What’s Up With Willamette? (Part 1)

his is the first installment in a two-part article about Willamette, the next generation x86 processor design from Intel. It includes a description of the development roots of Willamette and the basics of how its organization and operation differs from earlier P6 generation processors. In a future article I will examine the new technology and features of Willamette in more detail and speculate on its implementation, operational characteristics, and performance.

Intel Drops Its Sledgehammer

Willamette is the code name for a new high-end x86 microprocessor core Intel will introduce towards the end of the year. Willamette is quite an interesting processor for many different reasons, both technical and historical. It is the first entirely new x86 processor design Intel has come up with since the powerful and versatile “P6” processor core was introduced in the Pentium Pro in 1995. Willamette is also the processor that Intel originally thought it wouldn’t need nearly as badly as it obviously does today. To fully understand this tangled tale we have to go back to seven years ago when computer users frustrated with their 486DX2 systems were eagerly awaiting a new Intel processor with a funny name instead of a number.

In early 1993 Intel’s Santa Clara processor design team had just finished off the P5 project (Pentium) and started work on the P7. Intel had initiated a new strategy to operate two separate x86 processor development teams in parallel, in an overlapped fashion. Under this strategy, when the first team finishes the generation N processor design it starts work on the generation N+2 processor, while the second team is in the middle of the generation N+1 processor project. The hope was to cut the four-year intervals between new processor cores in half. When work on the P7 started up in Santa Clara, Intel’s P6 team in Hillsboro Oregon was about 18 months and a lot of hard work away from delivering the Pentium Pro. The P7 was a powerful 64-bit x86 compatible successor to the P6 envisioned to have around 20 million transistors or nearly four times as many as the Pentium Pro. In some ways Intel’s original P7 project conceptually resembles AMD’s K8 “sledgehammer”, the 64-bit successor to the K7 Athlon.

The P7 progressed only far along enough for Intel’s engineers to realize that extending x86 to 64 bits, and staying competitive with RISC processors, would be challenging to say the least. Around this time Intel entered into an alliance with Hewlett Packard to develop a high performance 64-bit processor incorporating variable length VLIW technology from HP’s “Wide word” extension to its Precision Architecture RISC architecture. In 1994 the Santa Clara team dropped all work on the 64 bit x86 processor design called P7 and started on the first implementation of the new IA-64 architecture arising from the Intel-HP alliance, a processor later known as Merced. The Merced project adopted the P7 designation, and its troubled offspring is targeted to reach the market later this year under the name “Itanium”.

By 1996 the P6-based Pentium Pro was shipping and plans were made to create high volume consumer versions of the P6 core. Also around this time Intel was touting the new P7 (Merced) with its dual x86 and IA-64 capability as the next generation Intel processor. Unfortunately for Intel, the development of Merced and its successors proved more difficult and time consuming than originally thought and the IA-64 deployment schedule started to slip, first by months, and then by years. Even as late as 1996 Intel publicly proclaimed that Merced would ship in volume in 1998.

Waiting for no project, Moore's Law inexorably pushed semiconductor technology forward and dragged along P6 based designs to ever higher levels of performance. Since Merced wasn't getting any faster as its schedule slipped, its performance on x86 code was starting to look weaker and weaker compared to the P6 designs it was supposed to replace. This relative slippage is reflected in a succession of public pronouncements about Merced's x86 performance from being the fastest x86 processor to being equivalent to a mid range x86 MPU. At the same time competition in the x86 market was starting to heat up with the introduction of the AMD K6, and the announcement of the K7 development project and its licensed use of Alpha EV6 bus technology.

Hillsboro's Revenge and Intel's Deliverance

Luckily for Intel, Andy Grove hadn't put all the eggs in the IA-64 basket. Sometime after Intel's Oregon based design team delivered the P6 it started work on a new generation x86 core. The project was changed in scope, from the P7's original mandate of extending x86 to 64-bits, to simply developing a new generation 32-bit x86 core. The project, originally named P67, was officially confirmed and given the public code name Willamette on Intel's roadmap released in October 1998. It is not clear whether the Willamette was intended to act as Merced's little brother in the low-end x86 market or was undertaken simply as an insurance policy.

What is clear is that Intel desperately needs Willamette. AMD's impressive K7 Athlon processor design is clearly more advanced than the aging P6 core. The K7 offers both higher architectural performance per clock cycle and can be clocked at significantly higher frequencies than the P6-based design manufactured in a similar process. AMD is currently poised to release a new generation of 0.18 um based K7 processors at several different price/performance points to simultaneously attack Intel at the high, middle, and low end of the x86 market. This unparalleled competitive pressure comes at a dangerous time given Intel's well known troubles ramping up the clock rate, yields, and production volumes of its P6-based 0.18 um Coppermine Pentium III, and difficulty in coming up with a widely acceptable successor to the venerable 440BX chipset.

For the last three or four years Intel's Hillsboro design team has labored nearly invisibly in the shadow of the over-hyped Merced processor. No doubt there is a healthy competition between Intel's Hillsboro and Santa Clara engineers, and it couldn't have been much fun to work hard on a project publicly treated as little more than a footnote to the IA-64 effort. But now the tables have turned. Merced gets so little respect it is treated almost as an industry joke, while everyone from PC

buyers to Intel share holders impatiently wait for every hint of how Willamette stacks up against AMD's powerhouse. Now, Intel management publicly downplays Merced ("wait until McKinley!"), while it mounts a full-blown, category five FUD (fear uncertainty and doubt) campaign against AMD and its K7 Athlon. Intel's widely reported demonstrations of prototype Willamette devices with clock rates up to 1.5GHz is the opening salvo of a tooth and nail struggle for the hearts and minds of customers, trade press and stock analysts. Indeed, Intel's fortunes over the next two or three years clearly ride heavily on Willamette's shoulders.

Starting Point: The Design Willamette Would Replace

To understand the significance of the Willamette one must examine the design that it is intended to replace. Five years ago the P6 core was first delivered in the form of the Pentium Pro high-end processor for technical workstations and servers. The P6 was a remarkable achievement, an out-of-order execution superscalar x86 processor whose integer performance briefly eclipsed the fastest RISC processors. Intel went on to sell about 7 million Pentium Pro's, a huge figure that would represent the jack pot for most high end processor families but is still quite modest by mainstream x86 standards.

But Intel had great plans for this versatile core. The P6 core rapidly proliferated into every market Intel targeted from the low end Celeron, the mainstream Pentium II and III, and the high end Xeons. The P6 core has been implemented in five different processes and had two major instruction set extensions as shown in Table 1.

Year	Process	ISA Extensions	Products
1995	0.5 um BiCMOS	–	"P6" Pentium Pro
1995	0.35 um BiCMOS	–	"P6" Pentium Pro
1997	0.35/0.28 um CMOS	MMX	"Klamath" Pentium II
1998	0.25 um CMOS	MMX	"Deschutes" Pentium II / Xeon "Covington" Celeron "Mendocino" Celeron
1999	0.25 um CMOS	MMX, KNI/SSE	"Katmai" Pentium III / Xeon
1999	0.18 um CMOS	MMX, KNI/SSE	"Coppermine" Pentium III / Xeon

Table 1 History of Intel's P6 Core

For the last several years the P6 core has been at the heart of just about every processor Intel has sold and is responsible for over $20 billion in annual sales and billions in profit for the chip giant. The basic design of the P6 core is shown in Figure 1. It is shown in its latest incarnation, the Pentium III, with 16 KB L1 caches and MMX and KNI/SSE functional unit extensions (the original Pentium Pro P6 design had 8 KB L1 caches)

