the same set-value and may therefore occasionally push each other out of the BTB. Since there are 16 ways per set, this won't happen too often.

The branch target buffer (BTB) of the PPro, P2 and P3 has 512 entries organized as 16 ways * 32 sets. Each entry is identified by bits 4-31 of the address of the last byte of the control transfer instruction it belongs to. Bits 4-8 define the set, and all bits are stored in the BTB as a tag. Control transfer instructions which are spaced 512 bytes apart have the same set-value and may therefore occasionally push each other out of the BTB. Since there are 16 ways per set, this won't happen too often.

The PPro, P2 and P3 allocate a BTB entry to any control transfer instruction the first time it is executed. The PMMX allocates it the first time it jumps. A branch instruction that never jumps will stay out of the BTB on the PMMX. As soon as it has jumped once, it will stay in the BTB, even if it never jumps again. An entry may be pushed out of the BTB when another control transfer instruction with the same set-value needs a BTB entry.

Misprediction penalty

In the PMMX, the penalty for misprediction of a conditional jump is 4 clocks in the U-pipe, and 5 clocks if it is executed in the V-pipe. For all other control transfer instructions it is 4 clocks.

In the PPro, P2 and P3, the misprediction penalty is higher due to the long pipeline. A misprediction usually costs between 10 and 20 clock cycles.

Pattern recognition for conditional jumps

The PMMX, PPro, P2 and P3 all use a two-level adaptive branch predictor with a local 4-bit history, as explained on page 13. Simple repetitive patterns are predicted well by this mechanism. For example, a branch which is alternately taken twice and not taken twice, will be predicted all the time after a short learning period. The rule on page 14 tells which repetitive branch patterns can be predicted perfectly. All patterns with a period of five or less are predicted perfectly. This means that a loop which always repeats five times will have no mispredictions, but a loop that repeats six or more times will not be predicted.

The branch prediction mechanism is also good at handling 'almost regular' patterns, or deviations from the regular pattern. Not only does it learn what the regular pattern looks like. It also learns what deviations from the regular pattern look like. If deviations are always of the same type, then it will remember what comes after the irregular event, and the deviation will cost only one misprediction. Likewise, a branch which switches back and forth between two different regular patterns is predicted well.

Tight loops (PMMX)

Branch prediction in the PMMX is not reliable in tiny loops where the pattern recognition mechanism doesn't have time to update its data before the next branch is met. This means that simple patterns, which would normally be predicted perfectly, are not recognized. Incidentally, some patterns which normally would not be recognized, are predicted perfectly in tight loops. For example, a loop which always repeats 6 times would have the branch pattern 111110 for the branch instruction at the bottom of the loop. This pattern would normally have one or two mispredictions per iteration, but in a tight loop it has none. The same applies to a loop which repeats 7 times. Most other repeat counts are predicted poorer in tight loops than normally.

To find out whether a loop will behave as 'tight' on the PMMX you may follow the following rule of thumb: Count the number of instructions in the loop. If the number is 6 or less, then the loop will behave as tight. If you have more than 7 instructions, then you can be reasonably sure that the pattern recognition functions normally. Strangely enough, it doesn't matter how many clock cycles each instruction takes, whether it has stalls, or whether it is paired or not. Complex integer instructions do not count. A loop can have lots of complex integer instructions and still behave as a tight loop. A complex integer instruction is a non-pairable integer instruction that always takes more than one clock cycle. Complex floating point instructions and MMX instructions still count as one. Note, that this rule of thumb is heuristic and not completely reliable.

Tight loops on PPro, P2 and P3 are predicted normally, and take minimum two clock cycles per iteration.

Indirect jumps and calls (PMMX, PPro, P2 and P3)

There is no pattern recognition for indirect jumps and calls, and the BTB can remember no more than one target for an indirect jump. It is simply predicted to go to the same target as it did last time.

JECXZ and LOOP (PMMX)

There is no pattern recognition for these two instructions in the PMMX. They are simply predicted to go the same way as last time they were executed. These two instructions should be avoided in time-critical code for PMMX. In PPro, P2 and P3 they are predicted using pattern recognition, but the `LOOP` instruction is still inferior to `DEC ECX` / `JNZ`.

3.4 Branch prediction in P4 and P4E

The organization of the branch target buffer (BTB) in the P4 and P4E is not known in detail. It has 4096 entries, probably organized as 8 ways * 512 sets. It is indexed by addresses in the trace cache which do not necessarily have a simple correspondence to addresses in the original code. Consequently, it is difficult for the programmer to predict or avoid BTB contentions. Far jumps, calls and returns are not predicted in the P4 and P4E.

The processor allocates a BTB entry to any near control transfer instruction the first time it jumps. A branch instruction which never jumps will stay out of the BTB, but not out of the branch history register. As soon as it has jumped once, it will stay in the BTB, even if it never jumps again. An entry may be pushed out of the BTB when another control transfer instruction with the same set-value needs a BTB entry. All conditional jumps, including `JECXZ` and `LOOP`, contribute to the branch history register. Unconditional and indirect jumps, calls and returns do not contribute to the branch history.

Branch mispredictions are much more expensive on the P4 and P4E than on previous generations of microprocessors. The time it takes to recover from a misprediction is rarely less than 24 clock cycles, and typically around 45 µops. Apparently, the microprocessor cannot cancel a bogus µop before it has reached the retirement stage. This means that if you have a lot of µops with long latency or poor throughput, then the penalty for a misprediction may be as high as 100 clock cycles or more. It is therefore very important to organize code so that the number of mispredictions is minimized.

Pattern recognition for conditional jumps in P4

The P4 uses an "agree" predictor with a 16-bit global history, as explained on page 15. The branch history table has 4096 entries, according to an article in Ars Technica (J. Stokes: The Pentium 4 and the G4e: an Architectural Comparison: Part I. arstechnica.com, Mar. 2001). The prediction rule on page 14 tells us that the P4 can predict any repetitive pattern with a period of 17 or less, as well as some patterns with higher history. However, this applies to the global history, not the local history. You therefore have to look at the preceding branches in order to determine whether a branch is likely to be well predicted. I will explain this with the following example:

```
    ; Example 3.4. P4 loops and branches
    mov  eax, 100
A:  ...
    ...
    mov  ebx, 16
B:  ...
    sub  ebx, 1
    jnz  B
    test eax, 1
    jnz  X1
    call EAX_IS_EVEN
    jmp  X2
X1: call EAX_IS_ODD
X2: ...
    mov  ecx, 0
C1: cmp  ecx, 10
    jnb  C2
    ...
    add  ecx, 1
```

```
        jmp   C1
    C2: ...
        sub   eax, 1
        jnz   A
```

The `A` loop repeats 100 times. The `JNZ A` instruction is taken 99 times and falls through 1 time. It will be mispredicted when it falls through. The `B` and `C` loops are inside the `A` loop. The `B` loop repeats 16 times, so without considering the prehistory, we would expect it to be predictable. But we have to consider the prehistory. With the exception of the first time, the prehistory for `JNZ B` will look like this: `JNB C2`: not taken 10 times, taken 1 time (`JMP C1` does not count because it is unconditional); `JNZ A` taken; `JNZ B` taken 15 times, not taken 1 time. This totals 17 consecutive taken branches in the global history before `JNZ B` is not taken. It will therefore be mispredicted once or twice for each cycle. There is a way to avoid this misprediction. If you insert a dummy branch that always falls through anywhere between the `A:` and `B:` labels, then `JNZ B` is likely to be predicted perfectly, because the prehistory now has a not taken before the 15 times taken. The time saved by predicting `JNZ B` well is far more than the cost of an extra dummy branch. The dummy branch may, for example, be `TEST ESP,ESP` / `JC B`.

`JNZ X1` is taken every second time and is not correlated with any of the preceding 16 conditional jump events, so it will not be predicted well.

Assuming that the called procedures do not contain any conditional jumps, the prehistory for `JNB C2` is the following: `JNZ B` taken 15 times, not taken 1 time; `JNZ X1` taken or not taken; `JNB C2`: not taken 10 times, taken 1 time. The prehistory of `JNB C2` is thus always unique. In fact, it has 22 different and unique prehistories, and it will be predicted well. If there was another conditional jump inside the `C` loop, for example if the `JMP C1` instruction was conditional, then the `JNB C2` loop would not be predicted well, because there would be 20 instances between each time `JNB C2` is taken.

In general, a loop cannot be predicted well on the P4 if the repeat count multiplied by the number of conditional jumps inside the loop exceeds 17.

Alternating branches

While the `C` loop in the above example is predictable, and the `B` loop can be made predictable by inserting a dummy branch, we still have a big problem with the `JNZ X1` branch. This branch is alternately taken and not taken, and it is not correlated with any of the preceding 16 branch events. Let's study the behavior of the predictors in this case. If the local predictor starts in state "weakly not taken", then it will alternate between "weakly not taken" and "strongly not taken" (see figure 3.1). If the entry in the global pattern history table starts in an agree state, then the branch will be predicted to fall through every time, and we will have 50% mispredictions (see figure 3.3). If the global predictor happens to start in state "strongly disagree", then it will be predicted to be taken every time, and we still have 50% mispredictions. The worst case is if the global predictor starts in state "weakly disagree". It will then alternate between "weakly agree" and "weakly disagree", and we will have 100% mispredictions. There is no way to control the starting state of the global predictor, but we can control the starting state of the local predictor. The local predictor starts in state "weakly not taken" or "weakly taken", according to the rules of static prediction, explained on page 36 below. If we swap the two branches and replace `JNZ` with `JZ`, so that the branch is taken the first time, then the local predictor will alternate between state "weakly not taken" and "weakly taken". The global predictor will soon go to state "strongly disagree", and the branch will be predicted correctly all the time. A backward branch that alternates would have to be organized so that it is not taken the first time, to obtain the same effect. Instead of swapping the two branches, we may insert a `3EH` prediction hint prefix immediately before the `JNZ X1` to change the static prediction to "taken" (see p. 36). This will have the same effect.

While this method of controlling the initial state of the local predictor solves the problem in most cases, it is not completely reliable. It may not work if the first time the branch is seen is after a mispredicted preceding branch. Furthermore, the sequence may be broken by a task switch or other event that pushes the branch out of the BTB. We have no way of predicting whether the branch will be taken or not taken the first time it is seen after such an event. Fortunately, it appears that the designers have been aware of this problem and implemented a way to solve it. While researching these mechanisms, I discovered an undocumented prefix, `64H`, which does the trick on the P4. This prefix doesn't change the static prediction, but it controls the state of the local predictor after the first event so that it will toggle between state "weakly not taken" and "weakly taken", regardless of whether the branch is taken or not taken the first time. This trick can be summarized in the following rule:

A branch which is taken exactly every second time, and which doesn't correlate with any of the preceding 16 branch events, can be predicted well on the P4 if it is preceded by a `64H` prefix. This prefix is coded in the following way:

```
; Example 3.5. P4 alternating branch hint
DB   64H          ; Hint prefix for alternating branch
jnz  X1           ; Branch instruction
```

No prefix is needed if the branch can see a previous instance of itself in the 16-bit prehistory.

The `64H` prefix has no effect and causes no harm on any previous microprocessor. It is an `FS` segment prefix. The `64H` prefix cannot be used together with the `2EH` and `3EH` static prediction prefixes.

Pattern recognition for conditional jumps in P4E

Branch prediction in the P4E is simpler than in the P4. There is no agree predictor, but only a 16-bit global history and a global pattern history table. This means that a loop can be predicted well on the P4E if the repeat count multiplied by the number of conditional jumps inside the loop does not exceed 17.

Apparently, the designers have decided that the improvement in prediction rate of the agree predictor is too small to justify the considerable complexity. However, it appears that the P4E has inherited a little peculiarity from the agree predictor of the P4. The `64H` prefix influences the first few predictions of a branch in a way that might have been optimal for an alternating branch if there was an agree predictor. This has no useful purpose, but causes no serious harm either.

3.5 Branch prediction in PM and Core2

The branch prediction mechanism is the same in PM and Core2.

Misprediction penalty

The misprediction penalty is approximately 13 clock cycles in the PM and 15 clock cycles in the Core2, corresponding to the length of the pipeline. Far jumps, far calls and far returns are not predicted.

Pattern recognition for conditional jumps

The branch prediction mechanism is more advanced than on previous processors. Conditional jumps are handled by a hybrid predictor combining a two-level predictor and a loop counter. In addition, there is a mechanism for predicting indirect jumps and indirect calls.

A branch instruction is recognized as having loop behavior if it goes one way n-1 times and then goes the other way one time. A loop counter makes it possible to predict branches with loop behavior perfectly if the period *n* is no longer than 64. The loop counters are stored for each branch without using the global history table. Instead, the loop counters have their own buffer with 128 entries. Therefore, the prediction of a loop does not depend on the number of other branches inside the loop. Nested loops are predicted perfectly.

Branches that do not have loop behavior are predicted using a two-level predictor with an 8-bit global history buffer and a history pattern table of unknown size. The ability to predict a repetitive branch pattern, other than a simple loop pattern, depends on the number of branches in a loop according to the rules for a predictor with a global history table, explained on p. 15.

A meta-predictor determines whether a branch has loop behavior or not and chooses the prediction mechanism accordingly. The mechanism of the meta predictor is not known.

Pattern recognition for indirect jumps and calls

Indirect jumps and indirect calls (but not returns) are predicted using the same two-level predictor principle as branch instructions. Branches without loop behavior and indirect jumps/calls share the same history buffer and history pattern table, but apparently not the same BTB. An indirect jump/call gets a new BTB entry every time it jumps to a new target. It can have more then four BTB entries even though the BTB has only four ways. The history buffer stores more than one bit for each entry, probably 6 or 7, in order to distinguish between more than two targets of an indirect jump/call. This makes it possible to predict a periodic jump pattern that switches between several different targets. The maximum number of different targets that I have seen predicted perfectly is 36, but such impressive predictions are rare for reasons explained below. A periodic pattern can be predicted if all 8-length history sub-patterns are different. This also improves the prediction of subsequent conditional jumps because they share the same history buffer. A conditional jump can make a prediction based on the distinction between different jump targets of a preceding indirect jump.

The above observations indicate that the history buffer must have at least 8*6 = 48 bits, but a history pattern table of 2^{48} entries is physically impossible. The 48 or more bits must be compressed by some hashing algorithm into a key of x bits to index the 2^x entries of the history pattern table. The value of x is not known, but it is probably between 10 and 16. The hashing algorithm may be a simple XOR combination of bits from the history buffer and the address or the BTB index of the branch instruction. A more complex hashing function is also possible.

My experiments show that the PM and Core2 make more mispredictions than expected in programs where there are more than a few branches with non-loop behavior or indirect jumps/calls in the innermost loop. Test results were not always the same for identical experiments. There is evidently a dependence on the state of the BTB and the history pattern table prior to the experiment.

The design gives three probable causes for poor predictions: The first cause is contention for entries in the BTB. The second cause is aliasing of keys generated by the hashing algorithm causing contention between history pattern table entries. It appears that this aliasing phenomenon occurs not only for indirect jumps/calls but also for conditional jumps predicted by the two-level predictor. The third possible cause is poor performance of the meta predictor. My guess is that an insufficient size of the history pattern table is the main reason for lower-than-expected prediction rates.

BTB organization

There appears to be different branch target buffers for different types of branches in the PM and Core2. My measurements on a Core2 indicate the following BTB organizations:

For unconditional jumps and branches without loop behavior:

4 ways by 512 sets = 2048 entries. Each entry is identified by bits 0 - 21 of the address of the last byte of the control transfer instruction it belongs to. Bits 4 - 12 define the set, and the remaining bits are stored in the BTB as a tag. Entries that differ only by bits 22 - 31 will clash into the same BTB entry. A BTB entry is allocated the first time a control transfer instruction is met.

For branches with loop behavior:
2 ways by 64 sets = 128 entries. Each entry is identified by bits 0 - 13 of the address of the last byte of the control transfer instruction it belongs to. Bits 4 - 9 define the set. Entries that differ only by bits 14 - 31 will clash into the same BTB entry.

For indirect jumps and indirect calls:
4 ways by 2048 sets = 8192 entries. Each entry is identified by bits 0 - 21 of the address of the last byte of the control transfer instruction it belongs to. Bits 0 - 10 define the set, and the remaining bits are stored in the BTB as a tag. Entries that differ only by bits 22 - 31 will clash into the same BTB entry.

For near returns:
The return stack buffer has 16 entries.

It should be noted that these figures are somewhat uncertain because my measurements on branches without loop behavior are inconsistent. The inconsistency is probably due to the fact that the hashing function for the history pattern table is not known.

(Literature: S. Gochman, et al.: The Intel Pentium M Processor: Microarchitecture and Performance. Intel Technology Journal, vol. 7, no. 2, 2003).

3.6 Branch prediction in Intel Nehalem

Little has been published about the branch prediction mechanism in the Nehalem, and the test results have not been fully interpreted.

Misprediction penalty

The misprediction penalty is longer than on Core2 due to a longer pipeline. The measured misprediction penalty is at least 17 clock cycles.

Pattern recognition for conditional jumps

The branch prediction mechanism has been improved to compensate for the longer misprediction delay. Conditional jumps are handled by a hybrid predictor combining one or more two-level predictor mechanisms and a loop counter. In addition, there is a mechanism for predicting indirect jumps and indirect calls.

A branch with loop behavior is predicted for loop counts up to 64. The loop counters are stored for each branch without using the global history table. Instead, the loop counters have their own buffer with 32 entries. The prediction of a loop does not depend on the number of other branches inside the loop. Nested loops are predicted perfectly.

Branches that do not have loop behavior are predicted using a two-level predictor with an 18-bit global history buffer and a history pattern table of unknown size. The ability to predict a repetitive branch pattern, other than a simple loop pattern, depends on the number of jumps in a loop according to the rules for a predictor with a global history table, explained on p. 15. Apparently, there are two 18-bit global history buffers. The first history buffer includes all jumps and branches including unconditional jumps but not including never-taken branches. The second history buffer includes only some branches, presumably the most important ones. This improves the prediction of some branches inside loops that contain up to nine jumps. I have not been able to fully interpret the test results. The criteria for which history buffer to use have not been found. The algorithm is asymmetric, so that the maximum number of consecutive not-taken events is higher than the maximum number of consecutive taken events. Both numbers depend on the number of jumps inside the loop.

Pattern recognition for indirect jumps and calls

Indirect jumps and indirect calls (but not returns) are predicted using the same two-level predictor as branch instructions. Branches without loop behavior and indirect jumps/calls share the same history buffer, but perhaps not the same BTB.

BTB organization

The branch target buffer has two levels, like a two-level cache system.

There may be different branch target buffers for different types of branches. I have not been able to measure the sizes of the branch target buffers.

Prediction of function returns

The return stack buffer has 16 entries for near returns.

Literature: "A Method and apparatus for branch prediction using a second level branch prediction table". WO Patent 2000/014628.

3.7 Branch prediction in Intel Sandy Bridge and Ivy Bridge

The Sandy Bridge reverses the trend of ever more complicated branch prediction algorithms by not having a separate predictor for loops. The redesign of the branch prediction mechanism has probably been necessary in order to handle the new µop cache (see page 121 below). A further reason for the simplification may be a desire to reduce the pipeline length and thereby the misprediction penalty. The Ivy Bridge appears to be very similar to the Sandy Bridge.

Misprediction penalty

The misprediction penalty is often shorter than on the Nehalem thanks to the µop cache (see page 121 below). The misprediction penalty was measured to 15 clock cycles or more for branches inside the µop cache and slightly more for branches in the level-1 code cache.

Pattern recognition for conditional jumps

There appears to be a two-level predictor with a 32-bit global history buffer and a history pattern table of unknown size. There is no specialized loop predictor. Nested loops and loops with branches inside are not predicted particularly well, though sometimes better than what follows from the rules on page 14.

Pattern recognition for indirect jumps and calls

Indirect jumps and indirect calls (but not returns) are predicted using the same two-level predictor as branch instructions.

BTB organization

The branch target buffer in Sandy Bridge is bigger than in Nehalem according to unofficial rumors. It is unknown whether it has one level, as in Core 2 and earlier processors, or two levels as in Nehalem. It can handle a maximum of four call instructions per 16 bytes of code. Conditional jumps are less efficient if there are more than 3 branch instructions per 16 bytes of code.

Prediction of function returns

The return stack buffer has 16 entries for near returns.

3.8 Branch prediction in Intel Haswell, Broadwell and Skylake

The branch predictor appears to have been redesigned in the Haswell, but very little is known about its construction.

The measured throughput for jumps and branches varies between one branch per clock cycle and one branch per two clock cycles for jumps and predicted taken branches. Predicted not taken branches have an even higher throughput of up to two branches per clock cycle.

The high throughput for taken branches of one per clock was observed for up to 128 branches with no more than one branch per 16 bytes of code. If there is more than one branch per 16 bytes of code then the throughput is reduced to one jump per two clock cycles. If there are more than 128 branches in the critical part of the code, and if they are spaced by at least 16 bytes, then apparently the first 128 branches have the high throughput and the remaining have the low throughput.

These observations may indicate that there are two branch prediction methods: a fast method tied to the μop cache and the instruction cache, and a slower method using a branch target buffer.

Misprediction penalty

The branch misprediction penalty varies a lot. It was measured to 15 - 20 clock cycles.

Pattern recognition for conditional jumps

The processor is able to predict very long repetitive jump patterns with few or no mispredictions. I found no specific limit to the length of jump patterns that could be predicted. Loops are successfully predicted up to a count of 32 or a little more. Nested loops and branches inside loops are predicted reasonably well.

Pattern recognition for indirect jumps and calls

Indirect jumps and indirect calls are predicted well.

BTB organization

The organization of the branch target buffer is unknown. It appears to be reasonably big.

Prediction of function returns

The return stack buffer has 16 entries for near returns.

3.9 Branch prediction in Intel Atom and Silvermont

The branch prediction mechanism in the Intel Atom and Silvermont processors is a two-level adaptive predictor with a global history table, following the principles described on page 15. The branch history register has 12 bits. The pattern history table on the Atom has 4096 entries and is shared between threads. The branch target buffer has 128 entries, organized as 4 ways by 32 sets. The size of these buffers on the Silvermont is unknown, but probably bigger, and not shared between threads.

Unconditional jumps make no entry in the global history table, but always-taken and never-taken branches do.

The Silvermont has branch prediction both at the fetch stage and at the later decode stage in the pipeline, where the latter can correct errors in the former, according to an article in Realworldtech.

Contrary to what some documents say, there is no special predictor for loops according to my tests. Loops are predicted in the same way as other branches, following the rules as described on page 15.

Misprediction penalty

The penalty for mispredicting a branch is 11 - 13 clock cycles.

It often occurs that a branch has a correct entry in the pattern history table, but no entry in the branch target buffer, which is much smaller. If a branch is correctly predicted as taken, but no target can be predicted because of a missing BTB entry, then the penalty will be approximately 7 clock cycles on the Atom.

Prediction of indirect branches

Some documents say that the Silvermont has a history predictor for indirect branches, but this is not confirmed in my tests. I have found no pattern prediction for indirect branches. Indirect branches are predicted to go to the same target as last time.

Return stack buffer

There is a return stack buffer with 8 entries on the Atom and 16 entries on the Silvermont.

3.10 Branch prediction in VIA Nano

The VIA Nano processor has a hybrid prediction mechanism with several different branch predictors, a majority vote mechanism, and a meta predictor, according to G. Glenn Henry: "The VIA Isaiah Architecture", Centaur Technology, 2008 (www.via.com.tw).

The branch target buffer has 4096 entries, 4-way set associative, according to the abovementioned document.

The branch target buffer is connected to the code cache with two entries for each 16 bytes of code. If a 16-bytes block of code contains more than two branches then the excessive branches use a simpler, and slower, mechanism. This was confirmed by my tests. Branch prediction is slower if a 16-bytes block of code contains more than two jumps, branches, calls or returns.

In my tests, the mechanism behaved almost like an alloyed two-level adaptive predictor with a history table that combines 12 bits of local history and 2 bits of global history. The local history information is stored in connection with the code cache with a certain amount of storage provided for each 16-bytes block of code.

The goodness of the prediction depends on the number of jumps in the same 16-bytes block of code. The prediction is best if there is no more than one branch or jump in each 16-bytes block of code. If there are two branches or jumps in one 16-bytes block of code then each branch has fewer bits for storing local history, even if one of the jumps is unconditional. The prediction is particularly bad if there are two jumps or branches, one of which is always taken. If there are more than two jumps or branches in the same 16-bytes block of code then the excessive jumps or branches will use a slower and inferior prediction method.

Loop branches behave slightly differently. A loop with no jumps inside can be predicted up to a loop count of 15 - 20. A loop with one or more jumps inside can be predicted up to a count of 12 - 16. The behavior varies and is not always reproducible.

Indirect jumps are predicted to go to the same target as last time.

The return stack buffer appears to be very deep. All returns were predicted correctly even in very deeply nested subroutines in my tests.

The misprediction penalty is typically 16 clock cycles, max. 20.

The branch prediction mechanism fails if a branch is followed by a serializing instruction such as CPUID. This is not a performance problem, but it must be taken into account when testing.

3.11 Branch prediction in AMD K8 and K10

BTB organization

The branch prediction mechanism of the K8 and K10 AMD processors is connected to the code cache. The level-1 code cache has 1024 lines of 4*16 bytes each, 2-way set associative.

Each 16-bytes block in the code cache has an associated set of branch indicators. There are nine branch indicators, associated with byte number 0, 1, 3, 5, 7, 9, 11, 13 and 15 of the code block. Most branch instructions are two or more bytes long. A branch instruction that is two or more bytes long will have at least one branch indicator even when there are only indicators for the odd addresses and address 0. The extra indicator at byte 0 covers the case where a branch instruction crosses a 16-bytes boundary and ends at byte 0.

Each branch indicator has two bits of information to cover the following cases: (0) no branch or never jump, (1) use branch selector 1, (2) use branch selector 2, (3) use branch selector 3. There are three branch selectors in addition to the nine branch indicators. Each branch selector has an index into the branch target buffer as well as a local branch prediction information. The local branch prediction information can be "never jump", "always jump", "use dynamic prediction", or "use return stack buffer". There is also an indication of whether the branch is a call. This is used for pushing a return address on the return address stack (see page 35).

The branch target addresses are saved in the branch target buffer (BTB), which has 2048 entries. Return addresses are stored in the return address stack, which has 12 entries in K8 and 24 entries in K10.

The branch prediction mechanism works as follows: A branch that is never taken gets no branch indicator, no branch selector and no BTB entry. The first time a branch is taken, it gets a branch selector which is set to "always jump" and a BTB entry to indicate the target address. If the branch is later not taken, then the branch selector is changed to "use dynamic prediction". It never goes back from "use dynamic prediction" to "always jump" or "never jump". The dynamic prediction mechanism is described below. The predictor bits can indicate at least the following values: "no jump", "always jump", "use dynamic prediction", "use return stack buffer". The K10 also has predictors for indirect jumps.

Some documents say that return instructions occupy a branch target entry which is used when the return stack buffer is exhausted, but experiments seem to indicate that returns are always mispredicted when the return stack buffer is exhausted.

The branch indicators and part of the local branch prediction information is copied to the level-2 cache when evicted from the level-1 cache. The index into the BTB is not copied to the level-2 cache due to lack of space. There is a branch target address calculator which can calculate the target address for direct jumps and calls in case a BTB entry is missing or the BTB index has been lost due to eviction to the level-2 cache. The calculation of a lost branch target address takes an extra 4 clock cycles, according to my measurements, which is less than the cost of a complete misprediction. This allows conditional and unconditional jumps and calls to be predicted correctly, even if they have been evicted from the BTB and/or the level-1 cache. The branch target address calculator cannot calculate the targets of indirect jumps and returns, of course. Returns are mispredicted if they have been evicted from the level-1 cache even they are still in the return stack buffer.

A drawback of this design is that there can be no more than three control transfer instructions for every aligned 16-bytes block of code, except for branches that are never taken. If there are more than three taken branches in the same 16-bytes block of code then they will keep stealing branch selectors and BTB entries from each other and cause two mispredictions for every execution. It is therefore important to avoid having more than three jumps, branches, calls and returns in a single aligned 16-bytes block of code. Branches that are never taken do not count. The three branch limit can easily be exceeded if for example a `switch`/`case` statement is implemented as a sequence of `dec` / `jz` instructions.

A further problem is that the design allows only one control transfer instruction for every two bytes of code. Most control transfer instructions use more than one byte of code, but return instructions can be coded as a single-byte instruction. This can cause two kinds of problems. This first kind of problem occurs if two branches share the same branch selector. If a branch instruction ending at an even address if followed by a single-byte return instruction at the following odd address, then the two instructions will share the same branch selector and will therefore be mispredicted most of the time.

The second kind of problem relates to single-byte return instructions having no branch selector. This can happen when there is a jump directly to a single-byte return instruction or a single-byte return instruction follows immediately after a mispredicted branch. If the single-byte return instruction is at an even address not divisible by 16 then the branch selector will not be loaded in these situations, and the return will be mispredicted.

The first kind of problem can be avoided by placing the single-byte return instruction at an even address, the second kind of problem by placing it at an odd address (or an address divisible by 16). Both kinds of problem can be avoided by making the return instruction longer than one byte. This can be done by inserting a segment prefix or F3 prefix before the return instruction to make it two bytes long, or by coding the return instruction with an offset operand of zero, which makes it three bytes long. It is recommended to make return instructions longer than one byte if there is a conditional jump immediately before it, or if there is a jump directly to it. A call instruction immediately before a return should be replaced by a jump.

Misprediction penalty

AMD manuals say that the branch misprediction penalty is 10 clock cycles if the code segment base is zero and 12 clocks if the code segment base is nonzero. In my measurements, I have found a minimum branch misprediction penalty of 12 and 13 clock cycles, respectively. The code segment base is zero in most 32-bit operating systems and all 64-bit systems. It is almost always nonzero in 16-bit systems (see page 177).

The misprediction penalty corresponds to the length of the pipeline. Far jumps, calls and returns are not predicted.

Pattern recognition for conditional jumps

The AMD uses a two-level adaptive branch predictor with a global 8 or 12-bit history, as explained on page 15. Simple repetitive patterns and small loops with a repeat count up to 9 or 13 can be predicted by this mechanism. The rule on page 14 tells which repetitive branch patterns can be predicted perfectly.

The AMD optimization guide tells that the global pattern history table has 16k entries. This table is indexed by the 8 or 12 bits of the global history combined with part of the branch address. My tests on K8 indicate that it is indexed by an 8-bit history and bits number 4-9 of the address of the last byte of the branch instruction. This makes 16k entries if none of the bits are combined. It is possible that some of the bits are XOR'ed with each other and/or with part of the branch target address.

Branches that always go the same way do not pollute the global branch history register. This is accomplished by the information stored in the branch selector block. A branch is assumed to be not taken until the first time it is taken. After a branch has been taken for the first time it is assumed always taken until next time it is not taken. After a branch has been taken and then not taken it is tagged as needing dynamic prediction. It is then predicted using the two-level adaptive algorithm with an 8 bit global history.

This mechanism makes it possible to predict loops with repeat counts up to 9 or 13 even if they contain branches, as long as the branches always go the same way.

My tests on K8 indicate that the dynamic branch prediction fails much more often than what can be expected from the design described above and that the pattern learning time can be quite long. I have no explanation for this, but it is possible that the branch pattern history table is smaller than indicated or that the mechanism is more complicated than described here.

Prediction of indirect branches

The K10 has a separate target buffer with 512 entries for indirect jumps and indirect calls. It probably shares the 12-bit history counter with the two-way branches. Indirect jumps with multiple target can thereby be predicted if they follow a regular pattern. Earlier processors predict indirect jumps to always go the same way, as described below.

Return stack buffer

The return stack buffer has 12 entries in K8 and 24 entries in K10. This is more than sufficient for normal applications except for recursive procedures. See page 35 for an explanation of the return stack buffer.

Literature:

The branch prediction mechanism is described in the following documents, though these may not be accurate:

AMD Software Optimization Guide for AMD64 Processors, 2005, 2008.
Hans de Vries: Understanding the detailed Architecture of AMD's 64 bit Core, Chip Architect, Sept. 21, 2003. www.chip-architect.com.
Andreas Kaiser: K7 Branch Prediction, 1999.
www.agner.org/optimize/KaiserK7BranchPrediction.pdf.
Talk at Stanford 2004 stanford-online.stanford.edu/courses/ee380/040107-ee380-100.asx.

3.12 Branch prediction in AMD Bulldozer, Piledriver and Steamroller

The AMD Bulldozer has a new branch prediction design which has no connection to the code cache, unlike previous models. The prediction mechanism is described as a hybrid with a local predictor and a global predictor. Most probably, the branch predictor is based on *perceptrons*. A perceptron is similar to a neuron, and it learns by tracking correlations in the branch history. Unlike the adaptive two-level predictor, the perceptron predictor can learn very long branch patterns.

The branch target buffer (BTB) has two levels, according to AMD's software optimization guide. The level-1 BTB is organized as a set-associative cache with 128 sets of 4 ways = 512 entries. The level-2 BTB has1024 sets of 5 ways = 5120 entries in the Bulldozer and Piledriver, and probably 10240 entries in the Steamroller.

The BTB and predictor is shared between the two cores of a compute unit.

My tests indicate that complex repetitive patterns are predicted well after a certain learning period. There appears to be no sharp limit to the length of branch patterns that can be predicted, and even very long patterns can be predicted. There seems to be no loop counter, and nested loops are not predicted particularly well. Indirect branches are predicted well. The prediction success rate is somewhat higher in the Steamroller than in the previous models.

Misprediction penalty

The misprediction penalty is specified as minimum 20 clock cycles for conditional and indirect branches and 15 clock cycles for unconditional jumps and returns. My measurements indicated up to 19 clock cycles for conditional branches and 22 clock cycles for returns.

Return stack buffer

The return stack buffer has 24 entries.

Literature:

Software Optimization Guide for AMD Family 15h Processors. AMD, January 2012.
Daniel A. Jiménez, and Calvin Lin. "Dynamic branch prediction with perceptrons." The Seventh International Symposium on High-Performance Computer Architecture, 2001., pp. 197-206. IEEE, 2001.

3.13 Branch prediction in AMD Bobcat and Jaguar

BTB organization

The position of jumps and branches is stored in two arrays, a "sparse branch marker array" which can hold 2 branches for each 64-bytes cache line, and a "dense branch marker array" which can hold 2 branches for every 8 bytes of code, according to the article cited below. In my tests, the Bobcat could predict 2 branches per 64 bytes of code in the level-2 cache, which indicates that the sparse branch array, but not the dense branch array, is coupled to both levels of cache. If both arrays are used, we would expect a maximum of 18 branches per 64 bytes of level-1 cache. In my tests, the Bobcat and Jaguar were able to predict 16 or 17 branches per line of level-1 cache, depending on the position of the branches, but not 18.

There are many mispredictions when these number of branches are exceeded, as the branches keep evicting each other from the arrays.

Branches that are never taken are included in the branch marker arrays.

Misprediction penalty

The article cited below indicates a misprediction penalty of 13 clock cycles. In my tests, however, the misprediction penalty ranged from 8 to 19 clock cycles depending on the subsequent instructions. There was no consistent difference between integer and floating point code in this respect.

Pattern recognition for conditional jumps

The branch predictor behaves approximately as a two-level adaptive branch predictor with a global 26-bit history (see page 15), or slightly better. There is no dedicated loop predictor. This means that loops with many branches or other loops inside are poorly predicted. I have no information on the size of the pattern history table, but we can surely assume that it is less than 2^{26}.

Branches that are always taken are included in the history buffer, while unconditional jumps and branches never taken are not.

Prediction of indirect branches

The processor can predict indirect jumps with more than two different targets, but mispredictions are frequent.

Return stack buffer

There is a return stack buffer with 12 entries on Bobcat, and 16 entries on Jaguar, according to my measurements.

Literature:

Brad Burgess, et. al.: Bobcat: AMD's Low-Power x86 Processor. IEEE Micro, March/April 2011, pp. 16-25.

3.14 Indirect jumps on older processors

Indirect jumps, indirect calls, and returns may go to a different address each time. The prediction method for an indirect jump or indirect call is, in processors older than PM and K10, simply to predict that it will go to the same target as last time it was executed. The first time an indirect jump or indirect call is seen, it is predicted to go to the immediately following instruction. Therefore, an indirect jump or call should always be followed by valid code. Don't place a list of jump addresses immediately after an indirect jump or call. Such a list should preferably be placed in the data segment, rather than the code segment.

A multi-way branch (`switch` statement) is implemented either as an indirect jump using a list of jump addresses, or as a tree of branch instructions. The indirect jump has the disadvantage that it is poorly predicted on many processors, but the branch tree method has other disadvantages, namely that it consumes more BTB entries and that many processors have poor performance for dense or consecutive branches.

3.15 Returns (all processors except P1)

A better method is used for returns. A Last-In-First-Out buffer, called the return stack buffer, remembers the return address every time a call instruction is executed, and it uses this for predicting where the corresponding return will go. This mechanism makes sure that return instructions are correctly predicted when the same subroutine is called from several different locations.

The P1 has no return stack buffer, but uses the same method for returns as for indirect jumps. Later processors have a return stack buffer. The size of this buffer is 4 in the PMMX, 8 in Atom, 12 in AMD k8, 16 in PPro, P2, P3, P4, P4E, PM, Core2 and Nehalem, and 24 in AMD k10. This size may seem rather small, but it is sufficient in most cases because only the innermost subroutines matter in terms of execution time. The return stack buffer may be insufficient, though, in the case of a deeply nesting recursive function.

In order to make this mechanism work, you must make sure that all calls are matched with returns. Never jump out of a subroutine without a return and never use a return as an indirect jump. It is OK, however, to replace a `CALL MYPROC` / `RET` sequence with `JMP MYPROC` in 16 and 32 bit mode. In 64 bit mode, obeying the stack alignment standard, you can replace `SUB RSP,8 / CALL MYPROC / ADD RSP,8 / RET` with `JMP MYPROC`.

On most processors, you must make sure that far calls are matched with far returns and near calls with near returns. This may be problematic in 16-bit code because the assembler will replace a far call to a procedure in the same segment with `PUSH CS` followed by a near call. Even if you prevent the assembler from doing this by hard-coding the far call, the linker is likely to translate the far call to `PUSH CS` and a near call. Use the /NOFARCALLTRANSLATION option in the linker to prevent this. It is recommended to use a small or flat memory model so that you don't need far calls, because far calls and returns are expensive anyway.

3.16 Static prediction

The first time a branch instruction is seen, a prediction is made according to the principles of static prediction.

Static prediction in P1 and PMMX

A control transfer instruction which has not been seen before or which is not in the branch target buffer (BTB) is always predicted to fall through on the P1 and PMMX.

A branch instruction will not get a BTB entry if it always falls through. As soon as it is taken once, it will get into the BTB. On the PMMX, it will stay in the BTB no matter how many times it falls through. Any control transfer instruction which jumps to the address immediately following itself will not get a BTB entry and will therefore always have a misprediction penalty.

Static prediction in PPro, P2, P3, P4, P4E

On PPro, P2, P3, P4 and P4E, a control transfer instruction which has not been seen before, or which is not in the BTB, is predicted to fall through if it goes forwards, and to be taken if it goes backwards (e.g. a loop). Static prediction takes longer time than dynamic prediction on these processors.

On the P4 and P4E, you can change the static prediction by adding prediction hint prefixes. The prefix `3EH` will make the branch predicted taken the first time, and prefix `2EH` will make it predicted not taken the first time. These prefixes can be coded in this way:

```
; Example 3.6. P4/P4E static branch prediction hint
DB  3EH     ; Prediction hint prefix
JBE LL      ; Predicted taken first time
```

The prediction hint prefixes are in fact segment prefixes, which have no effect and cause no harm on other processors.

It is rarely worth the effort to take static prediction into account. Almost any branch that is executed sufficiently often for its timing to have any significant effect is likely to stay in the BTB so that only the dynamic prediction counts. Static prediction only has a significant effect if context switches or task switches occur very often.

Normally you don't have to care about the penalty of static mispredictions. It is more important to organize branches so that the most common path is not taken, because this improves code prefetching, trace cache use, and retirement.

Static prediction does have an influence on the way traces are organized in a trace cache, but this is not a lasting effect because traces may be reorganized after several iterations.

Static prediction in PM and Core2

These processors do not use static prediction. The predictor simply makes a random prediction the first time a branch is seen, depending on what happens to be in the BTB entry that is assigned to the new branch. There is simply a 50% chance of making the right prediction of jump or no jump, but the predicted target is correct. Branch hint prefixes have no useful effect on PM and Core2 processors.

Static prediction in AMD

A branch is predicted not taken the first time it is seen. A branch is predicted always taken after the first time it has been taken. Dynamic prediction is used only after a branch has been taken and then not taken. Branch hint prefixes have no effect.

3.17 Close jumps

Close jumps on PMMX

On the PMMX, there is a risk that two control transfer instructions will share the same BTB entry if they are too close to each other. The obvious result is that they will always be mispredicted. The BTB entry for a control transfer instruction is identified by bits 2-31 of the address of the last byte in the instruction. If two control transfer instructions are so close together that they differ only in bits 0-1 of the address, then we have the problem of a shared BTB entry. The `RET` instruction is particularly prone to this problem because it is only one byte long. There are various ways to solve this problem:

1. Move the code sequence a little up or down in memory so that you get a DWORD boundary between the two addresses.

2. Change a short jump to a near jump (with 4 bytes displacement) so that the end of the instruction is moved further down. There is no way you can force the assembler to use anything but the shortest form of an instruction so you have to hard-code the near jump if you choose this solution.

3. Put in some instruction between the two control transfer instructions. This is the easiest method, and the only method if you don't know where DWORD boundaries are because your segment is not DWORD aligned or because the code keeps moving up and down as you make changes in the preceding code.

There is a penalty when the first instruction pair following the target label of a call contains another call instruction or if a return follows immediately after another return.

The penalty for chained calls only occurs when the same subroutines are called from more than one location. Chained returns always have a penalty. There is sometimes a small stall for a jump after a call, but no penalty for return after call; call after return; jump, call, or return after jump; or jump after return.

Chained jumps on PPro, P2 and P3

A jump, call, or return cannot be executed in the first clock cycle after a previous jump, call, or return on the PPro, P2 and P3. Therefore, chained jumps will take two clock cycles for each jump, and you may want to make sure that the processor has something else to do in parallel. For the same reason, a loop will take at least two clock cycles per iteration on these processors.

Chained jumps on P4, P4E and PM

The retirement station can handle only one taken jump, call or return per clock cycle, and only in the first of the three retirement slots. Therefore, preferably, no more than every third µop should be a jump.

Chained jumps on AMD

Taken jumps have a throughput of one jump per two clock cycles. It is delayed another clock cycle if there is a 16-byte boundary shortly after the jump target. Not taken branches have a throughput of three per clock cycle. Avoid a one-byte return instruction immediately after a branch instruction.

4 Pentium 1 and Pentium MMX pipeline

The P1 and PMMX processors cannot do out-of-order processing. But they can execute two consecutive instructions simultaneously by an instruction pairing mechanism described below.

4.1 Pairing integer instructions

Perfect pairing

The P1 and PMMX have two pipelines for executing instructions, called the U-pipe and the V-pipe. Under certain conditions it is possible to execute two instructions simultaneously, one in the U-pipe and one in the V-pipe. This can almost double the speed. It is therefore advantageous to reorder the instructions to make them pair.

The following instructions are pairable in either pipe:

- `MOV` register, memory, or immediate into register or memory
- `PUSH` register or immediate, `POP` register
- `LEA`, `NOP`
- `INC`, `DEC`, `ADD`, `SUB`, `CMP`, `AND`, `OR`, `XOR`,
- and some forms of `TEST` (See manual 4: "Instruction tables").

The following instructions are pairable in the U-pipe only:

- `ADC`, `SBB`
- `SHR`, `SAR`, `SHL`, `SAL` with immediate count
- `ROR`, `ROL`, `RCR`, `RCL` with an immediate count of 1

The following instructions can execute in either pipe but are only pairable when in the V-pipe:

- near call
- short and near jump
- short and near conditional jump.

All other integer instructions can execute in the U-pipe only, and are not pairable.

Two consecutive instructions will pair when the following conditions are met:

1. The first instruction is pairable in the U-pipe and the second instruction is pairable in the V-pipe.

2. The second instruction does not read or write a register which the first instruction writes to.

Examples:

```
; Example 4.1a. P1/PMMX pairing rules
mov eax, ebx / mov ecx, eax      ; Read after write, do not pair
mov eax, 1   / mov eax, 2        ; Write after write, do not pair
mov ebx, eax / mov eax, 2        ; Write after read, pair ok
mov ebx, eax / mov ecx, eax      ; Read after read, pair ok
mov ebx, eax / inc eax           ; Read and write after read, pair ok
```

3. In rule 2, partial registers are treated as full registers. Example:

```
; Example 4.1b. P1/PMMX pairing rules
mov al, bl  /  mov ah, 0
```

writes to different parts of the same register, do not pair.

4. Two instructions which both write to parts of the flags register can pair despite rule 2 and 3. Example:

```
; Example 4.1c. P1/PMMX pairing rules
shr eax, 4 / inc ebx              ; pair OK
```

5. An instruction that writes to the flags can pair with a conditional jump despite rule 2. Example:

```
; Example 4.1d. P1/PMMX pairing rules
cmp eax, 2 / ja LabelBigger       ; pair OK
```

6. The following instruction combinations can pair despite the fact that they both modify the stack pointer:

```
; Example 4.1e. P1/PMMX pairing rules
push + push,  push + call,  pop + pop
```

7. There are restrictions on the pairing of instructions with prefixes. Many instructions which were not implemented on the 8086 processor have a two-byte opcode where the first byte is `0FH`. The `0FH` byte behaves as a prefix on the P1. On PMMX and later processors the `0FH` byte behaves as part of the opcode. The most common instructions with `0FH` prefix are: `MOVZX`, `MOVSX`, `PUSH FS`, `POP FS`, `PUSH GS`, `POP GS`, `LFS`, `LGS`, `LSS`, `SETcc`, `BT`, `BTC`, `BTR`, `BTS`, `BSF`, `BSR`, `SHLD`, `SHRD`, and `IMUL` with two operands and no immediate operand.

On the P1, a prefixed instruction can execute only in the U-pipe, except for conditional near jumps.

On the PMMX, instructions with operand size or address size prefix can execute in either pipe, whereas instructions with segment, repeat, or lock prefix can execute only in the U-pipe.

8. An instruction which has both a displacement and immediate data is not pairable on the P1 and only pairable in the U-pipe on the PMMX:

```
; Example 4.1f. P1/PMMX pairing rules
mov dword ptr ds:[1000], 0     ; Not pairable or only in u-pipe
cmp byte ptr [ebx+8], 1        ; Not pairable or only in u-pipe
cmp byte ptr [ebx], 1          ; Pairable
cmp byte ptr [ebx+8], al       ; Pairable
```

Another problem with instructions which have both a displacement and immediate data on the PMMX is that such instructions may be longer than 7 bytes, which means that only one instruction can be decoded per clock cycle.

9. Both instructions must be preloaded and decoded. This will not happen on the P1 unless the first instruction is only one byte long.

10. There are special pairing rules for MMX instructions on the PMMX:

- MMX shift, pack or unpack instructions can execute in either pipe but cannot pair with other MMX shift, pack or unpack instructions.
- MMX multiply instructions can execute in either pipe but cannot pair with other MMX multiply instructions. They take 3 clock cycles and the last 2 clock cycles can overlap with subsequent instructions in the same way as floating point instructions can (see

page 44).

- an MMX instruction that accesses memory or integer registers can execute only in the U-pipe and cannot pair with a non-MMX instruction.

Imperfect pairing

There are situations where the two instructions in a pair will not execute simultaneously, or only partially overlap in time. They should still be considered a pair, though, because the first instruction executes in the U-pipe, and the second in the V-pipe. No subsequent instruction can start to execute before both instructions in the imperfect pair have finished.

Imperfect pairing will happen in the following cases:

1. If the second instruction suffers an AGI stall (see page 42).

2. Two instructions cannot access the same DWORD of memory simultaneously. The following examples assume that `ESI` is divisible by 4:

```
; Example 4.2a. P1/PMMX imperfect pairing
mov al, [esi] / mov bl, [esi+1]
```

The two operands are within the same DWORD, so they cannot execute simultaneously. The pair takes 2 clock cycles.

```
; Example 4.2b. P1/PMMX perfect pairing
mov al, [esi+3] / mov bl, [esi+4]
```

Here the two operands are on each side of a DWORD boundary, so they pair perfectly, and take only one clock cycle.

3. The preceding rule is extended to the case where bits 2 - 4 are the same in the two addresses (cache line conflict). For DWORD addresses this means that the difference between the two addresses should not be divisible by 32.

Pairable integer instructions, which do not access memory, take one clock cycle to execute, except for mispredicted jumps. MOV instructions to or from memory also take only one clock cycle if the data area is in the cache and properly aligned. There is no speed penalty for using complex addressing modes such as scaled index registers.

A pairable integer instruction that reads from memory, does some calculation, and stores the result in a register or flags, takes 2 clock cycles. (read/modify instructions).

A pairable integer instruction that reads from memory, does some calculation, and writes the result back to the memory, takes 3 clock cycles. (read/modify/write instructions).

4. If a read/modify/write instruction is paired with a read/modify or read/modify/write instruction, then they will pair imperfectly.

The number of clock cycles used is given in the following table:

First instruction	Second instruction		
	MOV or register only	read/modify	read/modify/write
MOV or register only	1	2	3
read/modify	2	2	3
read/modify/write	3	4	5

Table 4.1. Pairing complex instructions

Examples:

```
; Example 4.3. P1/PMMX pairing complex instructions
add [mem1], eax / add ebx, [mem2] ; 4 clock cycles
add ebx, [mem2] / add [mem1], eax ; 3 clock cycles
```

5. When two paired instructions both take extra time due to cache misses, misalignment, or jump misprediction, then the pair will take more time than each instruction, but less than the sum of the two.

6. A pairable floating point instruction followed by `FXCH` will make imperfect pairing if the next instruction is not a floating point instruction.

In order to avoid imperfect pairing you have to know which instructions go into the U-pipe, and which to the V-pipe. You can find out this by looking backwards in your code and search for instructions which are unpairable, pairable only in one of the pipes, or cannot pair due to one of the rules above.

Imperfect pairing can often be avoided by reordering instructions. Example:

```
; Example 4.4. P1/PMMX reorder instructions to improve pairing
L1:     mov     eax,[esi]
        mov     ebx,[esi]
        inc     ecx
```

Here the two `MOV` instructions form an imperfect pair because they both access the same memory location, and the sequence takes 3 clock cycles. You can improve the code by reordering the instructions so that `INC ECX` pairs with one of the `MOV` instructions.

```
; Example 4.5. P1/PMMX reorder instructions to improve pairing
L2:     mov     eax,offset a
        xor     ebx,ebx
        inc     ebx
        mov     ecx,[eax]
        jmp     L1
```

The pair `INC EBX` / `MOV ECX,[EAX]` is imperfect because the latter instruction has an AGI stall. The sequence takes 4 clocks. If you insert a `NOP` or any other instruction so that `MOV ECX,[EAX]` pairs with `JMP L1` instead, then the sequence takes only 3 clocks.

The next example is in 16-bit mode, assuming that `SP` is divisible by 4:

```
; Example 4.6. P1/PMMX imperfect pairing, 16 bit mode
L3:     push    ax
        push    bx
        push    cx
        push    dx
        call    Func
```

Here the `PUSH` instructions form two imperfect pairs, because both operands in each pair go into the same DWORD of memory. `PUSH BX` could possibly pair perfectly with `PUSH CX` (because they go on each side of a DWORD boundary) but it doesn't because it has already been paired with `PUSH AX`. The sequence therefore takes 5 clocks. If you insert a `NOP` or any other instruction so that `PUSH BX` pairs with `PUSH CX`, and `PUSH DX` with `CALL FUNC`, then the sequence will take only 3 clocks. Another way to solve the problem is to make sure that `SP` is not divisible by 4. Knowing whether `SP` is divisible by 4 or not in 16-bit mode can be difficult, so the best way to avoid this problem is to use 32-bit mode.

4.2 Address generation interlock

It takes one clock cycle to calculate the address needed by an instruction that accesses memory. Normally, this calculation is done at a separate stage in the pipeline while the preceding instruction or instruction pair is executing. But if the address depends on the result of an instruction executing in the preceding clock cycle, then we have to wait an extra clock cycle for the address to be calculated. This is called an AGI stall.

```
; Example 4.7a. P1/PMMX AGI
add ebx,4
mov eax,[ebx] ; AGI stall
```

The stall in this example can be removed by putting some other instructions in between these two, or by rewriting the code to:

```
; Example 4.7b. P1/PMMX AGI removed
mov eax,[ebx+4]
add ebx,4
```

You can also get an AGI stall with instructions that use `ESP` implicitly for addressing, such as `PUSH`, `POP`, `CALL`, and `RET`, if `ESP` has been changed in the preceding clock cycle by instructions such as `MOV`, `ADD`, or `SUB`. The P1 and PMMX have special circuitry to predict the value of `ESP` after a stack operation so that you do not get an AGI delay after changing `ESP` with `PUSH`, `POP`, or `CALL`. You can get an AGI stall after `RET` only if it has an immediate operand to add to `ESP`. Examples:

```
; Example 4.8. P1/PMMX AGI
add esp,4 / pop esi             ; AGI stall
pop eax   / pop esi             ; no stall, pair
mov esp,ebp / ret               ; AGI stall
call F1 / F1: mov eax,[esp+8]   ; no stall
ret / pop eax                   ; no stall
ret 8 / pop eax                 ; AGI stall
```

The `LEA` instruction is also subject to an AGI stall if it uses a base or index register that has been changed in the preceding clock cycle. Example:

```
; Example 4.9. P1/PMMX AGI
inc esi / lea eax,[ebx+4*esi]   ; AGI stall
```

PPro, P2 and P3 have no AGI stalls for memory reads and `LEA`, but they do have AGI stalls for memory writes. This is not very significant unless the subsequent code has to wait for the write to finish.

4.3 Splitting complex instructions into simpler ones

You may split up read/modify and read/modify/write instructions to improve pairing. Example:

```
; Example 4.10a. P1/PMMX Imperfect pairing
add [mem1],eax
add [mem2],ebx
```

This code may be split up into a sequence that reduces the clock count from 5 to 3 clock cycles:

```
; Example 4.10b. P1/PMMX Imperfect pairing avoided
mov ecx,[mem1]
mov edx,[mem2]
add ecx,eax
```

```
add edx,ebx
mov [mem1],ecx
mov [mem2],edx
```

Likewise you may split up non-pairable instructions into pairable instructions:

```
; Example 4.11a. P1/PMMX Non-pairable instructions
push [mem1]
push [mem2]       ; Non-pairable
```

Split up into:

```
; Example 4.11b. Split nonpairable instructions into pairable ones
mov eax,[mem1]
mov ebx,[mem2]
push eax
push ebx          ; Everything pairs
```

Other examples of non-pairable instructions that may be split up into simpler pairable instructions:

`; Example 4.12. P1/PMMX Split non-pairable instructions`
`CDQ` split into: `mov edx,eax / sar edx,31`
`not eax` change to `xor eax,-1`
`neg eax` split into `xor eax,-1` / `inc eax`
`movzx eax,byte ptr [mem]` split into `xor eax,eax / mov al,byte ptr [mem]`
`jecxz L1` split into `test ecx,ecx / jz L1`
`loop L1` split into `dec ecx, / jnz L1`
`xlat` change to `mov al,[ebx+eax]`

If splitting instructions does not improve speed, then you may keep the complex or nonpairable instructions in order to reduce code size. Splitting instructions is not needed on later processors, except when the split instructions generate fewer μops.

4.4 Prefixes

An instruction with one or more prefixes may not be able to execute in the V-pipe and it may take more than one clock cycle to decode.

On the P1, the decoding delay is one clock cycle for each prefix except for the `0FH` prefix of conditional near jumps.

The PMMX has no decoding delay for `0FH` prefix. Segment and repeat prefixes take one clock extra to decode. Address and operand size prefixes take two clocks extra to decode. The PMMX can decode two instructions per clock cycle if the first instruction has a segment or repeat prefix or no prefix, and the second instruction has no prefix. Instructions with address or operand size prefixes can only decode alone on the PMMX. Instructions with more than one prefix take one clock extra for each prefix.

Where prefixes are unavoidable, the decoding delay may be masked if a preceding instruction takes more than one clock cycle to execute. The rule for the P1 is that any instruction that takes N clock cycles to execute (not to decode) can 'overshadow' the decoding delay of N-1 prefixes in the next two (sometimes three) instructions or instruction pairs. In other words, each extra clock cycle that an instruction takes to execute can be used to decode one prefix in a later instruction. This shadowing effect even extends across a predicted branch. Any instruction that takes more than one clock cycle to execute, and any instruction that is delayed because of an AGI stall, cache miss, misalignment, or any other reason except decoding delay and branch misprediction, has such a shadowing effect.

The PMMX has a similar shadowing effect, but the mechanism is different. Decoded instructions are stored in a transparent first-in-first-out (FIFO) buffer, which can hold up to four instructions. As long as there are instructions in the FIFO buffer you get no delay. When the buffer is empty then instructions are executed as soon as they are decoded. The buffer is filled when instructions are decoded faster than they are executed, i.e. when you have unpaired or multi-cycle instructions. The FIFO buffer is emptied when instructions execute faster than they are decoded, i.e. when you have decoding delays due to prefixes. The FIFO buffer is empty after a mispredicted branch. The FIFO buffer can receive two instructions per clock cycle provided that the second instruction is without prefixes and none of the instructions are longer than 7 bytes. The two execution pipelines (U and V) can each receive one instruction per clock cycle from the FIFO buffer. Examples:

```
; Example 4.13. P1/PMMX Overshadow prefix decoding delay
cld
rep movsd
```

The `CLD` instruction takes two clock cycles and can therefore overshadow the decoding delay of the `REP` prefix. The code would take one clock cycle more if the `CLD` instruction were placed far from the `REP MOVSD`.

```
; Example 4.14. P1 Overshadow prefix decoding delay
cmp dword ptr [ebx],0
mov eax,0
setnz al
```

The `CMP` instruction takes two clock cycles here because it is a read/modify instruction. The `0FH` prefix of the `SETNZ` instruction is decoded during the second clock cycle of the `CMP` instruction, so that the decoding delay is hidden on the P1 (The PMMX has no decoding delay for the `0FH`).

4.5 Scheduling floating point code

Floating point instructions cannot pair the way integer instructions can, except for one special case, defined by the following rules:

- The first instruction (executing in the U-pipe) must be `FLD`, `FADD`, `FSUB`, `FMUL`, `FDIV`, `FCOM`, `FCHS`, or `FABS`.
- The second instruction (in V-pipe) must be `FXCH.`
- The instruction following the `FXCH` must be a floating point instruction, otherwise the `FXCH` will pair imperfectly and take an extra clock cycle.

This special pairing is important, as will be explained shortly.

While floating point instructions in general cannot be paired, many can be pipelined, i.e. one instruction can begin before the previous instruction has finished. Example:

```
; Example 4.15. Pipelined floating point instructions
 fadd st(1),st(0)    ; Clock cycle 1-3
 fadd st(2),st(0)    ; Clock cycle 2-4
 fadd st(3),st(0)    ; Clock cycle 3-5
 fadd st(4),st(0)    ; Clock cycle 4-6
```

Obviously, two instructions cannot overlap if the second instruction needs the result of the first one. Since almost all floating point instructions involve the top of stack register, `ST(0)`, there are seemingly not very many possibilities for making an instruction independent of the result of previous instructions. The solution to this problem is register renaming. The FXCH instruction does not in reality swap the contents of two registers; it only swaps their names. Instructions that push or pop the register stack also work by renaming. Floating point register renaming has been highly optimized on the Pentiums so that a register may be renamed while in use. Register renaming never causes stalls - it is even possible to rename a register more than once in the same clock cycle, as for example when FLD or FCOMPP is paired with FXCH.

By the proper use of FXCH instructions you may obtain a lot of overlapping in your floating point code. All versions of the instructions FADD, FSUB, FMUL, and FILD take 3 clock cycles and are able to overlap, so that these instructions may be scheduled. Using a memory operand does not take more time than a register operand if the memory operand is in the level 1 cache and properly aligned.

