Advanced Performance Features of the 64–bit PA–8000

Doug Hunt

Hewlett–Packard Company
Engineering Systems Lab
3404 East Harmony Road, MS#55
Fort Collins, Colorado 80525

Abstract*: The PA–8000 is Hewlett–Packard's first CPU to implement the new 64–bit PA2.0 architecture. It combines a high clock frequency with a number of advanced microarchitectural features to deliver industry–leading performance on commercial and technical applications while maintaining full compatibility with all previous PA–RISC binaries. Among these advanced features are a fifty–six entry instruction reorder buffer to support out–of–order execution, a branch target address cache, branch history table, support for multiple outstanding cache misses and dual integer, load/store, floating point multiply/accumulate, and divide/square root units which allow execution of four instructions per cycle. Together, these features will enable the PA–8000 to sustain superscalar operation on a wide variety of workloads.*

1. Introduction

Hewlett–Packard's PA–8000 CPU is designed to deliver industry–leading performance on today's commercial and technical applications while providing a growth path to future 64–bit applications. Maintaining industry–leading performance requires improvement in both clock frequency and the average number of clock cycles per instruction (CPI). Since RISC processors are already capable of starting one operation per cycle, reducing CPI further requires starting more than one operation per cycle. The PA–7100[1], PA–7100LC[2], and PA–7200[3] have already achieved success as two–way superscalar implementations, but adding still more functional units is not useful if the rest of the processor is not capable of supplying those functional units with a continuous stream of operations to perform. With the PA–8000, Hewlett–Packard introduces an entirely new microarchitecture with a carefully chosen set of features designed to *sustain* superscalar operation on real–world applications.

First, the PA–8000 includes two integer ALUs, two shift/merge units, two floating point multiply/accumulate units, two divide/square root units and two load/store units. These functional units are arranged to allow up to four instructions per cycle to begin execution. To supply these functional units with enough work to keep them busy, the PA–8000 incorporates a fifty–six entry Instruction Reorder Buffer (IRB) and a dual ported data cache. In order to keep the buffer as full as possible with instructions to choose from, the instruction fetch unit is designed to supply four instructions per cycle from a single–level, off–chip cache. Finally, in order to maximize the usefulness of the instruction and data caches, a high performance bus interface capable of supporting multiple outstanding cache misses is provided. This set of features will enable the PA–8000 to deliver >360 SPECint92 and >550 SPECfp92 at first release. Figure 1 is a block diagram of the PA–8000 which shows how these components are organized.

1.1. Why out of order?

The fundamental difficulty in achieving sustained superscalar operation is finding enough independent work to supply the multiple execution units. One way to handle this problem is to give the burden of finding the parallel work to the compiler by requiring that it order the instructions so that every instruction fetch includes more than one instruction which may be executed at the same time. Unfortunately, there are many situations where the compiler cannot take advantage of potential parallelism because of the limited information available at compile time. The design of the PA–8000 therefore leaves the scheduling of instructions up to the hardware, which can perform more aggressive reordering than the compiler thereby achieving a higher utilization of the functional units.

The PA–7100, PA–7100LC, and PA–7200 gain substantial benefit from their two–way superscalar operation while leaving scheduling to the compiler, but it became apparent during the investigation which led to the development of the PA–8000 that some problems would require a fundamentally different approach as the move was made from two–way superscalar to wider implementations. One of those problems is that legacy code which was compiled with an earlier compiler would not be optimally scheduled and would receive very little benefit from the added superscalar hardware. The more sequential instructions the hardware attempts to execute at once, the more likely it is that the instruction group will contain instructions which depend on one another and cannot be executed simultaneously. The PA–8000 addresses this problem by having the hardware scan a large portion of the program at one time in order to find opportunities for parallel execution, rather than only considering four instructions at once. With its large reorder buffer, the PA–8000 can examine over fifty instructions at one time to find four which are ready to be executed.