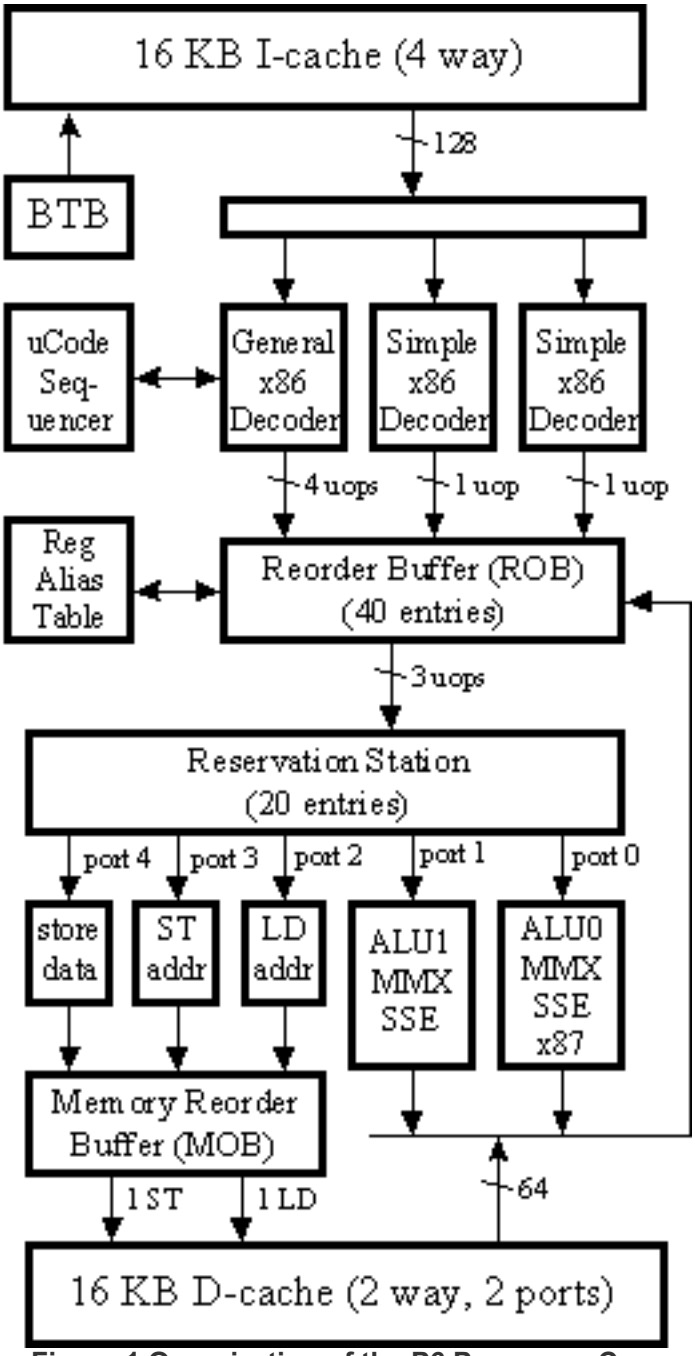

Figure 1 Organization of the P6 Processor Core

The primary characteristic of the P6 core is that it can decode up to one complex and two simple x86 instructions per clock cycle. The P6 instruction decoders effectively translate x86 instructions into one or more simpler operations encoded into control information parcels known as micro-ops or uops. A uop is a fixed length, 118 bit long control word that encodes an operation, two operand sources, and a result destination. The source and destination fields were wide enough to include a complete 32-bit operand value such as an immediate value or destination address offset. Uops are fed into a reorder buffer, which is a functional unit that tracks the overlapped out-of-order (OOO)

execution of up to 40 uops at once. Although the three decoders could theoretically generate up to 6 uops per clock cycle, the reorder buffer can only accept, process, and output 3 uops per cycle to the reservation station.

The OOO execution engine can issue up to three uops per clock cycle, each to one of five execution resources – two execution units, two address generation units, and a store data unit. When the MMX, and later on the KNI/SSE extensions, were added to the x86 instruction set, this basically expanded the two execution units. The port 0 execution unit originally supported just integer and basic x87 type FP instructions. It was extended to support MMX instructions and also the SIMD FP multiplication functions of KNI/SSE. The port 1 execution unit supported just integer instructions in the Pentium Pro but was later extended to support MMX, and ultimately SIMD FP addition functions of KNI/SSE.

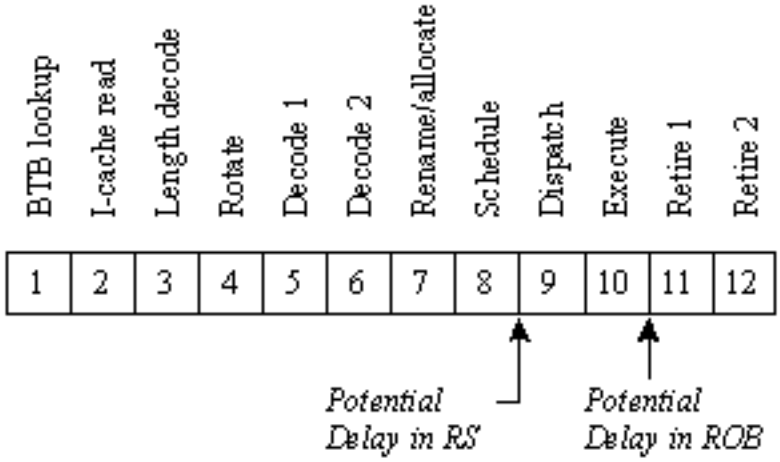

Figure 2 Basic P6 Execution Pipeline

The basic pipeline organization of the P6 design is shown in Figure 2. Simple instructions take a minimum of 12 clock cycles to flow through the pipeline. That is the minimum because there is the possibility of delays in the reservation station between pipe stages 8 and 9, or in the reorder buffer between pipe stages 10 and 11. Each x86 instruction executed on a P6 generates between 1.5 and 2.0 uops, with the preponderance of code closer towards 1.5 uops per instruction when running typical PC type applications. When running the SPEC95 benchmark suite, the uop per instruction figure ranges from 1.2 to 1.7 with an average around 1.35. This may indicate that the Intel reference compiler, typically used for benchmarking, uses more of a RISC-like code generation strategy that favors register to register instructions than the compilers typically used by commercial application developers. The average uops per instruction and average clock cycles per instruction (CPI) for several applications and benchmarks are shown in Table 2.

Benchmark	Uops/Instruction	Cycles per Instruction (CPI)
MaxEDA	1.5	1.7
Texim	1.7	1.7

PowerPoint	1.4	1.2
Excel	2.5	1.7
Word	1.9	2.4
SPECint95 (maximum)	1.6	1.6
SPECint95 (median)	1.4	1.1
SPECfp95 (maximum)	1.3	2.0

Table 2 Uops per Instruction and Cycles per Instruction (CPI) for Pentium Pro (from “Performance Characterization of the Pentium Pro Processor” by D. Bhandarkar and J. Ding, IEEE HPCA Symposium 1997)

Although uops can go through the P6 pipeline in as few as 9 to 10 cycles, about 90% of them take 12 to 14 clock cycles from decode to retirement. The uop lifetime distribution is actually bimodal with a minor lobe way out at 50 to 120+ cycles (depending on the processor clock to bus clock frequency ratio and the latency of main memory) for memory and memory dependent uops that miss both level of caches. There is also a bimodal distribution of uop delay passing through the reservation station. About half of the uops pass through in the minimum time while the other half are delayed 3 extra cycles waiting for the result of a preceding ALU operation. According to Bob Colwell, the P6 Architecture Manager, when all of these second order effects are taken into account the so-called 12 pipe stage P6 has an effective length of 15 to 20 cycles for integer instructions and 30 or so for floating point instructions.

The Basics of Willamette

The organization of the Willamette processor is shown in Figure 3. Compared to the P6 design, the primary defining features of Willamette are a trace cache, dual super-pipelined ALUs operating at twice the processor frequency, deeper pipelining, improved branch prediction, and a much higher bandwidth system interface.