By now you must be used to rules having exceptions, and the overlapping rule is no exception: You cannot start an FMUL instruction one clock cycle after another FMUL instruction, because the FMUL circuitry is not perfectly pipelined. It is recommended that you put another instruction in between two FMUL's. Example:

```
; Example 4.16a. Floating point code with stalls
fld     [a1]    ; Clock cycle 1
fld     [b1]    ; Clock cycle 2
fld     [c1]    ; Clock cycle 3
fxch    st(2)   ; Clock cycle 3
fmul    [a2]    ; Clock cycle 4-6
fxch    st(1)   ; Clock cycle 4
fmul    [b2]    ; Clock cycle 5-7    (stall)
fxch    st(2)   ; Clock cycle 5
fmul    [c2]    ; Clock cycle 7-9    (stall)
fxch    st(1)   ; Clock cycle 7
fstp    [a3]    ; Clock cycle 8-9
fxch    st(1)   ; Clock cycle 10     (unpaired)
fstp    [b3]    ; Clock cycle 11-12
fstp    [c3]    ; Clock cycle 13-14
```

Here you have a stall before FMUL [b2] and before FMUL [c2] because another FMUL started in the preceding clock cycle. You can improve this code by putting FLD instructions in between the FMUL's:

```
; Example 4.16b. Floating point stalls filled with other instructions
fld     [a1]    ; Clock cycle 1
fmul    [a2]    ; Clock cycle 2-4
fld     [b1]    ; Clock cycle 3
fmul    [b2]    ; Clock cycle 4-6
fld     [c1]    ; Clock cycle 5
fmul    [c2]    ; Clock cycle 6-8
fxch    st(2)   ; Clock cycle 6
fstp    [a3]    ; Clock cycle 7-8
fstp    [b3]    ; Clock cycle 9-10
fstp    [c3]    ; Clock cycle 11-12
```

In other cases you may put FADD, FSUB, or anything else in between FMUL's to avoid the stalls.

Not all floating point instructions can overlap. And some floating point instructions can overlap more subsequent integer instructions than subsequent floating point instructions. The FDIV instruction, for example, takes 39 clock cycles. All but the first clock cycle can overlap with integer instructions, but only the last two clock cycles can overlap with floating point instructions. A complete listing of floating point instructions, and what they can pair or overlap with, is given in manual 4: "Instruction tables".

There is no penalty for using a memory operand on floating point instructions because the arithmetic unit is one stage later in the pipeline than the read unit. The tradeoff of this comes when a floating point value is stored to memory. The `FST` or `FSTP` instruction with a memory operand takes two clock cycles in the execution stage, but it needs the data one clock earlier so you will get a one-clock stall if the value to store is not ready one clock cycle in advance. This is analogous to an AGI stall. In many cases you cannot hide this type of stall without scheduling the floating point code into four threads or putting some integer instructions in between. The two clock cycles in the execution stage of the `FST(P)` instruction cannot pair or overlap with any subsequent instructions.

Instructions with integer operands such as `FIADD`, `FISUB`, `FIMUL`, `FIDIV`, `FICOM` may be split up into simpler operations in order to improve overlapping. Example:

```
; Example 4.17a. Floating point code with integer operands
fild [a]
fimul [b]
```

Split up into:

```
; Example 4.17b. Overlapping integer operations
fild [a]
fild [b]
fmul
```

In this example, we save two clocks by overlapping the two `FILD` instructions.

5 Pentium 4 (NetBurst) pipeline

The Intel P4, which was introduced in 2000, and the later variant P4E were based on the so-called NetBurst microarchitecture, which was very different from the design of previous Intel processors. This architecture has turned out to be less efficient than expected and is no longer used in new designs.

The primary design goal of the NetBurst microarchitecture was to obtain the highest possible clock frequency. This can only be achieved by making the pipeline longer. The 6'th generation microprocessors PPro, P2 and P3 (see next chapter) have a pipeline of ten stages. The PM, which is an improvement of the same design, has approximately 13 stages. The 7'th generation microprocessor P4 has a 20 stage pipeline, and the P4E has even a few stages more. Some of the stages are just for moving data from one part of the chip to another.

An important difference from previous processors was that the code cache was replaced by a trace cache which contained decoded µops rather than instructions. The advantage of a trace cache is that the decoding bottleneck is removed and the design can use RISC technology. The disadvantage is that the information in the trace cache is less compact and takes up more chip space.

The out-of-order core in P4 is similar to the PPro design, but bigger. The reorder buffer can contain 126 µops in process. There is no limitation on register reads and renamings, but the maximum throughput is still limited to 3 µops per clock cycle, and the limitations in the retirement station are the same as in the PPro.

5.1 Data cache

The on-chip level-2 cache is used for both code and data. The size of the level-2 cache ranges from 256 kb to 2 MB for different models. The level-2 cache is organized as 8 ways, 64 bytes per line. It runs at full speed with a 256 bits wide data bus to the central processor, and is quite efficient.

The level-1 data cache is 8 or 16 kb, 8 ways, 64 bytes per line. The relatively small size of the level-1 data cache is compensated for by the fast access to the level-2 cache. The level-1 data cache uses a write-through mechanism rather than write-back. This reduces the write bandwidth.

The level-1 code cache is a trace cache, as explained below.

5.2 Trace cache

Instructions are stored in the trace cache *after* being decoded into µops. Rather than storing instruction opcodes in a level-1 cache, it stores decoded µops. One important reason for this is that the decoding stage was a bottleneck on earlier processors. An opcode can have any length from 1 to 15 bytes. It is quite complicated to determine the length of an instruction opcode; and we have to know the length of the first opcode in order to know where the second opcode begins. Therefore, it is difficult to determine opcode lengths in parallel. The 6'th generation microprocessors could decode three instructions per clock cycle. This may be more difficult at higher clock speeds. If µops all have the same size, then the processor can handle them in parallel, and the bottleneck disappears. This is the principle of RISC processors. Caching µops rather than opcodes enables the P4 and P4E to use RISC technology on a CISC instruction set. A trace in the trace cache is a string of µops that are executed in sequence, even if they are not sequential in the original code. The advantage of this is that the number of clock cycles spent on jumping around in the cache is minimized. This is a second reason for using a trace cache.

The µops take more space than opcodes on average. The following table shows the size of each trace cache entry:

Processor	instruction encoding, bits	immediate data or address, bits	address tag, bits	total bits per entry	number of lines	entries per line	total entries
P4	21	16	16	53	2048	6	12k
P4E	16	32	16	64	2048	6	12k

Table 5.1. Number of bits per trace cache entry and number of entries in trace cache.
(These numbers are approximate and speculative. See www.chip-architect.com 2003-04-20)

The trace cache is organized as 2048 lines of 6 entries each, 8-way set-associative. The trace cache runs at half clock speed, delivering up to 6 µops every two clock cycles.

Economizing trace cache use on P4

On the P4, 16 of the bits in each entry are reserved for data. This means that a µop that requires more than 16 bits of data storage must use two entries. You can calculate whether a µop uses one or two trace cache entries by the following rules, which have been obtained experimentally.

1. A µop with no immediate data and no memory operand uses only one trace cache entry.

2. A µop with an 8-bit or 16-bit immediate operand uses one trace cache entry.

3. A µop with a 32-bit immediate operand in the interval from -32768 to +32767 uses one trace cache entry. The immediate operand is stored as a 16-bit signed integer. If an opcode contains a 32-bit constant, then the decoder will investigate if this constant is within the interval that allows it to be represented as a 16-bit signed integer. If this is the case, then the µop can be contained in a single trace cache entry.

4. If a µop has an immediate 32-bit operand outside the $\pm 2^{15}$ interval so that it cannot be represented as a 16-bit signed integer, then it will use two trace cache entries unless it can borrow storage space from a nearby µop.

5. A µop in need of extra storage space can borrow 16 bits of extra storage space from a nearby µop that doesn't need its own data space. Almost any µop that has no immediate operand and no memory operand will have an empty 16-bit data space for other µops to borrow. A µop that requires extra storage space can borrow space from the next µop as well as from any of the preceding 3 - 5 µops (5 if it is not number 2 or 3 in a trace cache line), even if they are not in the same trace cache line. A µop cannot borrow space from a preceding µop if any µop between the two is double size or has borrowed space. Space is preferentially borrowed from preceding rather than subsequent µops.

6. The displacement of a near jump, call or conditional jump is stored as a 16-bit signed integer, if possible. An extra trace cache entry is used if the displacement is outside the $\pm 2^{15}$ range and no extra storage space can be borrowed according to rule 5 (Displacements outside this range are rare).

7. A memory load or store µop will store the address or displacement as a 16-bit integer, if possible. This integer is signed if there is a base or index register, otherwise unsigned. Extra storage space is needed if a direct address is $\geq 2^{16}$ or an indirect address (i.e. with one or two pointer registers) has a displacement outside

the ±2[15] interval.

8. Memory load µops can *not* borrow extra storage space from other µops. If 16 bits of storage is insufficient then an extra trace cache entry will be used, regardless of borrowing opportunities.

9. Most memory store instructions generate two µops: The first µop, which goes to port 3, calculates the memory address. The second µop, which goes to port 0, transfers the data from the source operand to the memory location calculated by the first µop. The first µop can always borrow storage space from the second µop. This space cannot be borrowed to any other µop, even if it is empty.

10. Store operations with an 8, 16, or 32-bit register as source, and no SIB byte, can be contained in a single µop. These µops can borrow storage space from other µops, according to rule 5 above.

11. Segment prefixes do not require extra storage space.

12. A µop cannot have both a memory operand and an immediate operand. An instruction that contains both will be split into two or more µops. No µop can use more than two trace cache entries.

13. A µop that requires two trace cache entries cannot cross a trace cache line boundary. If a double-space µop would cross a 6-entry boundary in the trace cache then an empty space will be inserted and the µop will use the first two entries of the next trace cache line.

The difference between load and store operations needs an explanation. My theory is as follows: No µop can have more than two input dependences (not including segment registers). Any instruction that has more than two input dependences needs to be split up into two or more µops. Examples are `ADC` and `CMOVcc`. A store instruction like `MOV [ESI+EDI],EAX` also has three input dependences. It is therefore split up into two µops. The first µop calculates the address `[ESI+EDI]`, the second µop stores the value of `EAX` to the calculated address. In order to optimize the most common store instructions, a single-µop version has been implemented to handle the situations where there is no more than one pointer register. The decoder makes the distinction by seeing if there is a SIB byte in the address field of the instruction. A SIB byte is needed if there is more than one pointer register, or a scaled index register, or `ESP` as base pointer. Load instructions, on the other hand, can never have more than two input dependences. Therefore, load instructions are implemented as single-µop instructions in the most common cases. The load µops need to contain more information than the store µops. In addition to the type and number of the destination register, it needs to store any segment prefix, base pointer, index pointer, scale factor, and displacement. The size of the trace cache entries has probably been chosen to be exactly enough to contain this information. Allocating a few more bits for the load µop to indicate where it is borrowing storage space from would mean that all trace cache entries would have a bigger size. Given the physical constraints on the trace cache, this would mean fewer entries. This is probably the reason why memory load µops cannot borrow storage space. The store instructions do not have this problem because the necessary information is already split up between two µops unless there is no SIB byte, and hence less information to contain.

The following examples will illustrate the rules for trace cache use (P4 only):

```
; Example 5.1. P4 trace cache use
add eax,10000    ; The constant 10000 uses 32 bits in the opcode, but
                 ; can be contained in 16 bits in uop. uses 1 space.
add ebx,40000    ; The constant is bigger than 215, but it can borrow
                 ; storage space from the next uop.
```

```
add ebx,ecx     ; Uses 1 space. gives storage space to preceding uop.
mov eax,[mem1]  ; Requires 2 spaces, assuming that address ≥ 216;
                ; preceding borrowing space is already used.
mov eax,[esi+4] ; Requires 1 space.
mov [si],ax     ; Requires 1 space.
mov ax,[si]     ; Requires 2 uops taking one space each.
movzx eax,word ptr[si]          ; Requires 1 space.
movdqa xmm1,es:[esi+100h]       ; Requires 1 space.
fld qword ptr es:[ebp+8*edx+16] ; Requires 1 space.
mov [ebp+4], ebx      ; Requires 1 space.
mov [esp+4], ebx      ; Requires 2 uops because sib byte needed.
fstp dword ptr [mem2] ; Requires 2 uops. the first uop borrows
                      ; space from the second one.
```

No further data compression is used in the trace cache besides the methods mentioned above. A program that has a lot of direct memory addresses will typically use two trace cache entries for each data access, even if all memory addresses are within the same narrow range. In a flat memory model, the address of a direct memory operand uses 32 bits in the opcode. The assembler listing will typically show addresses lower than 2^{16}, but these addresses are relocated twice before the microprocessor sees them. The first relocation is done by the linker; the second relocation is done by the loader when the program is loaded into memory. When a flat memory model is used, the loader will typically place the entire program at a virtual address space beginning at a value > 2^{16}. You may save space in the trace cache by accessing data through pointers. In high-level languages like C++, local data are always saved on the stack and accessed through pointers. Direct addressing of global and static data can be avoided by using classes and member functions. Similar methods may be applied in assembly programs.

You can prevent double-size µops from crossing 6-entry boundaries by scheduling them so that there is an even number (including 0) of single-size µops between any two double-size µops (A long, continuous 2-1-2-1 pattern will also do). Example:

```
; Example 5.2a. P4 trace cache double entries
mov eax, [mem1]    ; 1 uop, 2 TC entries
add eax, 1         ; 1 uop, 1 TC entry
mov ebx, [mem2]    ; 1 uop, 2 TC entries
mov [mem3], eax    ; 1 uop, 2 TC entries
add ebx, 1         ; 1 uop, 1 TC entry
```

If we assume, for example, that the first µop here starts at 6-entry boundary, then the `MOV [MEM3],EAX` µop will cross the next 6-entry boundary at the cost of an empty entry. This can be prevented by re-arranging the code:

```
; Example 5.2b. P4 trace cache double entries rearranged
mov eax, [mem1]    ; 1 uop, 2 TC entries
mov ebx, [mem2]    ; 1 uop, 2 TC entries
add eax, 1         ; 1 uop, 1 TC entry
add ebx, 1         ; 1 uop, 1 TC entry
mov [mem3], eax    ; 1 uop, 2 TC entries
```

We cannot know whether the first two µops are crossing any 6-entry boundary as long as we haven't looked at the preceding code, but we can be certain that the `MOV [MEM3],EAX` µop will not cross a boundary, because the second entry of the first µop cannot be the first entry in a trace cache line. If a long code sequence is arranged so that there is never an odd number of single-size µops between any two double-size µops then we will not waste any trace cache entries. The preceding two examples assume that direct memory operands are bigger than 2^{16}, which is usually the case. For the sake of simplicity, I have used only instructions that generate 1 µop each in these examples. For instructions that generate more than one µop, you have to consider each µop separately.

Trace cache use on P4E

The trace cache entries on the P4E need to be bigger than on the P4 because the processor can run in 64 bit mode. This simplifies the design considerably. The need for borrowing storage space from neighboring entries has been completely eliminated. Each entry has 32 bits for immediate data, which is sufficient for all µops in 32 bit mode. Only a few instructions in 64 bit mode can have 64 bits of immediate data or address, and these instructions are split into two or three µops which contain no more than 32 data bits each. Consequently, each µop uses one, and only one, trace cache entry.

Trace cache delivery rate

The trace cache runs at half clock speed, delivering one trace cache line with six entries every two clock cycles. This corresponds to a maximum throughput of three µops per clock cycle.

The typical delivery rate may be slightly lower on P4 because some µops use two entries and some entries may be lost when a two-entry µop crosses a trace cache line boundary.

The throughput on P4E has been measured to exactly 8/3 or 2.667 µops per clock cycle. I have found no explanation why a throughput of 3 µops per clock cannot be obtained on P4E. It may be due to a bottleneck elsewhere in the pipeline.

Branches in the trace cache

The µops in the trace cache are not stored in the same order as the original code. If a branching µop jumps most of the time, then the traces will usually be organized so that the jumping µop is followed by the µops jumped to, rather than the µops that follows it in the original code. This reduces the number of jumps between traces. The same sequence of µops can appear more than once in the trace cache if it is jumped to from different places.

Sometimes it is possible to control which of the two branches are stored after a branching µop by using branch hint prefixes (see page 36), but my experiments have shown no consistent advantage of doing so. Even in the cases where there is an advantage by using branch hint prefixes, this effect does not last very long because the traces are rearranged quite often to fit the behavior of the branch µops. You can therefore assume that traces are usually organized according to the way branches go most often.

The µop delivery rate is usually less than the maximum if the code contains many jumps, calls and branches. If a branch is not the last entry in a trace cache line and the branch goes to another trace stored elsewhere in the trace cache, then the rest of the entries in the trace cache line are loaded for no use. This reduces the throughput. There is no loss if the branching µop is the last µop in a trace cache line. In theory, it might be possible to organize code so that branch µops appear in the end of trace cache lines in order to avoid losses. But attempts to do so are rarely successful because it is almost impossible to predict where each trace begins. Sometimes, a small loop containing branches can be improved by organizing it so that each branch contains a number of trace cache entries divisible by the line size (six). A number of trace cache entries that is slightly less than a multiple of the line size is better than a number slightly more than a multiple of the line size.

Obviously, these considerations are only relevant if the throughput is not limited by any other bottleneck in the execution units, and the branches are predictable.

Guidelines for improving trace cache performance

The following guidelines can improve performance on the P4 if the delivery of µops from the trace cache is a bottleneck:

1. Prefer instructions that generate few µops.

2. Replace branch instructions by conditional moves if this does not imply extra dependences.

3. Keep immediate operands in the range between -2^{15} and $+2^{15}$ if possible. If a µop has an immediate 32-bit operand outside this range, then you should preferably have a µop with no immediate operand and no memory operand before or immediately after the µop with the big operand.

4. Avoid direct memory addresses. The performance can be improved by using pointers if the same pointer can be used repeatedly and the addresses are within $\pm 2^{15}$ of the pointer register.

5. Avoid having an odd number of single-size µops between any two double-size µops. Instructions that generate double-size µops include memory loads with direct memory operands, and other µops with an unmet need for extra storage space.

Only the first two of these guidelines are relevant to the P4E.

5.3 Instruction decoding

Instructions that are not in the trace cache will go directly from the instruction decoder to the execution pipeline. In this case, the maximum throughput is determined by the instruction decoder.

In most cases, the decoder generates 1 - 4 µops for each instruction. For complex instructions that require more than 4 µops, the µops are submitted from microcode ROM. The tables in manual 4: "Instruction tables" list the number of decoder µops and microcode µops that each instruction generates.

The decoder can handle instructions at a maximum rate of one instruction per clock cycle. There are a few cases where the decoding of an instruction takes more than one clock cycle:

An instruction that generates micro-code may take more than one clock cycle to decode, sometimes much more. The following instructions, which may in some cases generate micro-code, do not take significantly more time to decode: moves to and from segment registers, `ADC`, `SBB`, `IMUL`, `MUL`, `MOVDQU`, `MOVUPS`, `MOVUPD`.

Instructions with many prefixes take extra time to decode. The instruction decoder on P4 can handle one prefix per clock cycle. An instruction with more than one prefix will thus take one clock cycle for each prefix to decode on the P4. Instructions with more than one prefix are rare in a 32-bit flat memory model where segment prefixes are not needed.

The instruction decoder on P4E can handle two prefixes per clock cycle. Thus, an instruction with up to two prefixes can be decoded in a single clock cycle, while an instruction with three or four prefixes is decoded in two clock cycles. This capability was introduced in the P4E because instructions with two prefixes are common in 64 bit mode (e.g. operand size prefix and REX prefix). Instructions with more than two prefixes are very rare because segment prefixes are rarely used in 64 bit mode.

Decoding time is not important for small loops that fit entirely into the trace cache. If the critical part of the code is too big for the trace cache, or scattered around in many small pieces, then the µops may go directly from the decoder to the execution pipeline, and the decoding speed may be a bottleneck. The level-2 cache is so efficient that you can safely assume that it delivers code to the decoder at a sufficient speed.

If it takes longer time to execute a piece of code than to decode it, then the trace may not stay in the trace cache. This has no negative influence on the performance, because the code can run directly from the decoder again next time it is executed, without delay. This mechanism tends to reserve the trace cache for the pieces of code that execute faster than they decode. I have not found out which algorithm the microprocessor uses to decide whether a piece of code should stay in the trace cache or not, but the algorithm seems to be rather conservative, rejecting code from the trace cache only in extreme cases.

5.4 Execution units

Mops from the trace cache or from the decoder are queued when they are waiting to be executed. After register renaming and reordering, each µop goes through a port to an execution unit. Each execution unit has one or more subunits which are specialized for particular operations, such as addition or multiplication. The organization of ports, execution units, and subunits is outlined in the following two tables for the P4 and P4E, respectively.

port	execution unit	subunit	speed	latency	reciprocal throughput
0	alu0	add, sub, mov	double	0.5	0.5
		logic	double	0.5	0.5
		store integer	single	1	1
		branch	single	1	1
0	mov	move and store fp, mmx, xmm	single	6	1
		fxch	single	0	1
1	alu1	add, sub, mov	double	0.5	0.5
1	int	misc.	single		
		borrows mmx shift	single	4	1
		borrows fp mul	single	14	> 4
		borrows fp div	single	53 - 61	23
1	fp	fp add	single	4 - 5	1 - 2
		fp mul	single	6 - 7	2
		fp div	single	23 - 69	23 - 69
		fp misc.	single		
1	mmx	mmx alu	single	2	1 - 2
		mmx shift	single	2	1 - 2
		mmx misc.	single		
2	load	all loads	single		1
3	store	store address	single		2

Table 5.2. Execution units in P4

port	execution unit	subunit	speed	latency	reciprocal throughput
0	alu0	add, sub, mov	double	1	0.5
		logic	double	1	0.5
		store integer	single	1	1
		branch	single	1	1
0	mov	move and store fp, mmx, xmm	single	7	1
		fxch	single	0	1
1	alu1	add, sub, mov	double	1	0.5
		shift	double	1	0.5
		multiply	double	10	2.5
1	int	misc	single		

		borrows fp div	single	63 - 96	34
1	fp	fp add	single	5 - 6	1 - 2
		fp mul	single	7 - 8	2
		fp div	single	32 - 71	32 - 71
		fp misc.	single		
1	mmx	mmx alu	single	2	1 - 2
		mmx shift	single	2	1 - 2
		mmx misc.	single		
2	load	all loads	single		1
3	store	store address	single		2

Table 5.3. Execution units in P4E

Further explanation can be found in "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual". The table above deviates slightly from diagrams in the Intel manual in order to account for various delays.

A µop can be executed when the following conditions are met:

- All operands for the µop are ready.
- An appropriate execution port is ready.
- An appropriate execution unit is ready.
- An appropriate execution subunit is ready.

Two of the execution units run at double clock speed. This is alu0 and alu1, which are used for integer operations. These units are highly optimized in order to execute the most common µops as fast as possible. The double clock speed enables these two units to receive a new µop every half clock cycle. An instruction like `ADD EAX,EBX` can execute in either of these two units. This means that the execution core can handle four integer additions per clock cycle. alu0 and alu1 are both pipelined in three stages. The lower half of the result (16 bits on P4, 32 bits on P4E) is calculated in the first half clock cycle, the upper half is calculated in the second half clock cycle, and the flags are calculated in the third half-clock cycle. On the P4, the lower 16 bits are available to a subsequent µop already after a half clock cycle, so that the effective latency will appear to be only a half clock cycle. The double-speed execution units are designed to handle only the most common instructions in order to make them as small as possible. This is necessary for making the high speed possible.

This so-called "staggered addition" in three pipeline stages in alu0 and alu1 is revealed in "The Microarchitecture of the Pentium 4 Processor", Intel Technology journal 2001. I haven't been able to confirm this experimentally, since there is no difference in latencies between 8-bit, 16-bit, 32-bit and 64-bit additions.

It is unknown whether the floating point and MMX units also use staggered addition, and at what speed. See page 56 for a discussion.

The trace cache can submit approximately three µops per clock cycle to the queue. This sets a limit to the execution speed if all µops are of the type that can execute in alu0 and alu1. The throughput of four µops per clock cycle can thus only be obtained if µops have been queued during a preceding period of lower throughput (due to slow instructions or cache misses). My measurements show that a throughput of four µops per clock cycle can be obtained for a maximum of 11 consecutive clock cycles if the queue has been filled during a preceding period of lower throughput.

Each port can receive one µop for every whole clock tick. Port 0 and port 1 can each receive one additional µop at every half-clock tick, if the additional µop is destined for alu0 or alu1. This means that if a code sequence consists of only µops that go to alu0 then the throughput is two µops per clock cycle. If the µops can go to either alu0 or alu1 then the throughput at this stage can be four µops per clock cycle. If all µops go to the single-speed execution units under port 1 then the throughput is limited to one µop per clock cycle. If all ports and units are used evenly, then the throughput at this stage may be as high as six µops per clock cycle.

The single-speed execution units can each receive one µop per clock cycle. Some subunits have a lower throughput. For example, the FP-DIV subunit cannot start a new division before the preceding division is finished. Other subunits are perfectly pipelined. For example, a floating point addition may take 6 clock cycles, but the FP-ADD subunit can start a new `FADD` operation every clock cycle. In other words, if the first `FADD` operation goes from time T to T+6, then the second `FADD` can start at time T+1 and end at time T+7, and the third `FADD` goes from time T+2 to T+8, etc. Obviously, this is only possible if each `FADD` operation is independent of the results of the preceding ones.

Details about µops, execution units, subunits, throughput and latencies are listed in manual 4: "Instruction tables". The following examples will illustrate how to use this table for making time calculations. Timings are for P4E.

```
; Example 5.3. P4E instruction latencies
fadd st, st(1)          ; 0 - 6
fadd qword ptr [esi]    ; 6 - 12
```

The first `FADD` instruction has a latency of 6 clock cycles. If it starts at time T=0, it will be finished at time T=6. The second `FADD` depends on the result of the first one. Hence, the time is determined by the latency, not the throughput of the FP-ADD unit. The second addition will start at time T=6 and be finished at time T=12. The second `FADD` instruction generates an additional µop that loads the memory operand. Memory loads go to port 0, while floating point arithmetic operations go to port 1. The memory load µop can start at time T=0 simultaneously with the first `FADD` or perhaps even earlier. If the operand is in the level-1 or level-2 data cache then we can expect it to be ready before it is needed.

The second example shows how to calculate throughput:

```
; Example 5.4. P4E instruction throughput
                       ; Clock cycle
pmullw xmm1, xmm0      ; 0 - 7
paddw  xmm2, xmm0      ; 1 - 3
paddw  mm1,  mm0       ; 3 - 5
paddw  xmm3, [esi]     ; 4 - 6
```

The 128-bit packed multiplication has a latency of 7 and a reciprocal throughput of 2. The subsequent addition uses a different execution unit. It can therefore start as soon as port 1 is vacant. The 128-bit packed additions have a reciprocal throughput of 2, while the 64-bit versions have a reciprocal throughput of 1. Reciprocal throughput is also called issue latency. A reciprocal throughput of 2 means that the second `PADD` can start 2 clocks after the first one. The second `PADD` operates on 64-bit registers, but uses the same execution subunit. It has a throughput of 1, which means that the third `PADD` can start one clock later. As in the previous example, the last instruction generates an additional memory load µop. As the memory load µop goes to port 0, while the other µops go to port 1, the memory load does not affect the throughput. None of the instructions in this example depend on the results of the preceding ones. Consequently, only the throughput matters, not the latency. We cannot know if the four instructions are executed in program order or they are reordered. However, reordering will not affect the overall throughput of the code sequence.

5.5 Do the floating point and MMX units run at half speed?

Looking at the tables in manual 4: "Instruction tables", we notice that many of the latencies for 64-bit and 128-bit integer and floating point instructions are even numbers, especially for the P4. This has led to speculations that the MMX and FP execution units may be running at half clock speed. I have put up four different hypotheses in order to investigate this question:

Hypothesis 1

128-bit instructions are split into two 64-bit μops as in the P3. However, this hypothesis is not in accordance with the μop counts that can be measured with the performance monitor counters on the P4.

Hypothesis 2

We may assume that the P4 has two 64-bit MMX units working together at half speed. Each 128-bit μop will use both units and take 2 clock cycles, as illustrated on fig 5.1. A 64-bit μop can use either of the two units so that independent 64-bit μops can execute at a throughput of one μop per clock cycle, assuming that the half-speed units can start at both odd and even clocks. Dependent 64-bit μops will have a latency of 2 clocks, as shown in fig 5.1.

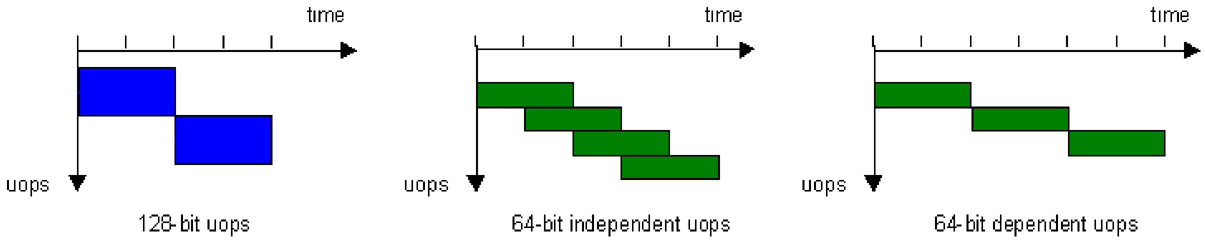

Figure 5.1

The measured latencies and throughputs are in accordance with this hypothesis. In order to test this hypothesis, I have made an experiment with a series of alternating 128-bit and 64-bit μops. Under hypothesis 2, it will be impossible for a 64-bit μop to overlap with a 128-bit μop, because the 128-bit μop uses both 64-bit units. A long sequence of n 128-bit μops alternating with n 64-bit μops should take $4 \cdot n$ clocks, as shown in figure 5.2.

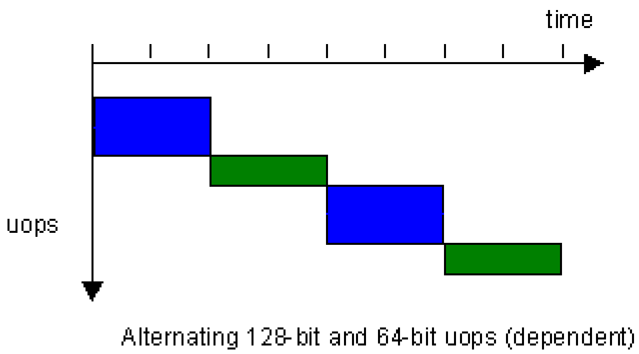

Figure 5.2

However, my experiment shows that this sequence takes only $3 \cdot n$ clocks. (I have made the 64-bit μops interdependent, so that they cannot overlap with each other). We therefore have to reject hypothesis 2.

Hypothesis 3

We may modify hypothesis 2 with the assumption that the internal data bus is only 64 bits wide, so that a 128-bit operand is transferred to the execution units in two clock cycles. If we still assume that there are two 64-bit execution units running at half speed, then the first 64-bit unit can start at time T = 0 when the first half of the 128-bit operand arrives, while the second 64-bit unit will start one clock later, when the second half of the operand arrives (see figure 5.3). The first 64-bit unit will then be able to accept a new 64-bit operand at time T=2, before the second 64-bit unit is finished with the second half of the 128-bit operand. If we have a sequence of alternating 128-bit and 64-bit µops, then the third µop, which is 128-bit, can start with its first half operand at time T=3, using the second 64-bit execution unit, while the second operand starts at T=4 using the first 64-bit execution unit. As figure 5.3 shows, this can explain the observation that a sequence of *n* 128-bit µops alternating with *n* 64-bit µops takes 3·*n* clocks.

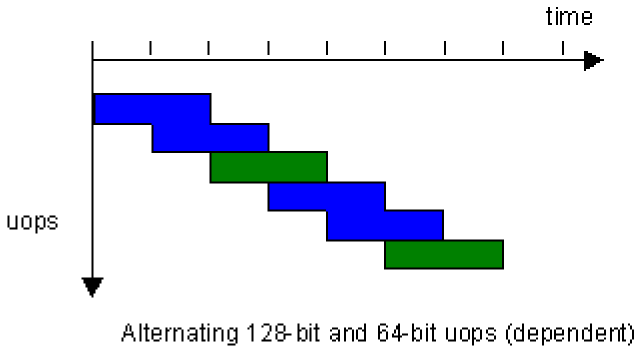

Figure 5.3

The measured latency of simple 128-bit µops is not 3 clocks, but 2. In order to explain this, we have to look at how a dependency chain of 128-bit µops is executed. Figure 5.4 shows the execution of a chain of interdependent 128-bit µops.

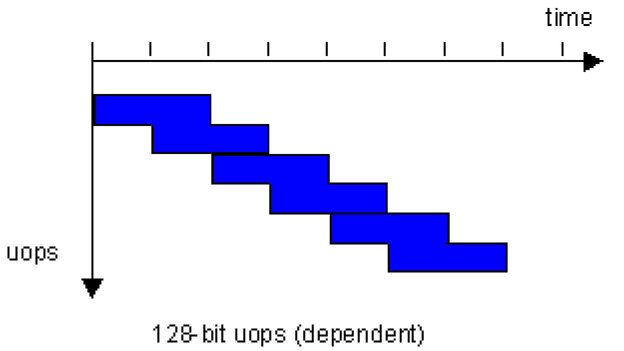

Figure 5.4

The first µop handles the first half of its operand from time T = 0 to 2, while the second half of the operand is handled from time T = 1 to time 3. The second µop can start to handle its first half operand already at time T = 2, even though the second half operand is not ready until time T = 3. A sequence of *n* interdependent 128-bit µops of this kind will thus take 2·*n*+1 clocks. The extra 1 clock in the end will appear to be part of the latency of the final instruction in the chain, which stores the result to memory. Thus, for practical purposes, we can calculate with a latency of 2 clocks for simple 128-bit µops.

Hypothesis 4

The assumption is now that there is only one 64-bit arithmetic unit running at full speed. It has a latency of 2 clocks and is pipelined in two stages, so that it can accept a new 64-bit operand every clock cycle. Under this assumption, the sequence of alternating 128-bit and 64-bit µops will still be executed as shown in figure 5.3.

There is no experimental way to distinguish between hypothesis 3 and 4 if the two units assumed under hypothesis 3 are identical, because all inputs and outputs to the execution units occur at the same times under both of these hypotheses. It would be possible to prove hypothesis 3 and reject hypothesis 4 if there were some 64-bit operations that could execute only in one of the two assumed 64-bit units. It is likely that some of the rarest operations would be supported only in one of the two units. And it would be possible to prove this by making an experiment where only the unit that does not support a particular operation is vacant when this operation is scheduled for execution. I have made a systematic search for operations that might be supported only by one of the two hypothetical units. The only candidate I have found is the 64-bit addition `PADDQ`. My experiments show that the 64-bit `PADDQ MM` executes in the MMX-ALU unit, while the 128-bit `PADDQ XMM` executes in the FP-ADD unit. However, further experiments show that if there are two 64-bit MMX-ADD units then they are both able to perform the `PADDQ MM`. This makes hypothesis 4 more likely than hypothesis 3.

If hypothesis 4 is right, then we have a problem explaining why it needs two pipeline stages. If the MMX-ALU unit is able to do a staggered 64-bit addition in 2 clock cycles, then it would be possible to do a packed 32-bit addition in 1 clock cycle. It is difficult to believe that the designers have given all MMX instructions a latency of 2 rather than 1 just for the sake of the rare `PADDQ MM` instruction. A more likely explanation is that each adder is fixed at a particular pipeline stage. I therefore consider hypothesis 4 the most likely explanation.

However, the following sentence may be read as support for hypothesis 3: "Intel NetBurst micro-architecture [...] uses a deeply pipelined design to enable high clock rates with different parts of the chip running at different clock rates, some faster and some slower than the nominally-quoted clock frequency of the processor" (Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, 2001). Letting different units run at different speeds may actually be a better design decision than letting the slowest unit determine the overall clock frequency. A further reason for this choice may be to reduce power consumption and optimize the thermal design. It is possible that some parts of e. g. the floating point unit run at half speed, but the above citation may just as well refer to the trace cache running at half speed.

Those 128-bit MMX µops where the two 64-bit halves are interdependent of each other all have a latency of 4 clocks. This is in accordance with hypothesis 3 and 4.

Floating point addition and multiplication µops operating on 80-bit registers have latencies that are one clock cycle more than the latencies of similar µops in 128-bit registers. Under hypothesis 3, the extra clock cycle can be explained as the extra time it takes to transfer an 80-bit operand over a 64-bit data bus. Under hypothesis 4, the extra clock cycle can be explained as the time needed to generate the extra 80-bit precision.

Scalar floating point operations in 80-bit registers have a throughput of 1 µop per clock cycle, while scalar floating point operations in 128-bit registers have half throughput, even though they only use 32 or 64 of the 128 bits. This is probably because the remaining 96 or 64 bits of the destination operand, which remain unchanged, are going through the execution unit to the new (renamed) destination register.

Divisions behave differently. There is a separate division unit which uses iteration and is not pipelined. Divisions can have both odd and even latencies, so it is likely that the division unit runs at full speed. Division uses the FP-MUL unit, which implies that the FP-MUL unit probably also runs at full speed.

5.6 Transfer of data between execution units

The latency of an operation is in most cases longer if the next dependent operation is not executed in the same execution unit. Example (P4E):

```
; Example 5.5. P4E transfer data between execution units
                      ; clock      ex.unit   subunit
paddw  xmm0, xmm1     ;  0 -  2    MMX       ALU
psllw  xmm0, 4        ;  2 -  4    MMX       SHIFT
pmullw xmm0, xmm2     ;  5 - 12    FP        MUL
psubw  xmm0, xmm3     ; 13 - 15    MMX       ALU
por    xmm6, xmm7     ;  3 -  5    MMX       ALU
movdqa xmm1, xmm0     ; 16 - 23    MOV
```

```
pand    xmm1, xmm4     ; 23 - 25    MMX      ALU
```

The first instruction PADDW runs in the MMX unit under port 1, and has a latency of 2. The shift instruction PSLLW runs in the same execution unit, though in a different subunit. There is no extra delay, so it can start at time T=2. The multiplication instruction PMULLW runs in a different execution unit, the FP unit, because there is no multiplication subunit in the MMX execution unit. This gives an extra delay of one clock cycle. The multiplication cannot start until T=5, even though the shift operation finished at T=4. The next instruction, PSUBW, goes back to the MMX unit, so again we have a delay of one clock cycle from the multiplication is finished till the subtraction can begin. The POR does not depend on any of the preceding instructions, so it can start as soon as port 1 and the MMX-ALU subunit are both vacant. The MOVDQA instruction goes to the MOV unit under port 0, which gives us another delay of one clock cycle after the PSUBW has finished. The last instruction, PAND, goes back to the MMX unit under port 1. However, there is no additional delay after a move instruction. The whole sequence takes 25 clock cycles.

There is no delay between the two double-speed units, ALU0 and ALU1, but on the P4 there is an additional delay of a half clock cycle from these units to any other (single-speed) execution unit. Example (P4):

```
; Example 5.6a. P4 transfer data between execution units
                    ;   clock         ex.unit   subunit
and  eax, 0fh       ;  0.0 -  0.5     ALU0      LOGIC
xor  ebx, 30h       ;  0.5 -  1.0     ALU0      LOGIC
add  eax, 1         ;  0.5 -  1.0     ALU1      ADD
shl  eax, 3         ;  2.0 -  6.0     INT       MMX SHIFT
sub  eax, ecx       ;  7.0 -  7.5     ALU0/1    ADD
mov  edx, eax       ;  7.5 -  8.0     ALU0/1    MOV
imul edx, 100       ;  9.0 - 23.0     INT       FP MUL
or   edx, ebx       ; 23.0 - 23.5     ALU0/1    MOV
```

The first instruction, AND, starts at time T=0 in ALU0. Running at double speed, it is finished at time 0.5. The XOR instruction starts as soon as ALU0 is vacant, at time 0.5. The third instruction, ADD, needs the result of the first instruction, but not the second. Since ALU0 is occupied by the XOR, the ADD has to go to ALU1. There is no delay from ALU0 to ALU1, so the ADD can start at time T=0.5, simultaneously with the XOR, and finish at T=1.0. The SHL instruction runs in the single-speed INT unit. There is a half clock delay from ALU0 or ALU1 to any other unit, so the INT unit cannot receive the result of the ADD until time T=1.5. Running at single speed, the INT unit cannot start at a half-clock tick so it will wait until time T=2.0 and finish at T=6.0. The next instruction, SUB, goes back to ALU0 or ALU1. There is a one-clock delay from the SHL instruction to any other execution unit, so the SUB instruction is delayed until time T=7.0. After the two double-speed instructions, SUB and MOV, we have a half clock delay again before the IMUL running in the INT unit. The IMUL, running again at single speed, cannot start at time T=8.5 so it is delayed until T=9.0. There is no additional delay after IMUL, so the last instruction can start at T=23.0 and end at T=23.5.

There are several ways to improve this code. The first improvement is to swap the order of ADD and SHL (then we have to add (1 SHL 3) = 8):

```
; Example 5.6b. P4 transfer data between execution units
                    ;   clock         ex.unit   subunit
and  eax, 00fh      ;  0.0 -  0.5     ALU0      LOGIC
xor  ebx, 0f0h      ;  0.5 -  1.0     ALU0      LOGIC
shl  eax, 3         ;  1.0 -  5.0     INT       MMX SHIFT
add  eax, 8         ;  6.0 -  6.5     ALU1      ADD
sub  eax, ecx       ;  6.5 -  7.0     ALU0/1    ADD
mov  edx, eax       ;  7.0 -  7.5     ALU0/1    MOV
imul edx, 100       ;  8.0 - 22.0     INT       FP MUL
or   edx, ebx       ; 22.0 - 22.5     ALU0/1    MOV
```

Here we are saving a half clock before the `SHL` and a half clock before the `IMUL` by making the data for these instructions ready at a half-clock tick so that they are available to the single-speed unit a half clock later, at an integral time. The trick is to reorder the instructions so that we have an odd number of double-speed μops between any two single-speed μops in a chain of interdependent instructions. We can improve the code further by minimizing the number of transitions between execution units. Even better, of course, is to keep all operations in the same execution unit, and preferably the double-speed units. `SHL EAX,3` can be replaced by 3 × (`ADD EAX,EAX`).

If we want to know why there is an additional delay when going from one execution unit to another, there are three possible explanations:

Explanation A

The physical distance between the execution units on the silicon chip is quite large, and this may cause a propagation delay in the traveling of electrical signals from one unit to another because of the induction and capacity in the wires.

Explanation B

The "logical distance" between execution units means that the data have to travel through various registers, buffers, ports, buses and multiplexers to get to the right destination. The designers have implemented various shortcuts to bypass these delaying elements and forward results directly to execution units that are waiting for these results. It is possible that these shortcuts connect to only execution units under the same port.

Explanation C

If 128-bit operands are handled 64 bits at a time in staggered additions, as figure 5.4 suggests, then we will have a 1 clock delay at the end of a chain of 128-bit instructions when the two halves have to be united. Consider, for example, the addition of packed double precision floating point numbers in 128-bit registers on P4. If the addition of the lower 64-bit operand starts at time T=0, it will finish at T=4. The upper 64-bit operand can start at time T=1 and finish at T=5. If the next dependent operation is also a packed addition, then the second addition can start to work on the lower 64-bit operand at time T=4, before the upper operand is ready.

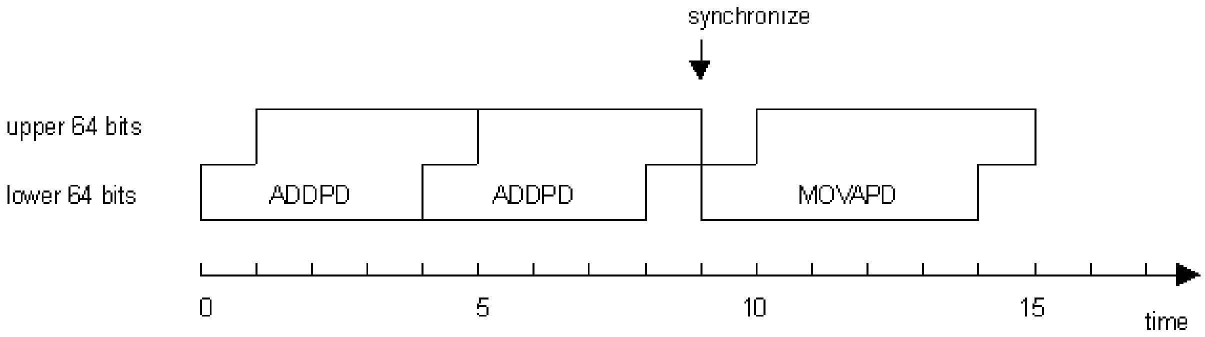

Figure 5.5

The latency for a chain of such instructions will appear to be 4 clock cycles per operation. If all operations on 128-bit registers can overlap in this way, then we will never see the 128-bit operations having higher latency than the corresponding 64-bit operations. But if the transport of the data to another execution unit requires that all 128 bits travel together, then we get an additional delay of 1 clock cycle for the synchronization of the upper and lower operands, as figure 5.5 shows. In the same way, the double-speed units ALU0 and ALU1 on P4 handle 32-bit operations as two 16-bit operations taking a half-clock cycle each. But if all 32 bits are needed together, then there is an extra delay of a half clock. It is not known whether the data buses between execution units are 32 bits, 64 bits or 128 bits wide.

5.7 Retirement

The retirement of executed µops works in the same way in the P4 and P4E as in the 6'th generation processors. This process is explained on page 80.

The retirement station can handle three µops per clock cycle. This may not seem like a problem because the throughput is already limited to 3 µops per clock in the trace cache. But the retirement station has the further limitation that taken jumps must retire in the first of the three slots in the retirement station. This sometimes limits the throughput of small loops. If the number of µops in the loop is not a multiple of 3, then the jump-back instruction in the bottom of the loop may go into the wrong retirement slot, at the penalty of one clock cycle per iteration. It is therefore recommended that the number of µops (not instructions) in small critical loops should be a multiple of 3. In some cases, you can actually save one clock cycle per iteration by adding one or two `NOP`'s to the loop to make the number of µops divisible by 3. This applies only if a throughput of 3 µops per clock cycle is expected.

5.8 Partial registers and partial flags

Registers `AL`, `AH`, and `AX` are all parts of the `EAX` register. These are called partial registers. On 6'th generation microprocessors, the partial registers could be split into separate temporary registers, so that different parts could be handled independently of each other. This caused a serious delay whenever there was a need to join different parts of a register into a single full register. This problem is explained on page 81 and 98.

The P4/P4E prevents this problem in a different way than the PM, namely by always keeping the whole register together. This solution has other drawbacks, however. The first drawback is that it introduces false dependences. Any read or write to `AL` will be delayed if a preceding write to `AH` is delayed.

Another drawback is that access to a partial register sometimes requires an extra µop. Examples:

```
; Example 5.7. Partial register access
mov  eax, [mem32]        ; 1 uop
mov  ax,  [mem16]        ; 2 uops
mov  al,  [mem8]         ; 2 uops
mov  ah,  [mem8]         ; 2 uops
add  al,  bl             ; 1 uop
add  ah,  bh             ; 1 uop
add  al,  bh             ; 2 uops
add  ah,  bl             ; 2 uops
```

For optimal performance, you may follow the following guidelines when working with 8-bit and 16-bit operands:

- Avoid using the high 8-bit registers `AH`, `BH`, `CH`, `DH`.

- When reading from an 8-bit or 16-bit memory operand, use `MOVZX` to read the entire 32-bit register, even in 16-bit mode.

- When sign-extension is needed then use `MOVSX` with the largest possible destination register, i.e. 32-bit destination in 16 or 32-bit mode, and 64-bit destination in 64-bit mode.

- Alternatively, use MMX or XMM registers to handle 8-bit and 16-bit integers, if they can be packed.

The problems with partial access also apply to the flags register when an instruction modifies some of the flags but leaves other flags unchanged.

For historical reasons, the INC and DEC instructions leave the carry flag unchanged, while the other arithmetic flags are written to. This causes a false dependence on the previous value of the flags and costs an extra µop. To avoid these problems, it is recommended that you always use ADD and SUB instead of INC and DEC. For example, INC EAX should be replaced by ADD EAX,1.

SAHF leaves the overflow flag unchanged but changes the other arithmetic flags. This causes a false dependence on the previous value of the flags, but no extra µop.

BSF and BSR change the zero flag but leave the other flags unchanged. This causes a false dependence on the previous value of the flags and costs an extra µop.

BT, BTC, BTR, and BTS change the carry flag but leave the other flags unchanged. This causes a false dependence on the previous value of the flags and costs an extra µop. Use TEST, AND, OR and XOR instead of these instructions. On P4E you can also use shift instructions efficiently. For example, BT RAX,40 / JC X can be replaced by SHR RAX,41 / JC X if the value of RAX is not needed again later.

5.9 Store forwarding stalls

The problems with accessing part of a memory operand are much bigger than when accessing part of a register. These problems are the same as for previous processors, see page 84. Example:

```
; Example 5.8a. Store forwarding stall
mov  dword ptr [mem1], eax
mov  dword ptr [mem1+4], 0
fild qword ptr [mem1]              ; Large penalty
```

You can save 10-20 clocks by changing this to:

```
; Example 5.8b. Avoid store forwarding stall
movd xmm0, eax
movq qword ptr [mem1], xmm0
fild qword ptr [mem1]              ; No penalty
```

5.10 Memory intermediates in dependency chains

The P4 has an unfortunate proclivity for trying to read a memory operand before it is ready. If you write

```
; Example 5.9. Memory intermediate in dependency chain
imul eax, 5
mov  [mem1], eax
mov  ebx, [mem1]
add  ebx, ecx
```

then the microprocessor may try to read the value of [MEM1] into EBX before the IMUL and the memory write have finished. It soon discovers that the value it has read is invalid, so it will discard EBX and try again. It will keep replaying the read instruction as well as the subsequent instructions until the data in [MEM1] are ready. There seems to be no limit to how many times it can replay a series of instructions, and this process steals resources from other processes. In a long dependency chain, this may typically cost 10 - 20 clock cycles! Using the MFENCE instruction to serialize memory access does not solve the problem because this instruction is even more costly. On other microprocessors, including P4E, the penalty for reading a memory operand immediately after writing to the same memory position is only a few clock cycles.

The best way to avoid this problem is, of course, to replace `MOV EBX,[MEM1]` with `MOV EBX,EAX` in the above example. Another possible solution is to give the processor plenty of work to do between the store and the load from the same address.

However, there are two situations where it is not possible to keep data in registers. The first situation is the transfer of parameters in high-level language procedure calls in 16-bit and 32-bit mode; the second situation is transferring data between floating point registers and other registers.

Transferring parameters to procedures

Calling a function with one integer parameter in C++ will typically look like this in 32-bit mode:

```
; Example 5.10. Memory intermediate in function call (32-bit mode)
push eax                ; Save parameter on stack
call _ff                ; Call function _ff
add esp,4               ; Clean up stack after call
...
_ff proc near           ; Function entry
push ebp                ; Save ebp
mov ebp,esp             ; Copy stack pointer
mov eax,[ebp+8]         ; Read parameter from stack
...
pop ebp                 ; Restore ebp
ret                     ; Return from function
_ff endp
```

As long as either the calling program or the called function is written in high-level language, you may have to stick to the convention of transferring parameters on the stack. Most C++ compilers can transfer 2 or 3 integer parameters in registers when the function is declared `__fastcall`. However, this method is not standardized. Different compilers use different registers for parameter transfer. To avoid the problem, you may have to keep the entire dependency chain in assembly language. The problem can be avoided in 64-bit mode where most parameters are transferred in registers.

Transferring data between floating point and other registers

There is no way to transfer data between floating point registers and other registers, except through memory. Example:

```
; Example 5.11. Memory intermediate in integer to f.p. conversion
imul  eax,    ebx
mov   [temp], eax       ; Transfer data from integer register to f.p.
fild  [temp]
fsqrt
fistp [temp]            ; Transfer data from f.p. register to integer
mov   eax,    [temp]
```

Here we have the problem of transferring data through memory twice. You may avoid the problem by keeping the entire dependency chain in floating point registers, or by using XMM registers instead of floating point registers.

Another way to prevent premature reading of the memory operand is to make the read address depend on the data. The first transfer can be done like this:

```
; Example 5.12. Avoid stall in integer to f.p. conversion
mov  [temp], eax
```

```
and  eax,    0          ; Make eax = 0, but keep dependence
fild [temp+eax]         ; Make read address depend on eax
```

The `AND EAX,0` instruction sets `EAX` to zero but keeps a false dependence on the previous value. By putting `EAX` into the address of the `FILD` instruction, we prevent it from trying to read before `EAX` is ready.

It is a little more complicated to make a similar dependence when transferring data from floating point registers to integer registers. The simplest way to solve the problem is:

```
; Example 5.13. Avoid stall in f.p. to integer conversion
fistp  [temp]
fnstsw ax                 ; Transfer status after fistp to ax
and    eax, 0             ; Set to 0
mov    eax, [temp+eax]    ; Make dependent on eax
```

Literature

A detailed study of the replay mechanism is published by Victor Kartunov, et. al.: "Replay: Unknown Features of the NetBurst Core", www.xbitlabs.com/articles/cpu/print/replay.html. See also US Patents 6,163,838; 6,094,717; 6,385,715.

5.11 Breaking dependency chains

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EBX,EBX`. The P4/P4E processor recognizes that these instructions are independent of the prior value of the register. So any instruction that uses the new value of the register will not have to wait for the value prior to the `XOR` or `SUB` instruction to be ready. The same applies to the `PXOR` instruction with a 64-bit or 128-bit register, but not to any of the following instructions: `XOR` or `SUB` with an 8-bit or 16-bit register, `SBB`, `PANDN`, `PSUB`, `XORPS`, `XORPD`, `SUBPS`, `SUBPD`, `FSUB`.

The instructions `XOR`, `SUB` and `PXOR` are useful for breaking an unnecessary dependence, but it doesn't work on e.g. the PM processor.

You may also use these instructions for breaking dependences on the flags. For example, rotate instructions have a false dependence on the flags in P4. This can be removed in the following way:

```
; Example 5.14. Break false dependence on flags
ror eax, 1
sub edx, edx    ; Remove false dependence on the flags
ror ebx, 1
```

If you don't have a spare register for this purpose, then use an instruction that doesn't change the register, but only the flags, such as `CMP` or `TEST`. You cannot use `CLC` for breaking dependences on the carry flag.

5.12 Choosing the optimal instructions

There are many possibilities for replacing less efficient instructions with more efficient ones. The most important cases are summarized below.

INC and DEC

These instructions have a problem with partial flag access, as explained on page 62. Always replace `INC EAX` with `ADD EAX,1`, etc.

8-bit and 16-bit integers

Replace `MOV AL,BYTE PTR [MEM8]` by `MOVZX EAX,BYTE PTR [MEM8]`
Replace `MOV BX,WORD PTR [MEM16]` by `MOVZX EBX,WORD PTR [MEM16]`

Avoid using the high 8-bit registers `AH`, `BH`, `CH`, `DH`.

If 8-bit or 16-bit integers can be packed and handled in parallel, then use MMX or XMM registers.

These rules apply even in 16-bit mode.

Memory stores

Most memory store instructions use 2 µops. Simple store instructions of the type `MOV [MEM],EAX` use only one µop if the memory operand has no SIB byte. A SIB byte is needed if there is more than one pointer register, if there is a scaled index register, or if `ESP` is used as base pointer. The short-form store instructions can use a general purpose register (see page 49). Examples:

```
; Example 5.15. uop counts for memory stores
mov array[ecx], eax      ; 1 uop
mov array[ecx*4], eax    ; 2 uops because of scaled index
mov [ecx+edi], eax       ; 2 uops because of two index registers
mov [ebp+8], ebx         ; 1 uop
mov [esp+8], ebx         ; 2 uops because esp used
mov es:[mem8], cl        ; 1 uop
mov es:[mem8], ch        ; 2 uops because high 8-bit register used
movq [esi], mm1          ; 2 uops because not a general purp.register
fstp [mem32]             ; 2 uops because not a general purp.register
```

The corresponding memory load instructions all use only 1 µop. A consequence of these rules is that a procedure which has many stores to local variables on the stack should use `EBP` as pointer, while a procedure which has many loads and few stores may use `ESP` as pointer, and save `EBP` for other purposes.

Shifts and rotates

Shifts and rotates on integer registers are quite slow on the P4 because the integer execution unit transfers the data to the MMX shift unit and back again. Shifts to the left may be replaced by additions. For example, `SHL EAX,3` can be replaced by 3 times `ADD EAX,EAX`. This does not apply to the P4E, where shifts are as fast as additions.

Rotates through carry (`RCL`, `RCR`) by a value different from 1 or by `CL` should be avoided.

If the code contains many integer shifts and multiplications, then it may be advantageous to execute it in MMX or XMM registers on P4.

Integer multiplication

Integer multiplication is slow on the P4 because the integer execution unit transfers the data to the FP-MUL unit and back again. If the code has many integer multiplications then it may be advantageous to handle the data in MMX or XMM registers.

Integer multiplication by a constant can be replaced by additions. Replacing a single multiply instruction by a long sequence of `ADD` instructions should, of course, only be done in critical dependency chains.

LEA

The `LEA` instruction is split into additions and shifts on the P4 and P4E. `LEA` instructions with a scale factor may preferably be replaced by additions. This applies only to the `LEA` instruction, not to any other instructions with a memory operand containing a scale factor.

In 64-bit mode, a `LEA` with a RIP-relative address is inefficient. Replace `LEA RAX,[MEM]` by `MOV RAX, OFFSET MEM`.

Register-to-register moves with FP, mmx and xmm registers

The following instructions, which copy one register into another, all have a latency of 6 clocks on P4 and 7 clocks on P4E: `MOVQ MM,MM`, `MOVDQA XMM,XMM`, `MOVAPS XMM,XMM`, `MOVAPD XMM,XMM`, `FLD ST(X)`, `FST ST(X)`, `FSTP ST(X)`. These instructions have no additional latency. A possible reason for the long latency of these instructions is that they use the same execution unit as memory stores (port 0, MOV).

There are several ways to avoid this delay:

- The need for copying a register can sometimes be eliminated by using the same register repeatedly as source, rather than destination, for other instructions.

- With floating point registers, the need for moving data from one register to another can often be eliminated by using `FXCH`. The `FXCH` instruction has no latency.

- If the value of a register needs to be copied, then use the old copy in the most critical dependence path, and the new copy in a less critical path. The following example calculates $Y = (a+b)^{2.5}$:

```
; Example 5.16. Optimize register-to-register moves
fld [a]
fadd [b]      ; a+b
fld st        ; Copy a+b
fxch          ; Get old copy
fsqrt         ; (a+b)0.5
fxch          ; Get new (delayed) copy
fmul st,st    ; (a+b)2
fmul          ; (a+b)2.5
fstp [y]
```

  The old copy is used for the slow square root, while the new copy, which is available 6-7 clocks later, is used for the multiplication.

If none of these methods solve the problem, and latency is more important than throughput, then use faster alternatives:

- For 80-bit floating point registers:

```
fld st(0)          ; copy register
```

  can be replaced by

```
fldz               ; make an empty register
xor eax, eax       ; set zero flag
fcmovz st, st(1)   ; conditional move
```

- For 64-bit MMX registers:

```
movq mm1, mm0
```

can be replaced by the shuffle instruction

```
pshufw mm1, mm0, 11100100B
```

- For 128-bit XMM registers:

```
movdqa xmm1, xmm0
```

can be replaced by the shuffle instruction

```
pshufd xmm1, xmm0, 11100100B
```

or even faster:

```
pxor xmm1, xmm1     ; Set new register to 0
por  xmm1, xmm0     ; OR with desired value
```

These methods all have lower latencies than the register-to-register moves. However, a drawback of these tricks is that they use port 1 which is also used for all calculations on these registers. If port 1 is saturated, then it may be better to use the slow moves, which go to port 0.

5.13 Bottlenecks in P4 and P4E

It is important, when optimizing a piece of code, to find the limiting factor that controls execution speed. Tuning the wrong factor is unlikely to have any beneficial effect. In the following paragraphs, I will explain each of the possible limiting factors. You have to consider each factor in order to determine which one is the narrowest bottleneck, and then concentrate your optimization effort on that factor until it is no longer the narrowest bottleneck.

Memory access

If the program is accessing large amounts of data, or if the data are scattered around everywhere in the memory, then we will have many data cache misses. Accessing uncached data is so time consuming that all other optimization considerations are unimportant. The caches are organized as aligned lines of 64 bytes each. If one byte within an aligned 64-bytes block has been accessed, then we can be certain that all 64 bytes will be loaded into the level-1 data cache and can be accessed at no extra cost. To improve caching, it is recommended that data that are used in the same part of the program be stored together. You may align large arrays and structures by 64. Store local variables on the stack if you don't have enough registers.