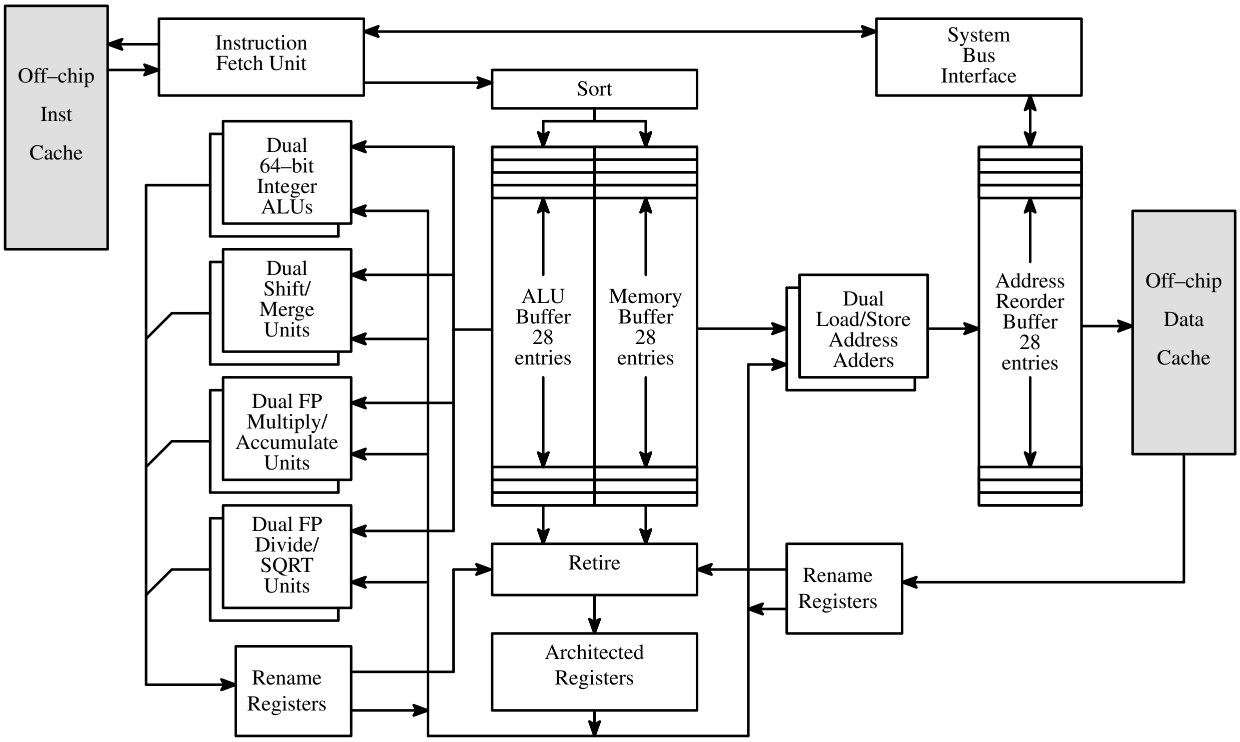

Figure 1: PA–8000 Block Diagram

A second problem with leaving scheduling up to the compiler is that the compiler often can only reorder instructions within a narrow window because of such unknowns as flow of control and pointer aliasing. The PA–8000 addresses this problem by performing speculative execution of instructions. Speculative execution is nothing more than guessing what course the program will take and executing instructions from the appropriate path. If it is later discovered that the guess was incorrect, the speculative work is discarded.

The PA–8000 actually performs speculative execution in a number of different ways. First, on every branch fetched, the instruction fetch unit makes an intelligent guess about whether the branch will be taken or not taken and fetches instructions down the appropriate path. When the branch is actually executed, the outcome (either taken or not taken) is compared with the predicted outcome. If the two outcomes do not agree, the correct address (either the branch target address or the inline address) is forwarded to the instruction fetch unit and fetching resumes from that point. All instructions in the IRB younger than the mispredicted branch are discarded.

Another way the PA–8000 performs speculative execution is by executing younger instructions before it is known whether an older instruction will signal an exception (trap). A common example of this situation is the execution of a younger instruction before an older load. The younger instruction must not be allowed to change any program state if the load signals a TLB miss or protection violation. In the PA–8000, the younger instruction is allowed to execute early, but its result is discarded if the older instruction traps.

A third example of speculative execution is the execution of a load before an older store. In this case, it is possible that the load and the store reference the same memory location. This should be a rare event since the compiler tends to keep a value in a register if it will be needed again shortly. However, there are situations where the compiler cannot know that a load and a store point to the same location in memory, especially if the load and store are generated by references through pointers. In the PA–8000, a load may execute before an older store and the hardware checks to see that the load received valid data before the result is committed to the general registers. If a load is determined to have received the incorrect data, the load and all subsequent instructions are flushed from the IRB and refetched.

In each of these three cases, the hardware is able to gain the advantage of performing work early in the common case where the program flow of control proceeds along expected lines and only suffers a performance penalty in the cases where the program flow is not as expected. A compiler often cannot take advantage of the same parallelism because it would have to add so many runtime checks to ensure the reordering was safe that the benefit of reordering the code would be lost.

2. Instruction fetch unit

The IRB can only do its job of supplying the execution units with plenty of work if the buffer itself has an adequate supply of incoming instructions. The PA–8000 instruction fetch unit fetches up to four quadword–aligned instructions per cycle from a single–level off–chip instruction cache. This is the same bandwidth as the maximum execution rate of the functional units. The instruction cache, which is constructed of synchronous SRAMs, has a two cycle latency. Adding the cycle it takes to calculate the target address of a branch means that there is a two cycle penalty for fetching the target of a correctly predicted taken branch from this cache.