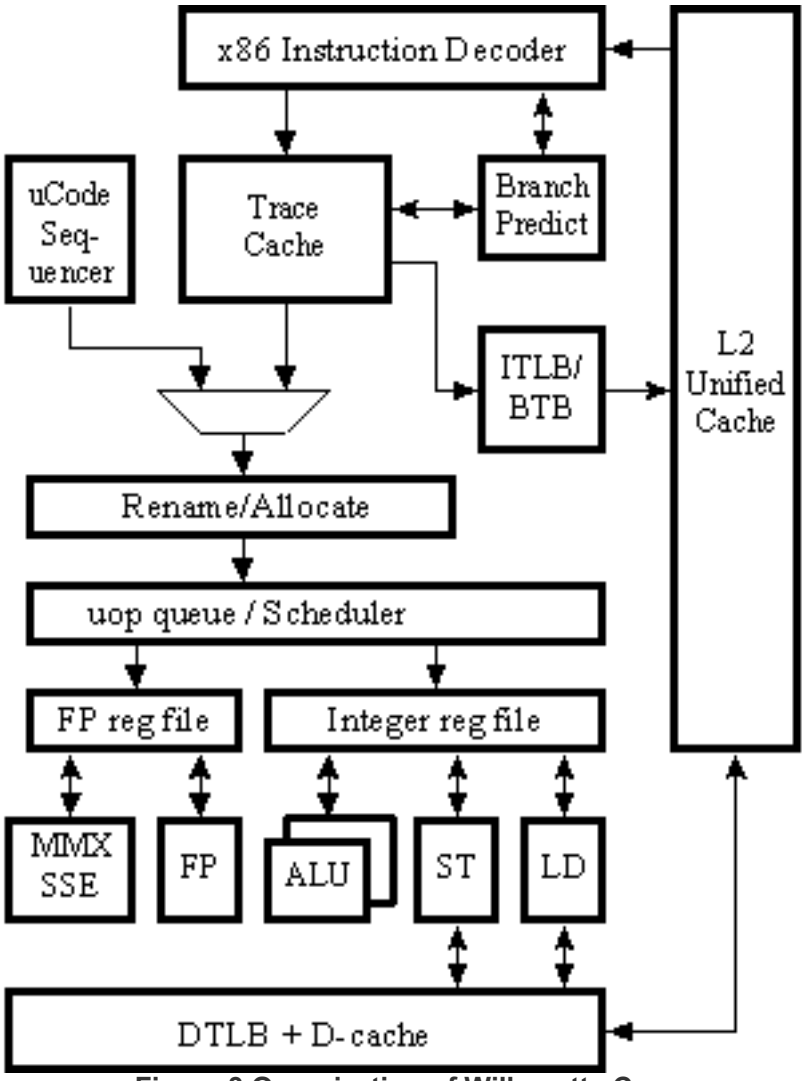

Figure 3 Organization of Willamette Core

The trace cache is an innovative first level instruction cache (I-cache) that stores sequences of micro-ops organized in dynamic program execution order, rather than the conventional I-cache in the P6, which stores x86 code in static program order organized by memory location. The effect of this radical change is to basically de-couple branch prediction and translation of x86 instructions to uops from the repetitive (loop based) execution of program code out of the trace cache. It also effectively performs loop unrolling (a code transform sometimes performed by sophisticated compilers for speeding up code) in hardware but without the associated code size expansion.

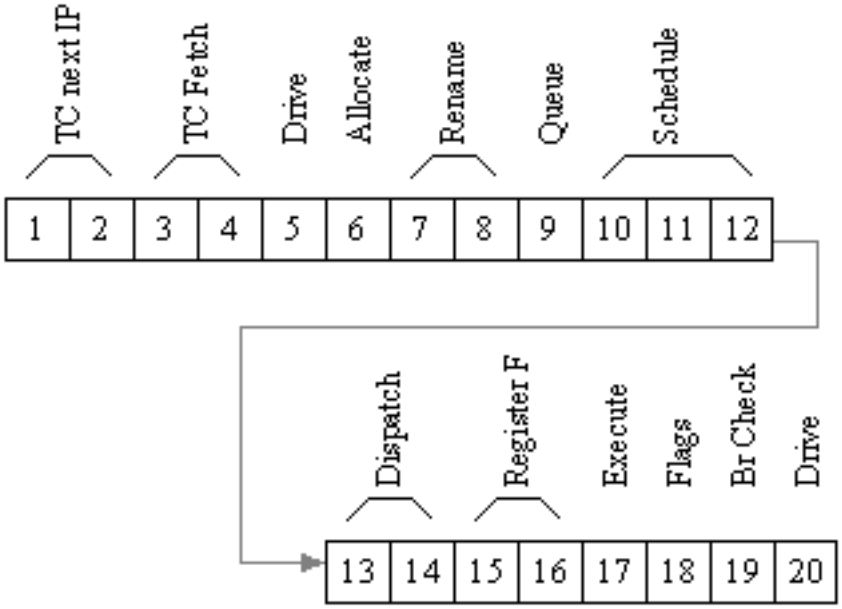

Figure 4 Basic Willamette Execution Pipeline

As shown in Figure 4, the Willamette is a very deeply pipelined processor. It uses 20 pipe stages to execute integer instructions including the 4 pipe stages associated with fetching uops from the trace cache. If you included the pipeline stages associated with fetching x86 code from the L2 cache, decoding it into uops, and loading uops and program mapping/flow information into the trace cache the total number of pipelines stages probably approaches 30 or more.

The branch misprediction penalty appears to be at least 19 clock cycles when the correct path is present in the trace cache. If the trace cache misses, then the branch mispredict penalty is considerably higher. This compares to a minimum branch mispredict penalty of 11 cycles for the P6 core. The P6 uses the two-level Yeh and Patt adaptive branch prediction scheme. Despite the fact that the P6 predicted branches correctly around 90% of the time it still lost about 30% of its potential performance due to branch mispredicts. Although the Willamette will no doubt use more modern branch prediction techniques like gshare and dynamic prediction strategy selection, its huge mispredict penalty will make its performance very sensitive to the efficacy of its branch prediction algorithm(s) on the particular code being run.

There is no doubt that Willamette will achieve much higher clock rates than a P6 core in the same process. The open question is whether or not it can deliver higher performance than P6 commensurate with, or exceeding, its higher clock rate. Or maybe Willamette is an expensive demonstration that microarchitectural innovation in implementations of the ancient x86 ISA have long gone past the point of diminishing returns.

In the second part of this article I will examine the unique features of Willamette like the trace cache and double speed ALUs in detail.

What's Up With Willamette? (Part 2)

April 4, 2000 by Paul DeMone

This is the second installment in a two part article about Willamette, the next generation x86 processor design from Intel. In the first part the technical and historical context of Willamette was presented along with how it differs from current Intel x86 processors. In this second part I'll examine two of Willamette's defining features – its trace cache and its double frequency arithmetic logic units (ALUs) in detail on the basis of available information and estimate their likely impact on Willamette's performance.

Peering Behind the Veil of Secrecy

When it comes to revealing information about a new mainstream microprocessor that is nearly a year away from volume production, Intel Corporation is usually about as forthcoming as the U.S. National Security Agency. However, these are strange and unsettling times for Intel as it is faced with a competitor, Advanced Micro Devices (AMD), offering a product for the expansive and lucrative mainstream x86 PC market that is at least the technical match of Intel's best. Worse still for Intel, AMD appears to be able to manufacture this processor, the K7 Athlon, at clock frequencies and in quantities to match its rapidly growing market acceptance, while Intel suffers prolonged and highly publicized difficulties in ramping the frequency and production volume of its Coppermine Pentium III. Because of the unusual competitive situation it finds itself in, Intel is under a lot of pressure to release details about its upcoming Willamette processor far in advance of its availability. At the recent Intel Developers Forum, Intel publicly demonstrated prototype Willamette devices operating at clock rates up to 1.5 GHz. Furthermore, it revealed that the integer arithmetic logic units (ALUs) in the Willamette operated at twice the processor frequency, or up to 3 GHz. These eye-popping frequencies generated a lot of ink within the technical press and helped divert attention from the fact that Intel was losing the race to ship standard product offerings at clock rates up to 1 GHz.

The description of Willamette that follows is based on a mixture of limited architectural disclosures by Intel, several U.S. patents recently issued to Intel that apparently relate to technology used within Willamette (namely 6,018,786 and 6,023,182), and a lot of speculation. In many cases I will propose possible design approaches to specific Willamette features where the actual details are unavailable. With any complex exercise in speculation it is inevitable that errors, false trails and red herrings will creep in. Even apparently concrete details from Intel's statements or patents shouldn't be taken as gospel as it quite possible (and even likely) that Intel has employed some degree of misinformation to try to conceal detailed operational characteristics, manufacturing costs, and performance levels of Willamette from its competitors.

What is a Trace Cache?

The most interesting feature of Willamette is its trace cache. The trace cache differs from a conventional instruction cache in two basic ways. First, the trace cache stores micro-ops (uops) instead of x86 instruction bytes, and second, the trace cache organizes program data by the expected execution flow rather than by memory location. To appreciate the large potential advantage of the Willamette trace cache consider the simplified block diagram of a modern decoupled execution x86 processor like the P6-based Pentium III or the K7 Athlon shown in Figure 1.