The level-1 data cache is only 8 kb on the P4 and 16 kb on P4E. This may not be enough to hold all the data, but the level-2 cache is more efficient on the P4/P4E than on previous processors. Fetching data from the level-2 cache will cost only a few clock cycles extra.

Data that are unlikely to be cached may be prefetched before they are used. If memory addresses are accessed consecutively, then they will be prefetched automatically. You should therefore preferably organize the data in a linear fashion so that they can be accessed consecutively, and access no more than four large arrays, preferably less, in the critical part of the program.

The `PREFETCH` instructions can improve performance in situations where you access uncached data and cannot rely on automatic prefetching. However, excessive use of the `PREFETCH` instructions can slow down program throughput on P4. If you are in doubt whether a `PREFETCH` instruction will benefit the program, then you may simply load the data needed into a spare register rather than using a `PREFETCH` instruction. If you have no spare register then use an instruction which reads the memory operand without changing any register, such as `CMP` or `TEST`. As the stack pointer is unlikely to be part of any critical dependency chain, a useful way to prefetch data is `CMP ESP,[MEM]`, which will change only the flags.

When writing to a memory location that is unlikely to be accessed again soon, you may use the non-temporal write instructions `MOVNTI`, etc., but excessive use of non-temporal moves will slow down performance on P4.

Further guidelines regarding memory access can be found in "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual".

Execution latency

The executing time for a dependency chain can be calculated from the latencies listed in manual 4: "Instruction tables". Many instructions have an additional latency of 1 clock cycle when the subsequent instruction goes to a different execution unit. See page 58 for further explanation.

If long dependency chains limit the performance of the program then you may improve performance by choosing instructions with low latency, minimizing the number of transitions between execution units, breaking up dependency chains, and utilizing all opportunities for calculating subexpressions in parallel.

Always avoid memory intermediates in dependency chains, as explained on page 62.

Execution unit throughput

If your dependency chains are short, or if you are working on several dependency chains in parallel, then the program is most likely limited by throughput rather than latency. Different execution units have different throughputs. ALU0 and ALU1, which handle simple integer instructions and other common μops, both have a throughput of 2 instructions per clock cycle. Most other execution units have a throughput of one instruction per clock cycle. When working with 128-bit registers, the throughput is usually one instruction per two clock cycles. Division and square roots have the lowest throughputs. Each throughput measure applies to all μops executing in the same execution subunit (see page 55).

If execution throughput limits the code then try to move some calculations to other execution subunits.

Port throughput

Each of the execution ports can receive one μop per clock cycle. Port 0 and port 1 can receive an additional μop at each half-clock tick if these μops go to the double-speed units ALU0 and ALU1. If all μops in the critical part of the code go to the single-speed units under port 1, then the throughput will be limited to 1 μop per clock cycle. If the μops are optimally distributed between the four ports, then the throughput may be as high as 6 μops per clock cycle. Such a high throughput can only be achieved in short bursts, because the trace cache and the retirement station limit the average throughput to less than 3 μops per clock cycle.

If port throughput limits the code then try to move some μops to other ports. For example, `MOV REGISTER,IMMEDIATE` can be replaced by `MOV REGISTER,MEMORY`.

Trace cache delivery

The trace cache can deliver a maximum of approx. 3 μops per clock cycle. On P4, some μops require more than one trace cache entry, as explained on page 48. The delivery rate can be less than 3 µops per clock cycle for code that contains many branches and for tiny loops with branches inside (see page 51).

If none of the abovementioned factors limit program performance, then you may aim at a throughput of approx. 3 µops per clock cycle.

Choose the instructions that generate the smallest number of µops. Avoid µops that require more than one trace cache entry on P4 (see page 48).

Trace cache size

The trace cache can hold less code than a traditional code cache using the same amount of physical chip space. The limited size of the trace cache can be a serious bottleneck if the critical part of the program doesn't fit into the trace cache.

µop retirement

The retirement station can handle 3 µops per clock cycle. Taken branches can only be handled by the first of the three slots in the retirement station.

If you aim at an average throughput of 3 µops per clock cycle then avoid an excessive number of jumps, calls and branches. Small critical loops should preferably have a number of µops divisible by 3 (see page 61).

Instruction decoding

If the critical part of the code doesn't fit into the trace cache, then the limiting stage may be instruction decoding. The decoder can handle one instruction per clock cycle, provided that the instruction generates no more than 4 µops and no microcode, and does not have an excessive number of prefixes (see page 52). If decoding is a bottleneck, then you may try to minimize the number of instructions rather than the number of µops.

Branch prediction

The calculations of latencies and throughputs are only valid if all branches are predicted. Branch mispredictions can seriously slow down performance when latency or throughput is the limiting factor. The inability of the P4 to cancel bogus µops after a misprediction can seriously degrade performance.

Avoid poorly predictable branches in critical parts of the code unless the alternative (e.g. conditional moves) outweighs the advantage by adding costly extra dependences and latency. See page 23 for details.

Replaying of µops

The P4 often wastes an excessive amount of resources on replaying bogus µops after cache misses, failed store-to-load forwarding, etc. This can result in a serious degradation of performance, especially when there are memory intermediates in long dependency chains.

6 Pentium Pro, II and III pipeline

6.1 The pipeline in PPro, P2 and P3

The Pentium Pro from 1995 was the first Intel processor with out-of-order execution. The microarchitecture design was quite successful. This design has been further developed through many generations to the processors that we have today - with a little detour to the less successful Pentium 4 or Netburst architecture.

The pipeline of the PPro, P2 and P3 microprocessors is explained in various manuals and tutorials from Intel, which unfortunately are no longer available. I will therefore explain the pipeline here.

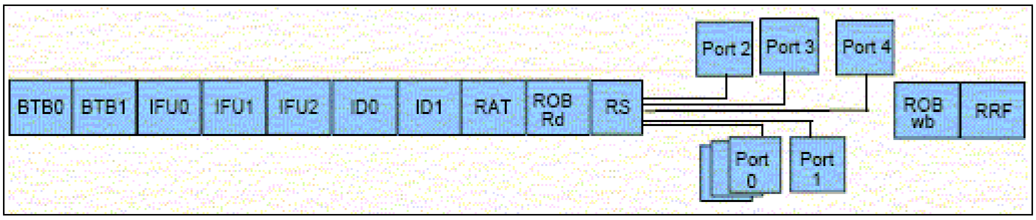

Figure 6.1. Pentium Pro pipeline.

The pipeline is illustrated in fig. 6.1. The pipeline stages are as follows:

BTB0,1:	Branch prediction. Tells where to fetch the next instructions from.
IFU0,1,2:	Instruction fetch unit.
ID0,1:	Instruction decoder.
RAT:	Register alias table. Register renaming.
ROB Rd:	µop re-ordering buffer read.
RS:	Reservation station.
Port0,1,2,3,4:	Ports connecting to execution units.
ROB wb:	Write-back of results to re-order buffer.
RRF:	Register retirement file.

Each stage in the pipeline takes at least one clock cycle. The branch prediction has been explained on p. 21. The other stages in the pipeline will be explained below (Literature: Intel Architecture Optimization Manual, 1997).

6.2 Instruction fetch

Instruction codes are fetched from the code cache in aligned 16-byte chunks into a double buffer that can hold two 16-byte chunks. The purpose of the double buffer is to make it possible to decode an instruction that crosses a 16-byte boundary (i.e. an address divisible by 16). The code is passed on from the double buffer to the decoders in blocks which I will call IFETCH blocks (instruction fetch blocks). The IFETCH blocks are up to 16 bytes long. In most cases, the instruction fetch unit makes each IFETCH block start at an instruction boundary rather than a 16-byte boundary. However, the instruction fetch unit needs information from the instruction length decoder in order to know where the instruction boundaries are. If this information is not available in time then it may start an IFETCH block at a 16-byte boundary. This complication will be discussed in more detail below.

The double buffer is not big enough for handling fetches around jumps without delay. If the IFETCH block that contains the jump instruction crosses a 16-byte boundary, then the double buffer needs to keep two consecutive aligned 16-bytes chunks of code in order to generate it. If the first instruction after the jump crosses a 16-byte boundary, then the double buffer needs to load two new 16-bytes chunks of code before a valid IFETCH block can be generated. This means that, in the worst case, the decoding of the first instruction after a jump can be delayed for two clock cycles. There is a one clock penalty for a 16-byte boundary in the IFETCH block containing the jump instruction, and also a one clock penalty for a 16-byte boundary in the first instruction after the jump. The instruction fetch unit can fetch one 16-byte chunk per clock cycle. If it takes more than one clock cycle to decode an IFETCH block then it is possible to use this extra time for fetching ahead. This can compensate for the penalties of 16-byte boundaries before and after jumps. The resulting delays are summarized in table 6.1 below.

If the double buffer has time to fetch only one 16-byte chunk of code after the jump, then the first IFETCH block after the jump will be identical to this chunk, that is, aligned to a 16-byte boundary. In other words, the first IFETCH block after the jump will not begin at the first instruction, but at the nearest preceding address divisible by 16. If the double buffer has had time to load two 16-byte chunks, then the new IFETCH block can cross a 16-byte boundary and begin at the first instruction after the jump. These rules are summarized in the following table:

Number of decode groups in IFETCH block containing jump	16-byte boundary in this IFETCH block	16-byte boundary in first instruction after jump	decoder delay	alignment of first IFETCH after jump
1	0	0	0	by 16
1	0	1	1	to instruction
1	1	0	1	by 16
1	1	1	2	to instruction
2	0	0	0	to instruction
2	0	1	0	to instruction
2	1	0	0	by 16
2	1	1	1	to instruction
3 or more	0	0	0	to instruction
3 or more	0	1	0	to instruction
3 or more	1	0	0	to instruction
3 or more	1	1	0	to instruction

Table 6.1. Instruction fetching around jumps

The first column in this table indicates the time it takes to decode all instructions in an IFETCH block (Decode groups are explained below).

Instructions can have any length from 1 to 15 bytes. Therefore, we cannot be sure that a 16-byte IFETCH block contains a whole number of instructions. If an instruction extends past the end of an IFETCH block then it will go into the next IFETCH block, which will begin at the first byte of this instruction. Therefore, the instruction fetch unit needs to know where the last full instruction in each IFETCH block ends before it can generate the next IFETCH block. This information is generated by the instruction length decoder, which is in stage IFU2 in the pipeline (fig. 6.1). The instruction length decoder can determine the lengths of three instructions per clock cycle. If, for example, an IFETCH block contains ten instructions then it will take three clock cycles before it is known where the last full instruction in the IFETCH block ends and before the next IFETCH block can be generated.

6.3 Instruction decoding

Instruction length decoding

The IFETCH blocks go to the instruction length decoder, which determines where each instruction begins and ends. This is a very critical stage in the pipeline because it limits the degree of parallelism that can be achieved. We want to fetch more than one instruction per clock cycle, decode more than one instruction per clock cycle, and execute more than one µop per clock cycle in order to gain speed. But decoding instructions in parallel is difficult when instructions have different lengths. You need to decode the first instruction in order to know how long it is and where the second instruction begins before you can start to decode the second instruction. So a simple instruction length decoder would only be able to handle one instruction per clock cycle. The instruction length decoder in the PPro microarchitecture can determine the lengths of three instructions per clock cycle and even feed back this information to the instruction fetch unit early enough for a new IFETCH block to be generated for the instruction length decoder to work on in the next clock cycle. This is quite an impressive accomplishment, which I believe is achieved by tentatively decoding all 16 possible start addresses in parallel.

The 4-1-1 rule

After the instruction length decoder comes the instruction decoders which translate instructions into µops. There are three decoders working in parallel so that up to three instructions can be decoded in every clock cycle. A group of up to three instructions that are decoded in the same clock cycle is called a decode group. The three decoders are called D0, D1, and D2. D0 can handle all instructions and can generate up to 4 µops per clock cycle. D1 and D2 can only handle simple instructions that generate no more than one µop each and are no more than 8 bytes long. The first instruction in an IFETCH block always goes to D0. The next two instructions go to D1 and D2 if possible. If an instruction that would go into D1 or D2 cannot be handled by these decoders because it generates more than one µop or because it is more than 8 bytes long, then it has to wait until D0 is vacant. The subsequent instructions are delayed as well. Example:

```
; Example 6.1a. Instruction decoding
mov  [esi], eax    ; 2 uops, D0
add  ebx, [edi]    ; 2 uops, D0
sub  eax, 1        ; 1 uop,  D1
cmp  ebx, ecx      ; 1 uop,  D2
je   L1            ; 1 uop,  D0
```

The first instruction in this example goes to decoder D0. The second instruction cannot go to D1 because it generates more than one µop. It is therefore delayed to the next clock cycle when D0 is ready. The third instruction goes to D1 because the preceding instruction goes to D0. The fourth instruction goes to D2. The last instruction goes to D0. The whole sequence takes three clock cycles to decode. The decoding can be improved by swapping the second and third instructions:

```
; Example 6.1b. Instructions reordered for improved decoding
mov  [esi], eax    ; 2 uops, D0
sub  eax, 1        ; 1 uop,  D1
add  ebx, [edi]    ; 2 uops, D0
cmp  ebx, ecx      ; 1 uop,  D1
je   L1            ; 1 uop,  D2
```

Now the decoding takes only two clock cycles because of a better distribution of instructions between the decoders.

The maximum decoding speed is obtained when instructions are ordered according to the 4-1-1 pattern: If every third instruction generates 4 µops and the next two instructions generate 1 µop each then the decoders can generate 6 µops per clock cycle. A 2-2-2 pattern gives the minimum decoding speed of 2 µops per clock because all the 2-µop instructions go to D0. It is recommended that you order instructions according to the 4-1-1 rule so that every instruction that generates 2, 3 or 4 µops is followed by two instructions that generate 1 µop each. An instruction that generates more than 4 µops must go into D0. It takes two or more clock cycles to decode, and no other instructions can decode in parallel.

IFETCH block boundaries

A further complication is that the first instruction in an IFETCH block always goes into D0. If the code has been scheduled according to the 4-1-1 rule and if one of the 1-µop instructions that was intended for D1 or D2 happens to be first in an IFETCH block, then that instruction goes into D0 and the 4-1-1 pattern is broken. This will delay the coding for one clock cycle. The instruction fetch unit cannot adjust the IFETCH boundaries to the 4-1-1 pattern because the information about which instruction generates more than 1 µop is only available, I suppose, two stages further down the pipeline.

This problem is difficult to handle because it is difficult to guess where the IFETCH boundaries are. The best way to address this problem is to schedule the code so that the decoders can generate more than 3 µops per clock cycle. The RAT and RRF stages in the pipeline (fig. 6.1) can handle no more than 3 µops per clock. If instructions are ordered according to the 4-1-1 rule so that we can expect at least 4 µops per clock cycle then maybe we can afford to lose one clock cycle at every IFETCH boundary and still maintain an average decoder throughput of no less than 3 µops per clock.

Another remedy is to make instructions as short as possible in order to get more instructions into each IFETCH block. More instructions per IFETCH block means fewer IFETCH boundaries and thus fewer breaks in the 4-1-1 pattern. For example, you may use pointers instead of absolute addresses to reduce code size. See manual 2: "Optimizing subroutines in assembly language" for more advices on how to reduce the size of instructions.

In some cases it is possible to manipulate the code so that instructions intended for decoder D0 fall at the IFETCH boundaries. But it is usually quite difficult to determine where the IFECTH boundaries are and probably not worth the effort. First you need to make the code segment paragraph-aligned in order to know where the 16-byte boundaries are. Then you have to know where the first IFETCH block of the code you want to optimize begins. Look at the output listing from the assembler to see how long each instruction is. If you know where one IFETCH block begins then you can find where the next IFETCH block begins in the following way: Make the IFETCH block 16 bytes long. If it ends at an instruction boundary then the next block will begin here. If it ends with an unfinished instruction then the next block will begin at the beginning of this instruction. Only the lengths of the instructions count here, it doesn't matter how many µops they generate or what they do. This way you can work your way all through the code and mark where each IFETCH block begins. The biggest problem is to know where to start. Here are some guidelines:

- The first IFETCH block after a jump, call, or return can begin either at the first instruction or at the nearest preceding 16-byte boundary, according to table 6.1. If you align the first instruction to begin at a 16-byte boundary then you can be sure that the first IFETCH block begins here. You may want to align important subroutine entries and loop entries by 16 for this purpose.

- If the combined length of two consecutive instructions is more than 16 bytes then you can be certain that the second one doesn't fit into the same IFETCH block as the first one, and consequently you will always have an IFETCH block beginning at the second instruction. You can use this as a starting point for finding where subsequent IFETCH blocks begin.

- The first IFETCH block after a branch misprediction begins at a 16-byte boundary. As explained on page 21, a loop that repeats more than 5 times will always have a misprediction when it exits. The first IFETCH block after such a loop will therefore begin at the nearest preceding 16-byte boundary.

I am sure you want an example now:

```
; Example 6.2. Instruction fetch blocks
```

```
address      instruction            length    uops  expected decoder
-------------------------------------------------------------------
1000h        mov ecx, 1000            5        1        D0
1005h   LL:  mov [esi], eax           2        2        D0
1007h        mov [mem], 0            10        2        D0
1011h        lea ebx, [eax+200]       6        1        D1
1017h        mov byte ptr [esi], 0    3        2        D0
101Ah        bsr edx, eax             3        2        D0
101Dh        mov byte ptr [esi+1],0   4        2        D0
1021h        dec edx                  1        1        D1
1022h        jnz LL                   2        1        D2
```

Let's assume that the first IFETCH block begins at address 1000h and ends at 1010h. This is before the end of the `MOV [MEM],0` instruction so the next IFETCH block will begin at 1007h and end at 1017h. This is at an instruction boundary so the third IFETCH block will begin at 1017h and cover the rest of the loop. The number of clock cycles it takes to decode this is the number of D0 instructions, which is 5 per iteration of the `LL` loop. The last IFETCH block contained three decode blocks covering the last five instructions, and it has one 16-byte boundary (1020h). Looking at table 6.1 above we find that the first IFETCH block after the jump will begin at the first instruction after the jump, that is the `LL` label at 1005h, and end at 1015h. This is before the end of the `LEA` instruction, so the next IFETCH block will go from 1011h to 1021h, and the last one from 1021h covering the rest. Now the `LEA` instruction and the `DEC` instruction both fall at the beginning of an IFETCH block which forces them to go into D0. We now have 7 instructions in D0 and the loop takes 7 clocks to decode in the second iteration. The last IFETCH block contains only one decode group (`DEC ECX` / `JNZ LL`) and has no 16-byte boundary. According to table 6.1, the next IFETCH block after the jump will begin at a 16-byte boundary, which is 1000h. This will give us the same situation as in the first iteration, and you will see that the loop takes alternately 5 and 7 clock cycles to decode. Since there are no other bottlenecks, the complete loop will take 6000 clocks to run 1000 iterations. If the starting address had been different so that you had a 16-byte boundary in the first or the last instruction of the loop, then it would take 8000 clocks. If you reorder the loop so that no D1 or D2 instructions fall at the beginning of an IFETCH block then you can make it take only 5000 clocks.

The example above was deliberately constructed so that fetch and decoding is the only bottleneck. One thing that can be done to improve decoding is to change the starting address of the procedure in order to avoid 16-byte boundaries where you don't want them. Remember to make the code segment paragraph aligned so that you know where the boundaries are. It may be possible to manipulate instruction lengths in order to put IFETCH boundaries where you want them, as explained in the chapter "Making instructions longer for the sake of alignment" in manual 2: "Optimizing subroutines in assembly language".

Instruction prefixes

Instruction prefixes can also incur penalties in the decoders. Instructions can have several kinds of prefixes, as listed in manual 2: "Optimizing subroutines in assembly language".

1. An operand size prefix gives a penalty of a few clocks if the instruction has an immediate operand of 16 or 32 bits because the length of the operand is changed by the prefix. Examples (32-bit mode):

```
; Example 6.3a. Decoding instructions with operand size prefix
add bx, 9       ; No penalty because immediate operand is 8 bits signed
add bx, 200     ; Penalty for 16 bit immediate. Change to ADD EBX, 200
mov word ptr [mem16], 9   ; Penalty because operand is 16 bits
```

The last instruction may be changed to:

```
; Example 6.3b. Decoding instructions with operand size prefix
mov eax, 9
```

```
mov word ptr [mem16], ax  ; No penalty because no immediate
```

2. An address size prefix gives a penalty whenever there is an explicit memory operand (even when there is no displacement) because the interpretation of the r/m bits in the instruction code is changed by the prefix. Instructions with only implicit memory operands, such as string instructions, have no penalty with address size prefix.

3. Segment prefixes give no penalty in the decoders.

4. Repeat prefixes and lock prefixes give no penalty in the decoders.

5. There is always a penalty if an instruction has more than one prefix. This penalty is usually one clock per extra prefix.

6.4 Register renaming

Register renaming is controlled by the register alias table (RAT) shown in figure 6.1. The µops from the decoders go to the RAT via a queue, and then to the ROB and the reservation station. The RAT can handle 3 µops per clock cycle. This means that the overall throughput of the microprocessor can never exceed 3 µops per clock cycle on average.

There is no practical limit to the number of renamings. The RAT can rename three registers per clock cycle, and it can even rename the same register three times in one clock cycle.

This stage also calculates IP-relative branches and send them to the BTB0 stage.

6.5 ROB read

After the RAT comes the ROB-read stage where the values of the renamed registers are stored in the ROB entry, if they are available. Each ROB entry can have up to two input registers and two output registers. There are three possibilities for the value of an input register:

1. The register has not been modified recently. The ROB-read stage reads the value from the permanent register file and stores it in the ROB entry.

2. The value has been modified recently. The new value is the output of a µop that has been executed but not yet retired. I assume that the ROB-read stage will read the value from the not-yet-retired ROB entry and store it in the new ROB entry.

3. The value is not ready yet. The needed value is the coming output of a µop that is queued but not yet executed. The new value cannot be written yet, but it will be written to the new ROB entry by the execution unit as soon as it is ready.

Case 1 appears to be the least problematic situation. But quite surprisingly, this is the only situation that can cause delays in the ROB-read stage. The reason is that the permanent register file has only two read ports. The ROB-read stage can receive up to three µops from the RAT in one clock cycle, and each µop can have two input registers. This gives a total of up to six input registers. If these six registers are all different and all stored in the permanent register file, then it will take three clock cycles to perform the six reads through the two read ports of the register file. The preceding RAT stage will be stalled until the ROB-read is ready again. The decoders and instruction fetch will also be stalled if the queue between the decoders and the RAT is full. This queue has only approximately ten entries so it will quickly be full.

The limitation of permanent register reads applies to all registers used by an instruction except those registers that the instruction writes to only. Example:

```
; Example 6.4a. Register read stall
mov [edi + esi], eax
mov ebx, [esp + ebp]
```

The first instruction generates two µops: one that reads EAX and one that reads EDI and ESI. The second instruction generates one µop that reads ESP and EBP. EBX does not count as a read because it is only written to by the instruction. Let's assume that these three µops go through the RAT together. I will use the word triplet for a group of three consecutive µops that go through the RAT together. Since the ROB can handle only two permanent register reads per clock cycle and we need five register reads, our triplet will be delayed for two extra clock cycles before it comes to the reservation station (RS). With 3 or 4 register reads in the triplet it would be delayed by one clock cycle. The same register can be read more than once in the same triplet without adding to the count. If the instructions above are changed to:

```
; Example 6.4b. No register read stall
mov [edi + esi], edi
mov ebx, [edi + edi]
```

then we will need only two register reads (EDI and ESI) and the triplet will not be delayed.

Case 2 and 3 do not cause register read stalls. The ROB-read can read a register without stall if it has not yet been through the ROB-writeback stage. It takes at least three clock cycles to get from RAT to ROB-writeback, so you can be certain that a register written to in one µop-triplet can be read without delay in at least the next three triplets. If the write-back is delayed by reordering, slow instructions, dependency chains, cache misses, or by any other kind of stall, then the register can be read without delay further down the instruction stream. Example:

```
; Example 6.5. Register read stall
mov eax, ebx
sub ecx, eax
inc ebx
mov edx, [eax]
add esi, ebx
add edi, ecx
```

These 6 instructions generate 1 µop each. Let's assume that the first 3 µops go through the RAT together. These 3 µops read register EBX, ECX, and EAX. But since we are writing to EAX before reading it, the read is free and we get no stall. The next three µops read EAX, ESI, EBX, EDI, and ECX. Since both EAX, EBX and ECX have been modified in the preceding triplet and not yet written back then they can be read for free, so that only ESI and EDI count, and we get no stall in the second triplet either. If the SUB ECX,EAX instruction in the first triplet is changed to CMP ECX,EAX then ECX is not written to, and we will get a stall in the second triplet for reading ESI, EDI and ECX. Similarly, if the INC EBX instruction in the first triplet is changed to NOP or something else then we will get a stall in the second triplet for reading ESI, EBX and EDI.

To count the number of register reads, you have to include all registers that are read by the instruction. This includes integer registers, the flags register, the stack pointer, floating point registers and MMX registers. An XMM register counts as two registers, except when only part of it is used, as e.g. in ADDSS and MOVHLPS. Segment registers and the instruction pointer do not count. For example, in SETZ AL you count the flags register but not AL. ADD EBX,ECX counts both EBX and ECX, but not the flags because they are written to only. PUSH EAX reads EAX and the stack pointer and then writes to the stack pointer.

The `FXCH` instruction is a special case. It works by renaming, but doesn't read any values so that it doesn't count in the rules for register read stalls. An `FXCH` instruction generates a µop that neither reads nor writes any registers with regard to the rules for register read stalls.

Don't confuse µop triplets with decode groups. A decode group can generate from 1 to 6 µops, and even if the decode group has three instructions and generates three µops there is no guarantee that the three µops will go into the RAT together.

The queue between the decoders and the RAT is so short (10 µops) that you cannot assume that register read stalls do not stall the decoders or that fluctuations in decoder throughput do not stall the RAT.

It is very difficult to predict which µops go through the RAT together unless the queue is empty, and for optimized code the queue should be empty only after mispredicted branches. Several µops generated by the same instruction do not necessarily go through the RAT together; the µops are simply taken consecutively from the queue, three at a time. The sequence is not broken by a predicted jump: µops before and after the jump can go through the RAT together. Only a mispredicted jump will discard the queue and start over again so that the next three µops are sure to go into the RAT together.

A register read stall can be detected by performance monitor counter number 0A2H, which unfortunately cannot distinguish it from other kinds of resource stalls.

If three consecutive µops read more than two different registers then you would of course prefer that they do not go through the RAT together. The probability that they do is one third. The penalty of reading three or four written-back registers in one triplet of µops is one clock cycle. You can think of the one clock delay as equivalent to the load of three more µops through the RAT. With the probability of 1/3 of the three µops going into the RAT together, the average penalty will be the equivalent of 3/3 = 1 µop. To calculate the average time it will take for a piece of code to go through the RAT, add the number of potential register read stalls to the number of µops and divide by three. You can see that it doesn't pay to remove the stall by putting in an extra instruction unless you know for sure which µops go into the RAT together or you can prevent more than one potential register read stall by one extra instruction that writes to a critical register.

In situations where you aim at a throughput of 3 µops per clock, the limit of two permanent register reads per clock cycle may be a problematic bottleneck to handle. Possible ways to remove register read stalls are:

- Keep µops that read the same register close together so that they are likely to go into the same triplet.

- Keep µops that read different registers spaced so that they cannot go into the same triplet.

- Place µops that read a register no more than 9-12 µops after an instruction that writes to or modifies this register to make sure it hasn't been written back before it is read (it doesn't matter if you have a jump between as long as it is predicted). If you have reason to expect the register write to be delayed for whatever reason then you can safely read the register somewhat further down the instruction stream.

- Use absolute addresses instead of pointers in order to reduce the number of register reads.

- You may rename a register in a triplet where it doesn't cause a stall in order to prevent a read stall for this register in one or more later triplets. This method costs an extra µop and therefore doesn't pay unless the expected average number of read stalls prevented is more than 1/3.

For instructions that generate more than one µop, you may want to know the order of the µops generated by the instruction in order to make a precise analysis of the possibility of register read stalls. I have therefore listed the most common cases below.

Writes to memory:
A memory write generates two µops. The first one (to port 4) is a store operation, reading the register to store. The second µop (port 3) calculates the memory address, reading any pointer registers. Example:

```
; Example 6.6. Register reads
fstp qword ptr [ebx+8*ecx]
```

The first µop reads `ST(0)`, the second µop reads `EBX` and `ECX`.

Read and modify
An instruction that reads a memory operand and modifies a register by some arithmetic or logic operation generates two µops. The first one (port 2) is a memory load instruction reading any pointer registers, the second µop is an arithmetic instruction (port 0 or 1) reading and writing to the destination register and possibly writing to the flags. Example:

```
; Example 6.7. Register reads
add eax, [esi+20]
```

The first µop reads `ESI`, the second µop reads `EAX` and writes `EAX` and flags.

Read / modify / write
A read / modify / write instruction generates four µops. The first µop (port 2) reads any pointer registers, the second µop (port 0 or 1) reads and writes to any source register and possibly writes to the flags, the third µop (port 4) reads only the temporary result that doesn't count here, the fourth µop (port 3) reads any pointer registers again. Since the first and the fourth µop cannot go into the RAT together, you cannot take advantage of the fact that they read the same pointer registers. Example:

```
; Example 6.8. Register reads
or [esi+edi], eax
```

The first µop reads `ESI` and `EDI`, the second µop reads `EAX` and writes `EAX` and the flags, the third µop reads only the temporary result, the fourth µop reads `ESI` and `EDI` again. No matter how these µops go into the RAT you can be sure that the µop that reads `EAX` goes together with one of the µops that read `ESI` and `EDI`. A register read stall is therefore inevitable for this instruction unless one of the registers has been modified recently, for example by `MOV ESI,ESI`.

Push register
A push register instruction generates 3 µops. The first one (port 4) is a store instruction, reading the register. The second µop (port 3) generates the address, reading the stack pointer. The third µop (port 0 or 1) subtracts the word size from the stack pointer, reading and modifying the stack pointer.

Pop register
A pop register instruction generates 2 µops. The first µop (port 2) loads the value, reading the stack pointer and writing to the register. The second µop (port 0 or 1) adjusts the stack pointer, reading and modifying the stack pointer.

Call
A near call generates 4 µops (port 1, 4, 3, 01). The first two µops read only the instruction pointer which doesn't count because it cannot be renamed. The third µop reads the stack pointer. The last µop reads and modifies the stack pointer.

Return
A near return generates 4 µops (port 2, 01, 01, 1). The first µop reads the stack pointer. The third µop reads and modifies the stack pointer.

6.6 Out of order execution

The reorder buffer (ROB) can hold 40 µops and 40 temporary registers (fig. 6.1), while the reservation station (RS) can hold 20 µops. Each µop waits in the ROB until all its operands are ready and there is a vacant execution unit for it. This makes out-of-order execution possible.

Writes to memory cannot execute out of order relative to other writes. There are four write buffers, so if you expect many cache misses on writes or you are writing to uncached memory then it is recommended that you schedule four writes at a time and make sure the processor has something else to do before you give it the next four writes. Memory reads and other instructions can execute out of order, except `IN`, `OUT` and serializing instructions.

If the code writes to a memory address and soon after reads from the same address, then the read may by mistake be executed before the write because the ROB doesn't know the memory addresses at the time of reordering. This error is detected when the write address is calculated, and then the read operation (which was executed speculatively) has to be re-done. The penalty for this is approximately 3 clocks. The best way to avoid this penalty is to make sure the execution unit has other things to do between a write and a subsequent read from the same memory address.

There are several execution units clustered around five ports. Port 0 and 1 are for arithmetic operations etc. Simple move, arithmetic and logic operations can go to either port 0 or 1, whichever is vacant first. Port 0 also handles multiplication, division, integer shifts and rotates, and floating point operations. Port 1 also handles jumps and some MMX and XMM operations. Port 2 handles all reads from memory and a few string and XMM operations, port 3 calculates addresses for memory write, and port 4 executes all memory write operations. A complete list of the µops generated by code instructions with an indication of which ports they go to is contained in manual 4: "Instruction tables". Note that all memory write operations require two µops, one for port 3 and one for port 4, while memory read operations use only one µop (port 2).

In most cases, each port can receive one new µop per clock cycle. This means that we can execute up to 5 µops in the same clock cycle if they go to five different ports, but since there is a limit of 3 µops per clock earlier in the pipeline you will never execute more than 3 µops per clock on average.

You must make sure that no execution port receives more than one third of the µops if you want to maintain a throughput of 3 µops per clock. Use the table of µops in manual 4: "Instruction tables" and count how many µops go to each port. If port 0 and 1 are saturated while port 2 is free then you can improve your code by replacing some `MOV register,register` or `MOV register,immediate` instructions with `MOV register,memory` in order to move some of the load from port 0 and 1 to port 2.

Most µops take only one clock cycle to execute, but multiplications, divisions, and many floating point operations take more. Floating point addition and subtraction takes 3 clocks, but the execution unit is fully pipelined so that it can receive a new `FADD` or `FSUB` in every clock cycle before the preceding ones are finished (provided, of course, that they are independent).

Integer multiplication takes 4 clocks, floating point multiplication 5, and MMX multiplication 3 clocks. Integer and MMX multiplication is pipelined so that it can receive a new instruction every clock cycle. Floating point multiplication is partially pipelined: The execution unit can receive a new `FMUL` instruction two clocks after the preceding one, so that the maximum throughput is one `FMUL` per two clock cycles. The holes between the `FMUL`'s cannot be filled by integer multiplications because they use the same execution unit. XMM additions and multiplications take 3 and 4 clocks respectively, and are fully pipelined. But since each logical XMM register is implemented as two physical 64-bit registers, you need two µops for a packed XMM operation, and the throughput will then be one arithmetic XMM instruction every two clock cycles. XMM add and multiply instructions can execute in parallel because they don't use the same execution port.

Integer and floating point division takes up to 39 clocks and is not pipelined. This means that the execution unit cannot begin a new division until the previous division is finished. The same applies to square root and transcendental functions.

You should, of course, avoid instructions that generate many µops. The `LOOP XX` instruction, for example, should be replaced by `DEC ECX` / `JNZ XX`.

If you have consecutive `POP` instructions then you may break them up to reduce the number of µops:

```
; Example 6.9a. Split up pop instructions
pop  ecx
pop  ebx
pop  eax
```

Can be changed to:

```
; Example 6.9b. Split up pop instructions
mov  ecx, [esp]
mov  ebx, [esp+4]
mov  eax, [esp+8]
add  esp, 12
```

The former code generates 6 µops, the latter generates only 4 and decodes faster. Doing the same with `PUSH` instructions is less advantageous because the split-up code is likely to generate register read stalls unless you have other instructions to put in between or the registers have been renamed recently. Doing it with `CALL` and `RET` instructions will interfere with prediction in the return stack buffer. Note also that the `ADD ESP` instruction can cause an AGI stall on earlier processors.

6.7 Retirement

Retirement is a process where the temporary registers used by the µops are copied into the permanent registers `EAX`, `EBX`, etc. When a µop has been executed, it is marked in the ROB as ready to retire.

The retirement station can handle three µops per clock cycle. This may not seem like a problem because the throughput is already limited to 3 µops per clock in the RAT. But retirement may still be a bottleneck for two reasons. Firstly, instructions must retire in order. If a µop is executed out of order then it cannot retire before all preceding µops in the order have retired. And the second limitation is that taken jumps must retire in the first of the three slots in the retirement station. Just like decoder D1 and D2 can be idle if the next instruction only fits into D0, the last two slots in the retirement station can be idle if the next µop to retire is a taken jump. This is significant if you have a small loop where the number of µops in the loop is not divisible by three.

All µops stay in the reorder buffer (ROB) until they retire. The ROB can hold 40 µops. This sets a limit to the number of instructions that can execute during the long delay of a division or other slow operation. Before the division is finished the ROB will possibly be filled up with executed µops waiting to retire. Only when the division is finished and retired can the subsequent µops begin to retire, because retirement takes place in order.

In case of speculative execution of predicted branches (see page 21) the speculatively executed µops cannot retire until it is certain that the prediction was correct. If the prediction turns out to be wrong then the speculatively executed µops are discarded without retirement.

The following instructions cannot execute speculatively: memory writes, `IN`, `OUT`, and serializing instructions.

6.8 Partial register stalls

Partial register stall is a problem that occurs when we write to part of a 32-bit register and later read from the whole register or a bigger part of it. Example:

```
; Example 6.10a. Partial register stall
mov al,  byte ptr [mem8]
mov ebx, eax              ; Partial register stall
```

This gives a delay of 5 - 6 clocks. The reason is that a temporary register has been assigned to `AL` to make it independent of `AH`. The execution unit has to wait until the write to `AL` has retired before it is possible to combine the value from `AL` with the value of the rest of `EAX`. The stall can be avoided by changing to code to:

```
; Example 6.10b. Partial register stall removed
movzx ebx, byte ptr [mem8]
and   eax, 0ffffff00h
or    ebx, eax
```

Of course we can also avoid the partial stalls by putting in other instructions after the write to the partial register so that it has time to retire before you read from the full register.

You should be aware of partial stalls whenever you mix different data sizes (8, 16, and 32 bits):

```
; Example 6.11. Partial register stalls
mov bh, 0
add bx, ax              ; Stall
inc ebx                 ; Stall
```

We don't get a stall when reading a partial register after writing to the full register, or a bigger part of it:

```
; Example 6.12. Partial register stalls
mov eax, [mem32]
add bl, al              ; No stall
add bh, ah              ; No stall
mov cx, ax              ; No stall
mov dx, bx              ; Stall
```

The easiest way to avoid partial register stalls is to always use full registers and use `MOVZX` or `MOVSX` when reading from smaller memory operands. These instructions are fast on the

PPro, P2 and P3, but slow on earlier processors. Therefore, a compromise is offered when you want your code to perform reasonably well on all processors. The replacement for `MOVZX EAX,BYTE PTR [MEM8]` looks like this:

```
; Example 6.13. Replacement for movzx
xor  eax, eax
mov  al,  byte ptr [mem8]
```

The PPro, P2 and P3 processors make a special case out of this combination to avoid a partial register stall when later reading from `EAX`. The trick is that a register is tagged as empty when it is `XOR`'ed with itself. The processor remembers that the upper 24 bits of `EAX` are zero, so that a partial stall can be avoided. This mechanism works only on certain combinations:

```
; Example 6.14. Removing partial register stalls with xor
xor eax, eax
mov al, 3
mov ebx, eax            ; No stall

xor ah, ah
mov al, 3
mov bx, ax              ; No stall

xor eax, eax
mov ah, 3
mov ebx, eax            ; Stall

sub ebx, ebx
mov bl, dl
mov ecx, ebx            ; No stall

mov ebx, 0
mov bl, dl
mov ecx, ebx            ; Stall

mov bl, dl
xor ebx, ebx            ; No stall
```

Setting a register to zero by subtracting it from itself works the same as the `XOR`, but setting it to zero with the `MOV` instruction doesn't prevent the stall.

We can set the `XOR` outside a loop:

```
; Example 6.15. Removing partial register stalls with xor outside loop
      xor eax, eax
      mov ecx, 100
LL:   mov al, [esi]
      mov [edi], eax          ; no stall
      inc esi
      add edi, 4
      dec ecx
      jnz LL
```

The processor remembers that the upper 24 bits of `EAX` are zero as long as you don't get an interrupt, misprediction, or other serializing event.

You should remember to neutralize any partial register you have used before calling a subroutine that might push the full register:

```
; Example 6.16. Removing partial register stall before call
add  bl, al
```

```
mov   [mem8], bl
xor   ebx, ebx              ; neutralize bl
call _highLevelFunction
```

Many high-level language procedures push `EBX` at the start of the procedure, and this would generate a partial register stall in the example above if you hadn't neutralized `BL`.

Setting a register to zero with the `XOR` method doesn't break its dependence on earlier instructions on PPro, P2, P3 and PM (but it does on P4). Example:

```
; Example 6.17. Remove partial register stalls and break dependence
div ebx
mov [mem], eax
mov eax, 0              ; Break dependence
xor eax, eax            ; Prevent partial register stall
mov al, cl
add ebx, eax
```

Setting `EAX` to zero twice here seems redundant, but without the `MOV EAX,0` the last instructions would have to wait for the slow `DIV` to finish, and without `XOR EAX,EAX` you would have a partial register stall.

The `FNSTSW AX` instruction is special: in 32-bit mode it behaves as if writing to the entire `EAX`. In fact, it does something like this in 32-bit mode:

```
; Example 6.18. Equivalence model for fnstsw ax
and    eax, 0ffff0000h
fnstsw temp
or     eax, temp
```

hence, you don't get a partial register stall when reading `EAX` after this instruction in 32 bit mode:

```
; Example 6.19. Partial register stalls with fnstsw ax
fnstsw ax / mov ebx,eax        ; Stall only if 16 bit mode
mov ax,0  / fnstsw ax          ; Stall only if 32 bit mode
```

Partial flags stalls

The flags register can also cause partial register stalls:

```
; Example 6.20. Partial flags stall
cmp   eax, ebx
inc   ecx
jbe   xx          ; Partial flags stall
```

The `JBE` instruction reads both the carry flag and the zero flag. Since the `INC` instruction changes the zero flag, but not the carry flag, the `JBE` instruction has to wait for the two preceding instructions to retire before it can combine the carry flag from the `CMP` instruction and the zero flag from the `INC` instruction. This situation is likely to be a bug in the assembly code rather than an intended combination of flags. To correct it, change `INC ECX` to `ADD ECX,1`. A similar bug that causes a partial flags stall is `SAHF` / `JL XX`. The `JL` instruction tests the sign flag and the overflow flag, but `SAHF` doesn't change the overflow flag. To correct it, change `JL XX` to `JS XX`.

Unexpectedly (and contrary to what Intel manuals say) we also get a partial flags stall after an instruction that modifies some of the flag bits when reading only unmodified flag bits:

```
; Example 6.21. Partial flags stall when reading unmodified flag bits
cmp eax, ebx
```

```
inc ecx
jc  xx          ; Partial flags stall
```

but not when reading only modified bits:

```
; Example 6.22. No partial flags stall when reading modified bits
cmp eax, ebx
inc ecx
jz  xx          ; No stall
```

Partial flags stalls are likely to occur on instructions that read many or all flags bits, i.e. `LAHF`, `PUSHF`, `PUSHFD`. The following instructions cause partial flags stalls when followed by `LAHF` or `PUSHF(D)`: `INC`, `DEC`, `TEST`, bit tests, bit scan, `CLC`, `STC`, `CMC`, `CLD`, `STD`, `CLI`, `STI`, `MUL`, `IMUL`, and all shifts and rotates. The following instructions do not cause partial flags stalls: `AND`, `OR`, `XOR`, `ADD`, `ADC`, `SUB`, `SBB`, `CMP`, `NEG`. It is strange that `TEST` and `AND` behave differently while, by definition, they do exactly the same thing to the flags. You may use a `SETcc` instruction instead of `LAHF` or `PUSHF(D)` for storing the value of a flag in order to avoid a stall.

Examples:

```
; Example 6.23. Partial flags stalls
inc eax   / pushfd      ; Stall
add eax,1 / pushfd      ; No stall

shr eax,1 / pushfd      ; Stall
shr eax,1 / or eax,eax / pushfd    ; No stall

test ebx,ebx / lahf     ; Stall
and  ebx,ebx / lahf     ; No stall
test ebx,ebx / setz al  ; No stall

clc / setz al           ; Stall
cld / setz al           ; No stall
```

The penalty for partial flags stalls is approximately 4 clocks.

<u>Flags stalls after shifts and rotates</u>

You can get a stall resembling the partial flags stall when reading any flag bit after a shift or rotate, except for shifts and rotates by one (short form):

```
; Example 6.24. Partial flags stalls after shift and rotate
shr eax,1 / jz xx                ; No stall
shr eax,2 / jz xx                ; Stall
shr eax,2 / or eax,eax / jz xx   ; No stall

shr eax,5 / jc xx                ; Stall
shr eax,4 / shr eax,1 / jc xx    ; No stall

shr eax,cl / jz xx               ; Stall, even if cl = 1
shrd eax,ebx,1 / jz xx           ; Stall
rol ebx,8 / jc xx                ; Stall
```

The penalty for these stalls is approximately 4 clocks.

6.9 Store forwarding stalls

A store forwarding stall is somewhat analogous to a partial register stall. It occurs when you mix data sizes for the same memory address:

```
; Example 6.25. Store-to-load forwarding stall
mov byte ptr [esi], al
mov ebx, dword ptr [esi]        ; Stall. Big read after small write
```

The large read after a small write prevents store-to-load forwarding, and the penalty for this is approximately 7 - 8 clock cycles.

Unlike the partial register stalls, you also get a store forwarding stall when you write a bigger operand to memory and then read part of it, if the smaller part doesn't start at the same address:

```
; Example 6.26. Store-to-load forwarding stall
mov dword ptr [esi], eax
mov bl, byte ptr [esi]          ; No stall
mov bh, byte ptr [esi+1]        ; Stall. Not same start address
```

We can avoid this stall by changing the last line to `MOV BH,AH`, but such a solution is not possible in a situation like this:

```
; Example 6.27. Store-to-load forwarding stall
fistp qword ptr [edi]
mov eax, dword ptr [edi]
mov edx, dword ptr [edi+4]      ; Stall. Not same start address
```

Interestingly, you can get a get a bogus store forwarding stall when writing and reading completely different addresses if they happen to have the same set-value in different cache banks:

```
; Example 6.28. Bogus store-to-load forwarding stall
mov byte ptr [esi], al
mov ebx, dword ptr [esi+4092]   ; No stall
mov ecx, dword ptr [esi+4096]   ; Bogus stall
```

6.10 Bottlenecks in PPro, P2, P3

When optimizing code for these processors, it is important to analyze where the bottlenecks are. Spending time on optimizing away one bottleneck doesn't make sense if another bottleneck is narrower.

If you expect code cache misses, then you should restructure your code to keep the most used parts of code together.

If you expect many data cache misses, then forget about everything else and concentrate on how to restructure the data to reduce the number of cache misses (page 8), and avoid long dependency chains after a data read cache miss.

If you have many divisions, then try to reduce them as described in manual 1: "Optimizing software in C++" and manual 2: "Optimizing subroutines in assembly language", and make sure the processor has something else to do during the divisions.

Dependency chains tend to hamper out-of-order execution. Try to break long dependency chains, especially if they contain slow instructions such as multiplication, division, and floating point instructions. See the manual 1: "Optimizing software in C++" and manual 2: "Optimizing subroutines in assembly language".

If you have many jumps, calls, or returns, and especially if the jumps are poorly predictable, then try if some of them can be avoided. Replace poorly predictable conditional jumps with conditional moves if it doesn't increase dependencies. Inline small procedures. (See manual 2: "Optimizing subroutines in assembly language").

If you are mixing different data sizes (8, 16, and 32 bit integers) then look out for partial stalls. If you use `PUSHF` or `LAHF` instructions then look out for partial flags stalls. Avoid testing flags after shifts or rotates by more than 1 (page 81).

If you aim at a throughput of 3 µops per clock cycle then be aware of possible delays in instruction fetch and decoding (page 71), especially in small loops. Instruction decoding is often the narrowest bottleneck in these processors, and unfortunately this factor makes optimization quite complicated. If you are making a modification in the beginning of your code in order to improve it, then this modification may have the side effect of moving the IFETCH boundaries and 16-byte boundaries in the subsequent code. This change of boundaries can have unpredicted effects on the total clock count which obfuscates the effect of the change you made.

The limit of two permanent register reads per clock cycle may reduce your throughput to less than 3 µops per clock cycle (page 75). This is likely to happen if you often read registers more than 4 clock cycles after they last were modified. This may, for example, happen if you often use pointers for addressing your data but seldom modify the pointers.

A throughput of 3 µops per clock requires that no execution port gets more than one third of the µops (page 79).

The retirement station can handle 3 µops per clock, but may be slightly less effective for taken jumps (page 80).

7 Pentium M pipeline

7.1 The pipeline in PM

This chapter applies to the Intel Pentium M, Core Solo and Core Duo, but not to Core 2. The abbreviation PM in this manual includes Pentium M, Core Solo and Core Duo.

The PM builds on the same basic microarchitecture as PPro, P2 and P3, while the P4/NetBurst design has been discontinued. The main stages in the pipeline are: branch prediction, instruction fetch, instruction decoding, register renaming, reorder buffer read, reservation station, execution ports, reorder buffer write-back, and retirement.

Several minor modifications have been made, but the overall functioning is almost identical to the PPro pipeline, as shown in figure 6.1 page 70. The exact structure of the PM pipeline has not been revealed by Intel. The only thing they have told is that the pipeline is longer, so the following discussion is mainly guesswork based on my own measurements.

The total length of the pipeline can be estimated from the branch misprediction penalty (p. 11). This penalty is 3-4 clock cycles more than for the P2 and P3. This indicates that the pipeline may have 3 or 4 extra stages. We may try to guess what these extra stages are used for.

The branch prediction mechanism is much more complicated in PM than in previous processors (p. 25), so it is likely that this mechanism requires three pipeline stages instead of two.

Instruction fetching has also been improved so that 16-byte boundaries or cache line boundaries do not cause delays in jumps (p. 88 below). This may require an extension of the instruction fetch unit from 3 to 4 stages.

The new stack engine (p. 92) is implemented near the instruction decoding, according to the Intel publication mentioned below. It is almost certain that at least one extra pipeline stage is required for the stack engine and for inserting stack synchronization µops (p. 92). This claim is based on my observation that an instruction that is decoded by D1 or D2 can generate a synchronization µop without adding to the decode time even though these decoders can only generate one µop. The extra synchronization µop must therefore be generated at a pipeline stage that comes after the stage that contains decoders D0-2.

One may wonder if the µop fusion mechanism (explained below, p. 90) requires extra stages in the pipeline. The number of stages from the ROB-read to the ROB-writeback stage can be estimated by measuring register read stalls (p. 75). My measurements indicate that this distance is still only 3 clocks. We can therefore conclude that no extra pipeline stage is used for splitting fused µops before the execution units. The two parts of a fused µop share the same ROB entry, which is submitted to two different ports, so there is probably not any extra pipeline stage for joining the split µops before the retirement station, either.

The RAT, ROB-read, and RS stages have all been modified in order to handle fused µops with three input dependences. It is possible that an extra pipeline stage has been added to the RAT because of the extra workload in this stage, but I have no experimental evidence supporting such a hypothesis. The RS still needs only one clock according to my measurements of the distance from `ROB-read` to ROB-writeback mentioned above. There have been speculations that the RS and ROB might be smaller than in previous processors, but this is not confirmed by my measurements. The RS and ROB can probably hold 20 and 40 fused µops respectively.

The conclusion is that the PM pipeline probably has 3 or 4 stages more than the PPro pipeline, including one extra stage for branch prediction, one extra stage for instruction fetching, and one extra stage for the stack engine.

The PM has many power-saving features that turn off certain parts of the internal buses, execution units, etc. when they are not used. Whether any of these new features require extra pipeline stages is unknown. These power-saving features have the positive side effect that the maximum clock frequency we can have without overheating the chip is increased.

The µop fusion, stack engine and complicated branch prediction are improvements which not only lower the power consumption but also speed up execution.

(Literature: S. Gochman, et al.: The Intel Pentium M Processor: Microarchitecture and Performance. Intel Technology Journal, vol. 7, no. 2, 2003).

7.2 The pipeline in Core Solo and Duo

I have not had the chance to do a thorough testing of the Core Solo and Core Duo yet, but a preliminary testing shows that its kernel is very similar to the Pentium M. The Core Solo/Duo have more advanced power saving features than the Pentium M, including the "SpeedStep" technology that enables it to lower the CPU voltage and clock frequency when the workload is small. The Core Duo has two processor cores with separate level-1 caches and a shared level-2 cache.

(Literature: S. Gochman, et al.: Introduction to Intel Core Duo Processor Architecture. Intel Technology Journal, vol. 10, no. 2, 2006).

7.3 Instruction fetch

Instruction fetching in the PM works the same way as in PPro, P2 and P3 (see p. 70) with one important improvement. Fetching of instructions after a predicted jump is more efficient and is not delayed by 16-byte boundaries. The delays in table 6.1 (p. 71) do not apply to the PM and it is possible to have one jump per clock cycle. For this reason, it is no longer important to align subroutine entries and loop entries. The only reason for aligning code on the PM is to improve cache efficiency.

Instructions are still fetched in IFETCH blocks (p. 70) which are up to 16 bytes long. The PM can fetch a maximum of one IFETCH block per clock cycle. The first IFETCH block after a predicted jump will normally begin at the first instruction. This is different from the previous processors where the placement of the first IFETCH block was uncertain. The next IFETCH block will begin at the last instruction boundary that is no more than 16 bytes away. Thus, you can predict where all IFETCH blocks are by looking at the output listing of the assembler. Assume that the first IFETCH block starts at a label jumped to. The second IFETCH block is found by going 16 bytes forward. If there is no instruction boundary there then go backwards to the nearest instruction boundary. This is where the second IFETCH block starts. Knowing where the IFETCH boundaries are can help improve decoding speed, as explained below.

7.4 Instruction decoding

Instruction decoding on the PM works the same way as on PPro, P2 and P3, as explained on page 71. There are three parallel decoders: D0, D1 and D2. Decoder D0 can handle any instruction. D1 and D2 can handle only instructions that generate no more than one µop, are no more than 8 bytes long, and have no more than one prefix.

It is possible to decode three instructions in the same clock cycle if they are contained in the same IFETCH block and the second and third instruction satisfy the criteria for going into decoders D1 and D2.

Some complex instructions take more than one clock cycle to decode:

- Instructions that generate more than four µops take more than one clock cycle to decode.

- Instructions with more than one prefix take 2+n clock cycles to decode, where n is the total number of prefixes. See manual 2: "Optimizing subroutines in assembly language" for an overview of instruction prefixes. Instructions with more than one prefix should be avoided.

- An operand size prefix causes problems if the size of the rest of the instruction is changed by the prefix. Decoder D0 will need an extra clock cycle to re-interpret the instruction. D1 and D2 are stalled in the meantime because the instruction length is re-adjusted. This problem happens when an instruction has a 16-bit immediate operand in 32-bit mode or a 32-bit immediate operand in 16-bit mode. See p. 74 for examples.

- The same problem occurs if an address size prefix changes the length of the rest of the instruction, for example `LEA EAX,[BX+200]` in 32-bit mode or `LEA AX,[EBX+ECX]` in 16-bit mode.

The maximum output of the decoders is six µops per clock cycle. This speed is obtained if the abovementioned problems are avoided and instructions are scheduled according to a 4-1-1 pattern so that every third instruction generates 4 µops and the next two instructions generate 1 µop each. The instruction that generates 4 µops will go into decoder D0, and the next two instructions will go into D1 and D2. See manual 4: "Instruction tables" for a list of how many µops each instruction generates, and use the values listed under "µops fused domain". The first instruction in an IFETCH block must go to decoder D0. If this instruction was intended for D1 or D2 according to the 4-1-1 pattern, then the pattern is broken and a clock cycle is lost.

It is superfluous to schedule the code for a decoding output of six µops per clock cycle because the throughput in later stages is only three µops per clock cycle. For example, a 2-1-2-1 pattern will generate three µops per clock cycle. But it is recommended to aim at an average output somewhat higher than three µops per clock cycle because you may lose a clock cycle when an instruction intended for decoder D1 or D2 falls at the beginning of an IFETCH block.

If decoding speed is critical then you may reduce the risk of single-µop instructions falling in the beginning of IFETCH blocks by reducing instruction sizes. See manual 2: "Optimizing subroutines in assembly language" about optimizing for code size. A possibly more effective strategy is to determine where each IFETCH block begins according to the method explained on p. 88 and then adjust instruction lengths so that only instructions intended for decoder D0 fall at the beginnings of IFETCH blocks. See the chapter "Making instructions longer for the sake of alignment" in manual 2: "Optimizing subroutines in assembly language" for how to make instruction codes longer. This method is tedious and should only be used if decoding is a bottleneck. Example:

```
; Example 7.1. Arranging IFETCH blocks
LL:  movq    mm0,[esi+ecx]    ; 4 bytes long
     paddd   mm0,[edi+ecx]    ; 4 bytes long
     psrld   mm0,1            ; 4 bytes long
     movq    [esi+ecx],mm0    ; 4 bytes long
     add     ecx,8            ; 3 bytes long
```

```
        jnz     LL              ; 2 bytes long
```

This loop calculates the average of two lists of integers. The loop has six instructions which generate one µop each. The first four instructions are all four bytes long. If we assume that the first IFETCH block starts at `LL:`, then the second IFETCH block will start 16 bytes after this, which is at the beginning of `ADD ECX,8`. The first three instructions will be decoded in one clock cycle. The fourth instruction will be decoded alone in the second clock cycle because there are no more instructions in the first IFETCH block. The last two instructions will be decoded in the third clock cycle. The total decoding time is three clock cycles per iteration of the loop. We can improve this by adding a DS segment prefix, for example to the fourth instruction. This makes the instruction one byte longer, so that the first IFETCH block now ends before the end of the fourth instruction. This causes the second IFETCH block to be moved up to the beginning of the fourth instruction. Now the last three instructions are in the same IFETCH block so that they can be decoded simultaneously in D0, D1 and D2, respectively. The decoding time is now only two clock cycles for the six instructions. The total execution time has been reduced from three to two clock cycles per iteration because nothing else limits the speed, except possibly cache misses.

7.5 Loop buffer

The PM has a loop buffer of 4x16 bytes storing predecoded instructions. This is an advantage in tiny loops where instruction fetching is a bottleneck. The decoders can reuse the fetched instructions up to 64 bytes back. This means that instruction fetching is not a bottleneck in a loop that has no more than 64 bytes of code and is aligned by 16. Alignment of the loop is not necessary if the loop has no more than 49 bytes of code because a loop of this size will fit into the 4*16 bytes even in the worst case of misalignment.

7.6 Micro-op fusion

The register renaming (RAT) and retirement (RRF) stages in the pipeline are bottlenecks with a maximum throughput of 3 µops per clock cycle. In order to get more through these bottlenecks, the designers have joined some operations together that were split in two µops in previous processors. They call this µop fusion. The fused operations share a single µop in most of the pipeline and a single entry in the reorder buffer (ROB). But this single ROB entry represents two operations that have to be done by two different execution units. The fused ROB entry is dispatched to two different execution ports but is retired as a single unit.

The µop fusion technique can only be applied to two types of combinations: memory write operations and read-modify operations.

A memory write operation involves both the calculation of the memory address and transfer of the data. On previous processors, these two operations have been split into two µops, where port 3 takes care of the address calculation and port 4 takes care of the data transfer. These two µops are fused together in most PM instructions that write to memory. A memory read operation requires only one µop (port 2), as it does in previous processors.

The second type of operations that can be fused is read-modify operations. For example, the instruction `ADD EAX,[mem32]` involves two operations: The first operation (port 2) reads from `[mem32]`, the second operation (port 0 or 1) adds the value that has been read to `EAX`. Such instructions have been split into two µops on previous processors, but can be fused together on the PM. This applies to many read-modify instructions that work on general purpose registers, floating point stack registers and MMX registers, but not to read-modify instructions that work on XMM registers.

A read-modify-write operation, such as `ADD [mem32],EAX` does not fuse the read and modify µops, but it does fuse the two µops needed for the write.

Examples of µop fusion:

```
; Example 7.2. Uop fusion
mov    [esi], eax              ; 1 fused uop
add    eax, [esi]              ; 1 fused uop
add    [esi], eax              ; 2 single + 1 fused uop
fadd   qword ptr [esi]         ; 1 fused uop
paddw  mm0, qword ptr [esi]    ; 1 fused uop
paddw  xmm0, xmmword ptr [esi] ; 4 uops, not fused
addss  xmm0, dword ptr [esi]   ; 2 uops, not fused
movaps xmmword ptr [esi], xmm0 ; 2 fused uops
```

As you can see, the packed addition (`PADDW`) read-modify instruction can be fused in the Pentium M if the destination is a 64-bit MMX register, but not if the destination is a 128-bit XMM register. The latter instruction requires two µops for reading 64 bits each, and two more µops for adding 64 bits each. The `ADDSS` instruction cannot be fused, even though it uses only the lower part of the XMM register. No read-modify instruction that involves an XMM register can be fused, but the XMM memory write instructions can be fused, as the last example shows.

The Core Solo/Duo has more opportunities for µop fusion of XMM instructions. A 128-bit XMM instruction is handled by two or more 64-bit µops in the 64-bit execution units. Two 64-bit µops can be "laminated" together in the decoders and early pipeline stages, according to the article cited in chapter 7.2. It is not clear whether there is any significant difference between "fusion" and "lamination" of µops. The consequence of this mechanism is that the throughput of XMM instructions is increased in the decoders, but not in the execution units of the Core Solo/Duo.