To reduce the penalty for taken branches, the PA–8000 incorporates a thirty–two entry fully associative Branch Target Address Cache (BTAC) which associates the address by which a branch is fetched with the address of the target of the branch for branches which are predicted taken. On every instruction fetch, the address sent to the instruction cache is also sent to the BTAC. Whenever the BTAC signals a hit, the address supplied by the BTAC is used as the next fetch address. This means that correctly predicted taken branches which hit the BTAC suffer no penalty, since the quadword containing the target of the branch will arrive on chip the cycle after the branch itself arrives. A new entry is inserted into the BTAC each time a predicted–taken branch is fetched for which there is not already an entry in the BTAC. This insert does not cause any additional instruction fetch penalty. A "round robin" replacement policy is employed.

2.1. Branch prediction

To achieve sustained superscalar operation, it is important that the number of mispredicted branches be minimized. To improve branch prediction accuracy, two different methods of branch prediction are provided: static and dynamic. In static prediction mode, the fetch unit follows the following policy: For most conditional branches, backward branches are predicted "taken" and forward branches are predicted "not taken". For the common compare and branch instruction, a hint is specifically encoded in the instruction to tell the instruction fetch unit which way to predict the branch. Compilers using either heuristic methods or Profile Based Optimization (PBO) can rearrange code segments or use the hinted branches to effectively communicate branch probabilities to the hardware.

In dynamic prediction mode, a 256–entry Branch History Table (BHT) is consulted to determine which way each branch should be predicted. Each entry in the BHT is a three–bit shift register which records the last three outcomes (taken or not taken) of a given branch. If a majority of the last three executions were actually taken, the fetch unit predicts that the branch will be taken again. This table is only updated as branch instructions are retired in order to prevent corrupting the history information with speculative executions of the branch.

The branch prediction mode used (either static or dynamic) is controlled on a page–by–page basis by an extra bit in each entry of the TLB. Thus, it is possible for programs compiled with PBO to take advantage of the profile information, while programs which have not been profiled use dynamic prediction. It is also possible for shared libraries to be profiled, if appropriate, in which case even non–profiled applications will gain the benefit of the profiling of the libraries. This also has the advantage that the library code will not displace the history information in the BHT, improving its effectiveness for the main body of the program.

Note that it is possible for the BTAC to signal a hit (indicating a predicted–taken branch) when the BHT signals that the branch should be predicted not–taken or that an older branch in the group should have been taken instead. In this case, the corresponding entry in the BTAC will be deleted to prevent another hit of the BTAC on that branch.

2.2. Why no on–chip instruction cache?

An instruction cache can improve performance in two main ways: it can reduce the latency of instruction fetches and it can be designed to provide more bandwidth to the processor than the next level of the memory hierarchy can provide. HP's low–cost processor, the PA–7100LC, is the only HP PA–RISC processor to include an on–chip instruction cache. Since the design of the PA–7100LC was driven by the need to minimize overall system cost, a single combined off–chip cache was provided. This necessitated including an on–chip instruction cache to provide sufficient instruction fetch bandwidth to the execution units without overly impacting the data cache performance.

The PA–8000, on the other hand, is designed to maximize performance. For many real–world applications, especially some commercial applications such as transaction processing, delivering high performance requires a larger instruction cache than can be included on–chip. Furthermore, a four–way superscalar design such as the PA–8000 requires a wide connection to this large instruction cache to avoid substantial fetch penalties. Once the decision has been made to design a high–bandwidth connection to a large off–chip cache, an on–chip cache provides no added benefit from a bandwidth perspective.

As far as latency is concerned, the only time the latency of an instruction fetch matters is when a branch is involved. This is because the instruction fetch unit does not need to see the incoming instructions to calculate the next address to fetch if the program is executing sequentially. As mentioned earlier, the BTAC avoids the taken branch penalty for most of the taken branches which are encountered. This means that the only time the reduced latency provided by an on–chip cache would come into play is in the case of a mispredicted branch. Since the aggressive design of the off–chip cache path resulted in a two–cycle latency, an on–chip cache could only save one cycle in the rare event of a mispredicted branch. This being the case, the die area was used for more effective performance features, particularly the fifty–six entry instruction reorder buffer, rather than for an on–chip cache.

3. Instruction reorder buffer

The fifty–six entry Instruction Reorder Buffer (IRB) is physically organized as two separate buffers of twenty–eight entries each. One buffer is used to hold instructions which are destined for either the integer units or the floating point units and the other buffer holds both integer and floating point load and store instructions. Some instructions are inserted into both buffers. These instructions are: (1) load–and–modify instructions, for which the modify is handled by an integer ALU; (2) branches, which go into both buffers to help in recovery from mispredicted branches; and (3) certain system control instructions.