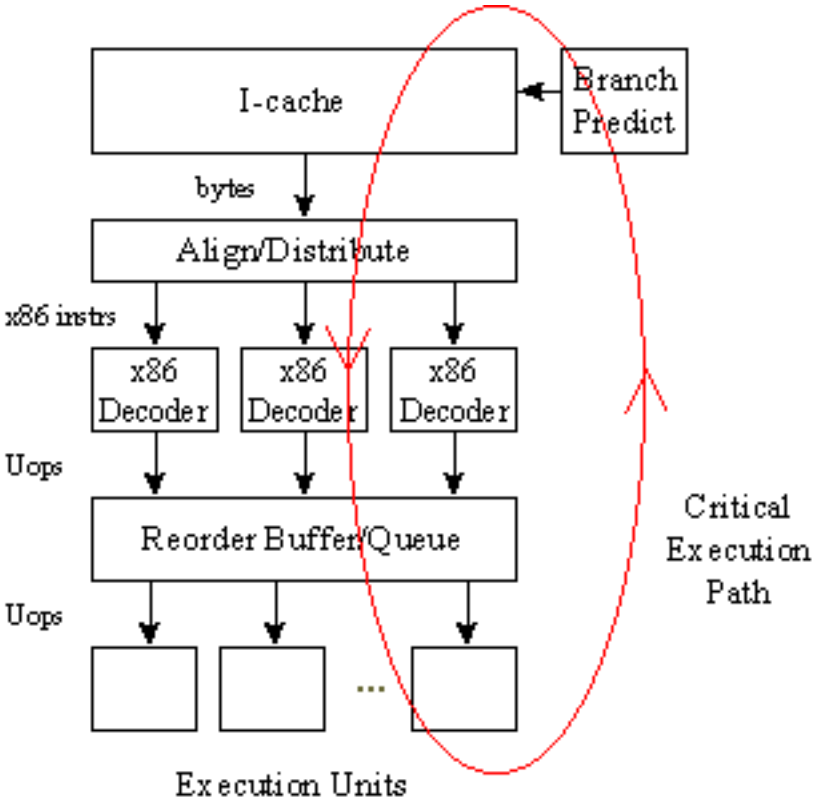

Figure 1. Organization of a Modern x86 CPU with a Conventional I-cache

Notice that in the conventional processor the critical execution path includes branch prediction, fetch of raw program bytes from the I-cache and the recognition, parsing, extraction, and decode of x86 instructions to obtain a stream of uops to drive the out-of-order execution back end. It takes quite a bit of logic and wide signal paths to be able to fetch, align, and decode multiple x86 instructions per clock cycle. Contrast that with the Willamette block diagram shown in Figure 2.

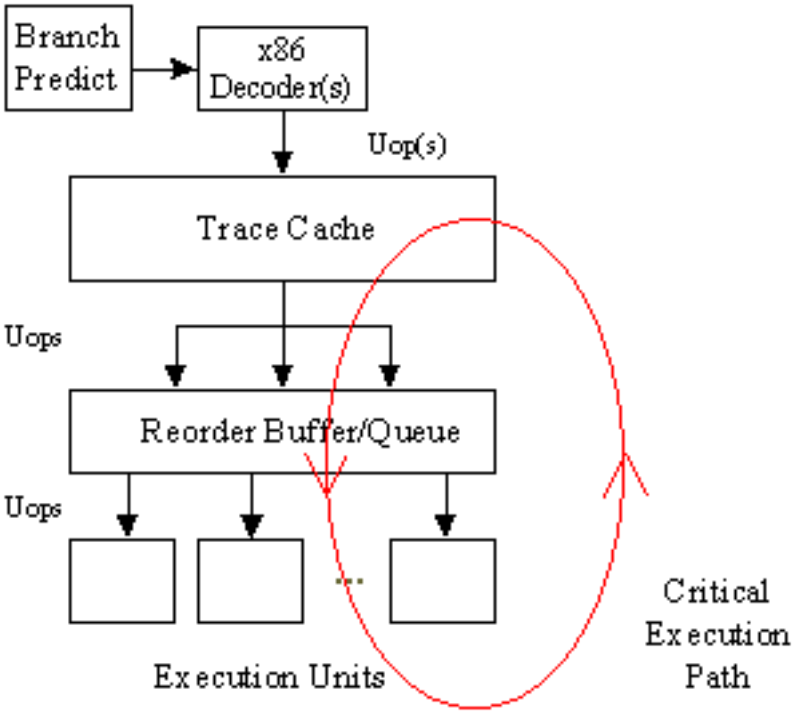

Figure 2. Organization of Willamette With Trace Cache

Notice that in Willamette the branch prediction and x86 decoding steps are no longer within the processor's critical execution path. Also notice that program execution parallelism is no longer limited by the number of x86 decoders. In fact the Willamette may only have one or two x86 instruction decoders.

The reason why this is possible is because the trace cache actually operates in two different modes. In "trace segment build mode" the control logic in the trace cache fetches x86 instructions for the predicted program execution path and decodes them into uops in response to a processor trace cache access that misses. These uops are gathered together and stored as a logical group called a trace segment in the cache. The uops can also be bypassed so that they feed the execution engine even while the trace segment is assembled.

Upon completion of a trace segment the trace cache enters "execute mode". In this mode, uops are fetched in predicted program order from the trace cache and executed. The degree of program parallelism available for the execution engines is limited only by the number of uops that can be fetched from the trace cache per clock cycle, rather than by the number of x86 instruction decoders present. The trick to getting good performance out of the trace cache is the same for all caches, program execution in loops. Willamette gets high performance whenever it can stay in trace cache execute mode (and out of trace segment build mode) for extended periods such as in program loops. An important point to remember is that while a uop coming out of an x86 instruction decoder in a P6 or K7 processor is only ever executed once, a uop coming out of the Willamette's x86 instruction decoder is retained and may be executed tens, hundred, or even thousands of times.

A second important issue is branches. In some ways, Willamette actually executes branches twice. The first time is when the trace cache is in trace segment build mode. When it encounters a

conditional branch, the control unit has to predict which way program flow is likely to go and it continues to assemble the trace segment accordingly. Later on, in execute mode, the branch's uop is executed and the actual branch condition resolved. If the branch goes against the trace segment's implicit branch direction then a misprediction has occurred. When this happens the processor immediately stops executing the current trace segment, discards all computation results calculated by instructions following the mispredicted branch, and tries to locate a different trace segment within the trace cache based on the true branch direction. Willamette has about a twenty-cycle branch misprediction penalty even when the alternative path trace segment is present in the trace cache, but most of this is due to the stretched out execution pipeline. The trace cache can actually lookup a new trace address and resume uop streaming within four clock cycles of receiving a misprediction signal. If this lookup fails then the processor drops back to trace segment build mode and the branch misprediction penalty is far greater than twenty cycles.

Trace Cache Implementation

The Willamette trace cache actually consists of two distinct memory blocks. Uops are stored in the so-called data array while trace segment address and control information is stored in the tag array. This organization is shown in Figure 3.