The reorder buffer (ROB) of PM has been redesigned so that each entry can have up to three input dependences, where previous designs allowed only two. For example, the instructions `MOV [ESI+EDI],EAX` and `ADD EAX,[ESI+EDI]` both have three input dependences, in the sense that both `EAX`, `ESI` and `EDI` have to be ready before all parts of the instructions can be executed. The unfused µops that go to the execution units still have only two input dependences. `MOV [ESI+EDI],EAX` is split into an address calculation µop that depends on `ESI` and `EDI`, and a store µop that depends on the output of the address calculation µop and on `EAX`. Similarly, `ADD EAX,[ESI+EDI]` is split into a read µop that depends on `ESI` and `EDI`, and an `ADD` µop that depends on the output of the read µop and on `EAX`. µops that are not fused can only have two input dependences. For example, the instructions `ADC EAX,EBX` and `CMOVE EAX,EBX` both have three input dependences: `EAX`, `EBX` and the flags. Since neither of these instructions can be fused, they must generate two µops each.

µop fusion has several advantages:

- Decoding becomes more efficient because an instruction that generates one fused µop can go into any of the three decoders while an instruction that generates two µops can go only to decoder D0.
- The load on the bottlenecks of register renaming and retirement is reduced when fewer µops are generated.
- The capacity of the reorder buffer (ROB) is increased when a fused µop uses only one entry.

If a program can benefit from these advantages then it is preferred to use instructions that generate fused µops over instructions that don't. If one or more execution unit is the only bottleneck, then µop fusion doesn't matter because the fused µops are split in two when sent to the execution units.

The table in manual 4: "Instruction tables" shows the fused and unfused µops generated by each instruction in the PM. The column "µops fused domain" indicates the number of µops generated by the decoders, where a fused µop counts as one. The columns under "µops unfused domain" indicates the number of µops that go to each execution port. The fused µops are split at the execution units so that a fused memory write µop is listed both under port 3 and 4, and a fused read-modify µop is listed both under port 2 and port 0 or 1. The instructions that generate fused µops are the ones where the number listed under "µops fused domain" is less than the sum of the numbers listed under "µops unfused domain".

7.7 Stack engine

Stack instructions such as `PUSH`, `POP`, `CALL` and `RET` all modify the stack pointer `ESP`. Previous processors used the integer ALU for adjusting the stack pointer. For example, the instruction `PUSH EAX` generated three µops on the P3 processor, two for storing the value of `EAX`, and one for subtracting 4 from `ESP`. On PM, the same instruction generates only one µop. The two store µops are joined into one by the mechanism of µop fusion, and the subtraction of 4 from `ESP` is done by a special adder that is dedicated for the stack pointer only, called the stack engine. The stack engine is placed immediately after the instruction decoders in the pipeline, before the out-of-order core. The stack engine can handle three additions per clock cycle. Consequently, no instruction will have to wait for the updated value of the stack pointer after a stack operation. One complication by this technique is that the value of `ESP` may also be needed or modified in the out-of-order execution units. A special mechanism is needed for synchronizing the value of the stack pointer in the stack engine and the out-of-order core. The true logical value of the stack pointer ESP_P is obtained as a 32-bit value ESP_O stored in the out-of-order core or the permanent register file and a signed 8-bit delta-value ESP_d stored in the stack engine:

$$ESP_P = ESP_O + ESP_d.$$

The stack engine puts the delta value ESP_d into the address syllable of every stack operation µop as an offset so that it can be added to ESP_O in the address calculation circuitry at port 2 or 3. The value of ESP_d cannot be put into every possible µop that uses the stack pointer, only the µops generated by `PUSH`, `POP`, `CALL` and `RET`. If the stack engine meets a µop other than these that needs `ESP` in the out-of-order execution units, and if ESP_d is not zero, then it inserts a synchronization µop that adds ESP_d to ESP_O and sets ESP_d to zero. The following µop can then use ESP_O as the true value of the stack pointer ESP_P. The synchronization µop is generated after the instruction has been decoded, and does not influence the decoders in any way.

The synchronization mechanism can be illustrated by a simple example:

```
; Example 7.3. Stack synchronization
push  eax
push  ebx
mov   ebp, esp
mov   eax, [esp+16]
```

This sequence will generate four µops. Assuming that ESP_d is zero when we start, the first `PUSH` instruction will generate one µop that writes `EAX` to the address [ESP_O-4] and sets ESP_d = -4. The second `PUSH` instruction will generate one µop that writes `EBX` to the address [ESP_O-8] and sets ESP_d = -8. When the stack engine receives the µop for `MOV EBP,ESP` from the decoders, it inserts a synchronization µop that adds -8 to ESP_O. At the same time, it sets ESP_d to zero. The synchronization µop comes before the `MOV EBP,ESP` µop so that the latter can use ESP_O as the true value of ESP_P. The last instruction, `MOV EAX,[ESP+16]`, also needs the value of `ESP`, but we will not get another synchronization µop here because the value of ESP_d is zero at this point, so a synchronization µop would be superfluous.

Instructions that require synchronization of ESP_O include all instructions that have `ESP` as source or destination operand, e.g. `MOV EAX,ESP`, `MOV ESP,EAX` and `ADD ESP,4`, as well as instructions that use `ESP` as a pointer, e.g. `MOV EAX,[ESP+16]`. It may seem superfluous to generate a synchronization µop before an instruction that only writes to `ESP`. It would suffice to set ESP_d to zero, but it would probably complicate the logic to make the distinction between, say, `MOV ESP,EAX` and `ADD ESP,EAX`.

A synchronization µop is also inserted when ESP_d is near overflow. The 8-bit signed value of ESP_d would overflow after 32 `PUSH EAX` or 64 `PUSH AX` instructions. In most cases, we will get a synchronization µop after 29 `PUSH` or `CALL` instructions in order to prevent overflow in case the next clock cycle gives `PUSH` instructions from all three decoders. The maximum number of `PUSH` instructions before a synchronization µop is 31 in the case that the last three `PUSH` instructions are decoded in the same clock cycle. The same applies to `POP` and `RET` instructions (Actually, you can have one more `POP` than `PUSH` because the value stored is -ESP_d and the minimum of a signed 8-bit number is -128, while the maximum is +127).

The synchronization µops are executed at any one of the two integer ALU's at port 0 and 1. They retire as any other µops. The PM has a recovery table that is used for undoing the effect of the stack engine in mispredicted branches.

The following example shows how synchronization µops are generated in a typical program flow:

```
; Example 7.4. Stack synchronization
        push 1
        call FuncA
        pop  ecx
        push 2
        call FuncA
        pop  ecx
...
...
FuncA PROC NEAR
        push ebp
        mov  ebp, esp        ; Synch uop first time, but not second time
        sub  esp, 100
        mov  eax, [ebp+8]
        mov  esp, ebp
        pop  ebp
        ret
FuncA ENDP
```

The `MOV EBP,ESP` instruction in `FuncA` comes after a `PUSH`, a `CALL`, and another `PUSH`. If ESP_d was zero at the start, then it will be -12 here. We need a synchronization µop before we can execute `MOV EBP,ESP`. The `SUB ESP,100` and `MOV ESP,EBP` don't need synchronization µops because there have been no `PUSH` or `POP` since the last synchronization. After this, we have the sequence `POP` / `RET` / `POP` / `PUSH` / `CALL` / `PUSH` before we meet `MOV EBP,ESP` again in the second call to `FUNCA`. ESP_d has now been counted up to 12 and back again to 0, so we don't need a synchronization µop the second time we get here. If `POP ECX` is replaced by `ADD ESP,4` then we will need a synchronization µop at the `ADD ESP,4` as well as at the second instance of `MOV EBP,ESP`. The same will happen if we replace the sequence `POP ECX` / `PUSH 2` by `MOV DWORD PTR [ESP],2` but not if it replaced by `MOV DWORD PTR [EBP],2`.

We can make a rule for predicting where the synchronization µops are generated by dividing instructions into the following classes:

1. Instructions that use the stack engine: `PUSH`, `POP`, `CALL`, `RET`, except `RET n`.
2. Instructions that use the stack pointer in the out-of-order core, i.e. instructions that have `ESP` as source, destination or pointer, and `CALL FAR`, `RETF`, `ENTER`.
3. Instructions that use the stack pointer both in the stack engine and in the out-of-order core, e.g. `PUSH ESP`, `PUSH [ESP+4]`, `POP [ESP+8]`, `RET n`.
4. Instructions that always synchronize ESP_O: `PUSHF(D)`, `POPF(D)`, `PUSHA(D)`, `POPA(D)`, `LEAVE`.
5. Instructions that don't involve the stack pointer in any way.

A sequence of instructions from class 1 and 5 will not generate any synchronization µops, unless ESP_d is near overflow. A sequence of instructions from class 2 and 5 will not generate any synchronization µops. The first instruction from class 2 after an instruction from class 1 will generate a synchronization µop, except if ESP_d is zero. Instructions from class 3 will generate synchronization µops in most cases. Instructions from class 4 generate a synchronization µop from the decoder rather than from the stack engine, even if $ESP_d = 0$.

You may want to use this rule for reducing the number of synchronization µops in cases where the throughput of 3 µops per clock is a bottleneck and in cases where execution port 0 and 1 are both saturated. You don't have to care about synchronization µops if the bottleneck is elsewhere.

(Literature: S. Gochman, et al.: The Intel Pentium M Processor: Microarchitecture and Performance. Intel Technology Journal, vol. 7, no. 2, 2003).

7.8 Register renaming

Register renaming is controlled by the register alias table (RAT) and the reorder buffer (ROB), shown in figure 6.1. The µops from the decoders and the stack engine go to the RAT via a queue that can hold approximately 10 µops, and then to the ROB-read and the reservation station. The RAT can handle 3 µops per clock cycle. This means that the overall throughput of the microprocessor can never exceed 3 fused µops per clock cycle on average.

The RAT can rename three registers per clock cycle, and it can even rename the same register three times in one clock cycle.

A code with many renamings can sometimes cause stalls, which are difficult to predict. My hypothesis is that these stalls occur in the RAT when it is out of temporary registers. The PM has 40 temporary registers.

The Core Solo/Duo can rename the floating point control word in up to four temporary registers, while the Pentium M cannot rename the floating point control word. This is important for the performance of floating point to integer conversions in C/C++ code.

7.9 Register read stalls

The PM is subject to the same kind of register read stalls as the PPro, P2 and P3, as explained on page 75. The ROB-read stage can read no more than three different registers from the permanent register file per clock cycle. This applies to all general purpose registers, the stack pointer, the flags register, floating point registers, MMX registers and

XMM registers. An XMM register counts as two, because it is stored as two 64-bit registers. There is no limitation on registers that have been modified recently by a preceding µop so that the value has not yet passed through the ROB-writeback stage. See page 75 for a detailed explanation of register read stalls.

Previous processors allowed only two permanent register reads per clock cycle. This value may have been increased to three in the PM, though this is uncertain. Three register read ports may not be sufficient for preventing register read stalls, because many instructions generate fewer µops on PM than on previous processors, thanks to µop fusion and the stack engine, but not fewer register reads. This makes the µop stream more compact and therefore increases the average number of register reads per µop. A fused µop can have up to three input registers, while previous processors allowed only two inputs per µop. If three fused µops with three input registers each go into the ROB-read stage in the same clock cycle then we can have a maximum of nine inputs to read. If these nine input registers are all different and all in the permanent register file, then it will take three clock cycles to read them all through the three register read ports.

The following example shows how register read stalls can be removed:

```
; Example 7.5. Register read stalls
inc   eax             ; (1) read eax, write eax
add   ebx, eax        ; (2) read ebx eax, write ebx
add   ecx, [esp+4]    ; (3) read ecx esp, write ecx
mov   edx, [esi]      ; (4) read esi, write edx
add   edi, edx        ; (5) read edi edx, write edi
```

These instructions generate one µop each through the ROB-read. We assume that none of the registers have been modified in the preceding three clock cycles. The µops go three by three, but we do not know which three go together. There are three possibilities:

A. (1), (2) and (3) go together. We need four register reads: `EAX`, `EBX`, `ECX`, `ESP`.

B. (2), (3) and (4) go together. We need four register reads: `EBX`, `ECX`, `ESP`, `ESI`. (`EAX` doesn't count because it has been written to in the preceding triplet)

C. (3), (4) and (5) go together. We need four register reads: `ECX`, `ESP`, `ESI`, `EDI`. (`EDX` doesn't count because it is written to before it is read).

All three possibilities involve a stall in the ROB-read for reading more than three registers that have not been modified recently. We can remove this stall by inserting a `MOV ECX,ECX` before the first instruction. This will refresh `ECX` so that we need only three permanent register reads, both in situation A, B and C. If we had chosen to refresh, for example, `EBX` instead then we would remove the stall in situation A and B, but not in situation C. So we would have a 2/3 probability of removing the stall. Refreshing `EDI` would have a 1/3 probability of removing the stall, because it works only in situation C. Refreshing `ESP` would also remove the stall in all three situations, but this would delay the fetching of the memory operands by approximately two clock cycles. In general, it doesn't pay to remove a register read stall by refreshing a register used as pointer for a memory read because this will delay the fetching of the memory operand. If we refresh `ESI` in the above example then this would have a 2/3 probability of removing a register read stall, but it would delay the fetching of `EDX`.

If `ESP` has been modified by a stack operation, e.g. `PUSH`, prior to the code in the above example so that ESP_d is not zero (see p. 92) then the stack engine will insert a stack synchronization µop between (2) and (3). This will remove the register read stall but delay the fetching of the memory operand.

7.10 Execution units

Unfused µops are submitted from the reservation station to the five execution ports which connect to all the execution units. Port 0 and 1 receive all arithmetic instructions. Port 2 is for memory read instructions. Memory write instructions are unfused into two µops which go to port 3 and 4, respectively. Port 3 calculates the address, and port 4 does the data transfer.

The maximum throughput of the execution units is five unfused µops per clock cycle, one at each port. On previous processors, such a high throughput could only be obtained in short bursts when the reservation station was full because of the limitation of three µops per clock cycle in the RAT and retirement station. The PM, however, can maintain the throughput of five µops per clock cycle for unlimited periods of time because three fused µops in the RAT can generate five unfused µops at the execution ports. The maximum throughput can be obtained when one third of the µops are fused read-modify instructions (port 2 and port 0/1), one third is fused store instructions (port 3 and 4), and one third is simple ALU or jump instructions (port 0/1).

The execution units are well distributed between port 0 and 1, and many instructions can go to either of these two ports, whichever is vacant first. It is therefore possible to keep both ports busy most of the time in most cases. Port 0 and 1 both have an integer ALU, so that both can handle the most common integer instructions like moves, addition and logic instructions. Two such µops can be executed simultaneously, one at each port. Packed integer ALU instructions can also go to any of the two ALU's.

Integer and floating point instructions share the same multiplication unit and the same division unit, but not the same ALU (addition and logic). This means that the PM can do floating point additions and integer additions simultaneously, but it cannot do floating point multiplications and integer multiplications simultaneously.

Simple instructions such as integer additions have a latency of one clock cycle. Integer multiplications and packed integer multiplications take 3-4 clock cycles. The multiplication unit is pipelined so that it can start a new multiplication every clock cycle. The same applies to floating point addition and single precision floating point multiplication. Double precision floating point multiplications use part of the multiplier unit twice so that it can start a new multiplication every second clock cycle, and the latency is 5.

128-bit XMM operations are split into two 64-bit µops, except if the output is only 64 bits. For example, the `ADDPD` instruction generates two µops for the two 64-bit additions, while `ADDSD` generates only one µop.

7.11 Execution units that are connected to both port 0 and 1

Some execution units are duplicated as mentioned above. For example, two integer vector addition µops can execute simultaneously, one at each ALU going through port 0 and 1, respectively.

Some other execution units are accessible through both port 0 and 1 but are not duplicated. For example, a floating point addition µop can go through either port 0 or port 1. But there is only one floating point adder so it is not possible to execute two floating point addition µops simultaneously.

This mechanism was no doubt implemented in order to improve performance by letting floating point addition µops go through whichever port is vacant first. But unfortunately, this mechanism is doing much more harm than good. A code where most of the µops are floating point additions or other operations that can go through either port takes longer time to execute than expected, typically 50% more time.

The most likely explanation for this phenomenon is that the scheduler will issue two floating point add µops in the same clock cycle, one for each of the two ports. But these two µops cannot execute simultaneously because they need the same execution unit. The consequence is that one of the two ports is stalled and prevented from doing something else for one clock cycle.

This applies to the following instructions:

- All floating point additions and subtractions, including single, double and long double precision, in `ST()` and `XMM` registers with both scalar and vector operands, e.g. `FADD`, `ADDSD`, `SUBPS`.

- xmm compare instructions with xmm result, e.g. `CMPEQPS`.

- xmm max and min instructions, e.g. `MAXPS`.

- xmm vector multiplications with 16 bit integers, e.g. `PMULLW`, `PMADDWD`.

Any code that contains many of these instructions is likely to take more time than it should. It makes no difference whether the instructions are all of the same type or a mixture of the types listed above.

The problem does not occur with µops that can go through only one of the ports, e.g. `MULPS`, `COMISS`, `PMULUDQ`. Neither does the problem occur with µops that can go through both ports where the ALU's are duplicated, e.g. `MOVAPS`, `PADDW`.

A further complication which has less practical importance but reveals something about the hardware design is that instructions such as `PMULLW` and `ADDPS` cannot execute simultaneously, even though they are using different execution units.

The poor performance of code that contains many instructions of the types listed above is a consequence of a bad design. The mechanism was no doubt implemented in order to improve performance. Of course it is possible to find examples where this mechanism actually does improve performance, and the designers may have had a particular example in mind. But the improvement in performance is only seen in code that has few of the instructions listed above and many instructions that occupy port 1. Such examples are rare and the advantage is limited because it applies only to code that has few of these instructions. The loss in performance is highest when many, but not all, of the instructions are of the types listed above. Such cases are very common in floating point code. The critical innermost loop in floating point code is likely to have many additions. The performance of floating point code is therefore likely to suffer quite a lot because of this poor design.

It is very difficult for the programmer to do anything about this problem because you cannot control the reordering of µops in the CPU pipeline. Using integers instead of floating point numbers is rarely an option. A loop which does nothing but floating point additions can be improved by unrolling because this reduces the relative cost of the loop overhead instructions, which suffer from the blocked ports. A loop where only a minority of the instructions are floating point additions can sometimes be improved by reordering the instructions so that no two FADD µops are issued to the ports in the same clock cycle. This requires a lot of experimentation and the result can be sensitive to the code that comes before the loop.

The above explanation of the poor performance of code that contains many floating point additions is based on systematic experimentation but no irrefutable evidence. My theory is based on the following observations:

- The phenomenon is observed for all instructions that generate µops which can go to either port 0 or port 1, but have only one execution unit.

- The phenomenon is not observed for µops that can go to only one of the two ports. For example, the problem disappears when `ADDPS` instructions are replaced by `MULPS`.

- The phenomenon is not observed for µops that can go to both ports where the execution unit is duplicated so that two µops can execute simultaneously.

- Tests are performed with two nested loops where the inner loop contains many floating point additions and the outer loop measures the number of clock cycles used by the inner loop. The clock count of the inner loop is not always constant but often varies according to a periodic pattern. The period depends on small details in the code. Periods as high as 5 and 9 have been observed. These periods cannot be explained by any of the alternative theories I could come up with.

- All the test examples are of course designed to be limited by execution port throughput rather than by execution latencies.

- The information about which µops go to which ports and execution units is obtained by experiments where a particular port or execution unit is saturated.

7.12 Retirement

The retirement station on the PM works exactly the same way as on PPro, P2 and P3. The retirement station can handle three fused µops per clock cycle. Taken jumps can only retire in the first of the three slots in the retirement station. The retirement station will therefore stall for one clock cycle if a taken jump happens to fall into one of the other slots. The number of fused µops in small loops should therefore preferably be divisible by three. See p. 80 for details.

7.13 Partial register access

The PM can store different parts of a register in different temporary registers in order to remove false dependences. For example:

```
; Example 7.6. Partial registers
mov  al, [esi]
inc  ah
```

Here, the second instruction does not have to wait for the first instruction to finish because `AL` and `AH` can use different temporary registers. `AL` and `AH` are stored into each their part of the permanent `EAX` register when the µops retire.

A problem occurs when a write to a part of a register is followed by a read from the whole register:

```
; Example 7.7. Partial register problem
mov  al,  1
mov  ebx, eax
```

On PM model 9, the read from the full register (`EAX`) has to wait until the write to the partial register (`AL`) has retired and the value in `AL` has been joined with whatever was in the rest of `EAX` in the permanent register. This is called a partial register stall. The partial register stalls on PM model 9 are the same as on PPro. See p. 81 for details.

On the later PM model D, this problem has been solved by inserting extra µops to join the different parts of the register. I assume that the extra µops are generated in the ROB-read stage. In the above example, the ROB-read will generate an extra µop that combines `AL` and the rest of `EAX` into a single temporary register before the `MOV EBX,EAX` instruction. This takes one or two extra clock cycles in the ROB-read stage, but this is less than the 5-6 clock penalty of partial register stalls on previous processors.

The situations that generate extra µops on PM model D are the same as the situations that generate partial register stalls on the earlier processors. Writes to the high 8-bit registers `AH`, `BH`, `CH`, `DH` generate two extra µops, while writes to the low 8-bit or 16-bit part of a register generate one extra µop. Example:

```
; Example 7.8a. Partial register access
mov  al, [esi]
inc  ax              ; 1 extra uop for read ax after write al
mov  ah, 2
mov  bx, ax          ; 2 extra uops for read ax after write ah
inc  ebx             ; 1 extra uop for read ebx after write ax
```

The best way to prevent the extra µops and the stalls in ROB-read is to avoid mixing register sizes. The above example can be improved by changing it to:

```
; Example 7.8b. Partial register problem avoided
movzx  eax, byte ptr [esi]
inc    eax
and    eax, 0ffff00ffh
or     eax, 000000200h
mov    ebx, eax
inc    ebx
```

Another way avoid the problem is to neutralize the full register by XOR'ing it with itself:

```
; Example 7.8c. Partial register problem avoided
xor    eax, eax
mov    al,  [esi]
inc    eax           ; No extra uop
```

The processor recognizes the `XOR` of a register with itself as setting it to zero. A special tag in the register remembers that the high part of the register is zero so that `EAX = AL`. This tag is remembered even in a loop:

```
; Example 7.9. Partial register problem avoided in loop
      xor eax, eax
      mov ecx, 100
LL:   mov al, [esi]
      mov [edi], eax          ; No extra uop
      inc esi
      add edi, 4
      dec ecx
      jnz LL
```

The rules for preventing extra µops by neutralizing a register are the same as the rules for preventing partial register stalls on previous processors. See p. 82 for details.

(Literature: Performance and Power Consumption for Mobile Platform Components Under Common Usage Models. Intel Technology Journal, vol. 9, no. 1, 2005).

Partial flags stall

Unfortunately, the PM doesn't generate extra µops to prevent stalls on the flags register. Therefore, there is a stall of 4-6 clock cycles when reading the flags register after an instruction that modifies part of the flags register. Examples:

```
; Example 7.10. Partial flags stalls
inc     eax     ; Modifies zero flag and sign flag, but not carry flag
jz      L1      ; No stall, reads only modified part
jc      L2      ; Stall, reads unmodified part
lahf            ; Stall, reads both modified and unmodified bits
pushfd          ; Stall, reads both modified and unmodified bits
```

The above stalls can be removed by replacing `INC EAX` by `ADD EAX,1` (modifies all flags) or `LEA EAX,[EAX+1]` (modifies no flags). Avoid code that relies on the fact that `INC` or `DEC` leaves the carry flag unchanged.

There is also a partial flags stall when reading the flags after a shift instruction with a count different from 1:

```
; Example 7.11. Partial flags stalls after shift
shr    eax, 1
jc     l1           ; No stall after shift by 1
shr    eax, 2
jc     l2           ; Stall after shift by 2
test   eax, eax
jz     l3           ; No stall because flags have been rewritten
```

See p. 83 for details about partial flags stalls.

7.14 Store forwarding stalls

Store forwarding stalls in the PM are the same as in previous processors. See p. 84 for details.

```
; Example 7.12. Store forwarding stall
mov byte ptr [esi], al
mov ebx, dword ptr [esi]        ; Stall
```

7.15 Bottlenecks in PM

It is important, when optimizing a piece of code, to find the limiting factor that controls execution speed. Tuning the wrong factor is unlikely to have any beneficial effect. In the following paragraphs, I will explain each of the possible limiting factors. You have to consider each factor in order to determine which one is the narrowest bottleneck, and then concentrate your optimization effort on that factor until it is no longer the narrowest bottleneck. As explained before, you have to concentrate on only the most critical part of the program - usually the innermost loop.

Memory access

If the program is accessing large amounts of data, or if the data are scattered around everywhere in the memory, then you will have many data cache misses. Accessing uncached data is so time consuming that all other optimization considerations are unimportant. The caches are organized as aligned lines of 64 bytes each. If one byte within an aligned 64-byte block has been accessed, then you can be certain that all 64 bytes will be loaded into the level-1 data cache and can be accessed at no extra cost. To improve caching, it is recommended that data that are used in the same part of the program be stored together. You may align large arrays and structures by 64. Store local variables on the stack if you don't have enough registers. The PM has four write ports. Having more than four writes immediately after each other can slow down the process by a few clock cycles, especially if there are memory reads simultaneously with the writes. Non-temporal writes to memory are efficient on PM. You may use `MOVNTI`, `MOVNTQ` and `MOVNTPS` for scattered writes to memory if you don't expect to read again soon from the same cache line.

The Core Solo/Duo has an improved data prefetching mechanism that predicts future memory reads.

Instruction fetch and decode

The instructions should be organized according to the 4-1-1 rule if you aim at a throughput of 3 µops per clock cycle. Remember that the 4-1-1 pattern can be broken at IFETCH boundaries. Avoid instructions with more than one prefix. Avoid instructions with 16-bit immediate operand in 32-bit mode. See p. 88.

Micro-operation fusion

µop fusion and the stack engine makes it possible to get more information into each µop. This can be an advantage if decoding or the 3 µops/clock limit is a bottleneck. Floating point registers allow µop fusion for read-modify instructions, but XMM registers do not. Use floating point registers instead of XMM registers for floating point operations if you can take advantage of µop fusion.

Register read stalls

Be aware of register read stalls if a piece of code has more than three registers that it often reads but seldom writes to. See p. 94.

There is a tradeoff between using pointers and absolute addresses. Object oriented code typically accesses most data through frame pointers and '`this`' pointers. The pointer registers are possible sources of register read stalls because they are often read but seldom written to. Using absolute addresses instead of pointers has other disadvantages, however. It makes the code longer so that the cache is used less efficiently and the problem with IFETCH boundaries is increased.

Execution ports

The unfused µops should be distributed evenly between the five execution ports. Port 0 and 1 are likely to be bottlenecks in a code that has few memory operations. You may move some of the load from port 0 and 1 to port 2 by replacing move-register-register and move register-immediate instructions by move-register-memory instructions if this can be done without cache misses.

Instructions using the 64-bit MMX registers are no less efficient than instructions using the 32-bit integer registers on PM. You may use the MMX registers for integer calculations if you are out of integer registers. The XMM registers are slightly less efficient because they do not use µop fusion for read-modify instructions. MMX and XMM instructions are slightly longer than other instructions. This may increase the problem with IFETCH boundaries if decoding is a bottleneck. Remember that you cannot use floating point registers and MMX registers in the same code.

Code with many floating point additions is likely to stall port 0 and 1 because of the design problem discussed on page 96.

Stack synchronization µops go to port 0 or 1. The number of such µops can sometimes be reduced by replacing `MOV` instructions relative to the stack pointer by `PUSH` and `POP` instructions. See page 92 for details.

Partial register access generates extra µops, see page 99.

Execution latencies and dependency chains

The execution units have reasonably low latencies on the PM, and many operations are faster than on P4.

The performance is likely to be limited by execution latencies when the code has long dependency chains with slow instructions.

Avoid long dependency chains and avoid memory intermediates in dependency chains. A dependency chain is not broken by an `XOR` or `PXOR` of a register with itself.

Partial register access

Avoid mixing register sizes and avoid using the high 8-bit registers `AH`, `BH`, `CH`, `DH`. Be aware of partial flags stalls when reading the flags after instructions that modify some of the flag bits and leave other flag bits unchanged, and after shifts and rotates. See p. 100.

Branch prediction

Branch prediction is more advanced in PM than in other processors. Loops with a constant repeat count of up to 64 are predicted perfectly. Indirect jumps and calls with varying targets can be predicted if they follow a regular pattern or if they are well correlated with preceding branches. But the branch target buffer (BTB) is much smaller than on other processors. Therefore, you should avoid unnecessary jumps in order to reduce the load on the BTB. Branch prediction will be good if most of the processor time is spent in a small piece of code with relatively few branches. But branch prediction will be poor if the processor time is distributed over large sections of code with many branches and no particular hot spot. See p. 25.

Retirement

The retirement of taken branches can be a bottleneck in small loops with many branches. See p. 80.

8 Core 2 and Nehalem pipeline

The microarchitecture named "Intel Core 2" is a further development of the PM design. The pipeline has been expanded to handle four micro-operations per clock cycle and the execution units have been expanded from 64 bits to 128 bits. A 45 nm version introduced in 2008 differs from the previous 65 nm version by faster division and shuffling operations.

The Core microarchitecture and its derivatives now form the basis of all Intel's x86 processors, including portable, desktop and server processors. The Intel Core 2 processors have two or more CPU cores with separate level-1 caches and a shared level-2 cache. The Nehalem has separate level-1 and level-2 caches and a shared level-3 cache.

The Nehalem is capable of running two threads in each of its four CPU cores, totaling eight threads.

The x64, SSE3 and Supplementary SSE3 instruction sets are supported in all Core 2 processors. SSE4.1 is supported in the 45 nm versions. SSE4.2 is supported in Nehalem.

The Core has the same power-saving features as the PM. It can turn off certain parts of the internal buses, execution units, etc. when they are not used. The clock frequency is reduced when the workload is small. These power-saving features have the positive side effect that the maximum clock frequency can be increased without overheating the chip. The Nehalem can increase the clock frequency in one core when other cores are idle.

8.1 Pipeline

The Core pipeline is very similar to the Pentium M, but with more of everything in order to increase the throughput from 3 to 4 µops per clock cycle. The advanced power saving technology makes it possible to use a high clock frequency without overheating the chip. The trace cache of the Netburst architecture has been trashed in favor of a traditional code cache.

The Core 2 microarchitecture allegedly has a pipeline of only fourteen stages in order to reduce power consumption, speculative execution and branch misprediction penalty. However, my measurements indicate that the pipeline is approximately two stages longer in the Core2 than in PM. This estimate is based on the fact that the branch misprediction penalty is measured to at least 15, which is 2 clock cycles more than on PM. The time a register stays in the ROB, as measured by register read stalls (p. 75), is approximately 2 clock cycles more than on PM, and partial flags stalls are at least one clock cycle longer on Core2 than on PM. These measurements are in accordance with the observation that instruction fetching and retirement have been improved. It is likely that one extra pipeline stage has been used for improving instruction fetch and predecoding, and one for improving retirement. The pipeline of the Nehalem is at least 2 stages longer with a branch misprediction penalty of at least 17.

The reorder buffer has 96 entries in Core 2 and 128 entries in Nehalem. The reservation station has 32 entries in Core 2 and 36 entries in Nehalem, according to Intel publications.

8.2 Instruction fetch and predecoding

Instruction fetching has been improved over previous Intel processors by adding a queue between branch prediction and instruction fetching. This can remove the delay bubble at taken branches in many cases. Unfortunately, the fetch bandwidth is still limited to 16 bytes per clock cycle. The limiting bottleneck is the predecoder, as explained below.

The instruction decoding machinery is split between a predecoder and a decoder, with a queue in between. This queue has an effective size of 64 bytes in Core 2. The main purpose of the predecoder is to detect where each instruction begins. This is quite difficult because each instruction can have any length from one to fifteen bytes and it can be necessary to inspect several bytes of an instruction in order to determine its length and know where the next instruction begins. The predecoders also identify instruction prefixes and other components of each instruction.

The maximum throughput of the predecoders is 16 bytes or 6 instructions per clock cycle, whichever is smallest. The throughput of the rest of the pipeline is typically 4 instructions per clock cycle, or 5 in case of macro-op fusion (see below, page 107). The throughput of the predecoders is obviously less than 4 instructions per clock cycle if there are less than 4 instructions in each 16-bytes block of code. The average instruction length should therefore preferably be less than 4 bytes on average.

The predecoder throughput can also be reduced if there are more than 6 instructions in a 16-bytes block of code. The reason for this is that the predecoder will not load a new 16-bytes block of code until the previous block is exhausted. If there are 7 instructions in a 16-bytes block then the predecoders will process the first 6 instructions in the first clock cycle and 1 instruction in the next clock cycle. This gives an average predecoder throughput of 3.5 instructions per clock cycle, which is less than desired. The optimal number of instructions per 16-bytes block of code is therefore 5 or 6, corresponding to an average instruction length of approximately 3. Any instruction that crosses a 16-bytes boundary will be left over until the next 16-bytes block is processed. It may be necessary to adjust instruction lengths in order to obtain the optimal number of instructions per 16-bytes block. See manual 2: "Optimizing subroutines in assembly language" for a discussion of how to make instructions shorter or longer.

Loopback buffer

The decoder queue in the Core2 can be used as a 64-bytes, 18 instructions, loop buffer. The predecoded instructions in the decoder queue can be reused in case a branch instruction loops back to an instruction that is still contained in the buffer. Predecoding is therefore not a bottleneck for a small loop that is completely contained in the 64-bytes buffer.

The Core2 loop buffer works almost as a 64 bytes level-0 code cache, organized as 4 lines of 16 bytes each. A loop that can be completely contained in four aligned blocks of 16 bytes each can execute at a rate of up to 32 bytes of code per clock cycle. The four 16-bytes blocks do not even have to be consecutive. A loop that contains jumps (but not calls and returns) can still exceed the predecoder throughput if all the code in the loop can be contained in four aligned 16-bytes blocks.

The Nehalem design is slightly different. The Core2 has the loop buffer between the predecoders and the decoders, while the Nehalem has the loop buffer after the decoders. The Nehalem loop buffer can hold 28 (possibly fused) μops. The size of the loop code is limited to 256 bytes of code, or up to 8 blocks of 32 bytes each. A loop containing more than 256 bytes of code cannot use the loop buffer. There is a one-clock delay in this loop process, so that a loop containing 4*N (possibly fused) μops will take N+1 clock cycles to execute if there are no other bottlenecks elsewhere. The Core2 does not have this one-clock delay.

The loop buffer can speed up execution considerably in cases where predecoding or decoding is a bottleneck, i.e. where instructions are longer than 4 bytes on average or contain length-changing prefixes. Critical loops should therefore preferably be aligned by 16 and be no bigger than 64 bytes or 18 instructions on the Core 2 and 256 bytes or 28 instructions on Nehalem.

Length-changing prefixes

The instruction length decoder has a problem with certain prefixes that change the meaning of the subsequent opcode bytes in such a way that the length of the instruction is changed. This is known as length-changing prefixes.

For example, the instruction `MOV AX,1` has an operand size prefix (`66H`) in 32-bit and 64-bit mode. The same code without operand size prefix would mean `MOV EAX,1`. The `MOV AX,1` instruction has 2 bytes of immediate data to represent the 16-bit value 1, while `MOV EAX,1` has 4 bytes of immediate data to represent the 32-bit value 1. The operand size prefix therefore changes the length of the rest of the instruction. The predecoders are unable to resolve this problem in a single clock cycle. It takes 6 clock cycles to recover from this error. It is therefore very important to avoid such length-changing prefixes. The Intel documents say that the penalty for a length-changing prefix is increased to 11 clock cycles if the instruction crosses a 16-bytes boundary, but I cannot confirm this. My measurements show a penalty of 6 clock cycles in this case as well. The penalty may be less than 6 clock cycles if there are more than 4 instructions in a 16-bytes block.

There are two prefixes that can cause this problem. This is the operand size prefix (`66H`) and the seldom used address size prefix (`67H`). The operand size prefix will change the length of an instruction and cause a delay in 32-bit and 64-bit mode in the following cases:

- If a `MOV` or `TEST` instruction has a 16-bit destination and an immediate constant as source. For example `MOV AX,1`. This should be replaced by `MOV EAX,1`.
- If any other instruction has a 16-bit destination and an immediate constant as source and the constant cannot be represented as an 8-bit sign-extended integer. For example `ADD AX,200`. This should be replaced by `ADD EAX,200`. But `ADD AX,100` does not have a length-changing prefix because 100 is within the range of 8-bit signed integers.
- If one of the instructions `NEG`, `NOT`, `DIV`, `IDIV`, `MUL` and `IMUL` with a single operand has a 16-bit operand and there is a 16-bytes boundary between the opcode byte and the mod-reg-rm byte. These instructions have a bogus length-changing prefix because these instructions have the same opcode as the `TEST` instruction with a 16-bit immediate operand, and the distinction between the `TEST` instruction and the other instructions is contained in the reg bits of the mod-reg-rm byte. Therefore, the decoder cannot determine if the prefix is length-changing or not until it has read the next 16-bytes block of code. You may want to avoid using the 16-bit versions of these instructions if you cannot control where the 16-bytes boundaries are.

These rules also apply to 32-bit operands in 16-bit mode. The disassembler in the `objconv` utility can be used for detecting these length-changing prefixes.

The address size prefix (`67H`) will always cause a delay in 16-bit and 32-bit mode on any instruction that has a mod/reg/rm byte, even if it doesn't change the length of the instruction. The only instructions on which the `67H` prefix makes sense and does not cause a stall are `JCXZ`/`JECXZ`/`JRCXZ`, string instructions and `XLAT`. The address size prefix is never length-changing in 64-bit mode and causes no delay in this mode. Address size prefixes are generally regarded as obsolete and should be avoided.

The REX.W prefix (`48H`) can also change the length of an instruction in the case of a `MOV` instruction with a 64-bit immediate operand, but the predecoder can resolve this case without penalty.

The penalty for length-changing prefixes occurs only the first time in a loop that fits into the loopback buffer because this buffer contains predecoded or decoded instructions.

8.3 Instruction decoding

Instruction decoding in the Core2 and Nehalem is very similar to previous processors, as explained on page 71, but extended from three to four decoders so that it can decode four instructions per clock cycle. The first decoder can decode any instruction that generates up to 4 µops in one clock cycle. The other three decoders can handle only instructions that generate no more than one µop each. There are no other restrictions on which instructions these decoders can handle. The maximum output of the decoders is 7 µops per clock cycle if instructions are organized in a 4-1-1-1 pattern so that the first decoder generates four µops and the other three decoders generate one µop each. The output from the decoders is a minimum of 2 µops per clock cycle if all the instructions generate 2 µops each. In this case, only the first decoder is active because the other three decoders cannot handle instructions that generate more than one µop. If the code contains many instructions that generate 2-4 µops each then these instructions should be spaced with two or three single-µop instructions between in order to optimize decoder throughput. Instructions that generate more than 4 µops use microcode ROM and take multiple clock cycles to decode. The number of µops generated by each instruction is listed in manual 4: "Instruction tables". The figure that is relevant to decoder throughput is the number listed under "µops fused domain".

The decoders can read two 16-bytes blocks of code per clock cycle from the 64-bytes buffer so that a total of 32 bytes can be decoded in one clock cycle. But the output of the predecoders is limited to 16 bytes or less per clock cycle so that the decoders can only receive more than 16 bytes in one clock cycle in linear code if they processed less than 16 bytes in the preceding clock cycle.

Previous processors had a limitation on the number of instruction prefixes that the decoders can handle per clock cycle. The Core2 and Nehalem have no such limitation. Instructions with any number of prefixes can be decoded by any of the four decoders in one clock cycle. The only limitation is set by the instruction set definition which limits the length of instruction plus prefixes to 15 bytes. Thus, it is possible to decode a one-byte instruction with 14 prefixes in a single clock cycle. No instruction needs so many prefixes, of course, but redundant prefixes can be used instead of `NOP`'s as fillers for aligning a subsequent loop entry. See manual 2: "Optimizing subroutines in assembly language" for a discussion of redundant prefixes.

8.4 Micro-op fusion

The Core2 and Nehalem use µop fusion in the same way as the PM, as described on page 90. Some instructions that need two µops in the execution units can use the µop fusion technique to keep these two µops together as one through most of the pipeline in order to save pipeline bandwidth. The fused µop is treated as two µops by the scheduler and submitted to two different execution units, but it is treated as one µop in all other stages in the pipeline and uses only one entry in the reorder buffer. The decoding throughput is also increased by µop fusion because a fused µop can be generated by those decoders that can handle only single-µop instructions.

There are two cases of µop fusion: read-modify instructions and write instructions. A read-modify instruction needs one µop for reading a memory operand and another µop for doing a calculation with this operand. For example, `ADD EAX,[MEM]` needs one µop for reading `MEM` and one for adding this value to `EAX`. These two µops can be fused into one. A write instruction needs one µop for calculating the address and one for writing to that address. For example, `MOV [ESI+EDI],EAX` needs one µop for calculating the address `[ESI+EDI]` and one for storing `EAX` to this address. These two µops are fused together.

The Core2 and Nehalem can use µop fusion in more cases than the PM can. For example, read-modify-write instructions can use both read-modify fusion and write fusion. Most XMM instructions can use µop fusion, but not all.

A fused µop can have three input dependencies, while an unfused µop can have only two. A write instruction may have three input dependencies, for example `MOV [ESI+EDI],EAX`. This is the reason why write instructions are split into two µops, while read instructions have only one µop.

Instructions that have both a rip-relative address and immediate data cannot use µop fusion. For example, `CMP BYTE PTR [RIP+m],AL` can fuse, but `CMP BYTE PTR [RIP+m],1` cannot.

You can see which instructions use µop fusion by looking at the tables in manual 4: "Instruction tables". Instructions with µop fusion have a higher number of µops listed under "unfused domain" than under "fused domain".

8.5 Macro-op fusion

The Core2 and Nehalem can also fuse two instructions into one µop in a few cases. This is called macro-op fusion. The decoders will fuse a compare or test instruction with a subsequent conditional jump instruction into a single compare-and-branch µop in certain cases. The compare-and-branch µop is not split in two at the execution units but executed as a single µop by the branch unit at execution port 5. This means that macro-op fusion saves bandwidth in all stages of the pipeline from decoding to retirement. Macro-op fusion does not help, however, if predecoding is the bottleneck.

Macro-op fusion is possible only if all of the following conditions are satisfied:

- The first instruction is a `CMP` or `TEST` instruction and the second instruction is a conditional jump instruction except `JECXZ` and `LOOP`.

- The `CMP` or `TEST` instruction can have two register operands, a register and an immediate operand, a register and a memory operand, but not a memory and an immediate operand.

- Core2 can do macro-op fusion only in 16-bit and 32-bit mode. Core Nehalem can also do this in 64-bit mode.

- Branches that test the zero flag and/or the carry flag (`JE`, `JNE`, `JB`, `JBE`, `JA`, `JAE`) can fuse with a preceding `CMP` or `TEST`. This includes all unsigned comparisons. Branches for signed comparisons (`JL`, `JLE`, `JG`, `JGE`) can fuse with a preceding `CMP` or `TEST` on Core Nehalem but only with `TEST` on Core2.
  Branches that test the overflow, parity or sign flag only (`JO`, `JNO`, `JP`, `JNP`, `JS`, `JNS`) can fuse with `TEST` but not with `CMP`.

- There can be no other instructions between the two instructions (branch hint prefixes are allowed, but ignored).

- The branch instruction should not start at a 16-bytes boundary or cross a 16-bytes boundary.

- If more than one such instruction pair reaches the four decoders in the same clock cycle then only the first pair is macro-fused.

Any one of the four decoders can make a macro-op fusion, but not simultaneously. Thus, we see that the four decoders can handle a maximum of five instructions in a single clock cycle in case of macro-op fusion.

It is possible to have both micro-op fusion and macro-op fusion at the same time. A `CMP` or `TEST` instruction with a memory operand and a register operand followed by a branch instruction can generate a single micro-macro-fused µop containing all the three operations: read, compare, and branch. However, there is a limit to how much information a µop can contain. This limit is probably defined by the size of a reorder-buffer (ROB) entry. There is not enough space for storing both an immediate operand, the address of a memory operand, and the address of a branch target in the same ROB entry. My guess is that this is the reason why we can't have macro-op fusion with both a memory operand and an immediate operand. This may also be the reason why macro-op fusion doesn't work in 64-bit mode on Core2: 64-bit branch addresses take more space in the ROB entry.

The programmer should keep `CMP` or `TEST` instructions together with the subsequent conditional jump rather than scheduling other instructions in-between; and there should preferably be at least three other instructions between one compare-branch pair and the next compare-branch pair in order to take advantage of macro-op fusion. The branch instruction after a `CMP` should preferably be of an unsigned type if it can be verified that none of the operands can be negative.

The fact that macro-op fusion doesn't work in 64-bit mode on Core2 should not make any programmer refrain from using 64-bit mode. The performance gain due to macro-op fusion is unlikely to be more than a few percent, and only if µop throughput is a bottleneck. Macro-op fusion has no effect in the much more likely case that the bottleneck lies elsewhere.

8.6 Stack engine

The Core2 and Nehalem has a dedicated stack engine which works the same way as on the PM, as described on page 92, with necessary adjustments for the larger pipeline.

The modification of the stack pointer by `PUSH`, `POP`, `CALL` and `RET` instructions is done by a special stack engine, which is placed immediately after the decoding stage in the pipeline and before the out-of-order core. This relieves the pipeline from the burden of µops that modify the stack pointer. This mechanism saves two copies of the stack pointer: one in the stack engine and another one in the register file and the out-of-order core. These two stack pointers may need to be synchronized if a sequence of `PUSH`, `POP`, `CALL` and `RET` instructions is followed by an instruction that reads or modifies the stack pointer directly, such as `ADD ESP,4` or `MOV EAX,[ESP+8]`. The stack engine inserts an extra stack-synchronization µop in every case where synchronization of the two stack pointers is needed. See page 92 for a more detailed explanation.

The stack synchronization µops can sometimes be avoided by not mixing instructions that modify the stack pointer through the stack engine and instructions that access the stack pointer in the out-of-order execution units. A sequence that contains only instructions from one of these two categories will not need stack synchronization µops, but a sequence that mixes these two categories will need these extra µops. For example, it is advantageous to replace an `ADD ESP,4` instruction after a function call by `POP ECX` if the preceding instruction was a `RET` and the next instruction touching the stack pointer is a `PUSH` or `CALL`.

It may be possible to avoid stack synchronization µops completely in a critical function if all function parameters are transferred in registers and all local variables are stored in registers or with `PUSH` and `POP`. This is most realistic with the calling conventions of 64-bit Linux. Any necessary alignment of the stack can be done with a dummy `PUSH` instruction in this case.

8.7 Register renaming

All integer, floating point, MMX, XMM, flags and segment registers can be renamed. The floating point control word can also be renamed.

Register renaming is controlled by the register alias table (RAT) and the reorder buffer (ROB), shown in figure 6.1. The µops from the decoders and the stack engine go to the RAT via a queue and then to the ROB-read and the reservation station. The RAT can handle 4 µops per clock cycle. The RAT can rename four registers per clock cycle, and it can even rename the same register four times in one clock cycle.

8.8 Register read stalls

The Core2 and Nehalem are subject to the same kind of register read stalls as the PM and earlier processors, as explained on page 75. The permanent register file has three read ports on the Core2 and Nehalem for reading instruction operands.

The ROB-read stage can read no more than three different registers from the permanent register file per clock cycle. This applies to all general purpose registers, the stack pointer, the flags register, floating point registers, MMX registers and XMM registers. An XMM register counts as one on the Core, while it counts as two 64-bit registers on previous processors.

The first two of the three register read ports can read registers for instruction operands, base pointers, and index pointers. The third read port can read only registers used as index pointers on the Core2. On Core Nehalem, all three read ports can be used for any operands. The same register can be read any number of times in the same clock cycle without causing stalls.

Registers that have been written to recently can be read directly from the ROB if they have not yet passed the ROB-writeback stage. Registers that can be read directly from the ROB do not need the read ports on the register file. My measurements indicate that it takes approximately 5 clock cycles for a µop to pass from the ROB-read stage to the ROB-writeback stage. This means that a register can be read without problems if it has been modified within the last 5 clock cycles. With a throughput of 4 µops per clock cycle, you can assume that a register can be read without using the register read ports if it has been modified within the last 20 µops unless the pipeline has been stalled for any reason in the meantime.

A unfused µop can contain up to two register reads, and a fused µop can contain up to three register reads. For example, the instruction `ADD EAX,[EBX+ECX]` reads register `EAX`, `EBX` and `ECX`, and then writes register `EAX`. The decoders can send up to four fused µops to the ROB-read stage in one clock cycle. The maximum number of register reads in a µop quartet is therefore twelve. The ROB-read stage may need four clock cycles to read these twelve registers in the worst case where all registers are in the permanent register file.

```
; Example 8.1a. Register read stall on Core2
L:  mov  eax, [esi+ecx]
    mov  [edi+ecx], ebx
    add  ecx, 4
    js   L
```

This loop has a register read stall because there are three registers that are read inside the loop, but not written to: `ESI`, `EDI` and `EBX`. `ECX` does not need a read port because it has been modified recently. There are three register read ports, but on Core2 the third read port can only be used for index registers, and none of the three read-only registers are used as index registers. Note that the SIB byte of the instruction code makes a distinction between base and index register. ESI and EDI are base registers in example 8.1a, while ECX is index register. An index register can have a scale factor, while a base register cannot.

A slight modification of the code can make EDI an index register so that the third read port can be used:

```
; Example 8.1b. Register read stall removed
L:  mov  eax, [ecx+esi*1]
    mov  [ecx+edi*1], ebx
    add  ecx, 4
    js   L
```

Here, we have applied the scale factor *1 to ESI and EDI to make sure the assembler uses ESI and EDI rather than ECX as the index register. ESI or EDI can now be read by the third register read port so that the stall disappears. Example 8.1b takes 1 clock cycle per iteration, while example 8.1a takes two clock cycles on Core2.

It is difficult to predict which µops will go into the ROB-read stage together. The µops arrive in order, but you don't know where each quartet begins unless the decoders have been stalled. A register read stall can therefore occur if more than two or three registers are read in any four consecutive µops and these registers have not been written to recently.

Removing register read stalls often requires some experimentation. The Core2 and Nehalem have performance monitor counters that you can use for detecting register read stalls.

It is possible to change a base pointer to an index register in instructions like `mov eax,[ebx]`. But this should be done only if there is experimental evidence that it prevents a register read stall, because the instruction `mov eax,[ebx*1]` is five bytes longer than `mov eax,[ebx]` (four bytes for a base address of zero, and one SIB byte).

Other methods for removing register read stalls are to minimize the number of registers that are often read from but rarely written to, replacing read-only registers by constants or memory operands, and to organize the code so as to limit the distance between writing to a register and subsequently reading from the same register. The stack pointer, frame pointer and 'this' pointer are common examples of registers that are often read but rarely modified.

8.9 Execution units

The execution units of the Core2 have been expanded a lot over previous processors. There are six execution ports. Port 0, 1 and 5 are for arithmetic and logic operations (ALU), port 2 for memory read, port 3 for write address calculation, and port 4 for memory write data. This gives a maximum throughput of six unfused µops per clock cycle.

All execution ports support full 128 bit vectors. Most ALU operations have a latency of 1 clock cycle. The different units are listed in table 8.1 below. All three ALU ports can handle 128-bit moves and Boolean operations. All three ports can handle additions on general purpose registers. Port 0 and 5 can also handle integer vector additions.

There are separate units for integer multiplication and floating point multiplication. The integer multiplier on port 1 is fully pipelined with a latency of 3 and a throughput of 1 full vector operation per clock cycle. The floating point multiplier on port 0 has a latency of 4 for single precision and 5 for double and long double precision. The throughput of the floating point multiplier is 1 operation per clock cycle, except for long double precision on Core2. The floating point adder is connected to port 1. It has a latency of 3 and is fully pipelined.

Integer division uses the floating point division unit. This is the only unit that is not pipelined.

The jump unit on port 5 handles all jump and branch operations, including the macro-fused compare-and-branch operations.

The floating point unit connected to port 0 and 1 handles all operations on the floating point stack registers and most floating point calculations on XMM registers. The Core2 makes no distinction between integer and floating point operands, while the Core Nehalem does. For example, `MOVAPS`, `MOVAPD` and `MOVDQA` are identical and all carried out by the integer unit on Core2. On Core Nehalem, `MOVAPS` and `MOVAPD` are different from `MOVDQA` and executed on port 5 only. Floating point XMM move, Boolean, and most floating point shuffle operations are done by the integer units on Core2 but use a dedicated unit on port 5 on the Core Nehalem.

The arithmetic/logic execution units are well distributed between port 0, 1 and 5. This makes it possible to execute three vector instructions per clock cycle, for example floating point vector multiplies on port 0, floating point vector additions on port 1, and a floating point moves on port 5.

Execution port	**Execution unit**	**Subunit**	**Max data size, bits**	**Latency, clocks**	**Processor**
0	int	move	128	1	
1	int	move	128	1	
5	int	move	128	1	
0	int	add	128	1	
1	int	add	64	1	Core 2 only
5	int	add	128	1	
0	int	Boolean	128	1	
1	int	Boolean	128	1	
5	int	Boolean	128	1	
1	int	multiply	128	3	
0	int	shift	128	1	Core 2 only
1	int	shift	128	1	Nehalem only
5	int	shift	64	1	Core 2 only
0	int	pack	128	1	
1	int	pack	64	1	Nehalem only
5	int	pack	128	1	Nehalem only
1	int	shuffle	128	1	Nehalem only
5	int	shuffle	128	1	
5	int	jump	64	1	
0	float	fp stack move	80	1	
1	float	fp add	128	3	
0	float	fp mul	128	4-5	
0	float	fp div and sqrt	128	> 5	
0	float	fp convert	128	1	
1	float	fp convert	128	3	
5	float	fp mov, shuffle	128	1	Nehalem only
5	float	fp boolean	128	1	Nehalem only
2	int	memory read	128	2	
3	store	store address	64	1	
4	store	store data	128	3	

Table 8.1. Execution units in Core2 and Nehalem

The latency for integer vector operations is the same as for operations in general purpose registers. This makes it convenient to use MMX registers or XMM registers for simple integer operations when you are out of general purpose registers. The vector operations are supported by fewer execution ports, though.

Data bypass delays on Core2

On the Core2, there is an extra latency of one clock cycle when the output of a µop in the integer unit is used as input for a µop in the floating point unit, or vice versa. This is illustrated in the following example.

```
; Example 8.2a. Bypass delays in Core 2
.data
align 16
signbits label xmmword        ; Used for changing sign
dq 2 dup (8000000000000000H)  ; Two qwords with sign bit set

.code
movaps    xmm0, [a]           ; Unit = int,   Latency = 2
mulpd     xmm0, xmm1          ; Unit = float, Latency = 5 + 1
xorps     xmm0, [signbits]    ; Unit = int,   Latency = 1 + 1
addpd     xmm0, xmm2          ; Unit = float, Latency = 3 + 1
```

In example 8.2a there are three additional latencies for moving data back and forth between the integer and floating point units. This code can be improved by reordering the instructions so that the number of switches between the integer and floating point units is reduced:

```
; Example 8.2b. Bypass delays in Core 2
.code
movaps    xmm0, [a]           ; Unit = int,   Latency = 2
xorps     xmm0, [signbits]    ; Unit = int,   Latency = 1
mulpd     xmm0, xmm1          ; Unit = float, Latency = 5 + 1
addpd     xmm0, xmm2          ; Unit = float, Latency = 3
```

In example 8.2b, we are changing the sign of `XMM0` before multiplying with `XMM1`. This reduces the number of transitions between the integer and floating point units from three to one, and the total latency is reduced by 2. (We have used `MOVAPS` and `XORPS` instead of `MOVAPD` and `XORPD` because the former instructions are shorter but have the same functionality).

The load/store unit is closely connected with the integer unit, so that there is no additional latency when transferring data between the integer unit and the load/store unit. There is a one clock latency when transferring data from memory (load unit) to the floating point unit, but there is no additional latency when transferring data from the floating point unit to memory (store unit). The execution units are listed in the tables in manual 4: "Instruction tables" where appropriate.

Data bypass delays on Nehalem

On the Nehalem, the execution units are divided into five "domains":

- The integer domain handles all operations in general purpose registers.
- The integer vector (SIMD) domain handles integer operations in vector registers.
- The FP domain handles floating point operations in XMM and x87 registers.
- The load domain handles all memory reads.
- The store domain handles all memory stores.

There is an extra latency of 1 or 2 clock cycles when the output of an operation in one domain is used as input in another domain. These so-called bypass delays are listed in table 8.2.

	To domain			
From domain	integer	integer vector	FP	store
integer	0	1	2	0
integer vector	1	0	2	1
FP	2	2	0	1
load	0	1	2	0

Table 8.2. Bypass delays in Nehalem

Several XMM instructions have multiple versions for the integer vector domain and the FP domain respectively. For example, for a register-to-register move, there is MOVDQA in the integer vector domain, and MOVAPS and MOVAPD in the FP domain. The extra latency for using an instruction in the wrong domain is considerable, as shown in the following example.

```
; Example 8.3a. Bypass delays in Nehalem
movaps    xmm0, [a]          ; Load domain,   Latency = 2
mulps     xmm0, xmm1         ; FP domain,     Latency = 4 + 2
pshufd    xmm2, xmm0, 0      ; int vec. dom., Latency = 1 + 2
addps     xmm2, xmm3         ; FP domain,     Latency = 3 + 2
pxor      xmm2, xmm4         ; int vec. dom., Latency = 1 + 2
movdqa    [b],  xmm1         ; Store domain,  Latency = 3 + 1
```

The bypass delays in this example can be reduced by using only instructions in the same domain as far as possible:

```
; Example 8.3b. Bypass delays in Nehalem
movaps    xmm0, [a]          ; Load domain,   Latency = 2
mulps     xmm0, xmm1         ; FP domain,     Latency = 4 + 2
movaps    xmm2, xmm0         ; FP domain,     Latency = 1 + 0
shufps    xmm2, xmm2, 0      ; FP domain,     Latency = 1 + 0
addps     xmm2, xmm3         ; FP domain,     Latency = 3 + 0
xorps     xmm2, xmm4         ; FP domain,     Latency = 1 + 0
movaps    [b],  xmm1         ; Store domain,  Latency = 3 + 1
```

Replacing PSHUFD with SHUFPS in example 8.3b requires an extra MOVAPS with a latency of 1 (if the value in XMM0 is needed later), but it saves 4 clock cycles in bypass delays. Replacing PXOR with XORPS is straightforward because these two instructions are functionally identical. Replacing the last MOVDQA with MOVAPS has no influence on latencies, but it may have on future processors.

The important conclusion here is that there is a penalty in terms of latency to using an XMM instruction of the wrong type on the Nehalem. On previous Intel processors there is no penalty for using move and shuffle instructions on other types of operands than they are intended for.

The bypass delay is important in long dependency chains where latency is a bottleneck, but not where it is throughput rather than latency that matters. In fact, the throughput may actually be improved by using the integer vector versions of the move and Boolean instructions, which have a throughput of 3 instructions per clock cycle, where the FP move and Boolean instructions have a throughput of only 1 instruction per clock cycle.

There is still no extra bypass delay for using load and store instructions on the wrong type of data. For example, it can be convenient to use MOVHPS on integer data for reading or writing the upper half of an XMM register.

Mixing µops with different latencies

There is a problem when µops with different latencies are issued to the same execution port. For example:

```
; Example 8.4. Mixing uops with different latencies on port 0
mulpd xmm1,xmm2    ; Double precision multiply has latency 5
mulps xmm3,xmm4    ; Single precision multiply has latency 4
```

Assume that the double precision multiplication with a latency of 5 starts at time T and ends at time T+5. If we attempt to start the single precision multiplication with a latency of 4 at time T+1 then this would also end at time T+5. Both instructions use port 0. Each execution port has only one write-back port, which can handle only one result at a time. It is therefore not possible to end two instructions at the same time. The scheduler will predict and avoid this write-back conflict by delaying the latter instruction to time T+2 so that it ends at time T+6. The cost is one wasted clock cycle.

This kind of conflict can occur when two or more µops issued to the same execution port have different latencies. The maximum throughput of one µop per clock cycle in each port can only be obtained when all µops that go to the same port have the same latency. In the example of floating point multiplications we can maximize the throughput by using the same precision for all floating point calculations, or by keeping floating point multiplications with different precisions apart from each other rather than mixing them.

The designers have tried to reduce this problem by standardizing µop latencies. Port 0 can handle only µops with a latency of 1 or ≥ 4. Port 1 can handle only µops with a latency of 1 or 3. Port 5 can handle only latency 1. Port 2, 3 and 4 handle memory operations and almost nothing else. There are no µops that use 2 clock cycles in any execution unit. (Instructions like `MOVD EAX,XMM0` have a latency of 2, but this in 1 clock cycle in the execution unit and 1 extra cycle for bypass delay between units).

The problem with mixing latencies can also occur at port 1, but less frequently:

```
; Example 8.5. Mixing uops with different latency on port 1 (Nehalem)
imul eax,10        ; Port 1. Latency 3
lea  ebx,[mem1]    ; Port 1. Latency 1
lea  ecx,[mem2]    ; Port 1. Latency 1
```

In example 8.5, we cannot issue the last `LEA` µop to port 1 two clock cycles after the `IMUL` µop because they would both finish at the same time on Nehalem (Core 2 has the `LEA` on port 0). This problem occurs only rarely on port 1 because most of the single-clock µops that can go to port 1 can also go to port 0 or 5.

8.10 Retirement

The retirement station seems to be more efficient than in the PM. I have not detected any delays due to bottlenecks in the retirement station on the Core2 or Nehalem.

8.11 Partial register access

There are three different ways that the Core2 and Nehalem uses for resolving writes to part of a register. These three different ways are used for general purpose registers, the flags register, and XMM registers, respectively.

Partial access to general purpose registers

Different parts of a general purpose register can be stored in different temporary registers in order to remove false dependences. For example:

```
; Example 8.6. Partial registers
mov  al, [esi]
inc  ah
```

Here, the second instruction does not have to wait for the first instruction to finish because `AL` and `AH` can use different temporary registers. `AL` and `AH` are stored into each their part of the permanent `EAX` register when the µops retire.

A problem occurs when a write to a part of a register is followed by a read from the whole register:

```
; Example 8.7. Partial register problem
mov  al,  1
mov  ebx, eax
```

This problem is solved by inserting an extra µop to join the different parts of the register. I assume that the extra µops are generated in the ROB-read stage. In the above example, the ROB-read will generate an extra µop that combines `AL` and the rest of `EAX` into a single temporary register before the `MOV EBX,EAX` instruction. This takes 2 - 3 extra clock cycles in the ROB-read stage, but this is less than the 5-6 clock penalty of partial register stalls on processors that don't have this mechanism.

Writes to the high 8-bit registers `AH`, `BH`, `CH`, `DH` generate two extra µops, while writes to the low 8-bit or 16-bit part of a register generate one extra µop. See page 99 for examples.

Partial flags stall

Unfortunately, the Core2 and Nehalem don't generate extra µops to prevent stalls on the flags register. Therefore, there is a stall of approximately 7 clock cycles when reading the flags register after an instruction that modifies part of the flags register. See page 100 for examples.

There is also a partial flags stall when reading the flags after a rotate instruction or a shift instruction with a count of `CL`. See page 100 for details.

Partial access to XMM registers

An XMM register is never split into its parts in the reorder buffer. Therefore, no extra µops are needed and there is no partial access stall when writing to part of an XMM register. But the write has a false dependence on the previous value of the register. Example:

```
; Example 8.8. Partial access to XMM register
mulss xmm0,   xmm1
movss [mem1], xmm0
movss xmm0,   xmm2  ; has false dependence on previous value
addss xmm0,   xmm3
```

The `MOVSS` and `MOVSD` instructions with register operands write to part of the destination register and leave the rest of the register unchanged. In example 8.8, the `MOVSS XMM0,XMM2` instruction has a false dependence on the preceding `MULSS` instruction because the lower 32 bits of the register cannot be separated from the unused upper part of the register. This prevents out-of-order execution. The false dependence in example 8.8 can be removed by replacing `MOVSS XMM0,XMM2` with `MOVAPS XMM0,XMM2`. Do not use the `MOVSS` and `MOVSD` instructions with two register operands unless it is necessary to leave the rest of the register unchanged.

8.12 Store forwarding stalls

The processor can forward a memory write to a subsequent read from the same address under certain conditions. This store forwarding will fail in most cases of misaligned or partial memory references as in previous processors (p. 84), but certain special cases have been dealt with to allow store forwarding of partial memory operands. A failed store forwarding will delay the subsequent read by approximately 10 clock cycles.

Store forwarding works if a write to memory is followed by a read from the same address when the read has the same operand size and the operand has its natural alignment:

```
; Example 8.9. Successful store-to-load forwarding
mov dword ptr [esi], eax        ; esi aligned by 4
mov ebx, dword ptr [esi]        ; No stall
```

The store forwarding also works with misaligned memory operands if the operand is less than 16 bytes and does not cross a 64-bytes boundary on 45 nm Core2, or an 8-bytes boundary on 65 nm Core2. Store forwarding works in all cases of misaligned memory operands on Nehalem.

```
; Example 8.10. Failed store forward because of misalignment on Core2
mov dword ptr [esi-2], eax     ; esi divisible by 64
mov ebx, dword ptr [esi-2]     ; Stall because 64 B boundary crossed
```

Store forwarding never works if the read has a bigger operand size than the preceding write:

```
; Example 8.11. Failed store forwarding when read bigger than write
mov dword ptr [esi], eax       ; Write lower 4 bytes
mov dword ptr [esi+4], edx     ; Write upper 4 bytes
movq mm0, qword ptr [esi]      ; Read 8 bytes. Stall
```

Store forwarding is possible if the read has a smaller operand size than the write and starts at the same address, and the write operand does not cross a 64-bytes boundary on 45 nm Core2 or the read operand does not cross an 8 bytes boundary on 65 nm Core2. On Nehalem there is no restriction on boundary crossing here.

```
; Example 8.12. Store forwarding to smaller read
mov dword ptr [esi], eax       ; Write 4 bytes
mov bx, word ptr [esi]         ; Successful store forwarding
mov cx, word ptr [esi+2]       ; Stall because not same start address
```

There are a few special cases where store forwarding is possible to a smaller read with a different start address. These special cases are:

(1) An 8-byte write can be followed by 4-byte reads of each of its halves if the read does not cross an 8-bytes boundary on 65 nm Core2 or the write does not cross a 64-bytes boundary on 45 nm Core2. On Nehalem there is no restriction on boundary crossing here.

(2) A 16-byte write can be followed by 8-byte reads of each of its halves and/or 4-byte reads of each of its quarters if the write is aligned by 16. On Nehalem there is no restriction on alignment or boundary crossing here.

```
; Example 8.13. Store forwarding in special case
movapd xmmword ptr [esi], xmm0 ; Write 16 bytes
fld qword ptr [esi]            ; Read lower half. Success
fld qword ptr [esi+8]          ; Read upper half. Success
mov eax, dword ptr [esi+12]    ; Read last quarter. Success
mov ebx, dword ptr [esi+2]     ; Not a quarter operand. Fail
```

The mechanism for detecting whether store forwarding is possible does not distinguish between different memory addresses with the same set-value in the cache. This can cause stalls for failed bogus store forwardings when addresses are spaced a multiple of 4 kb apart:

```
; Example 8.14. Bogus store forwarding stall
mov word ptr [esi], ax
mov ebx, dword ptr [esi+800h]  ; No stall
mov ecx, dword ptr [esi+1000h] ; Bogus stall
```

In example 8.14 there is a stall when reading `ecx` after writing `ax` because the memory addresses have the same set-value (the distance is a multiple of 1000h) and a large read after a small write would give a stall if the addresses were the same.

8.13 Cache and memory access

Cache	Core 2	Nehalem
Level 1 code	32 kB, 8 way, 64 B line size, latency 3, per core	32 kB, 8 way, 64 B line size, latency 4, per core
Level 1 data	32 kB, 8 way, 64 B line size, latency 3, per core	32 kB, 8 way, 64 B line size, latency 4, per core
Level 2	2, 4 or 6 MB, 16 or 24 way, 64 B line size, latency 15, shared	256 kB, 8 way, 64 B line size, latency 11, per core
Level 3	none	8 MB, 16 way, 64 B line size, latency 38, shared

Table 8.3. Cache sizes on Core 2 and Nehalem

There is one cache on each core, except for the last-level cache. All caches are shared between threads where a core can run two threads. It is likely that there will be more versions in the future with different last-level cache sizes. There is a 256 bit data path between the level-1 and level-2 caches.

The capability of reordering memory accesses is allegedly improved so that a memory read can be executed speculatively before a preceding write that is expected to go to a different address before the address is known with certainty.

The data prefetchers are able to automatically prefetch two data streams with different strides for both level-1 and level-2 caches.

Cache bank conflicts

Each 64-bytes line in the data cache is divided into 4 banks of 16 bytes each. It is not possible to do a memory read and a memory write in the same clock cycle if the two memory addresses have the same bank number, i.e. if bit 4 and 5 in the two addresses are the same on Core2. Example:

```
; Example 8.15. Core2 cache bank conflict
mov  eax, [esi]       ; Use bank 0, assuming esi is divisible by 40H
mov  [esi+100H], ebx  ; Use bank 0. Cache bank conflict
mov  [esi+110H], ebx  ; Use bank 1. No cache bank conflict
```

The Nehalem doesn't have these conflicts, but Core 2 and Nehalem both have a false dependence between memory addresses with the same set and offset, i.e. with a distance that is a multiple of 4 kB.

Misaligned memory accesses

The Core2 has a penalty for misaligned memory access when a cache line boundary (64 bytes) is crossed. The penalty is approximately 12 clock cycles for a misaligned read and 10 clock cycles for a misaligned write. The Nehalem has hardly any penalty for misaligned memory access.

8.14 Breaking dependency chains

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EBX,EBX`. The Core2 and Nehalem processors recognize that certain instructions are independent of the prior value of the register if the source and destination registers are the same.

This applies to all of the following instructions: `XOR`, `SUB`, `PXOR`, `XORPS`, `XORPD`, and all variants of `PSUBxxx` and `PCMPxxx` except `PCMPEQQ`.

The following instructions are not recognized as being independent when source and destination are the same: `SBB`, `CMP`, `PANDN`, `ANDNPS`, `ANDNPD`.

Floating point subtract and compare instructions are not truly independent when source and destination are the same because of the possibility of NAN's etc.

These instructions are useful for breaking an unnecessary dependence, but only on processors that recognize this independence.

8.15 Multithreading in Nehalem

Thread synchronization primitives, e.g. the `LOCK XCHG` instruction, are considerably faster than on previous processors.

The Nehalem can run two threads in each of its four cores. This means that each thread gets only half of the resources. The resources are shared in the following way between two threads running in the same core:

Cache: All cache resources are shared competitively between the threads. The more one thread uses the less the other thread can use.

Branch target buffer and branch history pattern table: These are shared competitively between threads.

Instruction fetch and decoding: The instruction fetcher and decoders are shared evenly between the two threads so that each thread gets every second clock cycle.

Loop buffer: There is one loop buffer for each thread.

Register renaming and register read ports. These are shared evenly so that each thread gets every second clock cycle. Register read stalls in one thread are independent of register reads in the other thread.

Reorder buffer and reservation station. These are shared competitively.

Execution ports and execution units. These are shared competitively. One thread can use one execution port while another thread uses another port.

Read and write buffers: These are shared competitively.

Permanent register file: There is one for each thread.

It is clear that there is no advantage to running two threads per core if any of the shared resources are limiting factors for the performance. In many cases, however, the execution resources are more than sufficient for a single thread. It can be particularly advantageous to run two threads per core in situations where a large fraction of the time goes to cache misses and branch misprediction. However, if any of the shared resources are bottlenecks then it is not advantageous to run two threads per core. On the contrary, each thread is likely to run at less than half the single-thread speed because of evictions in the cache and branch target buffers and other resource conflicts. There is no way to give one thread higher priority than the other in the CPU.

8.16 Bottlenecks in Core2 and Nehalem

Instruction fetch and predecoding

All parts of the pipeline in the Core2 and Nehalem design have been improved over the PM design so that the total throughput is increased significantly. The part that has been improved the least is instruction fetch and predecoding. This part cannot always keep up with the speed of the execution units. Instruction fetch and predecoding is therefore the most likely bottleneck in CPU-intensive code.

It is important to avoid long instructions in order to optimize instruction fetch and predecoding. The optimal average instruction length is approximately 3 bytes, which can be impossible to obtain.

Instruction fetch and predecoding is not a bottleneck in a loop that fits into the loop buffer. The performance of a program can therefore be improved if the innermost loop is sufficiently small for fitting into the loop buffer, or if it can be split into multiple smaller loops that each fit into the loop buffer.

Instruction decoding

The decoders can handle four instructions per clock cycle, or five in the case of macro-op fusion. Only the first one of the four decoders can handle instructions that generate more than one µop. The minimum output of the decoders is therefore 2 µops per clock cycle in the case that all instructions generate 2 µops each so that only the first decoder can be used. Instructions may be ordered according to the 4-1-1-1 pattern for optimal decoder throughput.

Fortunately, most of the instructions that generated multiple µops in previous designs generate only a single µop on the Core2 and Nehalem thanks to improved µop fusion, the stack engine, and the 128-bit width of buses and execution units. The decoders will generate four µops per clock cycle in an instruction stream where all of the instructions generate only a single µop each. This matches the throughput of the rest of the pipeline. Decoder throughput is therefore only critical if some of the instructions generate two µops each.

Length-changing prefixes cause long delays in the decoders. These prefixes should be avoided at all costs, except in small loops that fit into the loop buffer. Avoid instructions with 16-bit immediate operands in 32-bit and 64-bit mode.

Register read stalls

The number of register read ports on the permanent register file is insufficient in many situations. Register read stalls is therefore a very likely bottleneck.

Avoid having more than two or three registers that are often read but seldom written to in the code. The stack pointer, frame pointer, '`this`' pointer, and loop-invariant expressions that are stored in a register are likely contributors to register read stalls. Loop counters and other registers that are modified inside a loop may also contribute to register read stalls if the loop uses more than 5 clock cycles per iteration.

Execution ports and execution units

The capacity of the execution ports and execution units is quite high. Many µops have two or three execution ports to choose between and each unit can handle one full 128-bit vector operation per clock cycle. The throughput of the execution ports is therefore less likely to be a bottleneck than on previous designs.

Execution ports can be a bottleneck if the code generates many µops that all go to the same execution port. Memory operations can be a bottleneck in code that contains many memory accesses because there is only one memory read port and one memory write port.

Most execution units are pipelined to a throughput of one µop per clock cycle. The most important exceptions are division and square root.

Execution latency and dependency chains

Execution latencies on the Core2 and Nehalem are generally low. Most integer ALU operations have a latency of only one clock cycle, even for 128-bit vector operations. There is an additional latency of 1-2 clock cycles for moving data between the integer unit and the floating point unit. The execution latencies are critical only in long dependency chains. Long dependency chains should be avoided.

Partial register access

There is a penalty for reading a full register after writing to a part of the register. Use `MOVZX` or `MOVSX` to read 8-bit or 16-bit memory operands into a 32-bit register rather than using a smaller part of the register.

Retirement

The retirement of µops has not been observed to be a bottleneck in any of my experiments.

Branch prediction

The branch prediction algorithm is good, especially for loops. Indirect jumps can be predicted. Unfortunately, the branch history pattern table on the Core2 is so small that branch mispredictions are quite common.

A special branch target buffer for branches with loop behavior has only 128 entries which may be a bottleneck for a program with many critical loops.

Memory access

The cache bandwidth, memory bandwidth and data prefetching are significantly better than on previous processors.

Memory bandwidth is still a likely bottleneck, of course, for memory-intensive applications.

Literature

"Intel 64 and IA-32 Architectures Optimization Reference Manual". Intel 2009.
Jeff Casazza: "First the Tick, Now the Tock: Intel® Microarchitecture (Nehalem)". White Paper. Intel 2009.
Ofri Wechsler: "Inside Intel Core Microarchitecture: Setting New Standards for Energy-Efficient Performance". White Paper. Intel 2006.
David Kanter: "Inside Nehalem: Intel's Future Processor and System". www.realworldtech.com, 2008.

9 Sandy Bridge and Ivy Bridge pipeline

Intel's microarchitecture code named Sandy Bridge is a further development of the Core 2 and Nehalem design. A new µop cache has been added after the decoders, and the floating point execution units have been expanded from 128 bits to 256 bits.

The Sandy Bridge has 2 - 8 cores, and some versions are capable of running two threads in each core.

The new AVX instruction set is supported. This expands the sixteen 128-bit XMM registers to 256-bit YMM registers for floating point vector operations. Most XMM and YMM instructions have non-destructive three-operand versions in the AVX instruction set.

9.1 Pipeline

The Sandy Bridge and Ivy Bridge pipeline is very similar to the Core2 and Nehalem, but with an added µop cache after the decoders.

The reorder buffer has 168 entries, and the reservation station has 54 entries, according to Intel publications.

9.2 Instruction fetch and decoding

The pre-decoders and decoders can handle 16 bytes or 4 instructions per clock cycle, or 5 instructions in case of macro-op fusion. It appears to be almost identical to the decoders in the Core2 and Nehalem.

Instructions with any number of prefixes are decoded in a single clock cycle.

The penalty for length-changing prefixes (see page 105) has been removed in some cases, while a few cases remain:

- Move instructions with an immediate operand using an operand size prefix, e.g. `mov ax,1234` has no penalty.
- Arithmetic and logic instructions with an immediate operand using an operand size prefix, e.g. `add ax,1234` has a penalty of 2-3 clock cycles in the decoders, regardless of alignment. This applies to all arithmetic and logic instructions with a 16-bit immediate constant as operand in 32-bit or 64-bit mode.
- The instructions `NEG`, `NOT`, `DIV`, `IDIV`, `MUL` and `IMUL` with a single 16-bit operand no penalty.
- An address size prefix does not cause a penalty, even if it changes the length of the instruction.

There are four decoders, which can handle instructions generating one or more µops according to certain patterns. The following instruction patterns were successfully decoded in a single clock cycle in my experiments:

- 1-1-1-1
- 2-1-1
- 3
- 4

Instructions that generate 3 or 4 µops are decoded alone. Instructions that generate more than four µops are handled by microcode which is less efficient.

The multi-byte `NOP` instruction with opcode `0F 1F` can only be decoded at the first of the four decoders on Sandy Bridge, while a simple `NOP` with extra prefixes (opcode `66 66 90`) can be decoded at any of the four decoders. The Ivy Bridge does not have this limitation. Both types of long NOPs are decoded at a rate of four per clock on Ivy Bridge.

9.3 µop cache

The Sandy Bridge and Ivy Bridge have a cache for decoded µops after the decoders. This is useful because the limitation of 16 bytes per clock cycle in the fetch/decode units is a serious bottleneck if the average instruction length is more than four bytes. The throughput is doubled to 32 bytes per clock for code that fits into the µop cache.

The µop cache is organized as 32 sets × 8 ways × 6 µops, totaling a maximum capacity of 1536 µops. It can allocate a maximum of 3 lines of 6 µops each for each aligned and contiguous 32-bytes block of code.

Code that runs out of the µop cache are not subject to the limitations of the fetch and decode units. It can deliver a throughput of 4 (possibly fused) µops or the equivalent of 32 bytes of code per clock cycle.

The µop cache is rarely used to the maximum capacity of 1536 µops. The utilization is often less than optimal for the following reasons:

- A µop cache line is assigned to a specific 32-bytes block of code. A new µop cache line will be started every time a 32-bytes boundary in the code is passed, even if the previous µop cache line is only partially filled.

- Instructions that generate multiple µops cannot be split between two µop cache lines. If an instruction cannot be fully contained in the current line then the rest of this line will be unused and the instruction will be placed at the start of a new line.

- Instructions that generate more than 4 µops use microcode ROM. Such instructions use an entire µop cache line.

- An unconditional jump or call always ends a µop cache line

- The same piece of code can have multiple entries in the µop cache if it has multiple jump entries.

- Instructions that require more than 32 bits of storage may take up two entries in the µop cache and may take an extra clock cycle to load. The details are given below.

- It cannot load more than one µop cache line per clock cycle. This may be a bottleneck if many instructions use two entries each.

- A 32-bytes block of code which generates more than 18 µops or which would require more than 18 entries in the µop cache will not be allocated in the µop cache.

- The pipeline switches frequently between taking µops from the decoders and from the µop cache. Each switch may cost one clock cycle.

Each entry in the µop cache has 32 bits of storage space for address and data bits. A µop may need more than 32 bits of storage space, for example if it has both a memory operand and an immediate data operand. The system uses various methods for dealing with this problem. The 32 bits of storage space may be split into two 16-bit fields, one for address bits and one for immediate data bits. A 32-bit address may be stored as a 16-bit sign-extended number if it is in the range from -2^{15} to $+2^{15}$. Likewise, a 32-bit immediate data value may be stored as a 16-bit sign-extended number if it is in the range from -2^{15} to $+2^{15}$, and a 64-bit immediate data value may be stored as a 32-bit sign-extended number if it is in the range from -2^{31} to $+2^{31}$. However, a 64-bit address field cannot be converted to a 32-bit sign-extended number.

If the necessary data cannot be squeezed into a 32-bit number or two 16-bit numbers, then it will borrow unused data space from another entry in the same µop cache line if possible. If this is not possible, then it will use two entries in the µop cache.

A µop cache line takes an extra clock cycle to load if it has one or more entries that use extra data space, regardless of whether the extra storage space is borrowed from another entry or uses an extra entry, except for instructions with RIP-relative addresses.

Instructions with RIP-relative addresses behave slightly differently from other instructions. A disadvantage is that the 32-bits storage space cannot be split into two 16-bit fields. On the other hand, it has the advantage that it does not take an extra clock cycle to load if it uses extra data space for storing an immediate operand.

The details are given in table 9.1 below, according to my tests.

Address mode	**Address or offset bits**	**Immediate data bits**	**µop cache entries**	**µop cache load time**
no memory operand	0	0, 8, 16, 32	1	1
no memory operand	0	64small	1	1
no memory operand	0	64	2	2
32 bit absolute	32small	0, 8, 16, 32small	1	1
32 bit absolute	32small	32	2	2
32 bit absolute	32	0	1	1
32 bit absolute	32	8, 16, 32	2	2
pointer, index or both	0, 8, 16, 32small	0, 8, 16, 32small	1	1
pointer, index or both	0, 8, 16, 32small	32	2	2
pointer, index or both	32	0	1	1
pointer, index or both	32	8, 16, 32	2	2
rip relative	32small, 32	0	1	1
rip relative	32small, 32	8, 16, 32	2	1
64 bit absolute	64small, 64	0	2	2

Table 9.1. Size of different instructions in µop cache
Note: 32small means a 32-bit number in the range from -2^{15} to $+2^{15}$, 64small means a 64-bit number in the range from -2^{31} to $+2^{31}$.

Examples:

```
; Example 9.1. Instructions in µop cache
mov dword [rsi+4], 1000H         ; one entry
mov dword [rsi+4], 10000H        ; two entries
mov dword [rsi+40000H], 0        ; two entries
mov dword [rsi+40000H], eax      ; one entry
cmp dword fs:[8], 2              ; one entry
cmp dword fs:[8], 20000H         ; two entries
cmp dword fs:[80000H], 2         ; two entries
mov rax,-10000000H               ; one entry (even if long form)
mov rax,-100000000H              ; two entries
vinsertf128 ymm0,ymm1,[rip+x],1  ; two entries
```

The gain in performance due to the µop cache can be quite considerable if the average instruction length is more than 4 bytes. The following methods of optimizing the use of the µop cache may be considered:

- Make sure that critical loops are small enough to fit into the µop cache.

- Align the most critical loop entries and function entries by 32.

- Avoid unnecessary loop unrolling.

- Avoid instructions that have extra load time according to table 9.1.

- Instructions that require extra data space according to table 9.1 may be mixed with instructions that use no data space so that the vacant data space can be borrowed.

- If a 32-bytes block of code that is part of a critical loop generates more than 18 µops or does not fit into three µop cache lines then it may be useful to reorganize the code or make some instructions longer if this can make it fit into three µop cache lines. This will avoid the cost of switching between µop cache and decoder.

9.4 Loopback buffer

The 28 µop loop buffer of the Nehalem (see page 104) is preserved in Sandy Bridge and Ivy Bridge. The loop buffer is placed after the µop cache, but it can also receive µops from the decoders in case of misses in the µop cache. The loop buffer increases the performance of tiny loops that do not perform well in the µop cache for whatever reason. The loop buffer has no measurable effect in the cases where the µop cache is not a bottleneck, which is, in fact, most cases.

It is advantageous to make loops smaller than 28 µops in order to take advantage of the loop buffer.

9.5 Micro-op fusion

The processor uses µop fusion in the same way as previous processors. Some instructions that need two µops in the execution units can use the µop fusion technique to keep these two µops together as one through most of the pipeline in order to save pipeline bandwidth. See page 90 and 106.

You can see which instructions use µop fusion by looking at the tables in manual 4: "Instruction tables". Instructions with µop fusion has a higher number of µops listed under "unfused domain" than under "fused domain".

9.6 Macro-op fusion

The Sandy Bridge and Ivy Bridge can fuse two instructions into one µop in more cases than previous processors can (see page 107).

The decoders will fuse an arithmetic or logic instruction and a subsequent conditional jump instruction into a single compute-and-branch µop in certain cases. The compute-and-branch µop is not split in two at the execution units but executed as a single µop by the branch unit at execution port 5.

The `CMP`, `ADD` and `SUB` instructions can fuse with signed and unsigned branch instructions. `INC` and `DEC` can fuse with signed branch instructions, and `TEST` and `AND` instructions can fuse with all branch instructions (including useless combinations), as indicated in the following table:

First instruction	**can pair with these (and the inverse)**	**cannot pair with**

cmp	jz, jc, jb, ja, jl, jg	js, jp, jo
add, sub	jz, jc, jb, ja, jl, jg	js, jp, jo
adc, sbb	none	
inc, dec	jz, jl, jg	jc, jb, ja, js, jp, jo
test	all	
and	all	
or, xor, not, neg	none	
shift, rotate	none	

Table 9.2. Instruction fusion

The first instruction can have an immediate operand or a memory source operand, but not both. It cannot have a memory destination operand. It cannot have a RIP-relative memory operand. Examples:

```
; Example 9.2. Instruction fusion
dec ecx
jnz L1                ; Fusion is possible
cmp dword ptr [esi], 0
je  L2                ; No fusion. Both memory operand and immediate
dec dword ptr [esi]
jnz L3                ; No fusion. Memory destination operand
add eax, ebx
jo  L4                ; No fusion. See table 9.2
cmp eax,[mem]         ; Will use RIP-relative address in 64-bit mode
jg  L5                ; Fusion only in 32 bit mode
```

The `JECXZ` and `LOOP` instructions cannot be fused. The instruction fusion works even if instructions cross a 16-bytes boundary on the Ivy Bridge. It is uncertain whether this also applies to the Sandy Bridge.

If more than one fuseable instruction pair reaches the four decoders in the same clock cycle then only the first pair is macro-fused.

The programmer should keep fuseable arithmetic instructions together with a subsequent conditional jump rather than scheduling other instructions in-between; and there should preferably be at least three other instructions between one fuseable pair and the next fuseable pair in order to take advantage of macro-op fusion.

Instruction fusion can increase the throughput to a maximum of five instructions per clock cycle. Unfortunately, it can also decrease the throughput. The fuseable arithmetic/logic instructions (`ADD`, `SUB`, `INC`, `DEC`, `CMP`, `AND`, `TEST`) decode at a lower rate than similar non-fuseable instructions (e.g. `OR`). The probable explanation is this: If any of the fuseable arithmetic/logic instructions get into the last of the four decoders then the decoding will be postponed and the instruction will go into the first decoder in the next clock cycle in order to check if the next instruction is a fuseable branch. This means that the decoding throughput is lowered for these instructions for code that doesn't fit into the µop cache, even if the code contains no branches.

9.7 Stack engine

The Sandy Bridge has a dedicated stack engine which works the same way as on previous processors, as described on page 92.

The modification of the stack pointer by `PUSH`, `POP`, `CALL` and `RET` instructions is done by a special stack engine, which is placed immediately after the decoding stage in the pipeline and probably before the µop cache. This relieves the pipeline from the burden of µops that modify the stack pointer. This mechanism saves two copies of the stack pointer: one in the stack engine and another one in the register file and the out-of-order core. These two stack pointers may need to be synchronized if a sequence of PUSH, POP, CALL and RET instructions is followed by an instruction that reads or modifies the stack pointer directly, such as ADD ESP,4 or MOV EAX,[ESP+8]. The stack engine inserts an extra stack-synchronization µop in every case where synchronization of the two stack pointers is needed. See page 92 for a more detailed explanation.

9.8 Register allocation and renaming

All integer, floating point, MMX, XMM, YMM, flags and probably also segment registers can be renamed. The floating point control word can also be renamed.

Register renaming is controlled by the register alias table (RAT) and the reorder buffer (ROB), shown in figure 6.1. The µops from the decoders and the stack engine go to the RAT via a queue and then to the ROB-read and the reservation station. The RAT can handle 4 µops per clock cycle. The RAT can rename four registers per clock cycle, and it can even rename the same register four times in one clock cycle.

Special cases of independence

A common way of setting a register to zero is by XOR'ing it with itself or subtracting it from itself, e.g. XOR EAX,EAX. The processor recognizes that certain instructions are independent of the prior value of the register if the two operand registers are the same.

This applies to all of the following instructions: XOR, SUB, PXOR, XORPS, XORPD, VXORPS, VXORPD and all variants of PSUBxxx and PCMPGTxx, but not CMP, SBB, PANDN etc.

This register is set to zero at the rename stage by these instructions. The throughput is four of these zeroing instructions per clock cycle, because no execution unit is used.

The PCMPEQxx instructions set all bits to 1 if the two registers are the same. This instruction is recognized as being independent of the prior value of the register, but it does need an execution unit. A zeroing instruction with a 64-bit mmx register also uses an execution unit because of the overlap with the x87 style floating point stack registers.

Instructions that need no execution unit

The abovementioned special cases where registers are set to zero by instructions such as XOR EAX,EAX are handled at the register rename/allocate stage without using any execution unit. This makes the use of these zeroing instructions extremely efficient, with a throughput of four zeroing instructions per clock cycle. The carry flag can be zeroed with CLC in the same efficient way.

Previous processors can handle only the FXCH instruction at the register renaming stage. The Sandy Bridge can handle these special cases of zeroing instructions as well as NOP instructions at the register rename/allocate stage too.

NOP instructions, including multi-byte NOPs are therefore very efficient with a throughput of 4 NOPs per clock cycle. For reasons of efficiency, it is much better to use multi-byte NOPs than the commonly used pseudo-NOPs such as MOV EAX,EAX or LEA RAX,[RAX+0].

Elimination of move instructions

The Ivy Bridge (but not the Sandy Bridge) can eliminate register-to-register moves at the register allocation stage. The following example illustrates this:

```
; Example 9.3. Move elimintaion
add eax,4
mov ebx,eax      ; this move can be eliminated
sub ebx,ecx
```

In this example, the `mov ebx,eax` instruction is likely to be eliminated by register renaming. The physical register that represents the `ebx` input in the third instruction is simply the same as the register that represents the `eax` output value in the first instruction. Register renaming is explained on page 10.

Move elimination is not always successful. It fails when the necessary operands are not ready. But typically, move elimination succeeds in more than 80% of the possible cases. Chained moves can also be eliminated.

Move elimination is possible with all 32-bit and 64-bit general purpose registers and all 128-bit and 256-bit vector registers.

A zero-extended move from an 8-bit register to a 32-bit or 64-bit register can also be eliminated, e.g. `MOVZX EAX,BL`. Zero-extended moves from 16-bit registers cannot be eliminated. No move to 8-bit registers,16-bit registers or mmx registers can be eliminated. Sign-extended moves cannot be eliminated.

A move of a register to itself will never be eliminated. For example `mov eax,eax` is not eliminated.

An eliminated move has zero latency and does not use any execution port. But is does consume bandwidth in the decoders.

9.9 Register read stalls

Register read stalls has been a serious, and often neglected, bottleneck in previous processors since the Pentium Pro. All Intel processors based on the P6 microarchitecture and its successors, the Pentium M, Core and Nehalem microarchitectures have a limitation of two or three reads from the permanent register file per clock cycle.

This bottleneck has now finally been removed in the Sandy Bridge and Ivy Bridge. In my experiments, I have found no practical limit to the number of register reads.

9.10 Execution units

The Sandy Bridge and Ivy Bridge have six execution ports. Port 0, 1 and 5 are for arithmetic and logic operations (ALU). There are two identical memory read ports on port 2 and 3 where previous processors had only one. Port 4 is for memory write. The memory write unit on port 4 has no address calculation. All write operations use port 2 or 3 for address calculation. The maximum throughput is one unfused µop on each port per clock cycle.

Port 0, 1 and 5 support full 256 bit vector operations. Port 2, 3 and 4 use two clock cycles for 256 bit read and write operations. The different units are listed in table 9.3 and 9.4 below.

There are separate ports for multiplication in general purpose registers and vector registers. The general purpose register multiplier on port 1 has a latency of 3. The integer and floating point vector multiplier on port 0 has a latency of 5 for all precisions. These two multipliers can run simultaneously and both are fully pipelined with a throughput of 1 vector operation per clock cycle.

Integer division uses the floating point division unit on port 0. This is the only unit that is not pipelined.

The jump unit on port 5 handles all jump and branch operations, including the macro-fused compute-and-branch operations.

The processor has different execution units for integer and floating point vector operands. For example, the floating point vector moves (`MOVAPS` and `MOVAPD`) are executed on port 5 only, while integer vector moves (`MOVDQA`) can execute on port 0, 1 or 5.

The execution units are well distributed between port 0, 1 and 5. This makes it possible to execute three vector instructions per clock cycle, for example floating point vector multiplies on port 0, floating point vector additions on port 1, and a floating point shuffles on port 5.

Execution port	Data type	Operation	Max data size, bits	Latency, clocks
0	gp and ivec	move	128	1
1	gp and ivec	move	128	1
5	gp and ivec	move	128	1
0	gp and ivec	add	128	1
1	gp	add	64	1
5	gp and ivec	add	128	1
0	gp and ivec	Boolean	128	1
1	gp and ivec	Boolean	128	1
5	gp and ivec	Boolean	128	1
0	ivec	multiply	128	5
1	gp	multiply	64	3
0	gp	shift	64	1
1	ivec	shift	128	1
5	gp	shift	64	1
0	ivec	shuffle, pack	128	1
5	ivec	shuffle, pack	128	1
5	gp	jump	64	1
5	float	fp mov, shuffle	256	1
1	float	fp add	256	3
0	float	fp mul	256	5
0	float	fp div and sqrt	128	10-22
5	float	fp boolean	256	1
2	all	memory read	128	
3	all	memory read	128	
4	all	memory write	128	

Table 9.3. Execution ports in Sandy Bridge
Data types: gp = general purpose registers, ivec = integer vectors, float = floating point registers and floating point vectors.

Execution port	Data type	Operation	Max data size, bits	Latency, clocks
0	gp and ivec	move	128	1
1	gp and ivec	move	128	1
5	gp and ivec	move	128	1
0	gp	add	64	1
1	gp and ivec	add	128	1
5	gp and ivec	add	128	1
0	gp and ivec	Boolean	128	1
1	gp and ivec	Boolean	128	1
5	gp and ivec	Boolean	128	1
0	ivec	multiply	128	5

1	gp	multiply	64	3
0	gp and ivec	shift	128	1
5	gp	shift	64	1
1	ivec	shuffle, pack	128	1
5	ivec	shuffle, pack	128	1
5	gp	jump	64	1
5	float	fp mov, shuffle	256	1
1	float	fp add	256	3
0	float	fp mul	256	5
0	float	fp div and sqrt	128	10-20
5	float	fp boolean	256	1
2	all	memory read	128	
3	all	memory read	128	
4	all	memory write	128	

Table 9.4. Execution ports in Ivy Bridge
Data types: gp = general purpose registers, ivec = integer vectors, float = floating point registers and floating point vectors.

The latency for integer vector operations is the same as for operations in general purpose registers. This makes it convenient to use MMX registers or XMM registers for simple integer operations if there are not enough general purpose registers. The vector operations are supported by fewer execution ports, though.

Read and write bandwidth

The processor has two identical memory read ports (port 2 and 3) where previous Intel processors have only one read port. Port 2 and 3 are also used for address calculation. There is one write port (port 4). Port 4 has no address calculation unit. Instead, all write operations use port 2 or 3 for address calculation. All memory write operations require two µops: one µop for address calculation on port 2 or 3, and one µop for writing on port 4.

Memory operations of 128 bits or less have a throughput of two reads or one read and one write per clock cycle. It is not possible to do two reads and one write per clock cycle because there are only two address calculation units (port 2 and 3).

The situation is different for 256-bit operands in the new YMM registers. Each read port uses two clock cycles for a 256-bit read (but only one µop), and the write port uses two clock cycles for a 256-bit write. Obviously, a read port cannot execute another read µop in the second clock cycle of a 256-bit read, but it can execute an address calculation µop in the second clock cycle, which can be used by the write port. Therefore, it is possible to obtain a throughput of two 256-bit reads and one 256-bit write per two clock cycles.

The risk of cache bank conflicts (see below) is quite high when reading and writing at the maximum memory bandwidth.

Data bypass delays

The execution units are divided into domains in the same way as on the Nehalem, as described on page 112, and there are different register files for integer and floating point values. However, the delays for passing data between the different domains or different types of registers are smaller on the Sandy Bridge and Ivy Bridge than on the Nehalem, and often zero.

Instructions such as `MOVD` that move data between general purpose registers and vector registers have a latency of only 1 clock and no extra bypass delay. Likewise, instructions that move data from vector registers to the integer flags, such as `COMISS` and `PTEST` have no extra bypass delay.

There is often a data bypass delay of 1 clock cycle when passing data from a floating point instruction to an integer shuffle instruction, and 1 more clock cycle for passing the data back again to a floating point instruction, for example when `PSHUFD` is used between two `ADDPS` instructions. The same happens when using a floating point blend instruction on integer data, for example `PADDD` followed by `BLENDPS`. In some cases, there is no bypass delay when using the wrong type of shuffle or Boolean instruction. For example, I found no delay when mixing `PADDD` and `SHUFPS`. There is only rarely a bypass delay when using the wrong type of move instruction, for example `MOVAPS` instead of `MOVDQA`.

There is no extra bypass delay for using load and store instructions on the wrong type of data. For example, it can be convenient to use `MOVHPS` on integer data for reading or writing the upper half of an XMM register.

There is always a data bypass delay of 1 extra clock cycle on instructions that can move data between the upper 128 bits and the lower 128 bits of the YMM registers. For example, the `VINSERTF128` instruction has a latency of 2 clock cycles while other shuffle and blend instructions that do not cross the 128 bits boundary have a latency of 1 clock. The latency of the `VINSERTF128` instruction is independent of whether the data are actually inserted in the lower half or the upper half of the YMM register.

Mixing µops with different latencies

Previous processors have a write-back conflict when µops with different latencies are issued to the same execution port, as described on page 114. This problem is largely solved on the Sandy Bridge. Execution latencies are standardized so that all µops with a latency of 3 are issued to port 1 and all µops with a latency of 5 go to port 0. µops with a latency of 1 can go to port 0, 1 or 5. No other latencies are allowed, except for division and square root.

The standardization of latencies has the advantage that write-back conflicts are avoided. The disadvantage is that some µops have higher latencies than necessary.

256-bit vectors

The execution units on port 0, 1 and 5 have full 256-bit throughput. There appears to be two 128-bit data busses, one for the lower 128 bits and one for the upper 128 bits. There is a delay of one clock cycle for moving data between the upper and the lower 128 bits. The two halves of a 256-bit register are not treated as independent, except in the undesired "saved state", as explained below in chapter 9.12. Apparently the two halves are always transmitted simultaneously when in the "modified state".

Underflow and subnormals

Subnormal numbers (also called denormal) occur when floating point operations are close to underflow. The handling of subnormal numbers is very costly in some cases because the subnormal results are handled by microcode exceptions.

Some of these penalties have been removed by handling underflow and subnormal numbers in hardware rather than in microcode exceptions in some cases.

The processor has a penalty of approximately 140 - 150 clock cycles in all cases where an operation on normal numbers gives a subnormal result. There is a similar penalty for a multiplication between a normal and a subnormal number, regardless of whether the result is normal or subnormal. There is no penalty for adding a normal and a subnormal number, regardless of the result. There is no penalty for overflow, underflow, infinity or not-a-number results.

The penalties for subnormal numbers are avoided if the "flush-to-zero" mode and the "denormals-are-zero" mode are both set in the MXCSR register.

9.11 Partial register access

Different parts of a general purpose register can be stored in different temporary registers in order to remove false dependences. A problem occurs when a write to a part of a register is followed by a read from the whole register:

```
; Example 9.4. Partial register problem
mov  al,  1
mov  ebx, eax
```

This problem is solved in the Sandy Bridge by inserting an extra µop to join the different parts of the register. I assume that the extra µops are generated in the ROB-read stage. In the above example, the ROB-read will generate an extra µop that combines `AL` and the rest of `EAX` into a single temporary register before the `MOV EBX,EAX` instruction. There is no penalty for writing to a partial register unless there is a later read from a larger part of the same register. See page 99 for examples.

The Ivy Bridge inserts an extra µop only in the case where a high 8-bit register (`AH`, `BH`, `CH`, `DH`) has been modified, not in cases like example 9.4.

Partial flags stall

The Sandy Bridge and Ivy Bridge use the method of an extra µop to join partial registers not only for general purpose registers but also for the flags register, unlike previous processors which used this method only for general purpose registers. This occurs when a write to a part of the flags register is followed by a read from a larger part of the flags register. The partial flags stall of previous processors (See page 100) is therefore replaced by an extra µop. The Sandy Bridge also generates an extra µop when reading the flags after a rotate instruction, the Ivy Bridge does not.

Partial access to vector registers

An XMM register is never split into its parts in the reorder buffer. Therefore, no extra µops are needed and there is no partial access stall when writing to part of an XMM register. But a write to part of a vector register has a dependence on the previous value of the register. See example 8.8, on page 115.

The two halves of a YMM register are never treated as independent in VEX instructions, but the two halves can be separated when switching between VEX and non-VEX modes, as described below.

9.12 Transitions between VEX and non-VEX modes

The AVX instruction set defines three processor modes, as described in manual 2: "Optimizing subroutines in assembly language", chapter 13.6 "Using AVX instruction set and YMM registers". The three states are:

A. (Clean state) The upper half of all YMM registers is unused and known to be zero.

B. (Modified state) The upper half of at least one YMM register is used and contains data.

C. (Saved state) All YMM registers are split in two. The lower half is used by legacy XMM instructions which leave the upper part unchanged. All the upper-part halves are stored in a scratchpad. The two parts of each register will be joined together again if needed by a transition to state B.

State C is an undesired state. It appears when code that uses YMM registers is mixed with code that uses XMM registers in non-VEX instructions. The transitions B → C, C → B and C → A take approximately 70 clock cycles each on the Sandy Bridge, according to my measurements. The transitions A → B and B → A take zero or one clock cycle. The slow transitions to and from state C are best avoided by not mixing VEX and non-VEX instructions and by inserting a `VZEROUPPER` instruction after any sequence of code that uses the VEX-coded instructions.

9.13 Cache and memory access

Cache	Sandy Bridge and Ivy Bridge
µop cache	1536 µops, 8 way, 6 µop line size, per core
Level 1 code	32 kB, 8 way, 64 B line size, latency 4, per core
Level 1 data	32 kB, 8 way, 64 B line size, latency 4, per core
Level 2	256 kB, 8 way, 64 B line size, latency ca. 11, per core.
Level 3	up to 16 MB, 12 way, 64 B line size, latency ca. 28, shared

Table 9.5. Cache sizes on Sandy Bridge

There is one cache on each core, except for the last-level cache. All caches are shared between threads where a core can run two threads. It is likely that there will be more versions in the future with different last-level cache sizes.

Cache bank conflicts

Each consecutive 128 bytes, or two cache lines, in the data cache is divided into 8 banks of 16 bytes each. It is not possible to do two memory reads in the same clock cycle if the two memory addresses have the same bank number, i.e. if bit 4 - 6 in the two addresses are the same. Example:

```
; Example 9.5. Sandy bridge cache bank conflict
mov  eax, [rsi]        ; Use bank 0, assuming rsi is divisible by 40H
mov  ebx, [rsi+100H]   ; Use bank 0. Cache bank conflict
mov  ecx, [rsi+110H]   ; Use bank 1. No cache bank conflict
```

In addition, there is a false dependence between memory addresses with the same set and offset, i.e. with a distance that is a multiple of 4 Kbytes:

```
; Example 9.6. Sandy bridge false memory dependence
mov  [rsi], eax
mov  ebx, [rsi+1000H]       ; False memory dependence
```

Misaligned memory accesses

There is hardly any penalty for misaligned memory access with operand sizes of 64 bits or less, except for the effect of using multiple cache banks.

Prefetch instructions

The Ivy Bridge has a problem with the prefetch instructions. It appears that the Ivy Bridge is wasting time on prefetching data that are already in the cache. The measured throughput for repeated prefetch from the same address is one prefetch per 43 clocks on Ivy Bridge while the Sandy Bridge has a throughput of two prefetch instructions per clock cycle.

9.14 Store forwarding stalls

The processor can forward a memory write to a subsequent read from the same address under certain conditions. This store forwarding works in more cases than on previous processors, including misaligned cases. Store forwarding works in the following cases:

- When a write of 64 bits or less is followed by a read of the same size and the same address, regardless of alignment.
- When a write of 128 or 256 bits is followed by a read of the same size and the same address, aligned by 16.
- When a write of 64 bits or less is followed by a read of a smaller size which is fully contained in the write address range, regardless of alignment.
- When an aligned write of any size is followed by two reads of the two halves, or four reads of the four quarters, etc. with their natural alignment within the write address range.
- When an aligned write of 128 bits or 256 bits is followed by a read of 64 bits or less that does not cross an 8 bytes boundary.

There is no penalty for crossing a cache line boundary for store forwarding of 64 bits or less.

Store forwarding never works if the read has a bigger operand size than the preceding write or is partially overlapping the write address:

```
; Example 9.7. Failed store forwarding when read bigger than write
mov dword ptr [esi], eax        ; Write lower 4 bytes
mov dword ptr [esi+4], edx      ; Write upper 4 bytes
movq xmm0, qword ptr [esi]      ; Read 8 bytes. Stall
```

The penalty for a failed store forwarding is approximately 12 clock cycles in most cases.

The penalty can be exceptionally large for 128 bits or 256 bits store forwarding when writes are not aligned by at least 16. In this case I have measured a delay on the Ivy Bridge of approximately 50 clock cycles for 16 bytes read/write and 210 clock cycles for 32 bytes read/write.

9.15 Multithreading

Some versions of Sandy Bridge and Ivy Bridge can run two threads in each of its cores. This means that each thread gets only half of the resources. The resources are shared in the following way between two threads running in the same core:

Cache: All cache resources are shared competitively between the treads. The more one thread uses the less the other thread can use.

Branch target buffer and branch history pattern table: These are shared competitively between threads.

Instruction fetch and decoding: The instruction fetcher and decoders are shared evenly between the two threads so that each thread gets every second clock cycle.

Loop buffer: There is one loop buffer for each thread.

Register allocation and renaming resources are shared evenly so that each thread gets every second clock cycle.

Reorder buffer and reservation station. These are shared competitively.

Execution ports and execution units. These are shared competitively. One thread can use one execution port while another thread uses another port.

Read and write buffers: These are shared competitively.

Permanent register file: There is one for each thread.

It is clear that there is no advantage to running two threads per core if any of the shared resources are limiting factors for the performance. In many cases, however, the execution resources are more than sufficient for a single thread. It can be particularly advantageous to run two threads per core in situations where a large fraction of the time goes to cache misses and branch misprediction. However, if any of the shared resources are bottlenecks then it is not advantageous to run two threads per core. On the contrary, each thread is likely to run at less than half the single-thread speed because of evictions in the cache and branch target buffers and other resource conflicts. There is no way to give one thread higher priority than the other in the CPU.

9.16 Bottlenecks in Sandy Bridge and Ivy Bridge

Instruction fetch and predecoding

The instruction fetch and decoders are very similar to previous processors and continue to be a bottleneck. Fortunately, the new µop cache reduces the pressure on the decoders.

µop cache

The new µop cache is a very significant improvement because it removes the bottleneck of instruction fetch and decoding in cases where the critical part of the code fits into the µop cache. The maximum throughput of 4 instructions per clock (or 5 in case of macro-fusion) is now easily obtained even for longer instructions.

The programmer should be careful to economize the use of the µop cache in CPU-intensive code. The difference in performance between loops that fit into the µop cache and loops that do not can be quite remarkable if the average instruction length is more than four bytes.

The µop cache has some similarities with the trace cache in the old P4/NetBurst processor (see p. 47) and some of the same weaknesses. Instructions that need more than 32 bits of storage for address and data operands may use extra space in the µop cache and take an extra clock cycle to load. It is possible to optimize code to avoid these weaknesses, but it is unlikely that this will be done by anybody but the most dedicated assembly programmers.

It is a mystery to me why the processor is not marking instruction boundaries in the code cache. This would remove the bottleneck of instruction length decoding and thereby eliminate the need for the µop cache. AMD processors are doing this and the old Pentium MMX did the same.

Register read stalls

This old bottleneck, which has bothered Intel processors since the Pentium Pro, has finally been removed. No such stall was detected in my measurements.

Execution ports and execution units

The capacity of the execution ports and execution units is quite high. Many µops have two or three execution ports to choose between and each unit can handle one full 256-bit vector operation per clock cycle. The throughput of the execution ports is therefore not a serious bottleneck if instructions are evenly distributed between the ports.

Execution ports can be a bottleneck if the code generates many µops that all go to the same execution port.

Execution latency and dependency chains

Execution latencies are generally low. Most integer ALU operations have a latency of only one clock cycle, even for 256-bit vector operations. The execution latencies are critical only in long dependency chains.

Partial register access

There is a penalty for reading a full register after writing to a part of the register. Use `MOVZX` or `MOVSX` to read 8-bit or 16-bit memory operands into a 32-bit register rather than using a smaller part of the register.

Retirement

The retirement of µops has not been observed to be a bottleneck in any of my tests.

Branch prediction

The branch predictor does not have a special recognition of loops. Prediction of loops with many branches inside is inferior to the previous processors. The branch history pattern tables and branch target buffer may be bigger than on previous processors, but mispredictions are still common. The misprediction penalty is shorter for code that fits into the µop cache.

Memory access

The Sandy Bridge has two memory read ports where previous processors have only one. This is a significant improvement. Cache bank conflicts are quite common when the maximum memory bandwidth is utilized.

The Ivy Bridge has a serious problem with prefetch instructions, which are extremely slow.

Multithreading

Most of the critical resources are shared between threads. This means that the bottlenecks become even more critical in multithreaded applications.

Literature

"Intel 64 and IA-32 Architectures Optimization Reference Manual". Intel 2011.

10 Haswell and Broadwell pipeline

The Haswell has several important improvements over previous designs. The bandwidth to the data cache has been doubled to two reads and one write per clock cycle, of up to 32 bytes each. The number of execution units is increased from six to eight. Most execution units have a bandwidth of one full 256-bit vector per clock cycle.

The Haswell is currently available with 2 - 18 cores, and most versions are capable of running two threads in each core. Most of the critical resources are shared between the two threads running in the same core, as described on page 133 for the Sandy Bridge and Ivy Bridge processors.

The Haswell is the first processor to support the AVX2 instruction set, which can handle integer instructions on 256-bit vectors. It is also the first Intel processor to support fused multiply-and-add (FMA) instructions.

The Broadwell is a 14 nm die shrink of the Haswell, with just a few improvements in the execution units.

10.1 Pipeline

The pipeline is similar to previous designs, but improved with more of everything. It is designed for a throughput of four instructions per clock cycle.

Each core has a reorder buffer with 192 entries, the reservation station has 60 entries, and the register file has 168 integer registers and 168 vector registers, according to the literature listed on page 146 below.

All parts of the pipeline are shared between two threads in those CPU models that can run two threads in each core. Each thread gets half of the total throughput when two threads are running in the same core.

10.2 Instruction fetch and decoding

The instruction fetch unit can fetch a maximum of 16 bytes of code per clock cycle in single threaded applications.

There are four decoders, which can handle instructions generating up to four µops per clock cycle in the way described on page 121 for Sandy Bridge.

Instructions with any number of prefixes are decoded in a single clock cycle. There is no penalty for redundant prefixes.

The penalty for length-changing prefixes is the same as for Sandy Bridge (see page 121). Arithmetic and logic instructions with an immediate operand using an operand size prefix, e.g. `add ax,1234` has a penalty of 2-3 clock cycles in the decoders, regardless of alignment. This applies to all arithmetic and logic instructions with a 16-bit immediate constant as operand in 32-bit or 64-bit mode. Move instructions have no penalty for length-changing prefixes.

10.3 µop cache

The µop cache from the Sandy Bridge (see p. 122) has been preserved in the Haswell and Broadwell with the same size and organization. The most important effect of the µop cache is that the throughput is no longer limited by a fetch rage of 16 bytes per clock cycle for small code loops. A piece of code that fits into the µop cache can be delivered at a rate corresponding to up to a maximum of 32 bytes per clock cycle. This is a big advantage when the average instruction length is more than four bytes.

The limitations and weaknesses of the Sandy Bridge µop cache also apply to the Haswell and Broadwell. See page 122 for details.

10.4 Loopback buffer

The processor has a loop buffer which simply recycles µops from the µop queue. The loop buffer will rarely use all 56 entries of the queue, but small loops of up to 30 µops, or sometimes up to 40, will benefit from the loop buffer. The µop queue is dynamically shared between threads so that you get only half the size if two threads are running in the same core. The loop buffer gives a stable throughput of 4 µops per clock, regardless of instruction length for tiny loops.

To recapitulate, the pipeline can be fed from three different sources, depending on the size of critical loops:

- The loop buffer is used for tiny loops of up to 30 - 40 instructions. The throughput is 4 µops per clock cycle with no restriction on instruction length.
- The µop cache is used for loops up to approximately 1000 instructions. The throughput is up to 4 instructions or 32 bytes of code per clock cycle.
- The fetch and decode units are used for instructions that are not in the µop cache. The throughput is up to 4 instructions or 16 bytes of code per clock cycle.

Fused instruction pairs (see below) count as one in the µop cache and the loop buffer. With two fused not-taken branches per clock, it is possible to obtain a maximum throughput of six instructions per clock cycle from the loop buffer or µop cache.

There may be a difference in branch misprediction penalty between the three sources of µops, but I have not been able to verify such a difference because the variance in the measurements is high. The measured misprediction penalty varies between 16 and 20 clock cycles in all three cases.

10.5 Micro-op fusion

µop fusion is used in the same way as on previous processors. Some instructions that need two µops in the execution units can use the µop fusion technique to keep these two µops together as one from the decoders to the reservation station in order to save pipeline bandwidth. The reservation station will then submit two µops to two different ports. Most memory write instructions and most arithmetic and logic instructions with a memory operand use µop fusion, regardless of register size. See page 90 and 106 for further explanation.

The decoders can handle four µop-fused instructions per clock cycle.

You can see which instructions use µop fusion by looking at the tables in manual 4: "Instruction tables". Instructions with µop fusion has a higher number of µops listed under "unfused domain" than under "fused domain".

10.6 Macro-op fusion

The Haswell and Broadwell can fuse two instructions into one µop in the same way that previous processors can (see page 107).

The decoders will fuse an arithmetic or logic instruction and a subsequent conditional jump instruction into a single compute-and-branch µop in certain cases. The compute-and-branch µop is not split in two at the execution units but executed as a single µop by the branch unit at execution port 0 or 6.

The `CMP`, `ADD` and `SUB` instructions can fuse with signed and unsigned branch instructions. `INC` and `DEC` can fuse with signed branch instructions, and `TEST` and `AND` instructions can fuse with all branch instructions (including useless combinations), as indicated in table 9.2 page 125.

The first instruction can have an immediate operand or a memory source operand, but not both. It cannot have a memory destination operand. It cannot have a RIP-relative memory operand.

The `JECXZ` and `LOOP` instructions cannot be fused.

Unlike previous processors, it can make fusion even if a 16-bytes code boundary is crossed.

Two fuseable pairs can be decoded in the same clock cycle.

The programmer should keep any fuseable arithmetic instruction together with a subsequent conditional jump rather than scheduling other instructions in-between in order to take advantage of macro-op fusion. All four decoders support macro-op fusion.

10.7 Stack engine

The Haswell and Broadwell have a stack engine similar to the Sandy Bridge, as described on page 125. An extra stack synchronization µop is inserted automatically when stack operations such as push, pop, call or return are interspersed by instructions that access the stack pointer explicitly, such as add `rsp, 8` or `mov eax,[rsp+16]`.

10.8 Register allocation and renaming

All integer, floating point, MMX, XMM, YMM, flags and perhaps also segment registers can be renamed. The floating point control word can also be renamed.

Register renaming is controlled by the reorder buffer and the scheduler. Register allocation and renaming has not been observed to be a bottleneck.

Special cases of independence

A common way of setting a register to zero is by xor'ing it with itself or subtracting it from itself, e.g. `XOR EAX,EAX`. The processor recognizes that certain instructions are independent of the prior value of the register if the two input operands are the same register. This register is set to zero at the register allocation stage without using any execution unit and without waiting for the previous value of the register to be available.

The following instructions can set a register to zero in this way if xor'ed with or subtracted from itself: `XOR`, `SUB`, `PXOR`, `XORPS`, `XORPD` and all variants of `PSUBxxx` and `PCMPGTxx`. Instructions with `V`-prefix behave the same. No execution unit is used, and the throughput is four zeroing operations per clock cycle.

This works with all 32-bit and 64-bit general purpose registers and all 128-bit and 256-bit vector registers. It does not work with 8-bit and 16-bit registers, because only part of the register is set to zero. It works partially with 64-bit mmx registers: The register is set to zero without waiting for the previous value, but it does use an execution unit (in order to resolve the dual use as floating point stack register and mmx register).

All variants of the `PCMPEQxx` instruction can set a register to all ones without waiting for the previous value of the register. It does, however, use an execution unit.

The following instructions have no special case for the two input operands being the same register: `CMP`, `SBB`, `ANDN`, `PANDN`, `ANDNPS`, `ANDNPD`, `CMPEQPS`, `CMPEQPD`.

<u>Instructions that need no execution unit</u>

The abovementioned special cases where registers are set to zero by instructions such as `XOR EAX,EAX` are handled at the register allocation stage without using any execution unit.

A few other instructions are also handled without using any execution unit. These are `CLC`, `FXCH`, `NOP` (including long nops), but not `FNOP`.

<u>Elimination of move instructions</u>

Most register-to-register moves are eliminated at the register allocation stage in the same way as in the Ivy Bridge, as explained on page 126.

Move elimination typically succeeds in more than 80% of the possible cases. Chained moves can also be eliminated.

Move elimination is possible with all 32-bit and 64-bit general purpose registers and all 128-bit and 256-bit vector registers. It is not possible with 8-bit and 16-bit registers, and it is not possible with 64-bit mmx registers.

Unlike the Ivy Bridge, the Haswell and Broadwell cannot eliminate a zero-extended moves. But moves with implicit zero-extension can be eliminated, e.g. `MOV EAX,EBX` (zero-extends into `RAX`), and `VMOVAPS XMM0,XMM1` (zero-extends into `YMM0`).

A move of a register to itself will never be eliminated. For example `mov eax,eax` is not eliminated.

An eliminated move has zero latency and does not use any execution port. But is does consume bandwidth in the decoders.

10.9 Execution units

The processor has a number of execution units accessed through eight execution ports. This gives a theoretical maximum throughput of eight µops per clock cycle in the execution units. However, the typical throughput of the whole design is four instructions per clock. Thus, even with µop fusion it is impossible to keep all execution ports busy more than in temporary bursts.

Many of the execution units are duplicated so that there is always high probability that a vacant unit can be found that is applicable for a particular µop. There are four integer ALUs so that the most common integer operations can execute with a throughput of four instructions per clock cycle. There are three ports that can handle integer vector operations. Two ports can handle floating point vector operations. Two ports can handle branches. Two ports can handle memory read operations, and one port can handle memory writes.

It is strange that there is only one port for floating point addition, but two ports for floating point multiplication.

The eight ports and their common operations are listed in table 10.1.