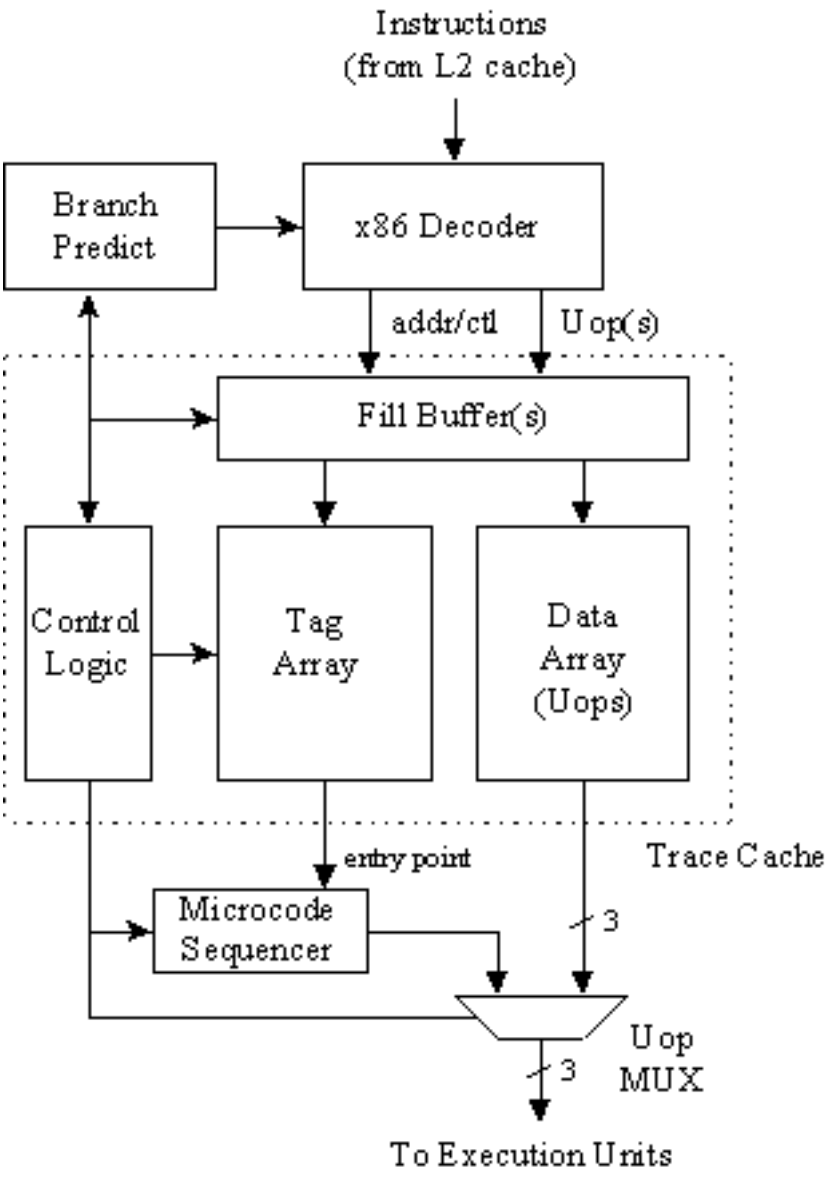

Figure 3. Trace Cache Organization

The processor initiates a trace cache access by performing a trace segment lookup based on a program address. This is essentially a fully associate CAM (content-addressable memory) based search for a matching program (“linear”) address within the tag array. If successful, i.e. a trace cache hit, the trace cache starts fetching trace segment members (groups of uops) from the data array. This fetching process continues in a sequential manner through the data array (wrapping at the end of the array if necessary) until the end of the trace is detected, at which point another trace segment lookup is performed or until a branch mispredict is detected. If the trace segment lookup fails then the processor enters trace segment build mode.

Some x86 instructions are quite complex and take many (possibly hundreds) of uop equivalent processing steps to execute. These so-called “CISCy” or complex x86 instructions are cleverly handled in a special way in order to not to pollute the trace cache with long “canned” sequences of

uops driven by microcode. In trace build mode when a CISCy instruction is encountered, the control logic adds a special microcode entry point address to the tag entry for that trace segment member and possibly a few prologue uops to the data array. In execute mode, the trace cache detects the presence of the microcode entry point in the tag and transitions to a special operating mode in which uops generated by a microcode sequencer unit are steered into the trace cache output path. Upon completion of a complex x86 instruction by the microsequencer, the Willamette trace cache resumes normal trace segment fetching from the next data line. The process of entering and exiting microsequencer mode is carefully integrated into normal trace cache operation and can be accomplished with very little overhead.

The organization of trace segment member uops within the data array is shown in Figure 4. A trace segment consists of one or more trace segment members that are groups of six uops stored within one of four “ways” within the trace cache data array. The first segment member in a trace is called the head, the last segment is called the tail, and every member in between are called body segment members. Two bits in the tag entry denote whether a trace segment member is a head, body, or tail element. The distinction is important to the operation of the finite state machines that drive the cache control logic. As previously mentioned, the head segment member of a trace is located by performing a lookup with the tag array. Subsequent members are stored in consecutive sets (rows) within the data array. The way location of trace data is stored as a couple of bits within the tag array elements which are effectively chained together like a doubly linked list for each distinct trace.

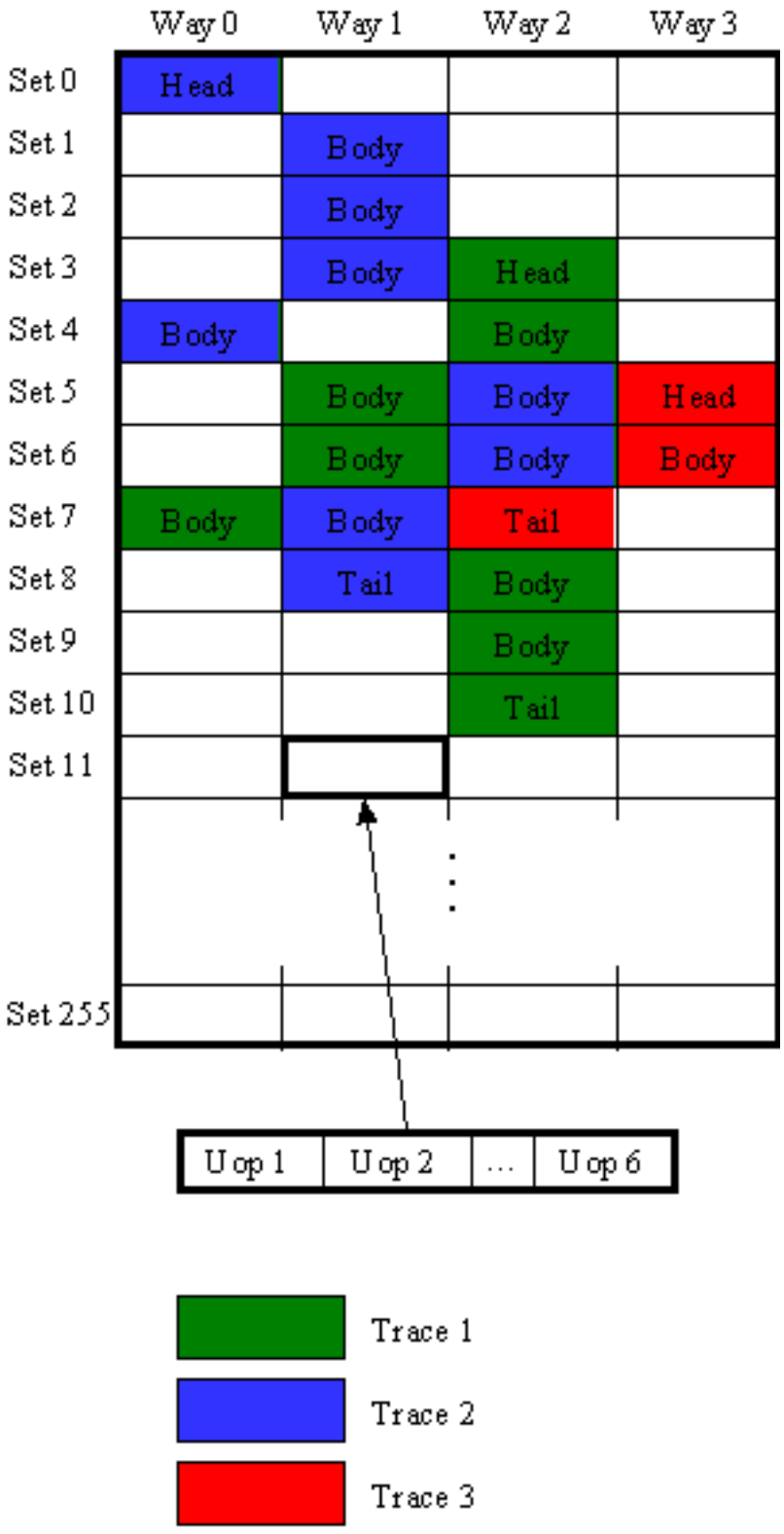

Figure 4 Trace Segment Organization Within the Data Array

In trace segment build mode the trace cache control logic assembles incoming uops from the x86 instruction decoder(s) into groups of up to 6 uops in size within the trace cache fill buffer. These

groups may be filled with fewer than 6 uops (and presumably padded with NOPs) for a variety of reasons, including implementation restrictions such as not splitting uops from a single x86 instruction across data lines or hitting a maximum number of branch uops permitted per data line. When a data line is finished (either 6 uops are collected or some restriction came into effect) the contents of the fill buffer are written into the data array. The choice of way is made based on a least recently used (LRU) strategy to avoid overwriting any portion of a recently active trace segment (which is more likely to be executed again). The sharing of the data array by trace segment members from three different traces (each a distinct color) is shown in Figure 4. In trace segment build mode, the trace cache logic continues to decode x86 instructions and append uops to the current trace until a trace segment termination condition occurs. Trace segments are terminated when 1) an indirect branch, call, or return instruction is encountered, 2) a branch misprediction or exception is raised, or 3) the trace segment length reaches a set limit (64 was suggested in the trace cache patent).