Port	Operations	Latency
0	integer and vector arithmetic, logic, shift	1
0	floating point multiplication	Haswell: 5 Broadwell: 3
0	floating point FMA and long double multiplication	5

0	integer vector multiplication	5
0	division	variable
0	branch	1-2
1	integer and vector arithmetic, logic	1
1	floating point addition	3
1	integer multiplication	3
1	bit scan and other complex integer operations	3
1	floating point multiplication	Haswell: 5 Broadwell: 3
1	floating point FMA	5
2	load, address generation	
3	load, address generation	
4	store	
5	integer and vector arithmetic, logic	1
5	vector permute	1 (3 if lane crossed)
5	AES encryption	7
6	integer arithmetic, logic, shift	1
6	branch	1-2
7	store address generation	

Table 10.1. Execution units in Haswell and Broadwell

All vector execution units have full 256-bit capacity, except for division, square root and encryption. A 256-bit unit cannot be split up and used for two 128-bit operations simultaneously.

The latency for integer vector operations is the same as for operations in general purpose registers. This makes it possible to use XMM registers for simple integer operations when you are out of general purpose registers. The vector operations are supported by fewer execution ports, though.

Fused multiply and add

The Haswell and Broadwell have two execution units that can handle fused multiply-and-add (FMA) instructions of the type $a = b * c + d$.

An FMA instruction does a multiplication and an addition or subtraction with a single instruction and a single µop. This can improve performance in floating point code where the combination of multiplication and addition often occurs. The throughput is two FMA instructions per clock cycle and the latency is 5.

Intel initially designed the FMA instructions with four operands (FMA4) but later changed their plans to a design with three operands (FMA3) where it is necessary to reuse one of the three input registers as output register, e.g. $a = b * c + a$.

How many input dependencies can a µop have?

All previous Intel processors with out-of-order capabilities had the design limitation that no µop can have more than two input dependencies. I am concluding this from the fact that all instructions with more than two input dependencies are split into at least two µops. The most common examples are:

```
; Example 10.1. Instructions with three input dependencies
mov [eax+ebx],ecx   ; depends on 3 registers
adc eax,ebx         ; depends on 2 registers and carry flag
cmovz eax,ebx       ; depends on 2 registers and zero flag
pblendvb xmm1,xmm2,xmm0 ; depends on 3 registers
```

All of these instructions are split into two µops on the Haswell and all previous Intel processors. The introduction of fused multiply-and-add (FMA) instructions in the Haswell made it necessary to get rid of the limitation of two input dependencies for a µop. Thus, the FMA instructions are the first instructions to use a µop with more than two input dependencies on an Intel processor. It has probably required a redesign of the out-of-order mechanism to allow µops with three input dependencies.

The Broadwell takes this redesign a little step further by using a single µop with three input dependencies for add-with-carry, subtract-with-borrow, and conditional moves on general purpose registers. All memory stores and blend instructions are still split into two µops on the Broadwell.

Read and write bandwidth

There are two identical memory read ports (port 2 and 3) and one write port (port 4). These ports have all been expanded to 256 bits. This makes it possible to make two memory reads and one memory write per clock cycle, all with any register size up to 256 bits. All write operations need an address calculation on port 2, 3 or 7.

Data bypass delays

The execution units are divided into domains as described on page 112, and there are sometimes a delay of one clock cycle when the output of an instruction in the integer domain is used as input for an instruction in the floating point domain or vice versa. For example:

```
; Example 10.2. Data bypass delays
addps xmm0, xmm1
por   xmm0, xmm2    ; 1 clock delay
mulps xmm0, xmm3    ; 1 clock delay
```

The delays in example 10.2 can be avoided by replacing the `POR` instruction with the more appropriate `ORPS`.

However, there are fewer such delays on Haswell and Broadwell than on previous processors. I found no such delays in the following cases:

- when a floating point Boolean instruction, such as `ORPS` is used with integer data
- when a wrong type of move instruction is used, e.g. `MOVPS` or `MOVDQA`
- when a wrong type of shuffle instruction is used, e.g. `SHUFPS` or `PFHUFD`

I did, however, find delays when a floating point blend instruction, such as `BLENDPS` is used with integer data.

Instructions such as `MOVD` that move data between general purpose registers and vector registers have a latency of only 1 clock and no extra bypass delay. Likewise, instructions that move data from vector registers to the integer flags, such as `COMISS` and `PTEST` have no extra bypass delay.

256-bit vectors

All vector execution units have full 256-bit throughput, except for division, square root and encryption.

The 256-bit data path that is used for all operations on YMM registers is divided into two lanes of 128 bits each. All instructions that can move data between these two lanes have a latency of 3 clock cycles, while other move instructions have a latency of only one clock cycle. For example:

```
; Example 10.3. Moving data between 128-bit lanes
vextracti128 xmm0,ymm1,1    ; move from upper to lower lane, 3 clocks
vextracti128 xmm0,ymm1,0    ; move in lover lane only, still 3 clocks
```

```
vmovdqa xmm0,xmm1          ; same operation in 0 - 1 clocks
```

The second instruction in example 10.3 has a latency of 3 clock cycles, even though all data stay in the lower lane, because the `VEXTRACTI128` instruction has the potential for moving data between lanes. This delay can be avoided by using the `VMOVDQA` instruction which does the same in this case, but has no potential for moving data across lanes.

There is a severe penalty for mixing 256-bit VEX code with 128-bit non-VEX code as explained in chapter 9.12. All transitions between state B (modified) and state C (saved) have a delay of 70 clock cycles. This delay is the same in 32-bit mode and 64-bit mode even though the number of registers to save is different. The delay should be avoided either by replacing all non-VEX vector instructions with the V-prefix version, or by issuing a `VZEROUPPER` instruction whenever a function containing 256-bit vector code calls another function that may use non-VEX code or returns to a function that may use non-VEX code.

The two halves of a 256-bit register are not treated as independent, except in the undesired state C (saved), as explained in chapter 9.12. It is not possible to split a 256-bit execution unit into two 128-bit units to do two 128-bit operations simultaneously.

Mixing µops with different latencies

Some older processors have a write-back conflict when µops with different latencies are issued to the same execution port, as described on page 114. I found no evidence of such a problem on Haswell and Broadwell. The problem has been solved partly by standardizing latencies, as is evident from table 10.1, and partly, perhaps, by having multiple write-back paths.

Underflow and subnormals

Subnormal numbers occur when floating point operations are close to underflow. The handling of subnormal numbers is very costly in some cases because the subnormal results are handled by microcode exceptions.

The Haswell and Broadwell have a penalty of approximately 124 clock cycles in all cases where an operation on normal numbers gives a subnormal result. There is a similar penalty for a multiplication between a normal and a subnormal number, regardless of whether the result is normal or subnormal. There is no penalty for adding a normal and a subnormal number, regardless of the result. There is no penalty for overflow, underflow, infinity or not-a-number results.

The penalties for subnormal numbers are avoided if the "flush-to-zero" mode and the "denormals-are-zero" mode are both set in the MXCSR register.

10.10 Partial register access

Different parts of a general purpose register can be stored in different temporary registers in order to remove false dependences. A problem occurs when a write to a part of a register is followed by a read from a larger part of the same register:

```
; Example 10.4. Partial register access
mov  al,  1
mov  ebx, eax
```

The Haswell and Broadwell solve this problem without any visible performance penalties. Perhaps it makes dual bookkeeping of both the partial register and the full register.

The situation is different if one of the high 8-bit registers (`AH`, `BH`, `CH`, `DH`) are modified, and a larger part of the register is read afterwards:

```
; Example 10.5. Partial register problem with AH
mov  ah,  1
mov  ebx, eax
```

Here, an extra µop is inserted to combine `AH` and the rest of `EAX` into a single temporary register before the `MOV EBX,EAX` instruction. This causes an extra latency of one clock cycle.

Partial flags access

A similar situation occurs when part of the flags register is modified and a larger part of the same register is read afterwards. Some of these cases require an extra µop like example 10.5.

```
; Example 10.6. Partial flags access
inc eax          ; modifies zero flag but not carry flag
jbe L1           ; reads both sero flag and carry flag

; Example 10.7. Partial register access
bt eax,2         ; modifies carry flag but not zero flag
cmovbe eax,ebx   ; reads both carry flag and zero flag
```

The processor will insert an extra µop to join the two parts of the flags register in these cases.

In cases like this, you may consider whether it is a programming error or a deliberate testing of two different conditions with a single instruction.

There is no penalty or extra µop when reading the flags after a shift or rotate instruction.

Partial access to vector registers

An XMM register is never split into its parts in the reorder buffer. Therefore, no extra µops are needed and there is no partial access stall when writing to part of an XMM register. But a write to part of a vector register has a dependence on the previous value of the register. See example 8.8, on page 115.

The two halves of a YMM register are never treated as independent in VEX instructions, but the two halves can be separated when switching between VEX and non-VEX modes, as described in chapter 9.12.

10.11 Cache and memory access

Cache	Haswell and Broadwell
µop cache	1536 µops, 8 way, 6 µop line size, per core
Level 1 code	32 kB, 8 way, 64 sets, 64 B line size, latency 4, per core
Level 1 data	32 kB, 8 way, 64 sets, 64 B line size, latency 4, per core
Level 2	256 kB, 8 way, 512 sets, 64 B line size, latency 12, per core.
Level 3	2 - 45 MB, 12-16 way, 64 B line size, latency 34, shared

Table 10.2. Cache sizes on Haswell and Broadwell

There is one cache on each core, except for the last-level cache. All caches are shared between two threads where a core can run two threads. It is likely that there will be more versions in the future with different level 3 cache sizes. Some versions have a level 4 cache.

The 256-bit read and write bandwidth (see p. 141) makes it advantageous to use 256-bit registers for copying or zeroing large blocks of memory. The `REP MOVS` instruction has full efficiency only if the source and destination are both aligned by 32. In all other cases, it is better to use a function library that uses 256-bit registers.

Cache bank conflicts

The phenomenon of cache bank conflicts has been a performance problem in previous processors. This problem has been removed now. It is always possible to do two cache reads in the same clock cycle without causing a cache bank conflict.

However, the problem with false dependence between memory addresses with the same set and offset remains. It is not possible to read and write simultaneously from addresses that are spaced by a multiple of 4 Kbytes:

```
; Example 10.8. False memory dependence
mov   [rsi], eax
mov   ebx, [rsi+1000H]        ; False memory dependence
```

Misaligned memory accesses

There is hardly any penalty for misaligned memory access, except for the effect of using multiple cache lines.

10.12 Store forwarding stalls

The processor can forward a memory write to a subsequent read from the same address under certain conditions. Store forwarding works in the following cases:

- When a write of 64 bits or less is followed by a read of the same size and the same address, regardless of alignment.

- When a write of 128 or 256 bits is followed by a read of the same size and the same address, fully aligned.

- When a write of 64 bits or less is followed by a read of a smaller size which is fully contained in the write address range, regardless of alignment.

- When an aligned write of any size is followed by two reads of the two halves, or four reads of the four quarters, etc. with their natural alignment within the write address range.

- When an aligned write of 128 bits or 256 bits is followed by a read of 64 bits or less that does not cross an 8 bytes boundary.

A delay of 2 clocks occur if the memory block crosses a 64-bytes cache line boundary. This can be avoided if all data have their natural alignment.

Store forwarding fails in the following cases:

- When a write of any size is followed by a read of a larger size

- When a write of any size is followed by a partially overlapping read

- When a write of 128 bits is followed by a smaller read crossing the boundary between the two 64-bit halves

- When a write of 256 bits is followed by a 128 bit read crossing the boundary between the two 128-bit halves

- When a write of 256 bits is followed by a read of 64 bits or less crossing any boundary between the four 64-bit quarters

A failed store forwarding takes 10 clock cycles more than a successful store forwarding. The penalty is much higher - approximately 50 clock cycles - after a write of 128 or 256 bits which is not aligned by at least 16.

10.13 Multithreading

Some versions of Haswell and Broadwell can run two threads in each of its cores. This means that each thread gets only half of the resources.

The resources are shared between two threads running in the same core in the same way as in Sandy Bridge (see p. 133). A small difference is that there is one loop buffer shared between the two threads.

There is no advantage to running two threads per core if any of the shared resources are limiting factors for the performance. There are so many execution ports and execution units that execution is rarely a limiting factor. If the code aims at more than two instructions per clock cycle, or if cache size is a limiting factor, then there is no advantage in running two threads in each core.

There is no way to give one thread higher priority than the other in the CPU.

10.14 Bottlenecks in Haswell and Broadwell

Instruction fetch and predecoding

The instruction fetch rate is still limited to 16 bytes per clock cycle, which is likely to be a bottleneck for code that doesn't fit well into the µop cache.

µop cache

The µop cache is efficient for loops of up to approximately a thousand instructions.

It is important to economize the use of the µop cache in CPU-intensive code. The difference in performance between loops that fit into the µop cache and loops that do not can be quite significant if the average instruction length is more than four bytes.

The µop cache has the same weaknesses as in the Sandy Bridge, as explained on page 122 and 134.

Execution ports and execution units

The capacity of the execution ports and execution units is quite high. The most common integer instructions have four execution units to choose between, and most floating point and vector instructions have two or three execution unit to choose between. Therefore, it is realistic to obtain a throughput of four instructions per clock cycle if the code has no long dependency chains.

Almost all vector execution units and data paths have full 256 bit width. This makes it very advantageous to use the 256-bit vector registers and the AVX or AVX2 instruction set.

The new AVX2 gather instructions are efficient for gathering non-contiguous data into vectors, and for vectorizing table-based lookup functions. Gather instructions are more efficient in Broadwell than in Haswell.

The fused multiply-and-add (FMA) instructions are useful for improving the performance of floating point code.

Register-to-register moves are eliminated to zero latency in most cases.

Floating point addition has lower throughput than multiplication

The throughput for floating point vector multiplications and FMA operations is two vector operations per clock cycle, but the throughput for floating point vector addition is only one vector operation per clock cycle. It is odd that there are two multiplication/FMA units and only one addition unit since additions are typically more common than multiplications in floating point code. The floating point performance will be sub-optimal if the code contains more additions than multiplications or FMA instructions. In my opinion, this is not the best design decision for typical floating point code, but at least it enables Intel to boast a floating point performance of 32 FLOPS per cycle.

For a code that contains mostly floating point additions, you can actually improve the throughput by replacing additions by FMA instructions with a multiplier of 1.0. The FMA instructions have a latency of 5 and a throughput of two instructions per clock, which means that you may need 10 accumulator registers to get the maximum throughput.

Execution latency and dependency chains

Execution latencies are generally low. Most integer ALU operations have a latency of only one clock cycle, even for 256-bit vector operations. Floating point addition has a latency of 3, and floating point multiplication has a latency of 3 on Broadwell and 5 on Haswell. These execution latencies are critical in long dependency chains.

Branch prediction

The size of the branch target buffer and the construction of the branch predictor is unknown, but at least the prediction rate seems good.

The throughput for taken branches is one jump per clock or one jump per two clocks, depending on the density of branches. Predicted not taken branches have a higher throughput of two per clock. Therefore, it is advantageous to organize branches so that they are most often not taken.

Memory access

The width of the memory ports has been doubled relative to previous processors. The maximum throughput to the level-1 cache is now two 32-byte reads and one 32-byte write per clock. This makes it possible to copy a block of memory at a speed of 32 bytes per clock cycle. The throughput for the level-2 cache is much lower than this.

Cache bank conflicts were a quite common performance problem on previous processors. This problem has been completely removed on Haswell and Broadwell.

Multithreading

Most of the critical resources are shared between threads. This means that the bottlenecks become even more critical in multithreaded applications.

Literature

Intel: "Intel 64 and IA-32 Architectures Optimization Reference Manual". July 2013.
David Kanter: "Intel’s Haswell CPU Microarchitecture". November 2012.
http://www.realworldtech.com/haswell-cpu/

Anand Lal Shimpi: "Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel". October 2012. http://www.anandtech.com/show/6355/intels-haswell-architecture
Anna Filatova: "4th Generation of Core Microarchitecture: Intel Haswell". September 2012. http://www.xbitlabs.com/articles/cpu/display/haswell-uarch-idf.html

11 Skylake pipeline

The Skylake is a further development of the Haswell and Broadwell design. The cache and decoder front end is basically the same as in Haswell with somewhat more bandwidth, while the execution engine has been reorganized somewhat to improve the throughput.

A version of Skylake with support for 512-bit vector registers has been announced, but is not available yet (late 2015).

The Skylake is currently available with 2 - 4 cores, and most versions are capable of running two threads in each core. Most of the critical resources are shared between the two threads running in the same core, as described on page 133.

The Skylake uses 14 nm technology like Broadwell, but supports the faster DDR4 RAM.

11.1 Pipeline

The pipeline is very similar to previous designs, but the execution units have been reorganized a little with more than one execution port for nearly all important instructions. It is designed for a throughput of four instructions per clock cycle.

The sizes of the reorder buffer, reservation station and register file have allegedly been increased over previous processors, but details have not been published.

All parts of the pipeline are shared between two threads in those CPU models that can run two threads in each core. Each thread gets half of the total throughput when two threads are running in the same core.

11.2 Instruction fetch and decoding

The instruction fetch unit can fetch a maximum of 16 bytes of code per clock cycle in single threaded applications.

There are four decoders, which can handle instructions generating up to four µops per clock cycle in the way described on page 121 for Sandy Bridge. Instructions with any number of prefixes are decoded in a single clock cycle. There is no penalty for redundant prefixes.

The penalty for length-changing prefixes is the same as for Sandy Bridge (see page 121). Arithmetic and logic instructions with an immediate operand using an operand size prefix, e.g. `add ax,1234` has a penalty of 2-3 clock cycles in the decoders, regardless of alignment. This applies to all arithmetic and logic instructions with a 16-bit immediate constant as operand in 32-bit or 64-bit mode. Move instructions have no penalty for length-changing prefixes.

11.3 µop cache

The µop cache has the same size and organization as in the Sandy Bridge (see p. 122). It is organized as 32 sets × 8 ways × 6 µops, totaling a maximum capacity of 1536 µops. It can allocate a maximum of 3 lines of 6 µops each for each aligned and contiguous 32-bytes block of code.

Code that runs out of the µop cache are not subject to the limitations of the fetch and decode units. It can deliver a throughput of 4 (possibly fused) µops or the equivalent of 32 bytes of code per clock cycle. This is a big advantage when the average instruction length is more than four bytes. The limitations and weaknesses of the Sandy Bridge µop cache still apply. See page 122 for details.

11.4 Loopback buffer

The processor has a loop buffer which simply recycles µops from the µop queue. The loop buffer will rarely use all 64 entries of the queue, but small loops of up to 30 µops, or sometimes up to 40, will benefit from the loop buffer. The µop queue is dynamically shared between threads so that you get only half the size if two threads are running in the same core. The loop buffer gives a stable throughput of 4 µops per clock, regardless of instruction length for tiny loops.

To recapitulate, the pipeline can be fed from three different sources, depending on the size of critical loops:

- The loop buffer is used for tiny loops of up to 30 - 40 instructions. The throughput is 4 µops per clock cycle with no restriction on instruction length.
- The µop cache is used for loops up to approximately 1000 instructions. The throughput is up to 4 instructions or 32 bytes of code per clock cycle.
- The fetch and decode units are used for instructions that are not in the µop cache. The throughput is up to 4 instructions or 16 bytes of code per clock cycle.

Fused instruction pairs (see below) count as one in the µop cache and the loop buffer. With two fused not-taken branches per clock, it is possible to obtain a maximum throughput of six instructions per clock cycle from the loop buffer or µop cache.

There may be a difference in branch misprediction penalty between the three sources of µops, but I have not been able to verify such a difference because the variance in the measurements is high. The measured misprediction penalty varies between 16 and 20 clock cycles in all three cases.

11.5 Micro-op fusion

µop fusion is used in the same way as on previous processors. Some instructions that need two µops in the execution units can use the µop fusion technique to keep these two µops together as one from the decoders to the reservation station in order to save pipeline bandwidth. The reservation station will then submit two µops to two different ports. Most memory write instructions and most arithmetic and logic instructions with a memory operand use µop fusion, regardless of register size. See page 90 and 106 for further explanation.

The decoders can handle four µop-fused instructions per clock cycle. You can see which instructions use µop fusion by looking at the tables in manual 4: "Instruction tables". Instructions with µop fusion have a higher number of µops listed under "unfused domain" than under "fused domain".

11.6 Macro-op fusion

The processor can fuse two instructions into one µop in the same way that previous processors can (see page 107).

The decoders will fuse an arithmetic or logic instruction and a subsequent conditional jump instruction into a single compute-and-branch µop in certain cases. The compute-and-branch µop is not split in two at the execution units but executed as a single µop by the branch unit at execution port 0 or 6.

The `CMP`, `ADD` and `SUB` instructions can fuse with signed and unsigned branch instructions. `INC` and `DEC` can fuse with signed branch instructions, and `TEST` and `AND` instructions can fuse with all branch instructions (including useless combinations), as indicated in table 9.2 page 125.

The first instruction can have an immediate operand or a memory source operand, but not both. It cannot have a memory destination operand. It cannot have a RIP-relative memory operand.

The `JECXZ` and `LOOP` instructions cannot be fused.

Unlike previous processors, it can make fusion even if a 16-bytes code boundary is crossed.

Two fuseable pairs can be decoded in the same clock cycle.

The programmer should keep any fuseable arithmetic instruction together with a subsequent conditional jump rather than scheduling other instructions in-between in order to take advantage of macro-op fusion. All four decoders support macro-op fusion.

11.7 Stack engine

The processor has a stack engine similar to the Sandy Bridge, as described on page 125. An extra stack synchronization µop is inserted automatically when stack operations such as push, pop, call or return are interspersed by instructions that access the stack pointer explicitly, such as add `rsp, 8` or `mov eax,[rsp+16]`.

11.8 Register allocation and renaming

All integer, floating point, MMX, XMM, YMM, flags and perhaps also segment registers can be renamed. The floating point control word can also be renamed.

Register renaming is controlled by the reorder buffer and the scheduler. Register allocation and renaming has not been observed to be a bottleneck.

Special cases of independence

A common way of setting a register to zero is by xor'ing it with itself or subtracting it from itself, e.g. `XOR EAX,EAX`. The processor recognizes that certain instructions are independent of the prior value of the register if the two input operands are the same register. This register is set to zero at the register allocation stage without using any execution unit and without waiting for the previous value of the register to be available.

The following instructions can set a register to zero in this way if xor'ed with or subtracted from itself: `XOR`, `SUB`, `PXOR`, `XORPS`, `XORPD` and all variants of `PSUBxxx` and `PCMPGTxx`. Instructions with `V`-prefix behave the same. No execution unit is used, and the throughput is four zeroing operations per clock cycle.

This works with all 32-bit and 64-bit general purpose registers and all 128-bit and 256-bit vector registers. It does not work with 8-bit and 16-bit registers, because only part of the register is set to zero. It works partially with 64-bit mmx registers: The register is set to zero without waiting for the previous value, but it does use an execution unit (in order to resolve the dual use as floating point stack register and mmx register).

All variants of the `PCMPEQxx` instruction can set a register to all ones without waiting for the previous value of the register. It does, however, use an execution unit.

The following instructions have no special case for the two input operands being the same register: `CMP`, `SBB`, `ANDN`, `PANDN`, `ANDNPS`, `ANDNPD`, `CMPEQPS`, `CMPEQPD`.

Instructions that need no execution unit

The abovementioned special cases where registers are set to zero by instructions such as `XOR EAX,EAX` are handled at the register allocation stage without using any execution unit.

A few other instructions are also handled without using any execution unit. These are `CLC`, `FXCH`, `NOP` (including long nops), but not `FNOP`.

Elimination of move instructions

Most register-to-register moves are eliminated at the register allocation stage in the same way as on the Ivy Bridge, as explained on page 126. Move elimination typically succeeds in more than 80% of the possible cases. Chained moves can also be eliminated.

Move elimination is possible with all 32-bit and 64-bit general purpose registers and all 128-bit and 256-bit vector registers. It is not possible with 8-bit and 16-bit registers, and it is not possible with 64-bit mmx registers.

The Skylake cannot eliminate zero-extended moves, but moves with implicit zero-extension can be eliminated, e.g. `MOV EAX,EBX` (zero-extends into `RAX`), and `VMOVAPS XMM0,XMM1` (zero-extends into `YMM0`).

A move of a register to itself will never be eliminated. For example `mov eax,eax` is not eliminated.

An eliminated move has zero latency and does not use any execution port. But is does consume bandwidth in the decoders.

11.9 Execution units

The Skylake has a number of execution units accessed through eight execution ports. This gives a theoretical maximum throughput of eight µops per clock cycle in the execution units. However, the throughput of the whole design rarely exceeds four instructions per clock. Thus, even with µop fusion it is impossible to keep all execution ports busy more than in temporary bursts.

Most of the execution units are duplicated so that a µop will rarely have to wait for a vacant unit. There are four integer ALUs so that the most common integer operations can execute with a throughput of four instructions per clock cycle. There are three ports that can handle integer vector operations. Two ports can handle floating point vector operations. Two ports can handle branches. Two ports can handle memory read operations, and one port can handle memory writes. The Skylake can handle more different latencies at the same port than on previous processors.

The eight ports and their common operations are listed in table 11.1.

Port	Operations	Latency
0	integer and vector arithmetic, logic, shift	1
0	vector string instructions	3
0	floating point add, multiply, FMA	4
0	AES encryption	4
0	integer vector multiplication	5
0	integer and floating point division, squareroot	variable
0	branch	1-2
1	integer and vector arithmetic, logic, shift	1
1	integer multiplication, bit scan	3
1	floating point add, multiply, FMA	4
1	integer vector multiplication	5
2	load, address generation	
3	load, address generation	
4	store	

5	integer and vector arithmetic, logic	1
5	vector permute	1 (3 if lane crossed)
5	x87 floating point add, SADBW	3
5	PCLMUL	7
6	integer arithmetic, logic, shift	1
6	jump and branch	1-2
7	load and store address generation	
Table 11.1. Execution units in Skylake		

All vector execution units have full 256-bit capability, except for division, square root and encryption. A 256-bit unit cannot be split up and used for two 128-bit instructions simultaneously.

The latency for integer vector operations is mostly the same as for operations in general purpose registers. This makes it possible to use XMM registers for simple integer operations when you are out of general purpose registers.

Fused multiply and add

The Skylake has two execution units that can handle floating point addition and multiplication, as well as fused multiply-and-add (FMA) instructions of the type $a = b * c + d$.

An FMA instruction does a multiplication and an addition or subtraction with a single instruction and a single µop in the same time as it takes to do only a multiplication or addition. The FMA instructions can improve performance in floating point code where the combination of multiplication and addition often occurs.

How many input dependencies can a µop have?

Until the Ivy Bridge, all Intel processors with out-of-order capabilities had the design limitation that no µop could have more than two input dependencies. The introduction of fused multiply-and-add (FMA) instructions in the Haswell made it necessary to get rid of the limitation of two input dependencies for each µop. Thus, the FMA instructions were the first instructions to use µops with more than two input dependencies on an Intel processor. A few more instructions with three input dependencies in a single µop have later been added: add-with-carry, subtract-with-borrow, and conditional moves in Broadwell, and a blend instruction in Skylake.

Read and write bandwidth

There are two identical memory read ports (port 2 and 3) and one write port (port 4). These ports all have the full 256 bits width. This makes it possible to make two memory reads and one memory write per clock cycle, with any register size up to 256 bits. All write operations need an address calculation on port 2, 3 or 7.

Data bypass delays

The execution units are divided into domains as described on page 112, and there is sometimes a delay of one clock cycle when the output of an instruction in the integer domain is used as input for an instruction in the floating point domain. For example:

```
; Example 11.1. Data bypass delays
addps xmm0, xmm1
por   xmm0, xmm2
mulps xmm0, xmm3   ; 1 clock delay
```

The delays in example 11.1 can be avoided by replacing the `POR` instruction with the more appropriate `ORPS`.

However, such delays are few on the Skylake processor. I found no such delays in the following cases:

- when a floating point Boolean instruction, such as `ORPS` is used with integer data
- when a wrong type of move instruction is used, e.g. `MOVPS` or `MOVDQA`
- when a wrong type of shuffle instruction is used, e.g. `SHUFPS` or `PFHUFD`
- when a wrong type of blend instruction is used, e.g. `VPBLENDD` or `BLENDPS`

Instructions such as `MOVD` that move data between general purpose registers and vector registers have a latency of 2 clock cycles.

256-bit vectors

All vector execution units have full 256-bit throughput, except for division, square root and encryption.

The 256-bit data path that is used for all operations on YMM registers is divided into two lanes of 128 bits each. All instructions that can move data between these two lanes have a latency of 3 clock cycles, while other move instructions have a latency of only one clock cycle. For example:

```
; Example 11.2. Moving data between 128-bit lanes on Haswell
vextracti128 xmm0,ymm1,1    ; move from upper to lower lane, 3 clocks
vextracti128 xmm0,ymm1,0    ; move in lover lane only, still 3 clocks
vmovdqa xmm0,xmm1           ; same operation in 0 - 1 clocks
```

The second instruction in example 11.2 has a latency of 3 clock cycles, even though all data stay in the lower lane, because the `VEXTRACTI128` instruction has the potential for moving data between lanes. This delay can be avoided by using the `VMOVDQA` instruction which does the same in this case, but has no potential for moving data across lanes.

The severe penalty for mixing 256-bit VEX code with 128-bit non-VEX code in previous processors (see chapter 9.12 page 131) is no longer found in the Skylake. This means that the two halves of a 256-bit register are treated as independent by 128-bit non-VEX instructions.

Warm-up period for 256-bit vector operations

I have observed a warm-up phenomenon for 256-bit instructions. Instructions with 256-bit vectors have a throughput that is approximately 4.5 times slower than normal during an initial warm-up period of approximately 56,000 clock cycles or 14 µs. A sequence of code containing 256-bit vector operations will run at full speed after this warm-up period. The processor returns to the mode of slow 256-bit execution 2.7 million clock cycles, or 675 µs, after the last 256-bit instruction (These times were measured on a 4 GHz processor).

It is possible to prepare the processor for executing 256-bit instructions by giving it a dummy 256-bit instruction at least 56,000 clock cycles before a critical sequence of 256-bit instructions. You may insert a 256-bit vector instruction such as, for example, `vxorps ymm0,ymm0,ymm0` somewhere before a piece of code that contains time-consuming 256-bit operations. Any instruction with YMM registers will start the warmup process, except `vzeroupper` and `vzeroall`. The first 256-bit instruction takes 150 - 250 clock cycles - probably to start a power-up process.

There are two possible explanations for this warm-up effect: (1) either the upper 128-bit lane runs at lower speed in periods where it is rarely used, in order to save power. Or (2) the processor turns off the upper lane and uses the lower 128-bit lane twice when executing a 256-bit instruction if the upper lane is powered down. Explanation (2) seems most plausible because timing measurements indicate that the performance during the cold period is limited by throughput rather than by latency.

This effect was measured on a Skylake i7-6700K, 4 GHz. Similar effects have been reported for some versions of Sandy Bridge and Haswell processors, but I have not seen it on any of the processors that I have tested myself other than the Skylake. It is possible that high-end processors have a feature for powering down the upper 128-bit lane, while low-end processors have no such feature.

Underflow and subnormals

Subnormal numbers occur when floating point operations are close to underflow. The handling of subnormal numbers is very costly in some cases because the subnormal results are handled by microcode exceptions.

The processor has a penalty of approximately 129 clock cycles in all cases where an operation on normal numbers gives a subnormal result. There is a similar penalty for a multiplication or division between a normal and a subnormal number, regardless of whether the result is normal or subnormal. There is no penalty for adding a normal and a subnormal number, regardless of the result. There is no penalty for overflow, underflow, infinity or not-a-number results.

The penalties for subnormal numbers are avoided if the "flush-to-zero" mode and the "denormals-are-zero" mode are both set in the MXCSR register.

11.10 Partial register access

Different parts of a general purpose register can be stored in different temporary registers in order to remove false dependences. A problem occurs when a write to a part of a register is followed by a read from a larger part of the same register:

```
; Example 11.3. Partial register access
mov  al,  1
mov  ebx, eax
```

The processor solves this problem without any visible performance penalties. Perhaps it makes dual bookkeeping of both the partial register and the full register. This also applies to the two halves of a 256-bit vector register on the Skylake.

The situation is different if one of the high 8-bit registers (`AH`, `BH`, `CH`, `DH`) are modified, and a larger part of the register is read afterwards:

```
; Example 11.4. Partial register problem with AH
mov  ah,  1
mov  ebx, eax
```

Here, an extra µop is inserted to combine `AH` and the rest of `EAX` into a single temporary register before the `MOV EBX,EAX` instruction. This causes an extra latency of one clock cycle.

Partial flags access

A similar situation occurs when part of the flags register is modified and a larger part of the same register is read afterwards. Some of these cases require an extra µop like example 11.4.

```
; Example 11.5. Partial flags access
inc eax         ; modifies zero flag but not carry flag
jbe L1          ; reads both sero flag and carry flag

; Example 11.6. Partial register access
bt eax,2        ; modifies carry flag but not zero flag
cmovbe eax,ebx  ; reads both carry flag and zero flag
```

The processor will insert an extra µop to join the two parts of the flags register in these examples.

In cases like this, you may consider whether it is a programming error or a deliberate testing of two different conditions with a single instruction.

There is no penalty or extra µop when reading the flags after a shift or rotate instruction.

Partial access to vector registers

An XMM register is never split into its parts in the reorder buffer. But a write to part of a vector register has a dependence on the previous value of the register. See example 8.8, on page 115.

The two halves of a YMM register are not treated as independent in VEX instructions, but the two halves can be separated when switching between VEX and non-VEX modes, as described in chapter 9.12. Unlike previous processors, the Skylake has no penalty for switching between VEX and non-VEX instructions.

11.11 Cache and memory access

Cache	Skylake
µop cache	1536 µops, 8 way, 6 µop line size, per core
Level 1 code	32 kB, 8 way, 64 sets, 64 B line size, latency 4, per core
Level 1 data	32 kB, 8 way, 64 sets, 64 B line size, latency 4, per core
Level 2	256 kB, 4 way, 1024 sets, 64 B line size, latency 14, per core.
Level 3	3-8 MB, 16 way, 64 B line size, latency 34, shared

Table 11.2. Cache sizes on Skylake

There is one cache on each core, except for the level 3 cache. All caches are shared between two threads where a core can run two threads. It is likely that there will be more versions in the future with different level 3 cache sizes. Some versions may have a level 4 cache.

The 256-bit read and write bandwidth (see p. 141) makes it advantageous to use 256-bit registers for copying or zeroing large blocks of memory. The `REP MOVS` instruction has full efficiency only if the source and destination are both aligned by 32. In all other cases, it is better to use a function library that uses 256-bit registers.

Cache bank conflicts

The phenomenon of cache bank conflicts has been a performance problem in some previous processors. This problem has been removed now. It is always possible to do two cache reads in the same clock cycle without causing a cache bank conflict.

However, the problem with false dependence between memory addresses with the same set and offset remains. It is not possible to read and write simultaneously from addresses that are spaced by a multiple of 4 Kbytes:

```
; Example 11.7. False memory dependence
mov   [rsi], eax
mov   ebx, [rsi+1000H]        ; False memory dependence
```

11.12 Store forwarding stalls

The Skylake processor can forward a memory write to a subsequent read from the same address under certain conditions. Store forwarding is one clock cycle faster than on previous processors. A memory write followed by a read from the same address takes 4 clock cycles in the best case for operands of 32 or 64 bits, and 5 clock cycles for other operand sizes.

Store forwarding has a penalty of up to 3 clock cycles extra when an operand of 128 or 256 bits is misaligned.

A store forwarding usually takes 4 - 5 clock cycles extra when an operand of any size crosses a cache line boundary, i.e. an address divisible by 64 bytes.

A write followed by a smaller read from the same address has little or no penalty.

A write of 64 bits or less followed by a smaller read has a penalty of 1 - 3 clocks when the read is offset but fully contained in the address range covered by the write.

An aligned write of 128 or 256 bits followed by a read of one or both of the two halves or the four quarters, etc., has little or no penalty. A partial read that does not fit into the halves or quarters can take 11 clock cycles extra.

A read that is bigger than the write, or a read that covers both written and unwritten bytes, takes approximately 11 clock cycles extra.

11.13 Multithreading

Most versions of the processor can run two threads in each of its cores. This means that each thread gets only half of the resources.

Level-1 cache, instruction fetch, decoding and execution units are shared between two threads running in the same core in the same way as in previous processors.

There is no advantage to running two threads per core if any of the shared resources are limiting factors for the performance. There are so many execution ports and execution units that execution is rarely a limiting factor. If the code aims at more than two instructions per clock cycle, or if cache size is a limiting factor, then there is no advantage in running two threads in each core. There is no way to give one thread higher priority than the other in the CPU.

11.14 Bottlenecks in Skylake

The pipeline and execution units in Skylake are quite efficient. Memory access is the most common bottleneck.

Instruction fetch and predecoding

The instruction fetch rate is still limited to 16 bytes per clock cycle, which is likely to be a bottleneck for code that doesn't fit well into the µop cache.

µop cache

The µop cache is efficient for loops of up to approximately a thousand instructions.

It is important to economize the use of the µop cache in CPU-intensive code. The difference in performance between loops that fit into the µop cache and loops that do not can be quite significant if the average instruction length is more than four bytes. The µop cache has the same weaknesses earlier processors.

Execution ports and execution units

The capacity of the execution ports and execution units is high. The most common integer instructions have four execution units to choose between, and most floating point and vector instructions have two or three execution unit to choose between. Therefore, it is realistic to obtain a throughput of four instructions per clock cycle if the code has no long dependency chains.

The vector execution units and data paths have full 256 bit width. This makes it very advantageous to use the 256-bit vector registers and the AVX or AVX2 instruction set.

The gather instructions are efficient for gathering non-contiguous data into vectors, and for vectorizing table-based lookup functions. Gather instructions are more efficient than in previous processors.

The fused multiply-and-add (FMA) instructions are useful for improving the performance of floating point code. Floating point vector additions, multiplications and FMA instructions all have a throughput of two instructions per clock cycle and a latency of 4 clock cycles.

Register-to-register moves are eliminated to zero latency in most cases. `MOVZX` instructions are not eliminated.

Execution latency and dependency chains

Execution latencies are generally low. Most integer ALU operations have a latency of only one clock cycle, even for 256-bit vector operations, and floating point arithmetic operations have a latency of 4. These execution latencies are critical mainly in long dependency chains.

Branch prediction

The size of the branch target buffer and the construction of the branch predictor is unknown, but at least the prediction rate seems good.

The throughput for taken branches is one jump per clock or one jump per two clocks, depending on the density of branches. Predicted not taken branches have a higher throughput of two per clock. Therefore, it is advantageous to organize branches so that they are most often not taken.

Memory access

The width of the memory ports is the same as for Haswell and Broadwell. The maximum throughput to the level-1 cache is now two 32-byte reads and one 32-byte write per clock. This makes it possible to copy a block of memory at a speed of 32 bytes per clock cycle. The throughput for the level-2 cache is much lower than this. The level-2 cache has 4 ways on the Skylake, while Haswell and Broadwell had 8 ways. Cache bank conflicts is no longer a problem.

Multithreading

Most of the critical resources are shared between threads. This means that the bottlenecks become more critical in multithreaded applications.

Literature

Intel: "Intel 64 and IA-32 Architectures Optimization Reference Manual". September 2015.
Orestis Bastounis: "Intel’s Skylake Core i7-6700K reviewed: Modest gains from a full Tick-Tock cycle". http://arstechnica.co.uk/gadgets/2015/08/intel-skylake-core-i7-6700k-reviewed/

12 Intel Atom pipeline

The Intel Atom processor has a simpler design where power saving has the highest priority. It has almost no out-of-order capabilities. It has one or two cores, where each core can run two threads, totaling up to four simultaneous threads. The caches, decoders, and execution units are all shared between two threads. It supports the Supplementary SSE3 instruction set. Some versions support x64.

The pipeline has sixteen stages: three stages for instruction fetch, three stages for instruction decoding, two for instruction dispatch, one for reading register operands, one for calculating the address of a memory operand, two for reading the data cache, one for execution, two for exception handling and multithreading, and one for committing the result. Instructions of the read-modify or read-modify-write type are handled as a single µop. Only more complex instructions are split into µops.

The pipeline can handle two instructions per clock cycle. There are two execution ports, each of which is connected to an integer unit and a floating point/SIMD unit.

This description is based on the limited information available from second hand sources in addition to my own test results (Scott Wasson: Intel's Atom processor unveiled. http://techreport.com/articles.x/14458/1, April 2008. Anand Lal Shimpi: Intel's Atom Architecture: The Journey Begins. http://www.anandtech.com/showdoc.aspx?i=3276&p=1, April 2008).

12.1 Instruction fetch

The instruction fetch rate is approximately 8 bytes per clock cycle on average when running a single thread. The fetch rate can get as high as 10.5 bytes per clock cycle in rare cases, such as when all instructions are 8 bytes long and aligned by 8, but in most situations the fetch rate is slightly less than 8 bytes per clock when running a single thread. The fetch rate is lower when running two threads in the same core.

The instruction fetch rate is likely to be a bottleneck when the average instruction length is more than 4 bytes. The instruction fetcher can catch up and fill the instruction queue in case execution is stalled for some other reason.

12.2 Instruction decoding

The two instruction decoders are identical. Instructions with up to three prefixes can be decoded in a single clock cycle. There are severe delays for instructions with more than three prefixes (which would almost never occur unless prefixes are used for padding). Most instructions generate only a single µop. There is no penalty for length-changing prefixes.

Decoded instructions go into a queue with 16 entries per thread. The two 16-entry queues can be combined to a single 32 entries queue if one thread is disabled.

12.3 Execution units

There are two clusters of execution units: an integer cluster which handles all instructions on general purpose registers, and a floating point and SIMD cluster which handles all instructions on floating point registers and SIMD vector registers. A memory access cluster is connected to the integer unit cluster. Moving data between the clusters is slow.

The µops from the decoders can be dispatched to two execution ports, which I will call port 0 and port 1. Each execution port has access to part of the integer cluster and part of the floating point/SIMD cluster. I will call the two parts of the integer cluster ALU0 and ALU1 and the two parts of the floating point/SIMD cluster FP0 and FP1, respectively. The two execution ports can thus handle two parallel streams of µops, with the work divided as follows:

Instructions that can be handled by both port 0 and port 1:

- Register-to-register moves
- Integer addition in general purpose or SIMD registers
- Boolean operations in general purpose or SIMD registers

Instructions that can be handled only by port 0:

- Memory read or write
- Integer shift, shuffle, pack in general purpose or SIMD registers
- Multiply
- Divide
- Various complex instructions

Instructions that can be handled only by port 1:

- Floating point addition
- Jumps and branches
- LEA instruction

The four units ALU0, ALU1, FP0 and FP1 probably have one integer ALU each, though it cannot be ruled out that there are only two integer ALUs, which are shared between ALU0 and FP0 and between ALU1 and FP1, respectively. There is one multiply unit in FP0, and one division unit in FP0. Integer multiplication and integer division go through port 0.

The SIMD integer adders and shift units have full 128-bit widths and a one clock latency. The floating point adder has full 128-bit capability for single precision vectors, but only 64-bit capability for double precision. The multiplier and the divider are 64-bits wide.

The floating point adder has a latency of 5 clock cycles and is fully pipelined to give a throughput of one single precision vector addition per clock cycle. The multiplier is partially pipelined with a latency of 4 clocks and a throughput of one single precision multiplication per clock cycle. Double precision and integer multiplications have longer latencies and a lower throughput. The time from one multiplication starts till the next multiplication can start varies from 1 clock cycle in the most favorable cases, to 2 clock cycles or more in less favorable cases. Double precision vector multiplication and some integer multiplications cannot overlap in time.

Division is slow and not pipelined. A single precision scalar floating point division takes 30 clock cycles. Double precision takes 60 clock cycles. A 64-bit integer division takes 207 clock cycles.

The list of instructions in manual 4: "Instruction tables" tell which instructions use which units.

12.4 Instruction pairing

The maximum throughput of two instructions per clock cycle can only be obtained when instructions are ordered so that they can execute two at a time. Two instructions can execute simultaneously when the following rules are obeyed:

- The core runs only one thread. The other thread, if any, must be idle or stalled by a cache miss etc.

- The two instructions must be consecutive with no other instructions between.

- The two instructions do not have to be contiguous. A predicted taken branch instruction can pair with the first instruction at the branch target.

- The second instruction does not read a register that the first instruction writes to. This rule has one exception: A branch instruction that reads the flags can pair with a preceding instruction that modifies the flags.

- The two instructions do not write to the same register, except for the flags register. For example: `INC EAX / MOV EAX,0` cannot pair because both modify `EAX`. `INC EAX / INC EBX` pair OK even though both modify the flags.

- The two instructions do not use the same execution port. The first instruction goes to port 0 and the second instruction to port 1; or the first instruction goes to port 1 and the second instruction to port 0.

- An instruction that uses resources from both ports or pipelines cannot pair with any other instruction. For example, a floating point add instruction with a memory operand uses FP1 under port 1 for floating point addition and the memory unit under port 0 for the memory operand.

It follows from these rules that it is not possible to do a memory read and a memory write at the same time because both use the memory unit under port 0. But it is possible to do a floating point addition (without memory operand) and a floating point multiplication simultaneously because they use FP1 and FP0 respectively.

12.5 X87 floating point instructions

Instructions that use the x87-style floating point registers are handled in a very unfortunate way by the Atom processor. Whenever there are two consecutive x87 instructions, the two instructions fail to pair and instead cause an extra delay of one clock cycle due to problems in the decoders. This gives a throughput of only one instruction every two clock cycles, while a similar code using XMM registers would have a maximum throughput of two instructions per clock cycle.

This applies to all x87 instructions (names beginning with F), even the FNOP. For example, a sequence of 100 consecutive FNOP instructions takes 200 clock cycles to execute in my tests. If the 100 FNOPs are interspersed by 100 NOPs then the sequence takes only 100 clock cycles. It is therefore important to avoid consecutive x87 instructions. If you have nothing else to put in between two x87 instructions then put in a NOP. Making every second instruction a NOP obviously takes half the bandwidth, but this is still better than the quarter bandwidth that you would have without the NOPs.

The FXCH instruction has a latency of one clock cycle, while many other processors give a zero latency for FXCH. This is a further disadvantage of running x87-style floating point code on the Atom because the floating point register stack structure makes it necessary to use many FXCH instructions.

It is therefore advantageous to replace any x87-style floating point code with SSE2-style code using XMM registers. The SSE2 instructions are often more than 4 bytes long while x87 instructions are shorter. With a maximum instruction fetch rate of 8 bytes per clock cycle we are likely to make instruction fetching a bottleneck, but the shorter length of the x87 instructions does not outweigh the severe disadvantages explained above.

12.6 Instruction latencies

Simple integer instructions have a latency of one clock cycle. Multiplications, divisions and floating point instructions have longer latencies.

Unlike most other processors, I have found no delays in the Atom processor when mixing instructions with different latencies in the same pipeline.

The LEA instruction uses the address generation unit (AGU) rather than the ALU. This causes a latency of 4 clock cycles when dependent on a pointer register or index register because of the distance between the AGU and the ALU. It is therefore faster to use addition and shift instructions than to use the LEA instruction in most cases.

Instructions that move data between a SIMD vector register and a general purpose register or flag have a latency of 4-5 clock cycles because the integer execution cluster and the floating point/SIMD cluster have separate register files.

There is no penalty for using XMM move, shuffle and Boolean instructions for other types of data than they are intended for. For example, you may use `PSHUFD` for floating point data or `MOVAPS` for integer data.

12.7 Memory access

Each core has three caches:

- Level 1 instruction cache. 32 kB, 8 way, set associative, 64 B line size
- Level 1 data cache. 24 kB, 6 way, set associative, 64 B line size
- Level 2 cache. 512 kB or 1 MB, 8 way, set associative, 64 B line size

Each cache is shared between two threads, but not between cores. All caches have a hardware prefetcher.

An instruction with a memory operand takes no more time to execute than a similar instruction with a register operand, provided that the memory operand is cached. It is not totally "free" to use memory operands, though, for two reasons. Firstly, the memory operand uses the memory unit under port 0 so that the instruction cannot pair with another instruction that would require port 0. And secondly, the memory operand may make the instruction code longer, especially if it has a full 4-bytes address. This can be a bottleneck when instruction fetching is limited to 8 bytes per clock cycle.

Cache access is fast for instructions running in the integer execution cluster, but slower for instructions running in the floating point/SIMD cluster because the memory unit is connected to the integer cluster only. Instructions using floating point or XMM registers typically take 4-5 clock cycles to read or write memory while integer instructions have only 1 clock cycle of effective cache latency thanks to store forwarding, as described below. The latency for a memory read that depends on a recently changed pointer register is 3 clock cycles.

Store forwarding is very efficient. A memory operand that is written in one clock cycle can be read back in the next clock cycle. Unlike most other processors, the Atom can do store forwarding even if the read operand is larger than the preceding write operand or differently aligned. The only situation I have found where store forwarding fails is when a cache line boundary is crossed.

Misaligned memory accesses are very costly when a cache line boundary is crossed. A misaligned memory read or write that crosses a 64 bytes boundary takes 16 clock cycles. The performance monitor counters indicate that the misaligned memory access involves four accesses to the level-1 cache, where two accesses would suffice. There is no cost to misaligned memory accesses when no 64 bytes boundary is crossed.

12.8 Branches and loops

The throughput for jumps and taken branches is one jump per two clock cycles. Not-taken branches go at one per clock cycle. The minimum execution time for a loop is thus 2 clock cycles if the loop contains no 16 bytes boundary, and 3-4 clock cycles if a 16 bytes boundary is crossed inside the loop.

Branch prediction uses a 12 bits global history register, as explained on page 29. This gives reasonably good predictions, but the branch target buffer (BTB) has only 128 entries. In some of my tests, there were more BTB misses than hits. A branch misprediction costs up to 13 clock cycles, sometimes a little less. If a branch is correctly predicted taken, but it fails to predict a target because the BTB entry has been evicted, then the penalty is approximately 7 clock cycles. This happens very often because the pattern history table has 4096 entries while the BTB has only 128.

12.9 Multithreading

Each processor core can run two threads. The two treads are competing for the same resources so that both threads run slower than they would when running alone.

The caches, decoders, ports and execution units are shared between the two threads of the core, while the prefetch buffers, instruction queues and register files are separate.

The maximum throughput for the whole core is still two instructions per clock cycles, which gives one instruction per clock cycle in each thread on average.

If both threads need the same resource, for example memory access, then each thread will get the contested resource half of the time. In other words, you will have one memory access every two clock cycles in each thread if there are no cache misses.

Interestingly, I found that the instruction fetch rate for each thread when running two threads is more than half the fetch rate for a single thread but less than the full single-thread rate. The instruction fetch rate per thread when running two threads is between 4 and 8 bytes per clock cycle, but never more than eight. The numbers depend heavily on instruction lengths and alignment. These findings indicate that there may be two instruction fetchers that are capable, to a limited extent, of serving the same thread when the other thread is idle.

The branch target buffer (BTB) and the pattern history table are shared between the two threads. If the two threads are running the same code (with different data) then we might expect the two threads to share identical entries in these two tables. However, this doesn't happen. The two tables are apparently indexed by some simple hash function of the branch address and the thread number, so that identical entries in the two threads don't use the same table index. My tests indicate that two threads running the same code have slightly more branch mispredictions and significantly more BTB misses than a single thread running alone. In the worst case of a code with many branches, each thread may run at less than half the speed of a single thread running alone.

These resource conflicts apply only to the case where two threads are running in the same processor core, of course. Some versions of the Atom processor have two cores capable of running two threads each, giving a maximum of four threads running simultaneously. If each core runs only one thread then there are no resource conflicts. Fortunately, most operating systems will preferably put two threads in two different cores rather than in the same core. But if there are more than two threads running then you will have some threads sharing the same processor core.

It is impossible to assign different priorities to two threads running in the same core. Thus, a low priority thread may take resources from a high priority thread running in the same core, with the unfortunate result that the high priority thread runs at only half the maximum possible speed. This has happened several times during my tests.

12.10 Bottlenecks in Atom

Some of the execution units in the Atom processor are quite powerful. It can handle two full 128-bit integer vector ALU instructions per clock cycle, though this capacity would rarely be fully utilized because of bottlenecks elsewhere in the system. The floating point addition unit is also reasonably good, while multiplication and division are slower.

The execution is likely to be limited by other factors than the execution units in most cases. The most likely bottlenecks are:

- In order execution. The processor can do nothing while it is waiting for a cache miss or a long-latency instruction, unless another thread can use its resources in the meantime.

- The instruction fetch rate is less than 8 bytes per clock cycle in most cases. This is insufficient if the average instruction length is more than 4 bytes.

- Memory access is limited to one read or one write per clock cycle. It cannot read and write simultaneously.

- Memory access has long latencies for floating point and SIMD instructions.

- The branch target buffer is rather small.

- x87 style floating point code executes much slower than SSE style code.

- The maximum throughput of two instructions per clock cycle can only be achieved if the code is optimized specifically for the Atom and instructions are ordered in a way that allows pairing.

- The throughput is halved if two threads are running simultaneously in the same core. The resources most likely to be bottlenecks, such as cache, memory port and branch target buffer, are shared between two threads.

The conclusion is that the Atom may be insufficient for highly CPU-intensive and memory intensive applications, such as games, graphics processing and floating point math. The low price and low power consumption makes it useful for less demanding purposes, such as office applications and embedded applications. The ability to run four threads makes the Atom useful for server applications with limited traffic.

13 Intel Silvermont pipeline

The low-power Atom design has finally moved a major step forward after several years of only small improvements. The Silvermont processor is now a serious competitor for the ARM processors. It has limited out-of-order execution capabilities for integer instructions, but not for floating point and vector instructions. It supports SSE4.2 instructions, but not AVX.

The pipeline has 14 stages where the previous Atom design had 16. There are three stages for instruction fetch, three stages for instruction decoding, two for register allocation and renaming, one for scheduling, one for execution, and four for retirement and commitment. The shorter pipeline should allegedly reduce the branch misprediction penalty to 10 clock cycles, but my measurements show approximately 12 clock cycles, which is almost the same as for the Atom.

The chip has one to four units with two execution cores in each for a total of up to eight threads. The two cores in a unit share the level-2 cache, but no pipeline or execution resources.

13.1 Pipeline

The Silvermont has five pipelines: One for memory read and write operations, two for integer operations, and two for floating point and vector operations. The maximum average throughput is two instructions per clock cycle, which can be achieved when the two instructions are going to two different pipelines.

The two integer pipelines have out-of-order execution. Each integer pipe has a reservation station which can hold 8 instructions in queue. Memory operations can also operate out of order with a 6-entry reservation station, while floating point and vector operations cannot be reordered within their respective pipelines.

13.2 Instruction fetch and decoding

Instruction boundaries are marked in the code cache. This is a simple technique for removing the bottleneck of instruction length decoding that Intel hasn't used since the Pentium MMX, while AMD has been doing it all the time.

The Silvermont has two decoders, and it can fetch and decode two simple instructions per clock cycle. Instructions longer than 8 bytes have a throughput of only one instruction per clock cycle.

Most instructions generate only 1 µop from the decoders. Read-modify, and read-modify-write instructions generate a single µop from the decoders, which is sent to both the memory unit and the execution unit. Instructions that generate more than one µop are using microcode ROM, except for the POP register and FXCH instructions. All instructions that use microcode ROM take at least 4 clock cycles to decode if they go into decoder 0. If such an instruction goes into decoder 1 first then it is redirected to decoder 0 with a further delay of 2 clocks. This means that the decoding process takes 6 clock cycles for an instruction that generates 2 µops and happens to go into decoder 1 first.

Instructions with more than three prefixes and escape bytes cause significant delays in the decoders. Unlike most other processors from Intel and AMD, the Silvermont includes not only the standard prefixes in this limitation, but also escape bytes such as 0F. Therefore, this limitation can easily be exceeded. All xmm instructions belonging to the SSSE3 and later instruction sets have one prefix byte (66) and two escape bytes (0F 38 or 0F 3A) in the instruction code. If any of the registers r8 – r15 or xmm8 – xmm15 is used in such an instruction then we need an extra REX prefix whereby the limit of three prefix and escape bytes is exceeded. For example:

```
; Example 13.1. Prefix limitation in Silvermont
pblendw xmm1, xmm2, 2  ; 1 prefix + 2 escape bytes
pblendw xmm1, xmm8, 2  ; 2 prefixes + 2 escape bytes
```

The first instruction in this example can decode normally in a single clock cycle, while the second instruction takes 4 clock cycles to decode if it goes into the first decoder, or 6 clock cycles if it goes into the second decoder. There is no penalty for length-changing prefixes.

The two decoders cannot handle two branch instructions simultaneously. Consecutive branch instructions should be avoided for this reason.

13.3 Loop buffer

The Silvermont has a loop buffer that recycles decoded µops. It can improve the performance of loops with up to 29 instructions by removing the bottleneck of decoding.

13.4 Macro-op fusion

The Silvermont has no fusion of multiple instructions into a single µop.

13.5 Register allocation and out of order execution

Instructions on general purpose registers can execute out of order to a limited degree. Apparently, no more than eight instructions can be pending at the same time. Instructions on floating point and vector registers cannot execute out of order with other instructions going to the same execution port and pipeline. But a floating point instruction in one pipeline can bypass a floating point instruction in the other pipeline. For example, a floating point multiply instruction (in FP0) can execute before a preceding floating point add instruction (in FP1).

The same logical register can be allocated to different physical registers in order to remove false dependencies. This also applies to floating point and vector registers, even though they cannot execute out of order in the same pipeline.

13.6 Special cases of independence

A common way of setting a register to zero is by xor'ing it with itself or subtracting it from itself, e.g. `XOR EAX,EAX`. The Silvermont processor recognizes that certain instructions are independent of the prior value of the register if the two input operands are the same register. This works only in a few cases. The `XOR` of a 32-bit register with itself is recognized as independent of the previous value, but this does not work with 8-, 16- or 64-bit registers. Thus, the best way to clear a general purpose register is to xor the 32-bit version of the register with itself. This will clear all 64 bits. `SUB`, `SBB` and `CMP` instructions are not recognized in this way. A vector register can be cleared of its dependence on previous values by xor'ing it with itself using the `PXOR`, `XORPS` or `XORPD` instructions, but not by any subtract, compare or other instructions.

13.7 Execution units

The Silvermont has five execution ports with each their scheduler:

Port	Operations
Memory	memory read and write
Integer port 0	ALU, multiply, shift
Integer port 1	ALU, jump
F.P. port 0	multiply, divide, shift, pack, convert
F.P. port 1	add

Table 13.1. Execution units in Silvermont

The execution units are partially pipelined. The integer multiplier as well as the floating point adder and multiplier all have a throughput of one instruction per clock for smaller data and one instruction per two clocks or less for larger data:

Port and unit	Data size	Latency	Reciprocal throughput
IP0 multiply	32 bit integer	3	1
IP0 multiply	64 bit integer	5	2

FP0 multiply	64 bit vector of 4 16-bit integers	4	1
FP0 multiply	128 bit vector of 8 16-bit integers	5	2
FP0 multiply	single prec. float	4	1
FP0 multiply	128 bit vector of 4 single prec.	5	2
FP0 multiply	double precision	5	2
FP0 multiply	128 bit vector of 2 double prec.	7	4
FP0/1 add	vectors of 8-, 16-, 32 bit integers	1	0.5
FP0/1 add	vectors of 64 bit integers	4	?
FP1 add	single prec. float	3	1
FP1 add	128 bit vector of 4 single prec.	3	1
FP1 add	double precision	3	1
FP1 add	128 bit vector of 2 double prec.	4	2

Table 13.2. Partially pipelined execution units in Silvermont

There is one division unit which is shared between integer port 0 and floating point port 0. It has latencies from 19 to 69 and is not pipelined. Floating point division has fixed latencies.

Read and write bandwidth

The memory bandwidth is one 128-bit read or write instruction per clock cycle. It is possible to do one read operation and one write operation per clock cycle, but only when the two operations are part of the same instruction, i.e. a read-modify-write instruction. In all other cases, the maximum bandwidth is one memory operation per clock cycle.

Data bypass delays

Instructions that move data between the integer units and the floating point/vector units have a latency of 3-4 clock cycles. Otherwise, there is no extra delay for moving data between different execution units, and there is no penalty for using integer vector instructions on floating point data or vice versa.

Underflow and subnormals

Operations that have subnormal numbers as input or output or generate underflow take approximately 160 clock cycles unless the flush-to-zero mode and denormals-are-zero mode are both used.

13.8 Partial register access

A write to a partial register has a false dependence on the rest of the register. The different parts of a general purpose register or vector register are never treated as independent.

The flags register may be treated as different parts. A read of any flag after writing to part of the flags has an extra delay of one clock cycle.

13.9 Cache and memory access

Cache	Silvermont
Level 1 code	32 kB, 8 way, 64 sets, 64 B line size, per core
Level 1 data	24 kB, 6 way, 64 sets, 64 B line size, latency 3, per core
Level 2	1 MB, 16 way, 1024 sets, 64 B line size, latency 19. Shared between two threads

Table 13.3. Cache sizes on Silvermont

Cache bank conflicts have not been observed. There is a false dependence between memory addresses spaced a multiple of 4 kB apart.

13.10 Store forwarding

A memory write can be forwarded to a subsequent read of the same size or a smaller size with the same start address. The latency of the write + subsequent read is 7 clock cycles. There is an extra delay of 3 clock cycles if an unaligned store forwarding is crossing a cache line boundary.

Store forwarding fails when the read is bigger than the write or does not start at the same address. It is not possible to forward a store to two reads of the two halves. A failed store forwarding has an extra delay of 5 clock cycles.

13.11 Multithreading

The Silvermont has one or more units with two cores each. The two cores in a unit share the same level-2 cache. The two threads running in the same unit do not compete for any resources other than the level-2 cache.

13.12 Bottlenecks in Silvermont

Instruction fetch and decoding

Decoding is definitely the weakest part of the Silvermont design. Instructions that generate more than one µop take at least 4 clock cycles to decode, and quite often more (with few exceptions). The same applies to instructions that need an extra prefix. All xmm instructions that belong to the SSSE3 and later instruction sets take 4 or 6 clock cycles to decode if they use any of the registers r8 – r15 or xmm8 – xmm15. Instructions that take extra time in the decoders should be avoided at all costs in code designed for the Silvermont.

Execution ports and execution units

Most of the execution units have full 128-bit capabilities, but some execution units are only partially pipelined and take an extra clock cycle for full vectors or double precision data.

The capacity of the execution units seems to be satisfactory for a small low-power processor.

The disastrous performance of the Atom on legacy x87 code has finally been repaired.

Out of order execution

The Silvermont has register renaming, but only very little capacity for out-of-order execution. Scheduling by the compiler may be necessary for best performance.

Branch prediction

The size of the branch target buffer is unknown. The prediction rate is fair. The misprediction penalty is relatively low. Indirect branches have no pattern prediction according to my tests.

Memory access

The throughput is one read/write instruction per clock cycle. Cache performance is good.

Multithreading

No critical resources are shared between threads other than the level-2 cache. This makes multithreading efficient.

Literature

Intel: "Intel 64 and IA-32 Architectures Optimization Reference Manual". July 2013.
Sebastian Anthony: "Intel's Silvermont revealed: After a five-year snooze, Intel is finally ready to crush ARM". http://www.extremetech.com/computing/155082-intels-silvermont-revealed-after-a-five-year-snooze-intel-is-finally-ready-to-crush-arm. 2013.
Anand Lal Shimpi: "Intel's Silvermont Architecture Revealed: Getting Serious about Mobile". http://www.anandtech.com/show/6936/intels-silvermont-architecture-revealed-getting-serious-about-mobile. 2013.
David Kanter: "Silvermont, Intel's Low Power Architecture". http://www.realworldtech.com/silvermont/. 2013.

14 VIA Nano pipeline

Despite their small low-power design, the VIA Nano processors have a complete out-of-order microarchitecture with a functionality and performance not far from the more power-hungry desktop processors from Intel and AMD. It has a few weak spots, though, where the performance is inferior. The versions presently available have one, two or four cores running a single thread each. The Supplementary SSE3 and x64 instruction sets are supported in the Nano 2000 series. The Nano 3000 series also supports the SSE4.1 instruction set and virtualization instructions.

The microarchitecture is described by G. Glenn Henry in: "The VIA Isaiah Architecture", Centaur Technology, 2008 (www.via.com.tw).

14.1 Performance monitor counters

My research on the other processors described in the present manual has relied heavily on the use of performance monitor counters. The Nano also has performance monitor counters, but these are completely undocumented and intended only for internal use. I have found several counters for branch prediction, µops etc., but these counters are somewhat unreliable, especially on the 2000 series. The findings presented below are therefore based mainly on clock count measurements.

The counters I have found are listed in the source code for my test program at www.agner.org/optimize.

14.2 Instruction fetch

The maximum instruction fetch rate is 16 bytes per clock cycle, which is likely to be sufficient in most cases.

14.3 Instruction decoding

The decoders can handle three instructions per clock cycle, including instructions that generate multiple µops. Instructions with any number of prefixes are decoded without delay. There is no penalty for length-changing prefixes.

14.4 Instruction fusion

The decoders can fuse an integer ALU instruction and a branch instruction into a single µop. The instruction fusion works only if the following conditions are satisfied:

- The first instruction is one of the following: `CMP`, `ADD`, `SUB`, `INC`, `DEC`, `TEST`, `AND`, `OR`, `XOR`.
- The first instruction has one or two register operands, no immediate operand, and no memory operand.
- On the Nano 2000 series, the second instruction can only be `JE` or `JNE` (same as `JZ` and `JNZ`).
- On the Nano 3000 series, the second instruction can be any conditional jump. A conditional jump that reads the carry flag (i.e. unsigned comparisons such as `JA`, `JB`) cannot fuse with an ALU instruction that doesn't use the carry flag (i.e. `INC`, `DEC`).

- There can be no other instructions in between (except a `NOP` on the 3000).

For example, instruction fusion works for the combination `DEC ECX` / `JNZ LOOP1`, but not for `CMP ECX,1000` / `JNE LOOP2` because there is an immediate operand.

On the Nano 3000 series, a `NOP` can be fused with a preceding instruction so that it uses no resources in the execution units. A `NOP` can fuse with many common instructions, including SIMD instructions, but not with all instructions. This applies to the single-byte `NOP` (opcode 90) and the multi-byte `NOP` (opcode 0F 1F xxx), but not to `FNOP` and not to other instructions used as NOPs (e.g. `LEA`, `MOV`).

14.5 Out of order system

Register renaming, reservation station and µop dispatch work well with a throughput of three instructions per clock cycle.

14.6 Execution ports

There are seven clusters of execution units, each served by its own execution port. These are named as follows:

Port	Nano 2000	Nano 3000
I1	Integer add, Boolean, shift, rotate	Integer add, Boolean, shift, rotate, move
I2	Integer add, Boolean, move, jump	Integer add, Boolean, move, shift, multiply (32 bit), jump
MA	Multiplication, division, square root on all operand types	Multiplication of F.P., SIMD and 64-bit integer operands. Division and square root.
MB	All other operations on F.P. and SIMD operands	All other operations on F.P. and SIMD operands
SA	Address calculation for memory store and LEA	Address calculation for memory store and LEA
ST	Memory store	Memory store
LD	Memory load	Memory load

Table 14.1. Execution units in VIA Nano

Each execution port can handle one µop per clock cycle. The latency is one clock cycle for most integer operations. 32 bit integer multiplication takes 2 clock cycles on the 3000 series, 4-6 clock cycles on the 2000 series. Floating point addition takes 2 clock cycles. Floating point multiplication takes 3-4 clock cycles. These operations are pipelined to a throughput of one operation per clock cycle. Floating point multiplication with double precision has half the throughput. Division is not pipelined.

A memory store instruction uses both the SA and the ST ports, apparently using a fused µop that goes to both ports. A `LEA` instruction uses the SA port. Memory read instructions use the LD port only. Read-modify instructions generate two µops, one for the LD port and one for an execution port. The maximum throughput is one µop per clock cycle for each port.

Manual 4: "Instruction tables" lists the latencies and port use of each instruction. The table entries do not include the additional latencies described below.

There is no penalty for mixing µops with different latencies on the same execution port.

There is an inefficiency when the code contains mainly instructions for port I1 and I2 that modify general purpose registers. When the queues for both of these ports are full then the scheduler will submit all subsequent I1/I2 instructions for port I1. This leads to suboptimal use of port I2. This can be avoided by making sure that no more than 2/3 of the instructions use port I1 and I2.

The `NOP` instruction is quirky on the Nano 2000 series. It appears to block all execution ports so that no other instructions can execute in parallel. This applies only to the single-byte `NOP` instruction (opcode 90). The multi-byte `NOP` (opcode 0F 1F xxx) and the `FNOP` do not have this problem. The Nano 3000 series does not have this problem.

14.7 Latencies between execution units

There is often an extra latency when the output of one instruction is needed as input for a subsequent instruction in a different execution unit or subunit. The measured delays for moving data from one unit to another are listed in table 14.2.

	To port					
From port	**I1 / I2**	**MA**	**MB**	**MB-fadd**	**LD, ST**	**SA**
I1 / I2	0	1-2			0	0
MA	1	0	1	1	0	0-1
MB		0-1	0	0-1	1-2	0-1
MB-fadd		0-1	0-1	0	1	0-1
LD, ST	0	0-1	1	1	0	0
SA	2	1-2			0	0

Table 14.2. Latencies between execution units

The missing entries in table 14.2 are impossible operations or cannot be measured. The latencies listed in manual 4: "Instruction tables" do not include the additional latencies in table 14.2, except for instructions that move data between XMM and general purpose registers. The latencies between units are sometimes higher in the Nano 3000 series than in the 2000 series.

The floating point adder under MB is named MB-fadd here. Note that there are additional latencies between the floating point adder under MB and other units under MB when XMM registers are used, but not when x87 registers are used.

The latencies between units are illustrated in the following examples.

```
; Example 14.1a. Latencies between units
                         ; Unit    Latency
mov  rax,  [m1]          ; LD       2
add  rax,  2             ; I12      1
imul rax,  10            ; MA       3+2
add  rax,  rbx           ; I12      1+1
mov  [m2], rax           ; ST       2
                         ; Total:  12 expected, 12-13 measured

; Example 14.1b. Instructions reordered
                         ; Unit    Latency
mov  rax,  [m1]          ; LD       2
```

```
imul rax,  10               ; MA       3
add  rax,  2*10             ; I12      1+1
add  rax,  rbx              ; I12      1
mov  [m2], rax              ; ST       2
                            ; Total:  10 expected, 10-11 measured

; Example 14.1c. Using I12 unit only
                            ; Unit    Latency
mov  rax,  [m1]             ; LD       2
add  rax,  2                ; I12      1
mov  rcx,  rax              ; I2      (1)
shl  rax,  3                ; I1       1
add  rcx,  rcx              ; I12     (1)
add  rax,  rbx              ; I12      1
add  rax,  rcx              ; I12      1
mov  [m2], rax              ; ST       2
                            ; Total:   8 expected, 9 measured

; Example 14.1d. Using SA unit only
                            ; Unit    Latency
mov  rax,  [m1]             ; LD       2
lea  rax,  [rax+4*rax+10]; SA       1
lea  rax,  [rbx+2*rax]      ; SA       1
mov  [m2], rax              ; ST       2
                            ; Total:   6 expected, 6 measured
```

Examples 14.1 a-d are all doing the same thing. In 14.1a we are wasting 3 clock cycles on moving data from I12 to MA and back again. In 14.1b we have avoided a delay by reordering the instructions. In 14.1c we are using shifts instead of multiplication to avoid the transition to the MA unit. In 14.1d we are using `LEA` for both multiplication and addition. We are avoiding delays by keeping all the calculations in the SA unit. We would have an extra 2 clock delay if there was a mixture of `LEA` and `ADD` instructions (SA and I1/I2 unit respectively).

These transportation delays are likely to occur in dependency chains in the following situations:

- Instructions in general purpose registers mix multiplication (MA), division (MA) or LEA instructions (SA) with any other integer instructions (I1/I2). (On 3000 series, MA is used for 64-bit multiplication, but not for multiplication with 32-bits or less).

- Integer instructions in XMM registers mix multiplication (MA) with any other instructions (MB).

- Floating point instructions in XMM registers mix multiplication, division or square root (MA), addition (MB-fadd) and any other instructions (MB).

- Floating point instructions in x87 registers mix multiplication, division or square root (MA) with addition and other instructions (MB).

The number of transitions between these units should be minimized. Transitions between different units are unavoidable when reading and writing to memory and when transferring data between general purpose registers and SIMD registers.

Latencies between integer and floating point type XMM instructions

There is a considerable delay when mixing XMM instructions intended for integer data with XMM instructions intended for floating point data. Several XMM instructions have three different versions, one for integer data, one for single precision floating point, and one for double precision. For example `MOVDQA`, `MOVAPS`, `MOVAPD` and `POR`, `ORPS`, `ORPD`. There is a serious penalty on the Nano processor for mixing instructions with different types. The penalty can be up to 2 clock cycles for a transition from an integer-type instruction to a floating point-type instruction, and 3 clock cycles for a transition from a floating point-type instruction to an integer-type instruction. The most probable explanation is that integer and floating point XMM instructions use different execution units, different registers or different data buses.

The use of instructions of the wrong type commonly occurs in the following situations:

- The single precision versions of the instructions are one byte shorter than the integer or double precision versions. For example, some compilers use `MOVAPS` instead of `MOVDQA` to make the code shorter. This should be avoided because instruction fetching is rarely a bottleneck on the Nano.

- Some integer XMM instructions have no floating point equivalent, for example shift instructions and `PSHUFD`. The penalty for using these instructions in a floating point context is typically 3 clock cycles. Use `SHUFPS` or `SHUFPD` instead if possible.

- Some floating point instructions have no integer equivalent, for example `MOVSS`, `MOVSD` and `SHUFPS`. The penalty for using these instructions in an integer context is typically 5 clock cycles.

These penalties are independent of what kind of data the registers actually contain, only the transition between differently typed instructions matter. For example, there is no penalty for using `MOVAPS` to move integer data as long as no calculation is done on these data.

14.8 Partial registers and partial flags

There are no partial register stalls when modifying part of a general purpose register or XMM register. The register is always treated as a whole, which can cause false dependences. For example, a move to `AX` will have to wait until any preceding write to `EAX` has finished because it cannot split out `AX` as an independent register. Replace `MOV AX,[mem]` by `MOVZX EAX,word ptr [mem]` to avoid this. It is not necessary to extend the write to `RAX` because a write to `EAX` will neutralize the upper part of `RAX`.

The flags register is treated in the opposite way. The flags register can be split into parts to avoid false dependences for instructions that modify part of the flags (e.g. `INC`). Consequently, there is a stall of approximately 7 clock cycles in case the partial flags registers have to be combined together. Example:

```
; Example 14.2. Partial flags stall
add eax, 2      ; modifies all flags
inc ebx         ; modifies all flags except carry
setbe cl        ; needs carry flag from add and zero flag from inc
```

The partial flags stall is unavoidable in the `PUSHF` instruction.

14.9 Breaking dependence

The processor recognizes that a register is cleared when XOR'ed with itself, for example `XOR EAX,EAX` is recognized as independent of the prior value of `EAX`. The same applies to the following instructions:

Instruction	Nano 2000	Nano 3000
XOR	x	x
SUB	-	x

SBB	-	-
PXOR, XORPS, XORPD	x	x
PANDN, ANDNPS, ANDNPD	-	-
PSUBB/W/D/Q	-	x
PCMPEQW/D	-	x
PCMPEQQ	-	-
PCMGTW/D/Q	-	-

14.10 Memory access

Caches:

- Level 1 instruction cache. 64 kB, 16 way, set associative, 64 bytes line size
- Level 1 data cache. 64 kB, 16 way, set associative, 64 bytes line size
- Level 2 cache. 1 MB, 16 way, set associative, 64 bytes line size

Misaligned memory accesses are very expensive if a 32-bytes boundary is crossed on the Nano 2000 series or a 64-bytes boundary on the 3000 series. The latency is approximately 18 clock cycles for a misaligned read and 37 clock cycles for a misaligned write. There is no penalty if no 32/64-bytes boundary is crossed.

Store forwarding works efficiently only in simple cases:

Read size	**Address offset**	**Extra delay Nano 2000**	**Extra delay Nano 3000**
= write size	0	0	0
< write size	0	0	0
> write size	0	11	9
< write size	1	33	7
< write size	2-4	7	7
< write size	> 4	33	7
any size	only partial overlap	21-33	18

14.11 Branches and loops

The throughput for jumps, calls and taken branches is one jump per three clock cycles. Not-taken branches have a throughput of 1 per clock cycle. The minimum execution time for a loop is 3 clock cycles if the loop contains no 16 bytes boundary, and 4 clock cycles if a 16 bytes boundary is crossed inside the loop. Tight loops should therefore be aligned by 16 to keep the clock count at 3.

The branch target buffer can address only two jumps in each 16-bytes line of code. If an aligned 16-bytes block of code contains more than two taken jumps, calls or returns then the excess jumps have a latency of 8 clock cycles each.

Branch prediction works best if there is no more than one branch or jump in each 16-bytes block of code, but prediction is also reasonably good when there are two branches in a 16-bytes block of code. See page 30 for details. Returns are predicted well.

14.12 VIA specific instructions

The processor has special instructions for encryption, decryption and hashing. These complex instructions run at a speed of 1-4 clock cycles per byte of data.

There is also an instruction for random number generation based on electrical noise. This instruction takes between 1300 and 19200 clock cycles to generate 8 bytes of random bits, depending on the desired quality and the processor version. Random bits are sampled continuously into an 8-bytes buffer while the processor is doing other tasks. The `XSTORE` instruction gets 8 random bytes only if the buffer is full, while the `REP XSTORE` instruction waits until enough random data have been sampled.

This instruction is too slow for generating a long sequence of random numbers for Monte Carlo applications, but it is sufficient for generating the seed for a pseudo random number generator or a cryptographic key.

I have tested the quality of the generated random numbers with the DIEHARD battery of tests (en.wikipedia.org/wiki/Diehard_tests). The random numbers generated by the `REP XSTORE` instruction fail many of the tests when the quality factor is set to the lowest value. This was expected because physical generators of randomness are known to be imperfect. With a higher value of the quality factor, it passes all the tests. The higher quality is obtained by sampling noise over a longer time interval and by manipulating the random data with a "whitener" with an undisclosed algorithm.

A programming reference for these instructions can be found in: "VIA PadLock Programming Guide", www.via.com.tw, May 2005.

14.13 Bottlenecks in Nano

The execution units in the Nano are quite powerful for such a small processor, and the execution latencies are very low. There are, however, a number of weak points:

- Misaligned memory accesses are very costly.
- Store forwarding works only in relatively simple cases.
- Data cache access is limited to one read or one write per clock cycle. Only rarely can it read and write simultaneously.
- Branch density limit. There is a limit of only two jumps or branches per 16 bytes of code. Performance is decreased when this limit is exceeded.
- Branch throughput. The Nano can make no more than one taken branch or jump every three clock cycles.
- There is suboptimal queueing when most instructions go to the I1 and I2 execution ports.
- There are many extra latencies for moving data between different execution units or subunits.

The main differences between the Nano 2000 and Nano 3000 series lie in the execution units. The Nano 3000 has two execution units for most integer instructions. It has a new integer multiplication unit with a 2-clock latency. Division is improved, and there are several minor improvements in SIMD instructions. Misaligned memory access and store forwarding is somewhat improved, but still less efficient than in bigger processors.

The conclusion is that the single-core Nano may be insufficient for the most demanding and memory intensive applications. The low price and low power consumption makes it useful for many common purposes, such as office applications, embedded applications and low traffic servers.

15 AMD K8 and K10 pipeline

15.1 The pipeline in AMD K8 and K10 processors

The AMD microprocessors are based on the same principles of out-of-order execution and register renaming as Intel desktop processors.

Instructions are split up as little as possible and as late as possible in the pipeline. Each read-modify macro-instruction is split into a read and a modify micro-instruction in the execution stage and joined together into the macro-operation before retirement. A macro-operation in AMD terminology is somewhat similar to a fused micro-operation in Intel terminology. The K8 microarchitecture has no execution units bigger than 64 or 80 bits, while the K10 microarchitecture has 128-bit execution units in the floating point pipeline so that 128-bit XMM instructions can be handled in a single macro-instruction.

The most important difference from Intel's microarchitecture is that the AMD micro-architecture contains three parallel pipelines. The instructions are distributed between the three pipelines right after the fetch stage. In simple cases, the instructions stay in each their pipeline all the way to retirement.

The exact length of the pipelines is not known but it can be inferred that it has approximately twelve stages, based on the fact that the branch misprediction penalty is measured to 12 clock cycles.

The following list of stages is based on publications from AMD as well as an independent analysis published by Chip Architect.

1. Instruction fetch 1. 32 bytes per clock cycle on K10, 16 bytes on K7 and K8.

2. Instruction fetch 2 and branch prediction.

3. Pick/Scan. Can buffer up to 7 instructions. Distributes three instructions into the three decoder pipelines. The following stages are all split into three parallel pipes.

4. Decode 1. Splits the instruction codes into their components.

5. Decode 2. Determines input and output registers.

6. Pack. Up to six macro-operations generated from the decoders are arranged into lines of three macro-operations for the three execution pipelines.

7. Pack/Decode. Register renaming. Integer registers are read from the "Integer Future File and Register File". Submits integer macro-operations to the three integer pipes. Submits floating point macro-operations to the floating point pipes.

   Integer pipes:

8. Dispatch. Sends macro-operations to a reservation station with 3x8 entries.

9. Schedule. Schedules macro-operations out of order. Read-modify and read-modify-write macro-operations are split into micro-operations which are submitted to the arithmetic logic units (ALU), the address generation units (AGU), and the load/store units.

10. Execution units and address generation units. Each of the three integer pipes has one ALU and one AGU. Integer arithmetic, logic and shift operations are executed in this stage. Integer multiplication can only be handled in pipe 0. All other integer

instructions can be handled by any of the three integer pipes.

11. Data cache access. Submits a read or write request to the data cache.

12. Data cache response. The data cache returns a hit or miss response for read operations.

13. Retirement. Macro-operations are retired in order. The data cache stages are skipped if there is no memory operation.

Floating point pipes:

8. Stack map. Maps floating point stack registers to virtual registers.

9. Register renaming.

10. Dispatch. Scheduler write.

11. Scheduler. Uses a reservation station with 3x12 entries to schedule instructions out of order. Micro-operations are targeted for one of the three floating point execution units.

12. Register read. Reads source operand register values.

13. Execution units. The three execution units in the floating point pipes are named FADD, FMUL and FMISC. These units are specialized for each their purpose. The floating point execution units are fully pipelined in three stages to handle floating point operations with latencies longer than one. Integer vector operations are handled by the floating point units, not the integer ALU's.

14. - 16. The processes of address generation, cache access and retirement are probably shared with the integer pipelines.

The floating point pipeline is longer than the integer pipeline because of the extra stages for stack map, register renaming, and register read. The minimum measured latency of floating point instructions is 2 clock cycles because of the extra register read stage. The maximum latency of floating point instructions is 4 clock cycles.

The length of the floating point pipeline is difficult to measure, because the branch misprediction penalty measures only the length of the integer pipeline.

You may think of the pipeline structure as consisting of an in-order front end, an out-of-order execution core, and an in-order retirement unit. However, the linear picture of the pipeline as sketched above is somewhat misleading, because some processes take place in parallel, more or less independently of each other. Address generation takes several clock cycles and may start before the ALU operation. Read-modify and read-modify-write macro-operations are split into micro-operations that go to different units and at different times in the out-of-order core. The in-order front end, the branch prediction unit, address-generation units, load-store units, integer arithmetic-logic units, and floating point units are all separate structures with each their pipeline. The integer ALU units and the floating point units can do operations out of order. The load-store units are doing all memory reads in order and all memory writes in order, while a read can be executed before a subsequent write. The other units are doing all operations in order.

The time it takes to calculate an address and read from that address in the level-1 cache is 3 clock cycles if the segment base is zero and 4 clock cycles if the segment base is nonzero, according to my measurements. Modern operating systems use paging rather than segmentation to organize memory. You can therefore assume that the segment base is zero in 32-bit and 64-bit operating systems (except for the thread information block which is accessed through `FS` or `GS`). The segment base is almost always nonzero in 16-bit systems in protected mode as well as real mode.

Complex instructions that require more than two macro-operations are so-called vector path instructions. These instructions make exclusive use of all three slots in a decode line, a reorder buffer line, etc. so that no other instructions can go in parallel. The macro-operations are generated from microcode ROM in stage 3 - 5 in the pipeline.

The K7 processor does not have double instructions. It uses the vector path process for all instructions that require more than one macro-operation. Otherwise, the microarchitecture of the K7 processor is very similar to the 64-bit architecture of K8 and K10, as outlined above. Earlier AMD processors have a different microarchitecture which will not be treated here.

Literature:
AMD Athlon™ Processor x86 Code Optimization Guide, Feb. 2002.
AMD Software Optimization Guide for AMD64 Processors, 2005, 2008.
Fred Weber: AMD’s Next Generation Microprocessor Architecture. Oct. 2001.
Hans de Vries: Understanding the detailed Architecture of AMD's 64 bit Core, 2003. www.chip-architect.com.
Yury Malich: AMD K10 Micro-Architecture. 2007. www.xbitlabs.com.

The research on the AMD pipeline has been carried out with the help of Andreas Kaiser and Xucheng Tang.

15.2 Instruction fetch

The instruction fetcher can fetch 32 bytes of code per clock cycle from the level-1 code cache on K10. On K7 and K8, it can fetch 16 bytes of code per clock cycle into a 32 bytes buffer. Instruction fetching can therefore be a bottleneck on the older processors if the code contains many long instructions or many jumps. The delivery bandwidth of code from the level-2 cache has been measured to 4.22 bytes per clock on average for K10, and 2.56 bytes per clock on K8.

The fetched packets are aligned by 32 on K10 and by 16 on K7 and K8. This has implications for code alignment. Critical subroutine entries and loop entries should not start near the end of a 16 byte block. You may align critical entries by 16 or at least make sure there is no 16-byte boundary in the first three instructions after a critical label.

Branch information stored in the code cache and the branch target buffer is used for fetching code after predicted branches. The throughput for jumps and taken branches is one jump per two clock cycles. I take this as an indication that the fetch buffer can only contain contiguous code. It cannot span across a predicted branch.

15.3 Predecoding and instruction length decoding

An instruction can have any length from 1 to 15 bytes. The instruction boundaries are marked in the code cache and copied into the level-2 cache. Instruction length decoding is therefore rarely a bottleneck, even though the instruction length decoder can handle only one instruction per clock cycle.

The level-1 code cache contains a considerable amount of predecode information. This includes information about where each instruction ends, where the opcode byte is, as well as distinctions between single, double and vector path instructions and identification of jumps and calls. Some of this information is copied to the level-2 cache, but not all. The low bandwidth for instructions coming from the level-2 cache may be due to the process of adding more predecode information.

My experiments show that it is possible to decode three instructions in one clock cycle on K8 even if the third instruction starts more than 16 bytes after the first one, provided that there are enough bytes left in the 32 byte buffer.

The throughput of the microprocessor is three instructions per clock cycle, even for an instruction stream that contains predicted jumps. We know that a jump incurs a delay bubble in the instruction fetch process, but there is a buffer between fetch and decoding which enables it to catch up after this delay.

A recommendation in some versions of AMD's Optimization Guide says that decoding can be improved by aligning groups of three instructions by eight. This is done by inserting dummy prefixes to make each group of three instructions exactly eight bytes long. This recommendation is obsolete, according to my measurements. The decoders can always handle three relatively short instructions per clock cycle regardless of alignment. There is no advantage in making instructions longer. Only in rare cases with relatively long instructions have I observed an improvement by making instructions longer to make groups of instructions a multiple of 8 bytes long (regardless of alignment). But making instructions longer is more likely to have a negative effect.

Each of the instruction decoders can handle three prefixes per clock cycle. This means that three instructions with three prefixes each can be decoded in the same clock cycle. An instruction with 4 - 6 prefixes takes an extra clock cycle to decode.

15.4 Single, double and vector path instructions

- Instructions that generate one macro-operation are called direct path single instructions.

- Instructions that generate two macro-operations are called direct path double instructions (K8 only).

- Instructions that generate more than two macro-operations are called vector path instructions.

The number of macro-operations generated by each instruction is listed in manual 4: "Instruction tables".

There is no difference in throughput between using one double instruction or two single instructions, except for the reduction in code size. The throughput is still limited to three macro-operations per clock cycle, not three instructions per clock cycle. The source of this bottleneck is most probably the retirement stage. If the bottleneck was in the schedulers then we would not expect a double instruction in the floating point scheduler to limit the throughput of the integer schedulers, or vice versa.

Vector path instructions are less efficient than single or double instructions because they require exclusive access to the decoders and pipelines and do not always reorder optimally. For example:

```
; Example 15.1. AMD instruction breakdown
xchg  eax, ebx   ; Vector path, 3 ops
nop              ; Direct path, 1 op
xchg  ecx, edx   ; Vector path, 3 ops
nop              ; Direct path, 1 op
```

This sequence takes 4 clock cycles to decode because the vector path instructions must decode alone.

Most read-modify and read-modify-write instructions generate only one macro-operation. These instructions are therefore more efficient than using separate read and modify instructions.

The latency from the memory operand to the result of a read-modify instruction is the same as the latency of a read plus the latency of the arithmetic operation. For example, the instruction `ADD EAX,[EBX]` has a latency of 1 from `EAX` input to `EAX` output, and a latency of 4 from `EBX` to `EAX` output in 32 bit mode. An 8-bit or 16-bit memory read behaves like a read-modify instruction. For example, `MOV AX,[EBX]` takes one clock cycle more than `MOV EAX,[EBX]`.

A macro-operation can have any number of input dependencies. This means that instructions with more than two input dependencies, such as `MOV [EAX+EBX],ECX`, `ADC EAX,EBX` and `CMOVBE EAX,EBX`, generate only one macro-operation, while they require two micro-operations on Intel processors.

15.5 Stack engine

The K10 has a stack engine very similar to the one on Intel processors (p. 92). This makes stack operations (`PUSH`, `POP`, `CALL`, `RET`) more efficient on K10 than on earlier processors.

15.6 Integer execution pipes

Each of the three integer execution pipes has its own ALU (Arithmetic Logic Unit) and its own AGU (Address Generation Unit). Each of the three ALU's can handle any integer operation except multiplication. This means that it is possible to do three single integer instructions in the same clock cycle if they are independent. The three AGU's are used for memory read, write and complex versions of the `LEA` instruction. It is possible to do two memory operations and one `LEA` in the same clock cycle. It is not possible to do three memory operations because there are only two ports to the data cache.

The K10 can do a `LEA` instructions with no more than two operands in the ALU's, even if it has a SIB byte. A `LEA` instruction with a scale factor or with both base register, index register and addend is executed in the AGU. It is unknown whether a `LEA` with a RIP-relative address is executed in the AGU. A `LEA` executed in the AGU has a latency of 2 clock cycles. If the AGU and ALU are at the same stage in the pipeline, as the model suggests, then the extra latency is most likely explained by assuming that there is no fast data forwarding path between these two units.

Integer multiplication can only be done in ALU0. A 32 bit integer multiplication takes 3 clock cycles and is fully pipelined so that a new multiplication can begin every clock cycle. Integer multiplication instructions with the accumulator as an implicit operand and only one explicit operand generate a double sized result in `DX`:`AX`, `EDX`:`EAX` or `RDX`:`RAX`. These instructions use ALU1 for the high part of the result. It is recommended to use multiplication instructions that do not produce a double sized result in order to free ALU1 and `EDX` for other purposes. For example, replace `MUL EBX` with `IMUL EAX,EBX` if the result can fit into 32 bits.

15.7 Floating point execution pipes

The three execution units in the floating point pipes are named FADD, FMUL and FMISC. FADD can handle floating point addition. FMUL can handle floating point multiplication and division. FMISC can handle memory writes and type conversions. All three units can handle memory reads. The floating point units have their own register file and their own 80-bits data bus.

The latency for floating point addition and multiplication is 4 clock cycles. The units are fully pipelined so that a new operation can start every clock cycle. Division takes 11 clock cycles and is not fully pipelined. The latency for move and compare operations is 2 clock cycles.

The 3DNow instructions have the same latency as the XMM instructions. There is hardly any advantage in using 3DNow instructions rather than XMM instructions unless you need the approximate reciprocal instruction which is more accurate and efficient in the 3DNow version than in the XMM version. The 3DNow instruction set is obsolete and not available on newer microprocessors.

SIMD integer operations in MMX and XMM registers are handled in the floating point pipelines, not the integer pipelines. The FADD and FMUL pipes both have an integer ALU that can handle addition, Boolean and shift operations. Integer multiplications are handled only by FMUL.

The minimum latency for the floating point units is 2 clock cycles. This latency is caused by the pipeline design, not by lower clock frequency or staggered addition. Most SIMD integer ALU operations have a latency of 2 clocks. Integer multiplication has a latency of 3 clocks. The units are fully pipelined so that a new operation can start every clock cycle.

Macro-operations for the floating point units can be divided into five categories according to which execution unit they are going to. I will designate these categories as follows:

macro-operation category	**Handled by unit**		
	FADD	**FMUL**	**FMISC**
FADD	X		
FMUL		X	
FMISC			X
FA/M	X	X	
FANY	X	X	X

Table 15.1. AMD floating point macro-operation categories

The floating point scheduler sends each macro-operation to a unit that can handle it. A macro-operation in the FA/M category can go to either the FADD or the FMUL unit. A macro-operation in the FANY category can go to any of the three units. The categories for all floating point instructions are listed under "Instruction timings and µop breakdown for AMD" in manual 4: "Instruction tables".

Unfortunately, the scheduler does not distribute the macro-operations optimally among the three units. Macro-operations in the FA/M category are scheduled according to the simplest possible algorithm: FA/M macro-operations go alternately to FADD and FMUL. This algorithm makes sure that two FA/M macro-operations submitted in the same clock cycle do not go to the same unit. A status bit is remembering which of the two units was used last, and this bit can be remembered indefinitely. I have found no way to reset it.

The scheduling algorithm for macro-operations in the FANY category is only slightly more sophisticated. A macro-operation in the FANY category goes preferentially to the unit that is determined by which of the three pipelines it happens to come from. The first FANY macro-operation after a series of integer macro-operations goes to FMISC. A second FANY macro-operation in the same clock cycle goes to FMUL, and a possible third FANY macro-operation goes to FADD. If other macro-operations submitted in the same clock cycle need a particular floating point unit then the FANY macro-operations can be redirected to a different unit.

The floating point scheduler does not check whether a particular unit is vacant or has a long queue when deciding where to send a macro-operation of category FA/M or FANY. If, for example, an instruction stream generates ten macro-operations of category FADD and then one macro-operation of category FA/M, then there is a 50% probability that the FA/M macro-operation will go to the FADD unit although it could save a clock cycle by sending it to FMUL.

This suboptimal scheduling of macro-operations can significantly slow down the execution of code with many floating point instructions. This problem can be diagnosed by testing a small critical piece of code with a performance monitor counter set up for each of the three floating point units. The problem is difficult to solve, however. Sometimes it is possible to improve the distribution of macro-operations by changing the instruction order, by using different instructions, or by inserting `NOP`s. But there is no general and reliable way to solve the problem.

Another scheduling problem which can have even worse consequences is explained in the next paragraph.

15.8 Mixing instructions with different latency

There is a scheduling problem when mixing instructions with different latencies. The floating point execution units are pipelined so that if a macro-operation with latency 4 starts at time=0 and finishes at time=3, then a second macro-operation of the same kind can start at time=1 and end at time=4. However, if the second macro-operation has latency 3 then it cannot start at time=1 because it would end at time=3, simultaneously with the preceding macro-operation. There is only one result bus for each execution unit and this prevents macro-operations from delivering their results simultaneously. The scheduler prevents a clash between two results by not dispatching a macro-operation to an execution unit if it can be predicted that the result bus will not be vacant when the macro-operation is finished. It is not able to redirect the macro-operation to another execution unit.

This problem is illustrated by the following example:

```
; Example 15.2. AMD mixing instruction with different latencies (K8)
                     ; Unit      time op 1   time op 2
mulpd  xmm0, xmm1    ; FMUL         0-3         1-4
mulpd  xmm0, xmm2    ; FMUL         4-7         5-8
movapd xmm3, xmm4    ; FADD/FMUL    0-1         8-9
addpd  xmm3, xmm5    ; FADD         2-5        10-13
addpd  xmm3, xmm6    ; FADD         6-9        14-17
```

Each instruction in this example generates two macro-operations, one for each of the 64-bit parts of the 128-bit register. The first two macro-operations are multiplications with a latency of 4 clock cycles. They start at time=0 and 1, and end at time=3 and 4, respectively. The second two multiplication macro-operations need the results of the preceding macro-operations. Therefore they cannot start until time=4 and 5, and end at time=7 and 8, respectively. So far so good. The `MOVAPD` instruction generates two macro-operations of category FA/M with latency 2. One of these macro-operations goes to the FADD pipe which is vacant so that this macro-operation can start immediately. The other macro-operation from `MOVAPD` goes to the FMUL pipe because macro-operations in the FA/M category alternate between the two pipes. The FMUL pipe is ready to start executing a new macro-operation at time=2, but the `MOVAPD` macro-operation can't start at time=2 because then it would end at time=3 where the first `MULPD` macro-operation finishes. It can't start at time=3 because then it would end at time=4 where the second `MULPD` macro-operation finishes. It can't start at time=4 or 5 because the next two `MULPD` macro-operations start there. It can't start at time=6 or 7 because then the result would clash with the results of the next two `MULPD` macro-operations. So time=8 is the first possible start time for this macro-operation. The consequence is that the subsequent additions, which are dependent on the `MOVAPD`, will be delayed 7 clock cycles even though the FADD unit is vacant.

There are two ways to avoid the problem in the above example. The first possibility is to reorder the instructions and put the `MOVAPD` before the two `MULPD` instructions. This would make the two macro-operations from MOVAPD both start at time=0 in the FADD and the FMUL unit, respectively. The subsequent multiplications and additions will then run in the two pipes without interfering with each other.

The second possible solution is to replace `XMM4` by a memory operand. The `MOVAPD XMM3,[MEM]` instruction generates two macro-operations for the FMISC unit which is vacant in this example. There is no conflict between macro-operations in different execution pipes, regardless of differences in latency.

The throughput is of course higher on K10 than on K8, but the deadlock problem can still occur for all instructions that use floating point registers, MMX registers or XMM registers. As a general guideline, it can be said that the deadlock can occur when a macro-operation with latency 2 follows after at least two macro-operations with a longer latency scheduled for the same floating point execution unit and the macro-operations are independent in the sense that one doesn't have to wait for the result of another. The deadlock can be avoided by putting the instructions with short latency first or by using instructions that go to different execution units.

The latencies and execution units of the most common macro-operations are listed below. A complete list can be found in manual 4: "Instruction tables". Remember that a 128-bit instruction typically generates one macro-operation on K10 and two macro-operations on K8.

Macro-operation type	Latency	Execution unit
register-to-register move	2	FADD/FMUL alternate
register-to-memory move	2	FMISC
memory-to-register 64 bit	4	any
memory-to-register 128 bit	4	FMISC
integer addition	2	FADD/FMUL alternate
integer Boolean	2	FADD/FMUL alternate
shift, pack, unpack, shuffle	2	FADD/FMUL alternate
integer multiplication	3	FMUL
floating point addition	4	FADD
floating point multiplication	4	FMUL
floating point division	11	FMUL (not pipelined)
floating point compare	2	FADD
floating point max/min	2	FADD
floating point reciprocal	3	FMUL
floating point Boolean	2	FMUL
type conversion	2-4	FMISC

Table 15.2. Execution units in AMD

15.9 64 bit versus 128 bit instructions

It is a big advantage to use 128-bit instructions on K10, but not on K8 because each 128-bit instruction is split into two 64-bit macro-operations on the K8.

128 bit memory write instructions are handled as two 64-bit macro-operations on K10, while 128 bit memory read is done with a single macro-operation on K10 (2 on K8).

128 bit memory read instructions use only the FMISC unit on K8, but all three units on K10. It is therefore not advantageous to use XMM registers just for moving blocks of data from one memory position to another on the k8, but it is advantageous on K10.

15.10 Data delay between differently typed instructions

XMM instructions come in three different types according to the types of operands they are intended for:

1. Integer instructions. Most of these instructions have a name that begins with `P` for Packed, for example `POR`.

2. Single precision floating point instructions. These instructions have a name that ends with `SS` (Scalar Single precision) or `PS` (Packed Single precision), for example `ORPS`.

3. Double precision floating point instructions. These instructions have a name that ends with `SD` (Scalar Double precision) or `PD` (Packed Double precision), for example `ORPD`.

The three instructions `POR`, `ORPS` and `ORPD` are doing exactly the same thing. They can be used interchangeably, but there is a delay when the output from an integer instruction is used as input for a floating point instruction, or vice versa. There are two possible explanations for this delay:

Explanation 1: The XMM registers have some tag bits that are used for remembering whether floating point values are normal, subnormal or zero. These tag bits have to be set when the output of an integer instruction is used as input for a single or double precision floating point instruction. This causes a so-called reformatting delay.

Explanation 2: There is no fast data forwarding path between the integer and floating point SIMD units. This gives a delay analogously to the delay between execution units on the P4.

Explanation 2 is supported by the fact that there is no delay between single precision and double precision floating point instructions, but there is a delay from floating point to integer instructions.

There is no difference in delay after instructions that read from memory and do no calculation. And there is no delay before instructions that write to memory and do no calculation. Therefore, you may use the `MOVAPS` instruction rather than the one byte longer `MOVAPD` or `MOVDQA` instructions for reading and writing memory.

There is often a two clock cycle delay when the output of a memory read instruction (regardless of type) is used as input to a floating point instruction. This is in favor of explanation 1.

It does not make sense to use the wrong type of instruction for arithmetic operations, but for instructions that only move data or do Boolean operations there may be advantages in using the wrong type in cases where no delay is incurred. The instructions with names ending in `PS` are one byte shorter than other equivalent instructions.

15.11 Partial register access

The processor always keeps the different parts of an integer register together. Thus, `AL` and `AH` are not treated as independent by the out-of-order execution mechanism. This can cause false dependences in a code that writes to part of a register. For example:

```
; Example 15.3. AMD partial register access
imul  ax, bx
mov   [mem1], ax
mov   ax, [mem2]
```

In this case the third instruction has a false dependence on the first instruction caused by the high half of EAX. The third instruction writes to the lower 16 bits of EAX (or RAX) and these 16 bits have to be combined with the rest of EAX before the new value of EAX can be written. The consequence is that the move to AX has to wait for the preceding multiplication to finish because it cannot separate the different parts of EAX. This false dependence can be removed by inserting an XOR EAX,EAX instruction before the move to AX, or by replacing MOV AX,[MEM2] by MOVZX EAX,[MEM2].

The above example behaves the same regardless of whether it is executed in 16 bit mode, 32 bit mode or 64 bit mode. To remove the false dependence in 64 bit mode it is sufficient to neutralize EAX. It is not necessary to neutralize the full RAX because a write to the lower 32 bits of a 64 bit register always resets the high half of the 64 bit register. But a write to the lower 8 or 16 bits of a register does not reset the rest of the register.

This rule does not apply to the XMM registers on K8. Each 128-bit XMM register is stored as two independent 64-bit registers on K8.

15.12 Partial flag access

The processor splits the arithmetic flags into at least the following groups:

1. Zero, Sign, Parity and Auxiliary flags
2. Carry flag
3. Overflow flag
4. Non-arithmetic flags

This means that an instruction that modifies only the carry flag has no false dependence on the zero flag, but an instruction that modifies only the zero flag has a false dependence on the sign flag. Examples:

```
; Example 15.4. AMD partial flags access
add  eax,1       ; Modifies all arithmetic flags
inc  ebx         ; Modifies all except carry flag. No false dependence
jc   L           ; No false dependence on EBX
bsr  ecx,edx     ; Modifies only zero flag. False depend. on sign flag
sahf             ; Modifies all except overflow flag
seto al          ; No false dependence on AH
```

15.13 Store forwarding stalls

There is a penalty for reading from a memory position immediately after writing to the same position if the read is larger than the write, because the store-to-load forwarding mechanism doesn't work in this case. Examples:

```
; Example 15.5. AMD store forwarding
mov    [esi],eax    ; Write 32 bits
mov    bx,[esi]     ; Read 16 bits. No stall
movq   mm0,[esi]    ; Read 64 bits. Stall
movq   [esi],mm1    ; Write after read. No stall
```

There is also a penalty if the read doesn't start at the same address as the write:

```
; Example 15.6. Store forwarding stall
mov    [esi],eax    ; Write 32 bits
mov    bl,[esi]     ; Read part of data from same address. No stall
```

```
mov     cl,[esi+1]    ; Read part of data from different address. Stall
```

There is also a penalty if the write originates from `AH`, `BH`, `CH` or `DH`:

```
; Example 15.7. Store forwarding stall for AH
mov     [esi],al      ; Write 8 bits
mov     bl,[esi]      ; Read 8 bits. No stall
mov     [edi],ah      ; Write from high 8-bit register
mov     cl,[edi]      ; Read from same address. Stall
```

15.14 Loops

The branch prediction mechanism for the AMD K8 and K10 processors is described on page 31.

The speed of small loops is often limited by instruction fetching on the AMD. A small loop with no more than 6 macro-operations can execute in 2 clock cycles per iteration if it contains no more than one jump and no 32-byte boundary on K10 or 16-byte boundary on K8. It will take one clock extra per iteration if there is a 32-byte boundary in the code because it needs to fetch an extra 32-byte block on K10, or 16-byte boundary on K7 or K8.

The maximum fetching speed can be generalized by the following rule:

> The minimum execution time per iteration for a loop is approximately equal to the number of 32-byte boundaries on K10, or 16-byte boundaries on K8, in the code plus 2 times the number of taken branches and jumps.

Example:

```
; Example 15.8. AMD branch inside loop
     mov ecx,1000
L1:  test bl,1
     jz L2
     add eax,1000
L2:  dec ecx
     jnz L1
```

Assume that there is a 32-byte boundary at the `JNZ L1` instruction. Then the loop will take 3 clocks if the `JZ L2` doesn't jump, and 5 clocks if the `JZ` jumps. In this case, we can improve the code by inserting a `NOP` before `L1` so that the 32-byte boundary is moved to `L2`. Then the loop will take 3 and 4 clocks respectively. We are saving one clock count in the case where `JZ L2` jumps because the 32-byte boundary has been moved into the code that we are bypassing.

These considerations are only important if instruction fetching is the bottleneck. If something else in the loop takes more time than the computed fetching time then there is no reason to optimize instruction fetching.

15.15 Cache

The level-1 code cache and the level-1 data cache are both 64 Kbytes, 2 way set associative and a line size of 64 bytes. The data cache has two ports which can be used for either read or write. This means that it can do two reads or two writes or one read and one write in the same clock cycle. Each read port is 128 bits wide on K10, 64 bits on K8. The write ports are 64 bits on both K8 and K10. This means that a 128-bit write operation requires two macro-operations.

Each 64 byte line in the code cache line is split into 4 blocks of 16 bytes each. Each 64 byte line in the data cache is split into 8 banks of 8 bytes each. The data cache cannot do two memory operations in the same clock cycle if they use the same bank, except for two reads from the same cache line:

```
; Example 15.9. AMD cache bank conflicts
mov eax,[esi]      ; Assume ESI is divisible by 40H
mov ebx,[esi+40h]  ; Same cache bank as EAX. Delayed 1 clock
mov ecx,[esi+48h]  ; Different cache bank

mov eax,[esi]      ; Assume ESI is divisible by 40H
mov ebx,[esi+4h]   ; Read from same cache line as EAX. No delay

mov [esi],eax      ; Assume ESI is divisible by 40H
mov [esi+4h],ebx   ; Write to same cache line as EAX. Delay
```

(See Hans de Vries: Understanding the detailed Architecture of AMD's 64 bit Core, Chip Architect, Sept. 21, 2003. www.chip-architect.com. Dmitry Besedin: Platform Benchmarking with Right Mark Memory Analyzer, Part 1: AMD K7/K8 Platforms. www.digit-life.com.)

Operations with memory access are executed with the use three different units with each their pipeline: (1) An Arithmetic Logic Unit (ALU) or one of the floating point units, (2) an Address Generation Unit (AGU), (3) a Load Store Unit (LSU). The ALU is used for read-modify and read-modify-write instructions, but not for instructions that only read or write. The AGU and LSU are used for all memory instructions. The LSU is used twice for read-modify-write instructions. While the ALU and AGU micro-operations can be executed out of order, the LSU micro-operations are processed in order in most cases. The rules are as follows, as far as I am informed:

1. Level-1 data cache hit reads are processed in order.

2. Level-1 data cache miss reads proceed in any order.

3. Writes must proceed in order.

4. A read can go before a preceding write to a different address.

5. A read depending on a prior write to the same address can proceed as soon as the forwarded data is available.

6. No read or write can proceed until the addresses of all prior read and write operations are known.

It is recommended to load or calculate the values of pointer and index registers as early as possible in the code in order to prevent the delaying of subsequent memory operations.

The fact that memory operations must wait until the addresses of all prior memory operations are known can cause false dependences, for example:

```
; Example 15.10. AMD memory operation delayed by prior memory
operation
imul eax, ebx       ; Multiplication takes 3 clocks
mov  ecx, [esi+eax] ; Must wait for EAX
mov  edx, [edi]     ; Read must wait for above
```

This code can be improved by reading `EDX` before `ECX` so that the reading of `EDX` doesn't have to wait for the slow multiplication.

There is a stall of one clock cycle for misaligned memory references if the data crosses an 8-bytes boundary. The misalignment also prevents store-to-load forwarding. Example:

```
; Example 15.11. AMD misaligned memory access
mov ds:[10001h],eax  ; No penalty for misalignment
mov ds:[10005h],ebx  ; 1 clock penalty when crossing 8-byte boundary
mov ecx,ds:[10005h]  ; 9 clock penalty for store-to-load forwarding
```

Level-2 cache

The level-2 cache has a size of 512 Kbytes or more, 16 ways set-associative with a line size of 64 bytes and a bus size of 16 bytes. Lines are evicted by a pseudo-LRU scheme.

Data streams can be prefetched automatically with positive or negative strides. Data are prefetched only to the level-2 cache, not to the level-1 cache.

The level-2 cache includes bits for automatic error correction when used for data, but not when used for code. The code is read-only and can therefore be reloaded from RAM in case of parity errors. The bits that have been saved by not using error correction for code are used instead for copying the information about instruction boundaries and branch prediction from the level-1 cache.

Level-3 cache

The K10 has a level-3 cache of 2 MB. It is likely that versions with different level-3 cache sizes will become available. The level-3 cache is shared between all cores, while each core has its own level-1 and level-2 caches.

15.16 Bottlenecks in AMD K8 and K10

It is important, when optimizing a piece of code, to find the limiting factor that controls execution speed. Tuning the wrong factor is unlikely to have any beneficial effect. In the following paragraphs, I will explain each of the possible limiting factors in the AMD microarchitecture.

Instruction fetch

The instruction fetch is limited to 16 bytes of code per clock cycle on K8 and earlier processors. This can be a bottleneck when the rest of the pipeline can handle three instructions per clock cycle. Instruction fetch is unlikely to be a bottleneck on K10.

The throughput for taken jumps is one jump per two clock cycles. Instruction fetch after a jump is delayed even more if there is a 16-byte boundary in the first three instructions after the jump. It is recommended to align the most critical subroutine entries and loop entries by 16 or at least make sure the critical jump targets are not near the end of an aligned 16-byte block. The number of jumps and 16-byte boundaries in small loops should be kept as small as possible. See above, page 186.

Out-of-order scheduling

The maximum reordering depth is 24 integer macro-operations plus 36 floating point macro-operations. Memory operations cannot be scheduled out of order.

Execution units

The execution units have a much larger capacity than it is possible to utilize. It is alleged that the nine execution units can execute nine micro-operations simultaneously, but it is virtually impossible to verify this claim experimentally since the retirement is limited to three macro-operations per clock cycle. All three integer pipelines can handle all integer operations except multiplication. The integer execution units can therefore not be a bottleneck except in code with an extraordinary high number of multiplications.

A throughput of 3 macro-operations per clock cycle can be obtained when no execution unit receives more than one third of the macro-operations. For floating point code, it is difficult to obtain a perfectly equal distribution of macro-operations between the three floating point units. Therefore, it is recommended to mix floating point instructions with integer instructions.

The floating point scheduler does not distribute macro-operations optimally between the three floating point execution units. A macro-operation may go to a unit with a long queue while another unit is vacant. See page 181.

All floating point units are pipelined for a throughput of one macro-operation per clock cycle, except for division and a few other complex instructions.

Mixed latencies

Mixing macro-operations with different latencies scheduled for the same floating point unit can seriously prevent out-of-order execution. See page 182.

Dependency chains

Avoid long dependency chains and avoid memory intermediates in dependency chains. A false dependence can be broken by writing to a register or by the following instructions on the register with itself: `XOR`, `SUB`, `SBB`, `PXOR`, `XORPS`, `XORPD`. For example, `XOR EAX,EAX`, `PXOR XMM0,XMM0`, but not `XOR AX,AX`, `PANDN XMM0,XMM0`, `PSUBD XMM0,XMM0` or compare instructions. Note that `SBB` has a dependence on the carry flag.

Accessing part of a register causes a false dependence on the rest of the register, see page 184. Accessing part of the flag register does not cause a false dependence, except in rare cases, see page 185.

Jumps and branches

Jumps and branches have a throughput of one taken branch every two clock cycles. The throughput is lower if there are 16-byte boundaries shortly after the jump targets. See page 186.

The branch prediction mechanism allows no more than three taken branches for every aligned 16-byte block of code. Jumps and branches that always go the same way are predicted very well if this rule is obeyed. See page 31.

Dynamic branch prediction is based on a history of only 8 or 12 bits. Furthermore, the pattern recognition often fails for unknown reasons. Branches that always go the same way do not pollute the branch history register.

Retirement

The retirement process is limited to 3 macro-operations per clock cycle. This is likely to be a bottleneck if any instructions generate more than one macro-operation.

16 AMD Bulldozer, Piledriver and Steamroller pipeline

16.1 The pipeline in AMD Bulldozer, Piledriver and Steamroller

The AMD Bulldozer, Piledriver and Steamroller processors can have from one to eight compute units with two execution cores per unit. It can run one thread per core or two threads per unit.

Instruction cache and fetching is shared between the two cores in an execution unit. The instruction decoder is also shared between two cores in Bulldozer and Piledriver, while the Steamroller has one decoder for each core.

Integer execution units and level-1 data cache is separate for each core. The floating point and vector execution units and level-2 cache are also shared between the two cores of an execution unit. A possible level-3 cache is shared between all compute units. Some versions have an integrated graphics processing unit.

Each core contains four parallel pipelines and can execute up to four instructions per clock cycle. Instructions are split up as little as possible and as late as possible in the pipeline. A read-modify or read-modify-write instruction generates only one macro-operation at the decode stage. The length of the pipeline is not known.

The design has more focus on power saving than previous designs. It saves power quite aggressively by slowing down the clock speed most of the time. Some versions also lower the voltage to the CPU when the clock speed is reduced. The maximum clock speed is only obtained after a long sequence of CPU-intensive code.

16.2 Instruction fetch

The instruction fetcher is shared between the two cores of an execution unit. The instruction fetcher can fetch 32 aligned bytes of code per clock cycle from the level-1 code cache. The measured fetch rate was up to 16 bytes per clock per core when two cores were active, and up to 21 bytes per clock in linear code when only one core was active. The fetch rate is lower than these maximum values when instructions are misaligned.

Critical subroutine entries and loop entries should not start near the end of a 32-bytes block. You may align critical entries by 16 or at least make sure there is no 16-bytes boundary in the first four instructions after a critical label.

16.3 Instruction decoding

Instruction boundaries are marked in the code cache. Each decoder can handle four instructions per clock cycle. The Bulldozer and Piledriver have one decoder in each unit, which is shared between two cores. When both cores are active, the decoders serve each core every second clock cycle, so that the maximum decode rate is two instructions per clock cycle per core. Instructions that belong to different cores cannot be decoded in the same clock cycle. The decode rate is four instructions per clock cycle when only one thread is running in each execution unit.

The Steamroller has one decoder per core so that it can decode four instructions per core per clock, even when running two threads in each unit. Therefore, the bottleneck is most likely to be instruction fetching rather than decoding on the Steamroller.

Instructions that generate two macro-ops are called double instructions. The decoders in the Piledriver and Steamroller can handle four single instructions (1-1-1-1) or one double instruction and two single instructions (2-1-1) or two double instructions (2-2) in one clock cycle. The Bulldozer can handle (1-1-1-1) and (2-1-1), but not (2-2).

Instructions that generate more than two macro-ops are using microcode. The decoders cannot do anything else while microcode is generated. This means that a decoder can stop decoding for several clock cycles after meeting an instruction that generates more than two macro-ops. On the Steamroller, this will affect only the thread that has the complex instruction, but on Bulldozer and Piledriver it will delay both threads in a unit because they share the same decoder. The number of macro-ops generated by each instruction is listed in manual 4: "Instruction tables".

Instructions with up to three prefixes can be decoded in one clock cycle. There is a very large penalty for instructions with more than three prefixes. Instructions with 4-7 prefixes take 14-15 clock cycles extra to decode. Instructions with 8-11 prefixes take 20-22 clock cycles extra, and instructions with 12-14 prefixes take 27 - 28 clock cycles extra. It is therefore not recommended to make NOP instructions longer with more than three prefixes. The prefix count for this rule includes operand size, address size, segment, repeat, lock, REX and XOP prefixes. A three-bytes VEX prefix counts as one, while a two-bytes VEX prefix does not count. Escape codes (0F, 0F38, 0F3A) do not count.

16.4 Loop buffer

The Steamroller has a queue of decoded macro-ops after the decoder. There is one queue for each core. The queue can hold up to 40 macro-ops, though sometimes a little less. Small loops can bypass the decoder and run from the macro-op queue. This saves power and removes the bottleneck of instruction fetching for tiny loops. A loop with no more then 4 instructions can execute in just one clock cycle per iteration on the Steamroller because it bypasses the decoder.

16.5 Instruction fusion

A `CMP` or `TEST` instruction immediately followed by a conditional jump can be fused into a single macro-op. This applies to all versions of the `CMP` and `TEST` instructions and all conditional jumps except if the `CMP` or `TEST` has a rip-relative address or both a displacement and an immediate operand. Examples:

```
; Example 16.1. Instruction fusion on Bulldozer
test eax,4
jnz L1                  ; fused into one op
cmp [Mydata],eax        ; rip-relative address in 64 bit mode
jb L2                   ; not fused if rip-relative address
cmp dword ptr[rsi+8],2  ; both displacement and immediate operand
jl  L3                  ; not fused
cmp [Mydata+rbx*4],eax  ; 32-bit absolute address + scaled index
jg L3                   ; fused
dec ecx
jnz L4                  ; not fused. Only cmp and test can fuse
```

(The AMD software optimization guide 3.06, Jan 2012 is inaccurate here)

No other ALU instruction can fuse with a conditional jump. The maximum decode rate is not increased by instruction fusion.

16.6 Stack engine

The processor has an efficient stack engine that renames the stack pointer, but we do not know exactly where it is placed in the pipeline.

Push, pop and return instructions use only a single macro-op. These instructions have zero latency with respect to the stack pointer, so that subsequent instructions that depend on the stack pointer, either as operand or as pointer, are not delayed. No extra stack synchronization µops of the type seen in Intel processors have been observed.

16.7 Out-of-order schedulers

Each core has an out-of-order Integer scheduler with 40 entries and a physical register file of 96 registers of 64-bit.

The shared floating point unit has its own out-of-order scheduler with 60 entries and a physical register file of 160 registers of 128-bit.

These numbers may be bigger for the Steamroller, but exact details are not available.

16.8 Integer execution pipes

There are four integer execution pipes:

Integer pipe	Used for
EX0	Most ALU operations, division
EX1	Most ALU operations, multiplication, jump
AGLU0	Memory read
AGLU1	Memory read

Table 16.1. Integer execution pipes

The execution pipes EX0 and EX1 are used for most integer and general purpose instructions. Memory read instructions use AGLU0 and AGLU1. Memory write instructions use both AGLU0/1 and EX0/1 simultaneously. AGLU0 and 1 can also handle simple register-to-register moves with 32-bit and 64-bit general purpose registers, except on early versions of Bulldozer. AGLU0 and 1 can not handle register move instructions with 8-bit or 16-bit registers or an immediate operand.

LEA instructions are executed as ALU operations in EX0 and EX1. Simple LEA instructions take one clock cycle. If shifting or more than one addition is involved then it takes two clocks. If the operand size or address size is 16 bits then it takes an extra clock.

Integer multiplication of operands up to 32 bits takes 4 clock cycles with a throughput or one multiplication per 2 clocks. Integer division is not pipelined.

16.9 Floating point execution pipes

There are four floating point/vector execution pipes on Bulldozer and Piledriver, but only three on Steamroller.