Notice that in Figure 3 the output path of the trace cache is shown as being 3 uops wide, yet a trace segment member actually consists of up to 6 uops. Interestingly, the Intel trace cache patent shows no possible mechanism to separately address two halves of a segment member. My conclusion is that the Willamette trace cache outputs 6 uops every clock cycle using two separate transfers of three uops each. (i.e. the output path is double pumped). This is also a smart implementation choice as it eliminates over 350 bused signals from one of the most congested areas of the processor. The ability of the trace cache to provide 6 uops every cycle also seems to be the most reasonable conclusion in light of the amount of parallel execution resources available within the Willamette.

What is an Adequate Size for a Trace Cache?

The uops in the original P6 core are 118 bits long. The uops stored in the Willamette are unlikely to be any smaller than P6 uops since the instructions set has increased in size and there are apparently a few extra bits needed to help correlate uops with the originating x86 macro instructions. Let’s assume the uops stored within the data array of Willamette trace cache are 120 bits long. The Intel trace cache patent suggests an implementation of 256 sets by 4 ways by 6 uops per way. This totals 256 * 4 * 6 * 120 or a total of 90 Kbytes of SRAM. This is quite large compared to the 16 Kbyte capacity of the I-cache found in recent P6 implementations. Also, the tag and control data for the trace cache are almost certain to be larger than the tags found in a conventional cache design. The Willamette has been attributed with 34 million transistors or about 6 million more than the Coppermine Pentium III. No doubt a good chunk of the extra transistors are swallowed up by the innovative but expansive trace cache.

How does one judge the adequacy of our hypothetical 90 Kbyte trace cache? Should it be larger? Or is it wastefully large? One simple way of assessing this issue is to look at the relationship between x86 instructions and uops. The average length of IA-32 x86 instructions has been reported as 3.2 to 3.7 bytes in length. Let’s assume it is 3.5 bytes (28 bits). In the P6 design an x86 instruction

generated, on average, about 1.5 uops. Assuming a rough equivalency between P6 and Willamette uops, an x86 instruction expands to about 180 bits of uop on average. This is an expansion ratio of about six to one. The Willamette trace cache does prevent complex x86 instructions from polluting the trace cache, but these instructions are probably so rare as not to significantly affect the expansion ratio. So our hypothetical 90 Kbyte trace cache data array holds, at most, the uop equivalent of about 15 Kbytes of x86 code. I say “at most” because trace segment members are not necessarily always fully packed with 6 useful uops. I doubt this 15 Kbyte equivalency figure I’ve derived is nearly the same as the 16 Kbyte I-cache size of the P6 by accident. No doubt the Willamette’s designers wanted to limit the size of the trace cache so as not to impact the clock cycle time but without making it less effective in exploiting program spatial locality than the conventional I-cache in the P6-based processors it will replace

Willamette’s Arithmetic Logic Units (ALUs)

One of the biggest surprises disclosed about Willamette concerned its arithmetic logic units, or ALUs. An ALU is the functional block within a processor that actually computes. The ALU performs such operations such as add, subtract, compare and bitwise logical instructions such as as ANDing a register value with a bitmask constant. In most processors the ALU is set up to be able to perform a new computation every clock cycle. This is accomplished by surrounding the ALU with input and output registers and inserting bypass multiplexers (MUXes) at the ALU inputs as shown in Figure 5.

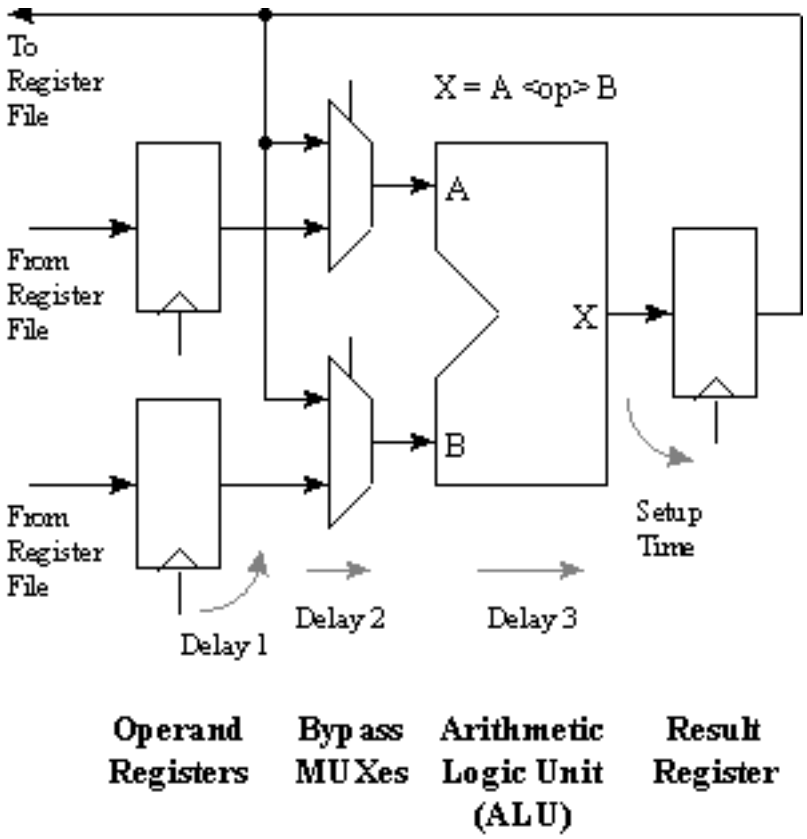

Figure 5. Conventional Single Cycle ALU Configuration

In this design, values from the register file, memory, or instruction immediate field are latched into the ALU operand registers on rising clock edge “N”. These values are then passed to the ALU though a set of bypass MUXes into the ALU. The result of the selected operation on the input values is latched temporarily into the result register on the following rising clock edge “N+1” before being sent on back the register file (and also back around to the ALU if the result is an input to an instruction executing a clock cycle later).

To see how the speed of the ALU affects the maximum clock rate of the processor we need to examine the path that signals must propagate through in one clock cycle. In this design, the input data takes some time from the rising clock edge “N” to appear at the output of the operand registers (Delay 1), time to pass through the bypass MUXes (Delay 2), and for the ALU to calculate the result (Delay 3). In order to reliably capture the operation result, this value must appear out of the ALU a little ahead of the second rising clock edge “N+1” (Setup time). We can express the maximum clock rate for this ALU pipeline organization as:

Clock Period (1/Clock Rate) &gt = Delay 1 + Delay 2 + Delay 3 + Setup Time

In most processors the ALU propagation delay dominates this speed path, but the other delays cannot be ignored. A simple rule of thumb for high clock rate MPUs employing this type of microarchitecture is the ALU delay can comprise at most about 65 to 70% of the minimum clock period value. For example, a 1 GHz processor has a minimum clock period of 1.00 ns so we would expect the worst case delay through the ALU to be no more than 0.65 to 0.70 ns.

ALU delay is generally dominated by the add/subtract circuit. Addition and subtraction are difficult operations to perform quickly because in the worst case a carry has to be propagated across all 32 bits. There are a variety of ways to build fast adders (I will talk only about adders from now on because addition and subtraction are nearly identical problems). In general, the faster an adder design is the more logic gates and chip area required to implement it. Table 1 shows some representative adder circuit delay values for processors from the past fourteen years.

Year	Processor	Technology	Clock Rate (MHz)	Delay
1986	Intel 386	1.5 mm CMOS	16	9 ns
1989	Intel 486	1.0 mm CMOS	25	5 ns
1990	HP PA-RISC	1.0 mm CMOS	90	3.5 ns
1994	PowerPC 603	0.50 mm CMOS	80	3 ns
1997	PowerPC G3	0.30 mm CMOS	250	1 ns
1999	PowerPC (64 bit)	0.22 mm CMOS (SOI, copper)	550	0.66 ns (64 bits)

Table 1 Adder Performance in Microprocessors, 1986 to 1999

When Intel demonstrated the Willamette processor at their developer's forum last month, not only were these 0.18 um aluminum interconnect devices running as fast as 1.5 GHz, Intel disclosed that the ALUs were in fact operating at twice the processor frequency or 3.0 GHz! This fact is incredible when one considers that if one extrapolates the data in Table 1 it is hard to see how Intel could build a 32 bit adder circuit with delay shorter than about 0.50 ns in their P858 0.18 um process. Using the rule of thumb previously described would seem to limit ALU operation to about 1.3 GHz which is well short of 3.0 GHz. So it is quite obvious that Intel has implemented an ALU pipeline completely different from that shown in Figure 5, and which is unlike any yet seen in a publicly disclosed microprocessor or digital signal processor.