Floating point pipe	Used for
P0	f.p. addition, multiplication, division, integer vector multiplication
P1	f.p. addition, multiplication, division, shuffle, shift, pack
P2	integer vector addition, boolean, move
P3	integer vector addition, boolean, move, store

Table 16.2. Bulldozer and Piledriver floating point execution pipes

Floating point pipe	Used for
P0	f.p. addition, multiplication, division, boolean, integer vector addition, multiplication
P1	f.p. addition, multiplication, division, shuffle, shift, pack
P2	integer vector addition, vector boolean, store

Table 16.3. Steamroller floating point execution pipes

All these units can handle 128-bit operands. 256-bit operations are split into two macro-ops.

All floating point additions and multiplications take 5 clock cycles if the next dependent instruction is also an addition or multiplication, otherwise possibly 6 clock cycles. A fused multiply-and-add instruction also takes 5 or 6 clock cycles. It is unknown how the execution unit saves one clock cycle when the result is used in the same unit. It might be due to a shorter data path, or it might be that the execution unit can save one pipeline stage of normalization, formatting or categorization of the floating point number.

The latency for move and compare operations and for simple integer vector instructions is 2 clock cycles.

Most execution units are doubled, as table 16.2 and 16.3 show, so that the throughput is two 128-bit operations or one 256-bit operation per clock cycle. Normally, a macro-op goes to the unit that is vacant first.

The store unit is not doubled, and 256-bit stores always take more than one clock cycle. The Bulldozer has a throughput of at most one 256-bit store per 3 clock cycles if aligned, and one per 10 clock cycles if unaligned. The Piledriver is particularly bad on 256-bit stores with a throughput of one aligned 256-bit store per 17 clock cycles in my measurements. The Steamroller is much better, with a throughput of one 256-bit store per 2 clock cycles if aligned, 4 clock cycles if unaligned.

The Steamroller has lower throughput than expected for floating point addition. The throughput is only one 128-bit vector per clock cycle in single-threaded applications, even though there are two 128-bit execution units for floating point addition. It is sometimes possible to use both add units simultaneously when running two threads.

Mixing instructions with different latency on the same pipe rarely causes problems.

The 3DNow instructions are no longer supported, except for the prefetch instructions.

Subnormal operands

The Bulldozer and Piledriver have a penalty of approximately 175 extra clock cycles when the result of a floating point operation is a subnormal or underflow, unless the flush-to-zero mode is activated. There is no penalty for overflow. The Steamroller has none of these penalties.

Fused multiply and add

The floating point units can do fused multiply-and-add (FMA) instructions of the type $d = a * b + c$. The Bulldozer supports FMA4 instructions, where the four operands can be different registers. The Piledriver and Steamroller support both FMA3 and FMA4. The FMA3 instructions have three operands, where the destination *d* must use the same register as one of the input operands *a*, *b* or *c*. The Piledriver has only half the expected throughput for FMA3, namely one 128-bit vector per clock, but full throughput for FMA4, i.e. two 128-bit vectors per clock. The Steamroller has full throughput for both FMA3 and FMA4.

16.10 AVX instructions

The throughput of 256-bit AVX instructions is generally one 256-bit vector per clock cycle while the throughput of 128-bit instructions is two 128-bit vectors per clock cycle for the most common instructions. Therefore, the overall throughput is roughly the same regardless of whether we are using 128-bit or 256-bit instructions.

There are a few disadvantages of using 256-bit instructions on Bulldozer and Piledriver:

- The instruction decoders cannot handle two double instructions per clock cycle on the Bulldozer.

- The throughput of 256-bit store instructions is less than half the throughput of 128-bit store instructions on Bulldozer and Piledriver. It is particularly bad on the Piledriver, which has a throughput of one 256-bit store per 17 - 20 clock cycles.
- 128-bit register-to-register moves have zero latency, while 256-bit register-to-register moves have a latency of 2 clocks plus a penalty of 2-3 clocks for using a different domain (see below) on Bulldozer and Piledriver. Register-to-register moves can be avoided in most cases thanks to the non-destructive 3-operand instructions.

Therefore, there is no advantage in using 256-bit instructions on Bulldozer and Piledriver when the bottleneck is execution unit throughput or instruction decoding. The poor throughput of 256-bit stores makes it a disadvantage to use 256-bit registers on the Piledriver. These problems have been fixed on the Steamroller.

While Intel processors have a large penalty for mixing 256-bit AVX instructions with non-AVX XMM instructions due to a mode switch (see page 131), there is no such penalty and apparently no mode switch on these AMD processors.

The AVX instruction set gives three-operand versions of most XMM instructions, both integer and floating point. This has the advantage that no operand register is overwritten so that most register-to-register moves can be avoided. There is no disadvantage of using the three-operand instructions other than incompatibility with older processors.

16.11 Data delay between different execution domains

There is often a delay when output data from one execution unit is used as input in another execution unit. The different execution units can be divided into six domains, where there is a transport delay when data are moved from one domain to another:

1. int domain. This includes all operations on general purpose registers.
2. ivec domain. This includes all integer vector operations as well as floating point move, pack, shuffle, blend and boolean operations.
3. fma domain. This includes all floating point add, subtract and multiply operations, including fused multiply-and-add instructions.
4. fp domain. This includes other floating point instructions, such as division, square root, round, etc.
5. load. This includes all memory read instructions.
6. store. This includes memory write instructions.

The transport delays between these domains have been measured as follows:

From domain	To domain				
	int	**ivec**	**fp**	**fma**	**store**
int	0	(10)	n.a.	n.a.	(4)
ivec	(8)	0	1	1	(5)
fp	n.a.	1	0	0	1(+5)
fma	n.a.	1	0	-1	1(+5)
load	(4)	(6)	(6)	(6)	n.a.

Table 16.4. Data transport delays, clock cycles. Numbers in parenthesis are included in the latency counts listed in manual 4: "Instruction tables".

Note that many floating point instructions belong to the integer vector (ivec) domain. For example, there are no special floating point versions of the boolean instructions. The `POR`, `ORPS` and `ORPD` instructions are all identical. Example:

```
; Example 16.2a. Data transport delays on Bulldozer
movaps   xmm0, [mem1]   ; 6   clock
```

```
mulps   xmm0, xmm1    ; 6+1 clock
xorps   xmm0, xmm2    ; 2+1 clock
addps   xmm0, xmm3    ; 6+1 clock
movaps  [mem2], xmm0  ; 5   clock
                      ; 28  clock total
```

This can be improved by reordering the instructions so that the number of transitions between domains is reduced (the `xorps` instruction is used for changing sign here, so that this reordering is allowed).

```
; Example 16.2b. Data transport delays on Bulldozer, improved
movaps  xmm0, [mem1]  ; 6   clock
xorps   xmm0, xmm2    ; 2+1 clock
mulps   xmm0, xmm1    ; 6-1   clock
addps   xmm0, xmm3    ; 6+1 clock
movaps  [mem2], xmm0  ; 5   clock
                      ; 26  clock total
```

The transport delays between the integer unit and the floating point/vector unit are much longer in my measurements than specified in AMD's Software Optimization Guide. Nevertheless, I cannot confirm that it is faster to move data from a general purpose register to a vector register through a memory intermediate, as recommended in that guide.

There is a large penalty when the output of a floating point calculation is input to a floating point calculation with a different precision, for example if the output of a double precision floating point addition is input to a single precision addition. This has hardly any practical significance since such a sequence is most likely to be a programming error, but it indicates that the processor stores extra information about floating point numbers beyond the 128 bits in an XMM register. This effect is not seen on Intel processors.

16.12 Instructions that use no execution units

The `NOP`, `FNOP` and `FWAIT` instructions are resolved without being sent to any execution unit. They have a throughput of 4 instructions per clock cycle.

128 bit register-to-register moves are implemented as register renaming without being sent to any execution unit. Therefore, they have a latency of zero and a throughput of four instructions per clock cycle. The same register can even be moved/renamed four times in one clock cycle. The instructions `MOVDQA`, `MOVDQU`, `MOVAPS`, `MOVUPS`, `MOVAPD` and `MOVUPD` are all identical when used with register operands.

256 bit register-to-register moves are different. The low half of the YMM register is renamed in the same way as with 128-bit register moves with zero latency, but the high half is moved by an execution unit in pipe P2 or P3 with a latency of 2 clock cycles. In addition to this latency comes a possible transport delay in floating point code because the move is executed in the integer vector domain (see table 16.4). Example:

```
; Example 16.3a. YMM move with transport delays on Bulldozer
vaddps  ymm0, ymm0, ymm1   ; 6   clock
vmovaps ymm2, ymm0         ; 2+2 clock
vmulps  ymm2, ymm2, ymm3   ; 6   clock
                           ; 16  clock total
```

Here, we can eliminate the move and save 5 clock cycles by taking advantage of the non-destructive 3-operand instructions:

```
; Example 16.3b. YMM move eliminated
vaddps  ymm0, ymm0, ymm1   ; 6-1 clock
vmulps  ymm2, ymm0, ymm3   ; 6   clock
```

```
; 11  clock total
```

The x87 floating point instructions `FINCSTP`, `FDECSTP` and `FFREE` are also resolved by renaming without using any execution unit. The `FXCH` instruction is only partly resolved by renaming: It has zero latency, but uses an execution unit at pipe P0 or P1.

All other register moves use execution units, including general purpose register moves.

16.13 Partial register access

The processor always keeps the different parts of an integer register together. For example, `AL` and `AH` are not treated as independent by the out-of-order execution mechanism. An instruction that writes to part of a register will therefore have a false dependence on any previous write to the same register or any part of it.

An instruction that writes to a 32-bit register will not have any false dependence on the corresponding 64-bit register because the upper part of the 64-bit register is set to zero.

A write to a part of an XMM register has a false dependence on the whole register, but this does not apply to the two halves of a YMM registers. A 256 bit YMM register is treated as two independent 128 bit XMM registers. However, this rarely has any practical consequences because the instructions `VEXTRACTF128` and `VINSERTF128` are treated by the sequencer as if they read/write both halves of the register.

16.14 Partial flag access

The processor treats certain parts of the arithmetic flags as independent. For example, an instruction that modifies only the carry flag has no false dependence on the zero flag.

16.15 Dependency-breaking instructions

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EBX,EBX`. The processor recognizes that certain instructions are independent of the prior value of the register if the source and destination registers are the same. The following instructions are recognized as independent of the input when both operands are the same register: `XOR`, `SUB`, `SBB` (depends on carry flag only), `CMP`, `PXOR`, `PANDN`, `PSUBx`, `PCMPEQB`, `PCMPEQW`, `PCMPEQD`, `PCMPGTB`, `PCMPGTW`, `PCMPGTD`, `XORPS`, `XORPD`, `ANDNPS`, `ANDNPD`.

This does not work with 8-bit and 16-bit registers due to the treatment of partial registers, but it works with 32-bit registers and larger.

The instructions `PCMPEQQ`, `PCMPGTQ` are recognized as independent on Steamroller, but not on Bulldozer and Piledriver.

Floating point subtractions and compares are never recognized as independent because they have to treat special cases such as NAN and INF.

The XOP instructions `VPCOMB`, `VPCOMW`, `VPCOMD`, `VPCOMQ` are recognized as independent of the input only when the immediate operand is 6 (set to 0) or 7 (set to all 1s).

The zeroing instructions are using the same execution units as when the two operands are different.

16.16 Branches and loops

The branch prediction mechanism is described on page 33. There is no longer any restriction on the number of branches per 16 bytes of code that can be predicted efficiently. The misprediction penalty is quite high because of a long pipeline.

The speed of small loops is most often limited by instruction fetching. On Bulldozer and Piledriver, a small loop takes at least 2 clock cycles per iteration if it contains no more than one taken jump and no 32-bytes boundary. Larger loops are limited by the instruction fetch rate or by the instructions they contain.

Tiny loops with up to four instructions can execute in just 1 clock cycle per iteration on the Steamroller.

16.17 Cache and memory access

Cache	Bulldozer	Piledriver	Steamroller
Level 1 code	64 kB, 2-way, 64 B line size, shared between two cores.	64 kB, 2-way, 64 B line size, shared between two cores.	96 kB, 3-way, 64 B line size, shared between two cores.
Level 1 data	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.
Level 2	1 - 2 MB, 16-way, 64 B line size, shared between two cores. Latency 21 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 20 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 19 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 6 clock.
Level 3	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	None

Table 16.5. Cache sizes on AMD Bulldozer, Piledriver and Steamroller

The data cache has two 128-bit ports which can be used for either read or write. This means that it can do two reads or one read and one write in the same clock cycle.

The measured throughput is two reads or one read and one write per clock cycle when only one thread is active. We would not expect the throughput to be less when multiple threads are active because each core has separate load/store units and level-1 data cache. But my measurements indicate that level-1 cache throughput is several times lower when multiple threads are running, even if the threads are running in different units that do not share any level-1 or level-2 cache. This phenomenon is seen on both Bulldozer, Piledriver and Steamroller. No explanation for this effect has been found. Level-2 cache throughput is shared between two threads running in the same unit, but not affected by threads running in different units.

Unaligned reads and writes have a throughput of one read or write per clock cycle. The throughput is one read or write in 2 or 3 clock cycles when a cache line boundary is crossed, and one in 21 clock cycles when a memory page boundary is crossed.

The processor can do a read before a pending write to a different address.

The level-2 cache has a disappointingly poor performance on the Bulldozer in certain benchmark tests for unknown reasons. The Piledriver and Steamroller has a much more efficient cache system. The Steamroller appears to have more write buffers than the previous models.

The level-1 data cache is organized as 16 banks of 16 bytes each. The data cache cannot do two memory operations in the same clock cycle if they use banks that are spaced by a multiple of 100h bytes, except for two reads from the same cache line. This kind of cache bank conflict occurs very often:

```
; Example 16.4. Cache bank conflicts
mov eax,[rsi]       ; Assume rsi is divisible by 100H
mov ebx,[rsi+200h]  ; Cache bank conflict. Delayed 1 clock

mov eax,[rsi]       ; Assume rsi is divisible by 100H
mov ebx,[rsi+210h]  ; Different cache bank, no conflict

mov eax,[rsi]       ; Assume rsi is divisible by 100H
mov ebx,[rsi+10h]   ; Same cache line, no conflict
```

There is a false dependence when the address of a memory read is spaced a multiple of 1000h bytes from a preceding write:

```
; Example 16.5. False memory dependence
mov  [rsi],eax
mov ebx,[rsi+2000h] ; False dependence on previous write
mov ecx,[rsi+2004h] ; No false dependence
```

16.18 Store forwarding stalls

Forwarding data from a store to a subsequent read from the same address is faster on Steamroller than on the previous designs, especially for vector registers.

There is a penalty of 25-26 clock cycles for reading from a memory position immediately after writing to the same position if the read is larger than the write, because the store-to-load forwarding mechanism doesn't work in this case. Examples:

```
; Example 16.6. AMD store forwarding
mov     [esi],eax    ; Write 32 bits
mov     bx,[esi]     ; Read 16 bits. No stall
movq    mm0,[esi]    ; Read 64 bits. Stall
movq    [esi],mm1    ; Write after read. No stall
```

On Bulldozer and Piledriver, there is a similar penalty if the read doesn't start at the same address as the write:

```
; Example 16.7. Bulldozer and Piledriver store forwarding stall
mov     [esi],eax    ; Write 32 bits
mov     bl,[esi]     ; Read part of data from same address. No stall
mov     cl,[esi+1]   ; Read part of data from different address. Stall
```

On the Steamroller, there is no penalty for reading part of the address written to:

```
; Example 16.8. Steamroller store forwarding stall
movdqa  [esi],xmm0   ; Write 128 bits
mov     eax,[esi+8]  ; Read part of the data. No stall
vmovaps [esi],ymm0   ; Write 256 bits as 2*128 bits
vmovaps xmm1,[esi+8] ; Crossing between two 128 bit writes. Stall
```

16.19 Bottlenecks in AMD Bulldozer, Piledriver and Steamroller

The AMD Bulldozer is a major redesign of previous microarchitectures. Some of the most important improvements are:

- Four pipelines giving a maximum throughput of 4 instructions per clock cycle.
- Improved floating point unit with high throughput
- Better scheduling of macro-ops to the first vacant execution unit
- Some register-to-register moves are translated into register renaming
- Branch prediction is no longer tied to the code cache and there is no limitation on the number of branches per code cache line
- AVX instruction set with non-destructive 3-operand instructions
- Efficient fused multiply-and-add instructions

The Piledriver and Steamroller are similar, with some improvements. Various possible bottlenecks are discussed in the following paragraphs.

Power saving

The power saving features are reducing the clock frequency most of the time. This often gives inconsistent results in performance tests because the clock frequency is varying. It is sometimes necessary to turn off the power saving features or put a long sequence of CPU-intensive code before the code under test in order to measure the maximum performance.

Shared resources

The instruction fetch is shared between the two cores that make a compute unit. The branch predictor and the floating point units are also shared. The instruction decoder is shared on the Bulldozer and Piledriver, while the Steamroller has one decoder per thread. Some operating systems do not have enough information about resource sharing so that they may put two threads into the same compute unit while another compute unit is idle.

Instruction fetch

The shared instruction fetch unit can fetch up to approximately 20 bytes per clock in single-threaded applications and 16 bytes per clock in multi-threaded applications. This is very likely to be a bottleneck when the average instruction length is more than 4 bytes or when frequent jumps produce bubbles in the pipeline.

Instruction decoding

On Bulldozer and Piledriver, the shared decode unit can handle four instructions per clock cycle. It is alternating between the two threads so that each thread gets up to four instructions every second clock cycle, or two instructions per clock cycle on average. This is a serious bottleneck because the rest of the pipeline can handle up to four instructions per clock.

The situation gets even worse for instructions that generate more than one macro-op each. All instructions that generate more than two macro-ops are handled with microcode. The microcode sequencer blocks the decoders for several clock cycles so that the other thread is stalled in the meantime.

Instruction decoding is less likely to be a bottleneck in the Steamroller because it has one decoder per core, giving a throughput of four instructions per clock per thread.

Out-of-order scheduling

On Bulldozer and Piledriver, the integer out-of-order scheduler has 40 entries, the shared floating point scheduler probably has somewhat more. This is a significant improvement over previous designs. The Steamroller has more entries and physical registers, according to some rumors, but this has not been independently confirmed.

Execution units

The integer execution units are poorly distributed between the four pipes. Two of the pipes have all the integer execution units while the other two pipes are used only for memory read instructions and address generation (not `LEA`), and on some models for simple register moves. This means that the processor can execute only two integer ALU instructions per clock cycle, where previous models can execute three. This is a serious bottleneck for pure integer code. The single-core throughput for integer code can actually be doubled by doing half of the instructions in vector registers, even if only one element of each vector is used.

The floating point execution units are better distributed so that all three or four pipes can be used. The most commonly used units are all doubled, including floating point addition, multiplication and division, as well as integer addition and boolean operations. All units are 128 bits wide. This gives a high throughput for 128-bit vector code which is likely sufficient to serve two threads simultaneously in many cases. All 256-bit vector instructions are split into two 128-bit operations so that there is generally no advantage in using 256-bit vectors. Bulldozer and Piledriver have four pipes for floating point and vector operations, while Steamroller has only three.

The fused multiply-and-add instructions are very efficient. They are doing one addition and one multiplication in the same time that it otherwise takes to do one addition *or* one multiplication. This effectively doubles the throughput of floating point code that has an equal number of additions and multiplications. The incompatibility of the Bulldozer's FMA4 instructions with Intel's FMA3 instructions is actually not AMD's fault, as discussed on my blog.

256-bit memory writes

256-bit memory write operations are exceptionally slow on the Piledriver. So slow, indeed, that it is better to not use 256-bit registers at all on the Piledriver. This problem has been fixed in the Steamroller.

Mixed latencies

Mixing operations with different latencies will cause less problems than on previous processors.

Dependency chains

Latencies for floating point instructions and integer vector instructions are relatively long. Long dependency chains should therefore be avoided. Accessing part of a register causes a false dependence on the rest of the register.

Jumps and branches

Jumps and branches have a throughput of one taken branch every two clock cycles. The throughput is lower if there are 32-byte boundaries shortly after the jump targets. Branch prediction is reasonably good, even for indirect jumps. The branch misprediction penalty is quite high because of a long pipeline.

Memory and cache access

The level-2 cache has rather poor performance on the Bulldozer, while it is much better on the Piledriver and Steamroller. Cache bank conflicts are very frequent and often impossible to avoid. Cache bank conflicts turned out to be a serious bottleneck in some of my tests. The code cache has only two or three ways which is quite low when we consider that it has to service two threads.

Retirement

There is no evidence that retirement can be a bottleneck.

16.20 Literature

Software Optimization Guide for AMD Family 15h Processors. AMD, January 2012.

David Kanter: AMD's Bulldozer Microarchitecture. Real World Technologies, 2010.

M. Golden, S. Arekapudi and J. Vinh: 40-Entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86–64 core. *Solid-State Circuits Conference Digest of Technical Papers* (ISSCC), 2011 IEEE International.

D. Weiss, *et. al.*: An 8MB level-3 cache in 32nm SOI with column-select aliasing. *Solid-State Circuits Conference Digest of Technical Papers* (ISSCC), 2011 IEEE International.

17 AMD Bobcat and Jaguar pipeline

The Bobcat and Jaguar are smaller microprocessors with low clock frequency and with focus on low power consumption. It saves power quite aggressively by slowing down the clock speed most of the time. In my tests on the Bobcat, the full speed was seen only after millions of instructions of CPU intensive code. This power saving feature ("CoolNQuiet") can be turned off in the BIOS setup when higher speed is desired.

The Bobcat has two cores while the successor Bobcat currently has up to four cores, and with plans about future 8-core versions. No execution resources are shared between the cores, and the cores do not interfere with each other unless they both access the same memory address.

The Jaguar supports the AVX instruction set, but not AMDs XOP instruction set. The AMD 3DNow instruction set is no longer supported, and may be regarded as obsolete. The `PREFETCH` instruction is still supported, though.

17.1 The pipeline in AMD Bobcat and Jaguar

The pipeline is designed for a throughput of 2 instructions or μops per clock cycle with two integer ALUs, two floating point/vector units, and two AGUs for load and store, respectively. The pipeline has schedulers for out-of-order execution. There are two separate pairs of pipelines, one for general purpose integer instructions and one for floating point and vector instructions. The general purpose integer unit has the two integer ALUs, the load unit and the store unit. The floating point and vector unit has two pipes.

The design has two physical register files, one for integer registers and one for floating point and vector registers. The size of the physical register files is stated as 64 entries on the Bobcat, which probably means 64 entries for each of the two physical register files. Temporary register values are not moved or shifted down the pipeline but stored in the physical register files while only an index or pointer to the physical register is stored in the pipeline. The physical registers can be renamed.

The Bobcat has 64-bit physical registers and uses two such registers to save a 128-bit vector. The Jaguar supports the AVX instruction set with 256-bit vectors. It has 128-bit physical registers and uses two such registers to save a 256-bit vector.

17.2 Instruction fetch

The instruction fetch rate is stated as "up to 32 bytes per cycle", but this is not confirmed by my measurements which consistently show a maximum of 16 bytes per clock cycle on average for both Bobcat and Jaguar. Some reports say that the Jaguar has a loop buffer, but I cannot detect any improvement in performance for tiny loops.

17.3 Instruction decoding

Unlike previous AMD designs, the Bobcat and Jaguar do not mark instruction boundaries in the code cache. It decodes two instructions per clock cycle. There is no penalty for multiple prefixes or length-changing prefixes. There is no fusion of branch instruction.

17.4 Single, double and complex instructions

The most common integer instructions generate only one µop, including read-modify-write, push and pop instructions. Call instructions generate two µops.

The Bobcat has 64-bit execution units and it splits all 128-bit vector instructions into at least two independent 64-bit µops. The Jaguar has 128-bit execution units and splits all 256-bit vector instructions into two independent 128-bit µops. (The Bobcat does not support the 256-bit AVX instructions). Instructions that generate more than two µops are generated from microcode ROM.

17.5 Integer execution pipes

There are two execution pipelines for integer µops with two almost identical ALUs. A few integer instructions, including multiplication, division, and some versions of `LEA`, can only be handled by integer pipe 0, other instructions can be handled by either pipe. Integer division uses the floating point division circuit with a large extra delay on Bobcat. Integer division is much faster on Jaguar.

There is one load unit and one store unit, both 64 bits wide on Bobcat and 128 bits wide on Jaguar.

The Bobcat reorder buffer has 56 entries, though the measured reordering capacity is generally less than 56 µops.

17.6 Floating point execution pipes

There are two floating point execution pipelines, which are also used for integer vector operations. Both pipelines can handle move, integer addition, and Boolean operations. Floating point pipe 0, here called FP0, handles integer multiplication and floating point addition. FP1 handles floating point multiplication and division. The division circuit is not pipelined and has no early out feature, hence the latency for division is independent of the values of the operands.

Memory read and write operations use the integer load/store unit with an extra delay of one or more clock cycles.

The penalty for underflow and subnormal operands is between 150 and 200 clock cycles on Bobcat. The Jaguar has penalties of 40 clock cycles for addition of subnormal numbers and 170 - 210 clock cycles when a multiplication gives a subnormal result or underflow.

While Intel processors have a large penalty for mixing 256-bit AVX instructions with non-AVX XMM instructions due to a mode switch (see page 131), there is no such penalty the AMD Jaguar.

17.7 Mixing instructions with different latency

There is little or no measurable penalty for mixing µops with different latency on the same execution pipeline.

17.8 Dependency-breaking instructions

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EBX,EBX`. The Bobcat processor recognizes that certain instructions are independent of the prior value of the register if the source and destination registers are the same. The following instructions are recognized as independent of the input when both operands are the same register: `XOR`, `SUB`, `SBB` (depends on carry flag only), `CMP`, `PXOR`, `PANDN`, `PSUBx`, `PCMPEQx`, `PCMPGTx`, `XORPS`, `XORPD`. The instructions `ANDNPS` and `ANDNPD` are recognized so on Jaguar but not on Bobcat.

The Jaguar can eliminate a move from a 128-bit register that has been zeroed in this way (or by `VZEROALL`), for example:

```
; Example 17.1. Propagation of zero on Jaguar
pxor   xmm0,xmm0     ; set to zero
movdqa xmm1,xmm0     ; move eliminated
movdqa xmm2,xmm1     ; move eliminated
movdqa xmm3,xmm2     ; move eliminated
```

No other case of move elimination is supported.

17.9 Data delay between differently typed instructions

There is an extra delay of 1 clock cycle when output from an integer vector instruction is used as input for a floating point vector instruction, or vice versa. For example, there is a penalty for using `MOVAPS` in connection with integer vector instructions, or `MOVDQA` in connection with floating point instructions.

There is a penalty of 40 clock cycles when the output of a floating point calculation is input to a floating point calculation with a different precision, for example if the output of a double precision floating point addition is input to a single precision addition. This has hardly any practical significance since such a sequence is most likely to be a programming error, but it indicates that the processor stores extra information about floating point numbers beyond the 128 bits in an XMM register.

17.10 Partial register access

The processor always keeps the different parts of a register smaller than 128 bits together. For example, `AL` and `AH` are not treated as independent by the out-of-order execution mechanism. But the high and low 64 bits of a 128 bit register are treated as independent on Bobcat, and the high and low 128 bits of a 256 bit register are treated as independent on Jaguar.

17.11 Cache

The cores have separate level-1 caches. The level-2 cache is separate for each core on Bobcat, but shared between up to four cores on Jaguar. Future Jaguar processors with eight cores are expected to have two level-2 caches, each shared between four cores.

Cache	Bobcat	Jaguar

Level 1 code	32 kB, 2 way, 64 B line size, latency 3, per core	32 kB, 2 way, 64 B line size, latency 3, per core
Level 1 data	32 kB, 8 way, 64 B line size, latency 3, per core	32 kB, 8 way, 64 B line size, latency 3, per core
Level 2	512 kB, 16 way, 64 B line size, half clock frequency, per core.	Up to 2 MB, 16 way, 64 B line size, half clock frequency, latency 26, shared between four cores.

Table 17.1. Cache sizes on Bobcat and Jaguar

The bandwidth is one read and one write per clock cycle to the level-1 data cache. The reads and writes are up to 64 bits wide on Bobcat, and 128 bits wide on Jaguar.

There is a penalty of typically 3 clock cycles for misaligned memory read or write that crosses a 16 bytes boundary.

17.12 Store forwarding stalls

Forwarding of data from a memory write to a subsequent read works well when the read is the same size or smaller than the write unless it is crossing a 16 bytes boundary. The combined latency of the read and subsequent write is at least 8 clock cycles for integer registers and 11 clock cycles for vector registers. The penalty for a failed store forwarding is 4-11 clock cycles.

Store forwarding is particularly fast on Jaguar for 32-bit and 64-bit general purpose registers.

Register size	Bobcat	Jaguar
8	8	8
16	8	8
32	8	3
64 g.p	8	3
64 mmx	11	7
128	12	7
256	n.a.	7

Table 17.2. Store forwarding times on Bobcat and Jaguar
Values are clock cycles, best case

17.13 Bottlenecks in Bobcat and Jaguar

The Bobcat and Jaguar have a well balanced pipeline design with no obvious bottlenecks other than what is obvious from the low power design.

Power saving is quite aggressive on the Bobcat so that the full speed is obtained only when the CPU load is heavy. The maximum throughput is two instructions per clock cycle, similar to the Intel Atom and VIA Nano processors.

The level-1 code cache has only 2-way associativity. This can slow down noncontiguous code. Integer division is slower than floating point division on Bobcat. Memory store forwarding is somewhat slow on Bobcat, but much higher on Jaguar. A store forwarding time of just 3 clock cycles for general purpose registers on the Jaguar is faster then the bigger processors from both AMD and Intel.

Integer vector instructions are generally faster on Jaguar than on Bobcat due to the larger size of internal registers (See manual 4: Instruction tables). All 128-bit instructions are split into two 64-bit µops on Bobcat which gives half the throughput. Similarly, all 256-bit instructions are split into two 128-bit µops on Jaguar.

The MASKMOVQ and MASKMOVDQU instructions are very slow, partially due to the fact that these instructions require uncached writes. The VMASKMOVPS instruction with a memory source operand takes more than 300 clock cycles on the Jaguar when the mask is zero, in which case the instruction should do nothing. This appears to be a design flaw.

The PALIGNR instruction is quite slow on the Bobcat. This instruction is used in many implementations of the memcpy (memory copying) function.

Prefetch instructions are very slow on the Jaguar, but not on Bobcat.

17.14 Literature:

Brad Burgess, et. al.: Bobcat: AMD's Low-Power x86 Processor. IEEE Micro, March/April 2011, pp. 16-25.

Anand Lal Shimpi: AMD’s Jaguar Architecture: The CPU Powering Xbox One, PlayStation 4, Kabini & Temash. May 2013. www.anandtech.com.

18 Comparison of microarchitectures

The state-of-the-art microprocessors that have been investigated here represent different microarchitecture kernels: the AMD, the Pentium 4 (NetBurst), the Pentium M, and the Intel Core 2 kernel. I will now discuss the advantages and disadvantages of each of these microarchitectures. I am not discussing differences in memory bandwidth because this depends partly on external hardware and on cache sizes. Each of the four microprocessor types is available in different variants with different cache sizes. The comparison of microarchitectures is therefore mostly relevant to CPU-intensive applications and less relevant to memory-intensive applications.

18.1 The AMD K8 and K10 kernel

The AMD microarchitecture is somewhat simpler than the microarchitecture of Intel processors. This has certain advantages because the processor spends less resources on complicated logistics. It also has drawbacks in terms of suboptimal out-of-order scheduling. The throughput of 3 macro-operations per clock cycle is obtained simply by having a 3-way pipeline all the way through. The ability to move macro-operations from one of the three pipelines to another is limited.

The execution units are less specialized than in Intel processors. All of the three integer execution units can handle almost any integer operation, including even the most obscure and seldom used instructions. Only the integer multiplication unit has been too expensive to make in three copies. This generality makes the logistics simple at the cost of bigger execution units.

The three floating point execution units are more specialized. Only the simplest operations are supported by more than one of these units. The distribution of macro-operations between the three floating point units is far from optimal. There is certainly room for improvement here.

The floating point execution units also have a problem with mixing macro-operations with different latencies. This problem is handled in a rather complicated and inefficient way by blocking the dispatch of a macro-operation with a short latency if it can be predicted that it would need the result bus simultaneously with a preceding macro-operation with a longer latency. A simpler and more efficient solution would be to keep the result of a short-latency macro-operation in the pipeline until the result bus is ready. This would solve the problem that a macro-operation with a short latency can be postponed for a long time if the pipeline is dominated by long-latency macro-operations, and it would get rid of the complicated logic for predicting when the result bus will be vacant.

All execution units have fairly short latencies and are fully pipelined so that they can receive a new macro-operation every clock cycle.

It is alleged that the nine execution units (three integer ALU's, three address generation units, and three floating point units) can execute nine micro-operations in the same clock cycle. Unfortunately, it is virtually impossible to verify this claim experimentally because the retirement stage is limited to three macro-operations per clock cycle. In other words, the AMD has ample execution resources which are never fully utilized. If the bottleneck in the retirement stage were widened then it would be possible to execute more macro-operations per clock cycle.

The AMD generates quite few macro-operations per instruction. Even read-modify and read-modify-write instructions generate only one macro-operation which is split into micro-operations only in the execute stage in the pipeline. This is similar to a fused micro-operation in the PM and Core2 design. The AMD K8 design has no execution units bigger than 64 bytes, so that 128-bit XMM instructions generate two macro-operations in the AMD K8. The subsequent AMD K10 processors have 128 bit units in the floating point pipeline to handle vector instructions in a single macro-operation.

The AMD design has no strict limitation to the number of input dependencies on a single macro-operation. Thus, instructions like `ADC EAX,EBX`, `CMOVBE EAX,EBX`, and `MOV [EAX+EBX],ECX` are implemented with a single macro-operation. The same instructions have to be split up into at least two micro-operations on Intel processors where a micro-operation can have no more than two input dependencies, including the condition flags.

The throughput for instruction fetching has been increased from 16 to 32 bytes per clock cycle on K10. There is still a lower throughput after taken jumps. Instruction fetching from the level-2 cache is particularly slow.

Instruction decoding seems to be quite efficient. It can decode at least three instructions per clock cycle.

Branch prediction is good for branches that always go the same way because branch prediction information is stored both in the level-1 and the level-2 cache. But the mechanism for predicting regular branch patterns has a lower prediction rate than on Intel processors.

The cache system can make two memory reads per clock cycle, while Intel processors prior to Sandy Bridge could do only one read per clock cycle.

18.2 The AMD Bulldozer, Piledriver and Steamroller kernel

The Bulldozer microarchitecture is a significant improvement over previous models.

The Bulldozer has two to eight compute units which have two CPU cores each. The code cache, instruction fetcher, branch prediction, and floating point units are shared between the two cores of each compute unit. The design allows a throughput of up to four instructions per clock cycle when only one core in a compute unit is used, or two instructions per clock cycle when both cores are used.

Most execution units are doubled so that most instructions can execute at a throughput of two instructions per clock cycle.

The latencies for floating point instructions and integer vector instructions are often somewhat longer than in Intel's Sandy Bridge.

The single-thread throughput for floating point operations is two 128-bit operations per clock cycle, which can be addition, multiplication or fused multiply-and-add.

The AMD design has no strict limitation to the number of input dependencies on a single µop. Thus, instructions like `ADC EAX,EBX`, `CMOVBE EAX,EBX`, and `MOV [EAX+EBX],ECX` are implemented with a single µop. The same instructions have to be split up into at least two µops on Intel processors where a µop can have no more than two input dependencies, including the condition flags, except for fused multiply-and-add instructions.

The instruction decoders can handle four instructions per clock cycle. The Bulldozer and Piledriver have one decoder per unit, shared between two threads, while the Steamroller has one decoder for each thread. The maximum throughput of four instructions per clock cycle is best obtained by mixing integer instructions and vector instructions.

The cache system is poor in Bulldozer, better in Piledriver, and still better in Steamroller.

18.3 The Pentium 4 kernel

The Intel Pentium 4 (NetBurst) kernel has been designed with a very one-sided focus on clock speed. Leading the clock frequency race surely has advantages in terms of marketing and prestige, but it also has considerable costs in terms of the design considerations that are needed in order to make the high clock frequency possible. A long pipeline is needed because the circuits can do less work per pipeline stage. Some pipeline stages in the P4 microarchitecture are used simply for transporting data from one part of the chip to another. Physical distances really matter at these high frequencies. The long pipeline means a high branch misprediction penalty.

Each of the execution units is kept as small as possible. Not only because physical distances matter but also because heat dissipation is a limiting factor. The smaller units are more specialized and can handle fewer different operations. This means that all but the simplest instructions require more than one µop, and some instructions require many µops.

The level-1 data cache is also quite small (8 or 16 kb), but access to the level-2 cache is fast.

The P4 doesn't have a code cache as other processors do, but a trace cache. The trace cache runs at half clock frequency, possibly because of its size. The trace cache doesn't store raw code but decoded µops. These µops are very similar to RISC instructions so that the kernel can use RISC technology. Decoding instructions into RISC-like µops before they are cached is a logical thing to do when decoding is a bottleneck. The P4 surely performs better in some cases where instruction decoding or predecoding is a bottleneck on other processors.

But the trace cache is not as advantageous as it first looks. For example, a `PUSH` or `POP` instruction takes only a single byte in a code cache, but 16 bytes in the P4E trace cache. This means that it can cache less code on the same chip area. Furthermore, the same code may appear in more than one trace in the trace cache. The consequence is less code in the cache when cache size is limited by physical factors. The 32-bit P4 has some data compression in the trace cache, but this goes against the idea of making decoding simple and efficient. Arranging code into traces eliminates the fetching delay when jumping to another part of the code, but there is still a delay when jumping to a different trace, and this delay is higher because the trace cache runs at half clock speed. Whether to store the same code in multiple traces or not is a tradeoff between eliminating jumps and putting more code into the trace cache.

Anyway, I seriously doubt that the trace cache makes the design simpler. I don't know how it keeps track of the traces, mapping physical addresses to multiple trace cache addresses, deciding whether to jump to another trace or extend the existing trace on a branch instruction, and deciding when to rearrange traces. This looks like a logistical nightmare to me.

The instruction decoder can only handle one instruction per clock cycle, while other designs can decode three or more instructions per clock cycle. Obviously, the decoder has got a low priority because it is used only when building new traces from the level-2 cache.

The P4 has two execution ports for ALU and other calculations (port 0 and 1) and two additional ports for memory operations and address calculation (port 2 and 3). Each execution port has one or more execution units. It is not possible to dispatch two µops simultaneously to two different execution units if they are using the same execution port. These ports are therefore a bottleneck. The execution units are not optimally distributed between port 0 and 1. All floating point and SIMD (vector) operations, except simple moves, go through port 1. This makes port 1 a serious bottleneck in floating point and SIMD code.

Two small integer ALU's run at double clock speed. The 32-bit P4 can do staggered additions with a virtual latency of only a half clock cycle. The latency for the condition flags is longer. The 64-bit P4E still uses double-speed ALU's, but with a latency of a full clock cycle. The throughput is four µops per clock for those instructions that can be handled by both ALU's. The double speed ALU's are specialized to handle only the most common operations such as move and addition on general purpose registers.

The other execution units have longer latencies, sometimes much longer. The most ridiculous example is a 7 clock latency for a floating point register-to-register move on P4E. There is also an additional latency of one clock cycle when a result from one execution unit is needed as input in another execution unit.

A 128-bit XMM instruction is handled by a single µop, but the execution units can handle only 64 bits at a time so the throughput is only 64 bits per clock cycle. Only memory reads have a throughput of 128 bits per clock cycle. This makes the P4/P4E an efficient microprocessor for memory-intensive applications that can use 128-bit operands.

The branch prediction algorithm is reasonably good, but the misprediction penalty is unusually high for two reasons. The first reason is obviously that the pipeline is long (20 or more stages). The second reason is that bogus µops in a mispredicted branch are not discarded before they retire. A misprediction typically involves 45 µops. If these µops are divisions or other time-consuming operations then the misprediction can be extremely costly. Other microprocessors can discard µops as soon as the misprediction is detected so that they don't use execution resources unnecessarily.

The same problem applies to bogus reads of a store-forwarded memory operand that is not ready. The P4 will keep re-playing the read, as well as subsequent µops that depend on it, until the memory operand is ready. This can waste a lot of execution resources in a read-after-write memory dependence. This typically occurs when parameters are transferred on the stack to a subroutine. There is also an excessive replaying of µops after cache misses and other events. The amount of resources that are wasted on executing bogus µops is so high that it is a serious performance problem.

The retirement is limited to slightly less than 3 µops per clock cycle.

18.4 The Pentium M kernel

The PM is a modification of the old Pentium Pro kernel with the main focus on saving power. A lot of effort has been put into turning off parts of the execution units and buses when they are not used. The lowered power consumption has a beneficial side effect. A low power consumption means that the clock frequency can be increased without overheating the chip.

Instruction decoding is limited by the 4-1-1 rule (page 88). Software must be tuned specifically to the 4-1-1 rule in order to optimize instruction decoding. Unfortunately, the 4-1-1 pattern is broken by instruction fetch boundaries, which are difficult to predict for the programmer or compiler maker. This problem reduces the instruction fetch rate to less than 16 bytes per clock cycle and the decode rate to less than three instructions per clock cycle (page 88). Instruction fetch and decoding is definitely a weak link in the PM design.

The execution units are clustered around five execution ports in a design very similar to the P4. Port 0 and 1 are for ALU and other calculations, while port 2, 3 and 4 are for memory operations and address calculation. The execution units are more evenly distributed between port 0 and 1, and many µops can go to any of these two ports. This makes it easier to keep both ports busy than in the P4 design.

SIMD integer instructions are quite efficient with an ALU on each of the two execution ports and a latency of only one clock cycle. Floating point latencies are also quite low.

The PM generates fewer µops per instruction than the P4. While both designs have a throughput of 3 µops per clock, the PM gets more instructions executed per clock cycle because of the lower number of µops. The low number of µops is partially due to µop fusion (page 90) and a dedicated adder for the stack pointer (page 92). Unfortunately, the µop fusion mechanism doesn't work for XMM registers. This makes the PM more efficient for MMX registers and floating point registers than for XMM registers.

The PM has a limitation of three register reads per clock cycle from the permanent register file. This can very well be a bottleneck.

The PM has an advanced branch prediction mechanism. The loop counter is something that we have been wishing for several years. But this doesn't make up for the very small branch target buffer of probably only 128 entries. The improved ability to predict indirect jumps is probably what has made it necessary to reduce the size of the BTB.

The PM doesn't support the 64-bit instruction set.

The throughput of the retirement stage is exactly as in P4. While both designs are limited to 3 µops per clock cycle, the PM has fewer µops per instruction and shorter latencies in the execution units. This makes the PM so efficient that it may run CPU-intensive code faster than the P4 even though the latter has a 50% higher clock frequency.

18.5 Intel Core 2 and Nehalem microarchitecture

This design takes the successful philosophy of the PM even further with a high focus on saving power. The low power consumption makes it possible to increase the clock frequency.

The pipeline and execution units are extended to allow a throughput of four µops per clock cycle. The throughput is further increased by issuing fewer µops per instruction and by extending the data busses and execution units to 128 bits. Cache and memory bandwidth have also been improved.

The Core2 and Nehalem have so many execution units and execution ports that the execution stage will rarely be a bottleneck. Most execution units and internal data busses have been extended to 128 bits unlike previous x86 processors from Intel and AMD, which have only 64-bit execution units.

A few weak spots in the design remain, however. I want to point out three areas in the design that do not match the performance of the rest of the pipeline, and these weak spots are likely to be bottlenecks:

1. Instruction predecoding. The mechanism of instruction fetch and predecoding has been improved, but the bandwidth is still limited to 16 bytes of code per clock cycle. This is a very likely bottleneck in CPU-intensive code.

2. Register read ports. The design can read no more than two or three registers from the permanent register file per clock cycle. This is likely to be insufficient in many cases.

3. Branch history pattern table. The branch history pattern table is so small that it may compromise the otherwise quite advanced branch prediction mechanism.

4. Read ports. The Core2 can read one memory operand per clock cycle, where AMD processors can read two. The read port cannot always match the high throughput of the execution units.

18.6 Intel Sandy Bridge and later microarchitectures

This new design is a significant improvement. Many of the bottlenecks of previous designs have been dealt with in the Sandy Bridge.

Instruction fetch and predecoding has been a serious bottleneck in Intel designs for many years. In the NetBurst architecture they tried to fix this problem by caching decoded µops, without much success. In the Sandy Bridge design, they are caching instructions both before and after decoding. The limited size of the µop cache is therefore less problematic, and the µop cache appears to be very efficient.

The limited number of register read ports has been a serious, and often neglected, bottleneck since the old Pentium Pro. This bottleneck has now finally been removed in the Sandy Bridge.

Previous Intel processors have only one memory read port where AMD processors have two. This was a bottleneck in many math applications. The Sandy Bridge has two read ports, whereby this bottleneck is removed.

The branch prediction has been improved by having bigger buffers and a shorter misprediction penalty, but it has no loop predictor, and mispredictions are still quite common.

The AVX instruction set is an important improvement. The throughput of floating point addition and multiplication is doubled when the new 256-bit YMM registers are used. The new non-destructive three-operand instructions are quite convenient for reducing register pressure and avoiding register move instructions. There is, however, a serious performance penalty for mixing vector instructions with and without the VEX prefix. This penalty is easily avoided if the programming guidelines are followed, but it is a common programming error to inadvertently mix VEX and non-VEX instructions, and such errors are difficult to detect.

Whenever the narrowest bottleneck is removed from a system, the next less narrow bottleneck will become the limiting factor. The new bottlenecks that require attention in the Sandy Bridge are the following:

1. The µop cache. This cache can ideally hold up to 1536 µops. The effective utilization will be much less in most cases. The programmer should pay attention to make sure the most critical inner loops fit into the µop cache.

2. Instruction fetch and decoding. The fetch/decode rate has not been improved over previous processors and is still a potential bottleneck for code that doesn't fit into the µop cache.

3. Data cache bank conflicts. The increased memory read bandwidth means that the frequency of cache conflicts will increase. Cache bank conflicts are almost unavoidable in programs that utilize the memory ports to their maximum capacity. This problem has been removed in the Haswell processor.

4. Branch prediction. While the branch history buffer and branch target buffers are probably bigger than in previous designs, mispredictions are still quite common.

5. Sharing of resources between threads. Many of the critical resources are shared between the two threads of a core when hyperthreading is on. It may be wise to turn off hyperthreading when multiple threads depend on the same execution resources.

19 Comparison of low power microarchitectures

Intel Atom, VIA Nano, and AMD Bobcat are all small processors designed for low power consumption. These processors have lower performance than the more powerful processors but are sufficient for ordinary office applications, embedded applications, and even low-traffic servers. The low price and low power consumption makes these processors a good choice for less demanding applications.

19.1 Intel Atom microarchitecture

The Intel Atom is a small low-power processor with lower performance than the Core brand. It has no out-of-order capabilities and it can do two instructions simultaneously only under certain rather restrictive conditions, just like the first Pentium processor fifteen years ago. The internal design is very different from the Pentium, however.

The main drawbacks of the Atom design are the in-order execution and a relatively low instruction fetch rate.

The hyperthreading support is of limited use because the resources that are shared between two threads are quite small, even for single-threaded use. Some versions have two cores capable of running two threads each.

19.2 VIA Nano microarchitecture

The VIA Nano is a small low-power processor targeted at the same type of applications as the Intel Atom. The Nano has a quite advanced out-of-order pipeline and powerful execution units. In some respects, the performance of the execution units is similar to that of the much bigger desktop processors from Intel and AMD. The floating point performance is particularly good.

The Nano has longer latencies than the desktop processors for memory access and for jumps, but very low latencies for integer and floating point calculations. The overall performance is significantly better than the Atom for single-threaded applications. The versions with two or more independent cores give good performance in multitasking environments.

19.3 AMD Bobcat microarchitecture

The AMD Bobcat has an efficient out-of-order pipeline with good performance and no obvious bottlenecks. The execution units are only 64 bits wide, which gives a lower throughput for vector instructions. The Bobcat has two independent cores, which gives it a good performance in multitasking environments.

19.4 Conclusion

The AMD, P4 and Intel Core designs can all execute instructions out of order with a maximum throughput of 3 µops per clock cycle. The Core2, Nehalem and Sandy bridge can do 4 µops per clock cycle. AMD Bulldozer and later designs can execute four µops per clock cycle when there is a good mixing of integer and vector instructions.

The NetBurst (Pentium 4) design has a higher clock frequency which means more µops per second. But each instruction generates more µops on NetBurst than on other processors, which means fewer instructions per clock cycle. A further disadvantage of the high clock frequency is a long pipeline and thus a high branch misprediction penalty. The NetBurst microarchitecture has so many inherent performance problems that it has been discontinued.

The PM, Core and the AMD designs all use the opposite strategy of a lower clock frequency but fewer µops per instruction. This strategy appears to give the best performance for CPU-intensive applications. The NetBurst has longer latencies than other processors for many instructions. This makes it inferior for code with long dependency chains.

All these designs have a RISC-like execution core that works on simple µops rather than on complex instructions. The NetBurst and Sandy Bridge designs have pushed the RISC philosophy even further by caching µops rather than instructions. The NetBurst design is not convincing since it reduces the amount of information per cache area and the management of the trace cache has become no less time consuming than a CISC decoder. The Sandy Bridge uses the compromise of caching instructions both before and after decoding. The result appears to be quite successful.

AMD uses the different strategy of marking instruction boundaries in the code cache, whereby the bottleneck of instruction length decoding is removed. Intel did the same in the Pentium MMX, and I don't understand why they have left this strategy. Instruction length decoding continues to be a serious bottleneck in Intel processors. The problem has been reduced by caching decoded instructions, but the capacity of the NetBurst trace cache and the Sandy Bridge µop cache is still limited because µops take more cache space than CISC instructions.

For many years, the RISC philosophy has been considered the best way to high performance. The Intel Core microarchitecture as well as the AMD design indicates a new trend away from RISC and back to the CISC principle. The RISC-like design of the P4 with its very long pipeline and long execution latencies was not convincing, and the trace cache appears to have been inefficient. The advantages of going back to CISC design are threefold:

1. The compact CISC code gives better utilization of the limited code cache area.

2. Fewer µops per instruction gives higher bandwidth in the pipeline.

3. Fewer µops per instruction gives lower power consumption.

The main disadvantage of a CISC design is that instruction decoding becomes a bottleneck. The AMD processor has a higher decoding bandwidth than the Intel design because of the technique of storing instruction boundaries in the code cache. The Intel design is still limited to a decoding rate of 16 bytes of code per clock cycle, which is insufficient in many cases.

The initial RISC philosophy required that all instructions should have the same latency in order to provide a smooth pipelining. However, this principle soon proved untenable because multiplication, division and floating point addition take longer time than integer addition. This problem is partially solved in the Intel microarchitecture by sending micro-operations with different latencies to different execution ports (see table 8.1 page 111).

The increased focus on power-saving features in the Intel Core and later designs makes it possible to use a relatively high clock frequency despite a design that does more work per pipeline stage than the NetBurst.

The AMD also uses a CISC design with few µops per clock cycle. Early versions were kept as simple as possible in order to reduce the necessary amount of logistics overhead. This, however, reduced the out-of-order capabilities and the optimal utilization of execution units.

The Intel Core and AMD K10 processors have full 128-bit execution units. The earlier processors use a 64-bit unit twice when computing a 128-bit result. It is therefore not possible to take full advantage of the 128-bit XMM instructions in these processors. The Sandy Bridge has full 256-bit execution units, except for memory read and write operations. It is likely that we will see processors from AMD or VIA with AVX support that use 128-bit units twice to compute a 256-bit vector result. The same may happen with future extensions of the vector size to 512 bits or more.

The throughput of 3 µops per clock cycle in both AMD, P4 and PM is increased to 4 µops in the Core2 and later designs, but I do not expect this number to increase much further in the future because the advantage of a higher throughput cannot be fully utilized unless the code has a high degree of inherent parallelism.

Instead, the trend goes towards multiple execution cores. Unfortunately, we need multi-threaded programs to take full advantage of multiple cores in single-user systems. Calculation tasks that cannot be divided into multiple threads cannot take much advantage of multiple cores or multiple processors.

Contemporary Intel and AMD processors have multiple cores. A half-way solution was introduced in the NetBurst and again in the Nehalem and Sandy Bridge with the so-called hyperthreading technology. The hyperthreading processor has two logical processors sharing the same execution core. The advantage of this is limited if the two threads compete for the same resources, but hyperthreading can be quite advantageous if the performance is limited by something else, such as memory access. The AMD processors do something similar, but with fewer resources shared between threads.

The extension of the 32 bit architecture to 64 bits has been a logical and necessary thing to do. Intel came before AMD with their advanced Itanium RISC architecture, but it lacks the backwards compatibility that the market demands. The AMD 64-bit architecture, which has doubled the number of registers and still retained backwards compatibility, has turned out to be a commercial success which Intel had to copy. All x86 processors produced today support the x64 instruction set.

Table 19.1 below compares execution latencies for various operations on the three designs. A table comparing cache sizes etc. is provided in manual 4: "Instruction tables".

Execution latencies, typical	AMD	P4E	Core2 45 nm	Sandy Bridge
integer addition	1	1	1	1
integer multiplication	3	10	3	3
integer division, 32 bits	40	79	23	28
packed integer move	2	7	1	1
packed integer addition	2	2	1	1
packed integer multiplication	3	8	3	3
floating point move	2	7	1-3	1
floating point addition	4	6	3	3
floating point division, double	20	45	21	22
floating point vector addition	4	5	3	3
floating point vector multiplication	4	7	5	5

Table 19.1. Comparison of instruction latencies

20 Future trends

The high clock frequency in the NetBurst design turned out to be too costly in terms of energy consumption and chip heating. This led to a change of focus away from the gigahertz race. Since then, there has been an increasing focus on energy efficiency, and this factor will become no less important in the future since small battery-operated computers are becoming very popular.

The speed and computing power of execution units is growing faster than the speed of memory access. We can therefore expect an intense focus on improving caching and memory bandwidth in the future. Three-level caching and wider data paths will be common.

Branch misprediction is very costly in present microprocessors because of the long pipelines. We will most likely see more advanced multilevel branch prediction algorithms, bigger branch target buffers and history tables, decoding and speculative execution of multiple branches simultaneously, as well as predicated instructions in future processors in order to reduce the cost of mispredictions. We have not yet seen radically shorter pipelines as a means of reducing misprediction penalties.

Instruction length decoding is a bottleneck in many CISC computers. A transition to a RISC instruction set is prevented by the requirement for backwards compatibility. Dual instruction set computers is a possible solution, but quite costly. Caching decoded µops was not a very successful solution in the Intel NetBurst architecture, but more successful in Intel's Sandy Bridge and its successors. A disadvantage of the µop cache is that it takes much more die space per instruction than a traditional code cache. More focus on improving the instruction length decoders is possibly a more efficient solution. The AVX and later instructions are using the new VEX coding scheme which includes information on the instruction length. This opens the possibility for a compromise where the decoders have a higher throughput for instructions with VEX encoding than for the most complicated non-VEX encodings.

An alternative to caching µops is to mark instruction boundaries in the instruction cache. This relieves the critical bottleneck of instruction length decoding. Most AMD processors use this method, and it was also used in Intel's Pentium MMX processor.

There is a remarkable convergence between the Intel and AMD microarchitectures thanks to a patent sharing agreement between the two companies. Intel's stack engine and the mechanism for predicting indirect branches have been copied by AMD. We can also expect Intel to some day match AMD's 32-bytes instruction fetch rate.

There is unfortunately not always convergence on the instruction set extensions. Intel's SSE4.1 and SSE4.2 instruction sets are very different from AMD's SSE4.A and XOP (formerly known as SSE5), and the intersection between these two sets is quite small. Intel has never copied AMD's 3DNow instruction set which is now obsolete, but they have copied the successful x64 extension from AMD and a few other instructions. AMD has traditionally copied all Intel instructions, but sometimes with a lag of several years. Fortunately, AMD has revised their proposed SSE5 instruction set to make it compatible with the AVX coding scheme (as I have previously argued that it would be wise of them to do). The AMD Bulldozer processors now support Intel's AVX instruction set, including the 256-bit YMM vector registers. The incompatible fused multiply-and-add instructions is a pathetic story as discussed on my blog. Fortunately, Intel's Haswell and AMD's Piledriver now both support the FMA3 instructions.

The transition to three- and four-operand instructions has been easier for AMD than for Intel because, for many generations, the Intel microarchitecture has not allowed a μop to have more than two input dependencies, while AMD has never had such a limitation. The limitation on input dependencies is partially broken in the Haswell with the fused multiply-and-add (FMA) instructions.

The amount of parallelism in CPU pipelines will not grow much beyond the four μops per clock cycle, because dependency chains in the software is likely to prevent further parallelism. Instead, we are seeing an increasing number of cores. Even small low-power processors now have multiple cores. This puts a high demand on software developers to make multithreaded applications. Both Intel and AMD are making hybrid solutions where some or all of the execution units are shared between two processor cores (hyperthreading in Intel terminology).

Lower instruction latencies is one way to higher performance that we have yet seen only in low power processors. AMD's Bulldozer was the first processor to translate register-to-register moves to register renaming with zero latency, but Intel soon followed. Another way to reduce latencies is instruction fusion. We will most likely see an automatic fusion of multiply and add instructions as well as other common instruction combinations. The store-to-load forwarding mechanism may also be improved to reduce the latency.

The extension of the 128-bit XMM registers to 256 bits (YMM) was realized in the Intel Sandy Bridge processor in January 2011, and AMDs Bulldozer later in the same year. The next extension to 512 bits has been announced, and a later extension to 1024 bits is also expected. Apparently, there are no plans of vector registers bigger than 1024 bits. The `XSAVE` and `XRESTOR` instructions are prepared for these future extensions.

There are two versions of the instruction set for 512-bit vectors. The Intel MIC/Xeon Phi Coprocessor uses a new 4-bytes prefix called MVEX to increase the vector size, double the number of vector registers to 32 and allow various extra attributes to be added to each instruction, such as masked operations, type conversion, broadcast, permutation, cache eviction hint, rounding mode, and suppression of exceptions. The forthcoming AVX-512 instruction set uses an almost identical prefix called EVEX. The EVEX prefix does not allow type conversion and permutation, but includes bits for specifying vector size instead. The MVEX and EVEX instruction sets not compatible with each other. They differ by a single bit in the prefix, even for otherwise identical instructions. The MVEX and EVEX instruction sets are both backwards compatible with previous instruction sets, though. I expect the AVX-512 instruction set with the EVEX prefix to be the standard for future processors, while MVEX will probably not be continued.

Strangely, the MIC/Xeon Phi processors also support the old x87 floating point instructions, which appear to be obsolete. It does not look like we are getting completely rid of the x87 instructions in a foreseeable future even though they are expensive to implement in hardware.

The x86 instruction set now has far more than a thousand logically different instructions, including specialized instructions for text processing, graphics, cryptography, CRC check, and complex numbers. The instruction set is likely to be increased with every new processor generation, at least for marketing reasons. We may see more application-specific instructions, such as Galois field algebra vector instructions for encryption purposes.

However, the ever-increasing number of instructions may not be optimal from a technical point of view because it increases the die area of the execution units whereby the clock frequency is limited. A more viable solution might be user-definable instructions. We are already seeing FPGA chips that combine a dedicated microprocessor core with programmable logic. A similar technology may be implemented in PC processors as well. Such a processor will have logical arrays similar to FPGAs that can be programmed in a hardware definition language to implement application-specific microprocessor instructions. Each processor core will have a cache for the hardware definition code in addition to the code cache and the data cache. The cost of task switching will increase, of course, but it will be easier to reduce the number of task switches when the number of cores is increased.

The drive towards ever-increasing CPU speed according to Moores law has a downside in terms of power consumption, size and price. We are now seeing an alternative trend of smaller low-power processors, represented by Intel's Atom, VIA's Nano and AMD's Bobcat and Jaguar. These lightweight processors are as fast as the desktop processors were a few years ago, yet smaller, cheaper and with much lower power consumption. We will see these small processors replace the big and power-hungry processors in a lot of applications where we have previously paid for more CPU power than we need.

21 Literature

The present manual is part three of a series of five manuals available from www.agner.org/optimize as mentioned in the introduction on page 5.

Other relevant literature on microarchitecture:

- Intel 64 and IA-32 Architectures Optimization Reference Manual.
- Software Optimization Guide for AMD64 Processors.
- J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach, 3'rd ed. 2002.
- www.xbitlabs.com
- www.arstechnica.com
- www.realworldtech.com
- www.aceshardware.com
- www.digit-life.com

22 Copyright notice