The Willamette's 3.0 GHz ALUs are surprising, but existing techniques such as superpipelining (letting the ALU propagation delay stretch over two or more clock periods) are available that could easily explain such high clock rates. But Intel actually packed a double whammy into their Willamette ALU disclosure. Normal application of superpipelining provides high clock rates but at the cost of extra latency. If one stretches an addition across two pipe stages then one wouldn't expect that an instruction that uses the result of an addition to be able to execute until two or more clock cycles after the instruction performing the addition. Yet Intel has clearly indicated that the Willamette can cascade the result of an add instruction into a subtraction and then through a move instruction and

finally feed an xor instruction in just two clock cycles. This is as shocking as if the Olympic track and field 10 km champion also won the 100 m dash.

How the Heck do They Do That?

The obvious conclusion is that Intel is has rolled out some serious innovation in integer datapath microarchitecture for Willamette and possibly some new circuit techniques too. I will put forward one possible explanation of how the Willamette ALUs are implemented. It may not be exactly what Intel has done but the odds are it is not too far off. First of all, the double clocking of the ALU has a lot of people amazed. There are three basic techniques for making a small section of circuitry operate at twice the global clock rate. The first is to divide the unit into two sections and clock one normally on the rising edge of the clock and the second section on the falling edge of the clock and then interleave and combine the results. The second is to employ flip-flops (memory circuits) that operate on both edges of the clock The third method is to simply provide a special clock signal to the unit that toggles at twice the frequency as the processor clock.

Intel has apparently chosen the third option. In U.S. patent 6,023,182 Intel discloses a clock buffering circuit that, among other things, generates an out put clock at twice the frequency of the input clock using one shot pulse generators triggered on both the rising and falling edges of the input clock. This circuit can be dropped down anywhere on a microprocessor to drive a functional block such as an ALU at twice the rate of the globally distributed processor clock. The advantage of this scheme is that it avoids the formidable challenge of generating and distributing both a regular clock and a double frequency clock over the entire device. The operation and likely use of this clock frequency doubling buffer in Willamette is shown in Figure 6.

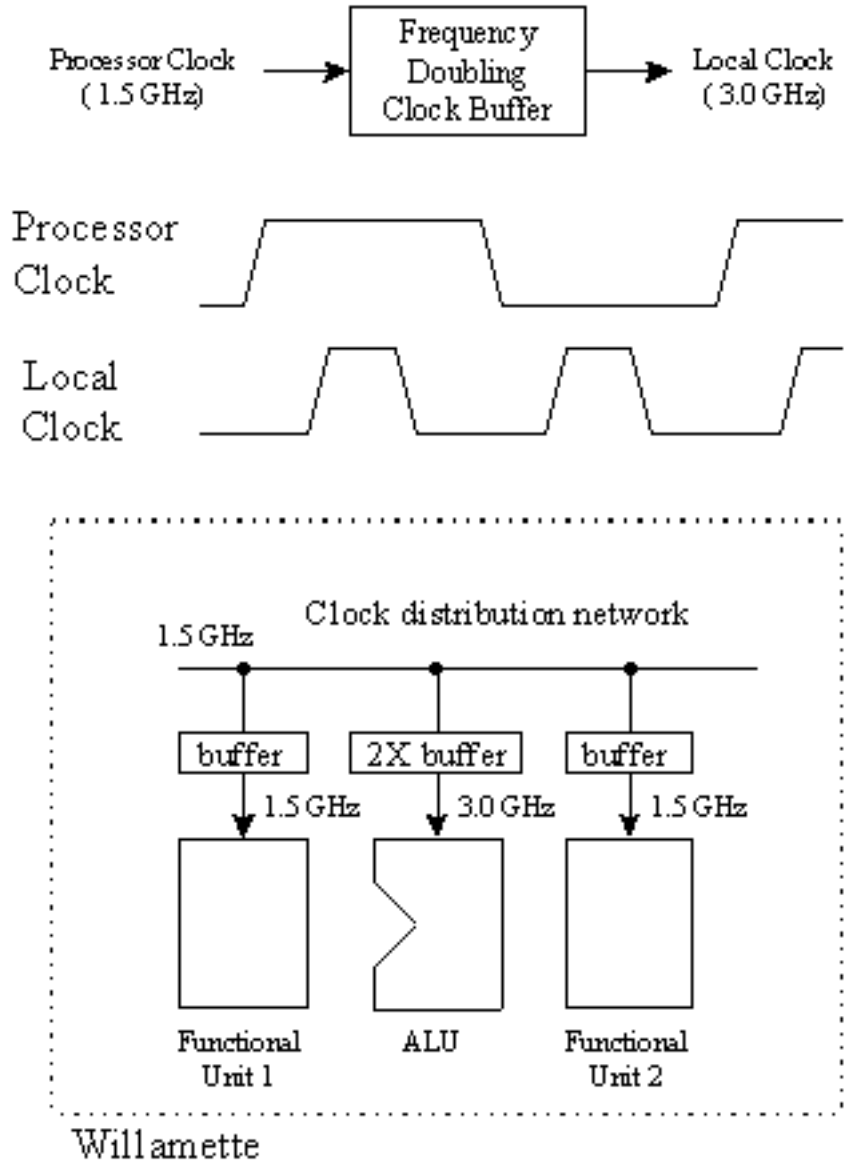

Figure 6. Hypothetical Willamette ALU Clocking scheme

So, we know how Intel can clock its ALUs at 3.0 GHz while surrounded by a sea of logic operating at 1.5 GHz., but I still haven't explained how an addition can apparently be performed in the 0.20 ns or so needed to fit into a single stage of a pipeline operating at 3.0 GHz. Could Intel have discovered a new way of performing addition that only takes 40% of the time needed by standard adder circuits used today? This possibility is extremely remote. Engineers and mathematicians having been studying binary math and arithmetic circuit design since the days computers were built using electromechanical relays 55 years ago and all the possibilities have been pretty well gone over. There are known techniques, such as logarithmic adders, that can add faster than the carry lookahead design typically used in microprocessors, but the speed up for a 32 bit addition is not that great and comes at a fairly large cost in area.

As I said before, the really time consuming part of addition occurs when a carry is generated at the least significant digit and needs to be propagated up 32 bits to the most significant bit. This is analogous to manually adding 1 to 999999 using pencil and paper arithmetic. However there is a trick often used in multiplier arrays called carry-save addition which gets around this. A carry-save adder basically puts off the task of carry propagation by generating two results, a partial sum value and a bit vector of carries. A decimal version of a carry-save adder, given our previous example of 1 plus 999999, would say the answer is 999990 (partial sum) plus 000010 (carry vector). As we see here, the drawback of the carry-save adder is that we eventually need a true adder circuit to combine the partial sum and carry. Although it appears practically useless, carry-save addition has the advantage of being able to perform a long series of additions very quickly, and requiring a single regular addition at the end to generate the true answer.

So, I think the secret of the Willamette adder is that it cannot perform a complete addition in half a processor clock cycle. Instead, it performs a carry-save addition (which is very fast – basically the delay of a single bit full adder circuit which is likely 0.1 ns or less in 0.18 um CMOS) in the first adder pipeline stage, and feeds back the partial sum and carry vector to two of the three adder inputs through bypass MUXes. This allows a second instruction which depends on an addition result for one of its inputs to be started an ALU clock period (half a processor clock period) after the addition enters the adder. This still leaves us with the problem of performing a true full 32-bit addition in one or more subsequent pipeline stages within the adder. One solution is to place the first part of binary look-ahead carry (BLC) adder in the back end of the first adder pipeline stage, and the remainder of the BLC adder in the second and third stage. The BLC adder is a logarithmic class adder that has the advantage of being easily pipelined because it has the same number of logic levels from its inputs to its outputs. Spreading a BLC addition across two and half ALU clock periods gives about 0.5 ns to perform the operation after accounting for pipelining overhead and should be quite readily accomplished in Intel’s P858 0.18 um process. This hypothetical Willamette adder scheme is shown in Figure 7. The drawback of this adder design as shown is that it cannot issue a dependent addition two ALU clock cycles after the first addition (although this would be possible if the partial sum and carry vector were retained through one more pipe stage and brought back to the bypass MUXes.)

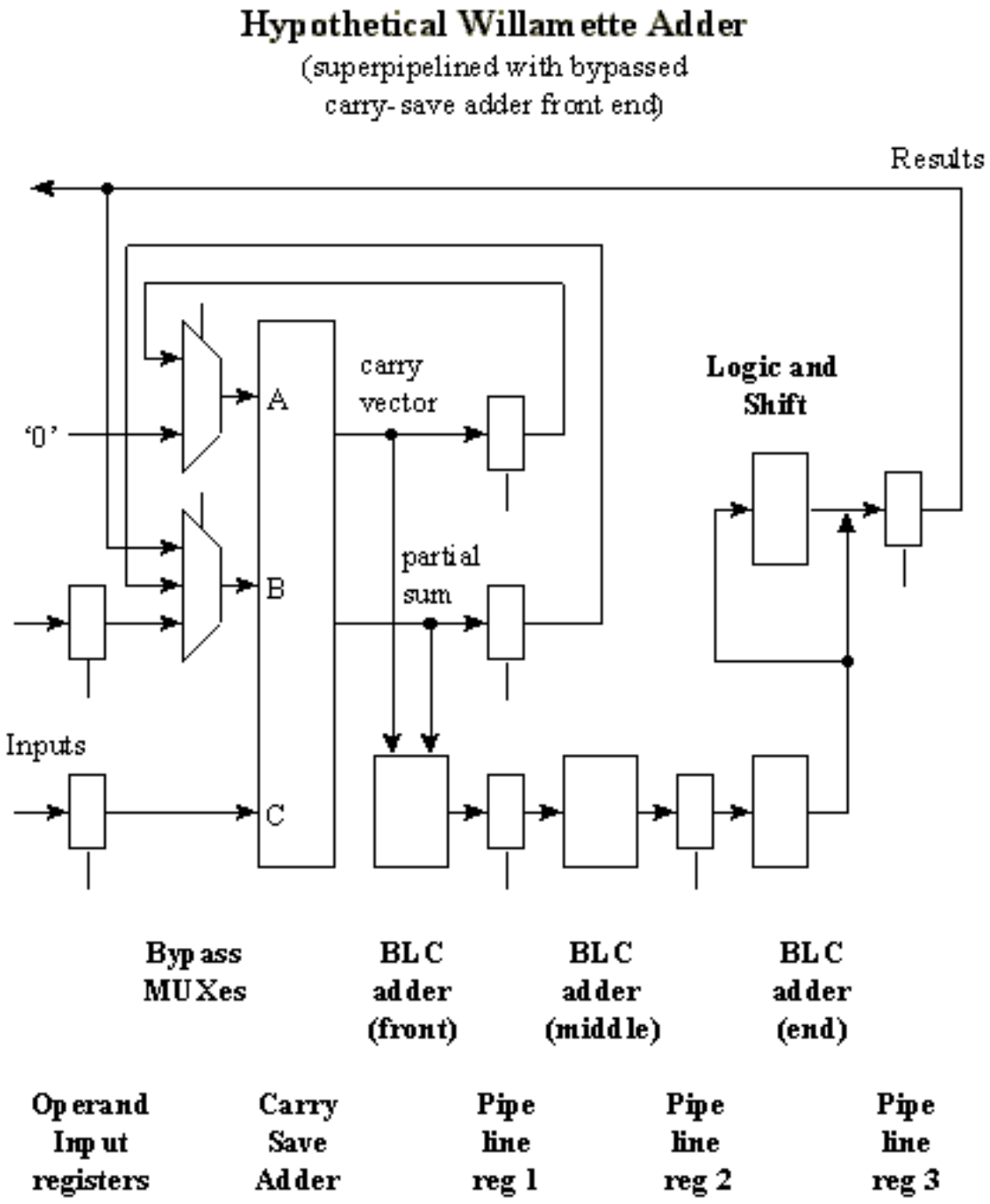

Figure 7 Hypothetical Willamette ALU Adder

Putting It Altogether

It should be obvious by now that the trace cache and doubled frequency ALUs in Willamette shows that Intel has come up with a delightful bag of new tricks to teach the old x86 dog. So how fast is it going to be? Well it is obvious from the pipeline diagram of Willamette in part one of my article that it

is designed for speed. It even has two complete pipeline stages (named “Drive”) which ostensibly reserve an entire processor clock cycle just to move signals from one part of the chip to another. A conservative guess is that the Willamette will achieve at least 50% higher clock frequencies than the P6 core in the same manufacturing process.

It gets trickier trying to estimate how Willamette will perform compared to P6 on a clock for clock basis. One should realize that it takes a great deal of renovation to a microarchitecture just to keep from losing clock normalized performance from non-scaling memory latency when one increases the clock rate. There are also a huge number of details about Williamette that Intel has not disclosed yet, including all the warts. We don’t know the instruction scheduling and dispatch rules. We don’t know how the Willamette’s branch prediction hardware works and how effective it will be. The efficacy and robustness of the trace cache on a variety of new and existing applications has yet to be assessed. But it is likely that Willamette can issue 6 uops per clock cycle, twice as many as P6. The Willamette can issue up to four integer operations to its two double frequency ALUs each processor clock cycle, again twice as many as P6 (although we don’t what restrictions apply). The trace cache hides branch prediction and x86 instruction decoding from the Willamette’s out-of-order execution engine, performs limited loop unrolling, and reduces the time to switch the uop stream to the correct address when a branch mispredict is detected and the alternate path code is resident in the cache. The Willamette also has a much larger instruction re-ordering window than the P6 and can support more than a hundred instructions in flight at a given time along with 48 outstanding loads and 24 outstanding stores.

I would go so far as estimate that Willamette might achieve 20 to 30% higher performance than P6 on a clock for clock basis on most integer code. Including the 50% or more higher clock rate, that is equivalent to an absolute integer performance approaching twice that of the P6 in the same manufacturing process. That increase seems staggering, but it is much less, for example, than the over three times higher system bus bandwidth advantage Willamette enjoys over current P6 implementations. On memory performance limited code the Willamette might dominate the P6 even more.

The Willamette is a serious threat to end the success AMD has been enjoying with its K7 Athlon device in the high end of the x86 processor marketplace. It will likely support clock frequencies about 50% higher than P6 and 30% higher than K7 in a similar process (although AMD will soon benefit from a switchover to a copper interconnect based 0.18 um). Everything else being equal, Willamette will likely enjoy at least a similar level of clock normalized performance or IPC advantage running general purpose integer code over K7 that the K7 enjoys over the P6 core. The Willamette will likely be larger than the K7 core in similar processes, although not dramatically so. The trace cache occupies a great deal more area than the 16 Kbyte I-cache in the P6 and maybe 50% or more area than the 64 KB I-cache in the K7. However, this is balanced by the fact that Willamette likely devotes less area to x86 instruction parsing, alignment and decoding than the K7 or even the P6. Willamette runs at very high clock rates and will likely consume at least twice as much power as P6, or somewhere in region of 60 Watts or more

Conclusion

Intel has come up a surprising number of microprocessor design innovations to power its new Willamette 32 bit x86 processor targeted at replacing the venerable, but aging, P6 core. The trace cache extends previous work in this area in several important ways, and solves some of the known nagging problems. The trace cache control logic handles complex x86 instructions separately so as to reserve cache capacity for the common and fast instructions that translate to just one or a few uops. It also removes the x86 decoders from the critical execution path and decouples uop issue rate from the number of x86 decoders for the majority of the time when previously translated uops are re-executed in program loops.

The Willamette's double clock rate arithmetic and logic units (ALUs) are an interesting development in an area of microprocessor design that has changed very little since the earliest days. The double clock rate ALUs provide the expected benefit of allowing two physical ALUs to perform the work of four logical ALUs as far as the rest of the processor is concerned. The biggest surprise of all is that dependent chains of instructions can issue to this superpipelined ALU at half processor clock intervals. The exact details of how this is accomplished by Intel and what restrictions apply are not yet disclosed, but I have shown one possible way this feat could have been done without imposing onerous issue rules.

The effect of Willamette on the competitive landscape will be enormous. It puts the onus back on AMD to seriously improve on the K7 and make more than cosmetic changes to its basic design to create the K8. Willamette represents a double-edged sword for Intel outside the x86 world. It will keep up the heavy pressure on RISC processors in the workstation and low-end server market, especially in the form of the Foster high end variant. It might even challenge the mighty Alpha EV68 for the SPECint crown. On the other hand it will make it very difficult for Intel's own Merced/Itanium IA-64 processor to make a mark for itself on the basis of performance, except for very floating point intensive applications. In summary, the Willamette appears to be a tremendous technical achievement. If Intel can put even half of the imagination and innovation shown by their Hillsboro design team into McKinley, then its competitors in the 64 bit market could suffer the same fate as competing x86 designs when Willamette ships in volume.