Insertion of instructions into the two buffers in the IRB is controlled by the sort unit. This unit receives the four instructions from the instruction fetch unit and routes each of them to one or both of the buffers in the IRB. Each buffer can accept up to four instructions per cycle, so an arbitrary collection of four instructions may be inserted simultaneously.

Once an instruction has been inserted into a slot of the IRB, the hardware watches each of the instructions launching to the functional units and checks to see whether any of them supplies any of the operands which the instruction in the slot requires. Once the last instruction upon which the slot is waiting has been launched, the slot begins to arbitrate for launch to the functional units. Even though the instructions are segregated into two different buffers, all of the launch information is visible to both buffers. No extra penalty is incurred for bypassing information from instructions in one buffer to instructions in the other buffer.

Up to two instructions per cycle may be launched from each buffer in the IRB. Arbitration in each buffer is handled in two groups. All of the even–numbered slots in the ALU buffer which are ready to launch arbitrate for launch to alu0 and all of the odd slots arbitrate for launch to alu1, and similarly for the memory buffer. In each buffer, the even–numbered slot containing the oldest instruction and the odd–numbered slot containing the oldest instruction win arbitration and are launched to the execution units or the address adders.

3.1. Retirement

Instructions are removed from the IRB in program order after they have successfully executed or their trap status is known. Up to four instructions may be retired per cycle. At retire time, the contents of the rename register associated with a given instruction are committed to the architected registers, and store data is forwarded to the store queue (discussed later). If an instruction needs to signal a trap, the trap parameters are recorded in the architected state and the appropriate trap vector is forwarded to the instruction fetch unit which begins fetching from that address. The fact that instructions are retired in program order and that traps are signalled when an instruction retires enables the PA–8000 to provide a precise exception signalling model.

4. Loads and stores

A frequent cause of pipeline stalls in pipelined in–order machines is that instructions must often wait for the result of preceding load operations. Previous implementations of PA–RISC[4] have implemented stall–on–use and hit–under–miss policies to avoid these penalties in the case of data cache misses. Unfortunately, these techniques are insufficient to avoid large performance penalties when more instructions are executed simultaneously. In fact, load/use penalties can be a serious performance limiter even when loads hit the data cache. As applications demand larger caches to support bigger working sets and as the operating frequency of processors increases, the number of clock cycles required to load data from the data cache increases. This problem is exacerbated in a wide superscalar machine because the distance which must separate a load from the use of its data to avoid a stall is likely to be more than the compiler can accommodate (refer to section 1.1).

Out–of–order execution is obviously a substantial advantage in being able to avoid load/use penalties. Given that the PA–8000 can dynamically schedule instructions over a window of more than fifty instructions, the hardware can look beyond the instructions dependent on a load and find other instructions ready to be executed. This is especially helpful in the case of data cache misses since, if the hardware finds another load or store which misses the data cache, that miss will also be issued on the system bus. Since the two miss transactions are overlapped, the total performance penalty is less than the cost of two sequential data cache misses. The PA–8000 can support up to ten such outstanding data cache misses at one time. This is accomplished without sacrificing a strongly ordered programming model.

When a slot containing a load or store operation determines that the operands required for calculating its effective address are available, it arbitrates for launch to the address adders, just as instructions in the ALU buffer launch to the integer and floating point units. Once the address is calculated, the address is stored in the address reorder buffer (ARB). The effective address is also sent to the TLB, which is dual ported, and the physical page number associated with the effective address is also stored in the ARB. The ARB is twenty–eight slots deep, and each slot of the ARB is associated with a slot of the memory buffer in the IRB.

The ARB is the interface to the dual–ported, single–level off–chip data cache. The two ports of the data cache are connected to separate banks of synchronous SRAMs, one of which contains even–numbered doublewords and the other odd–numbered doublewords. The data cache may be up to four Mbytes in size.

Once an address has been sent to the ARB, if no other instruction is arbitrating for access to the appropriate bank of the data cache, the cache access is immediately launched to the RAMs. In this case, load data arrives back on the chip in time for a dependent instruction to launch on the third cycle after the load launched to calculate its effective address.

In the event that a load cannot immediately access the data cache port it needs, it begins to arbitrate for access on each successive cycle until it wins arbitration. Arbitration is granted based on the age of the originating instruction, not the length of time a load has been in the address reorder buffer. Instructions in the IRB are informed of the status of loads in progress so that instructions waiting for load data do not arbitrate for launch until the load has won access to the data cache. In this way, the execution units continue to work on other, younger instructions which do have all their operands available.

Store instructions merely perform a tag lookup at the time when a load would read the cache. In the event that the store misses the cache, it proceeds to issue its miss to the system bus. Store data is copied from the register file to the store queue at retire time.

The store queue is a structure which can hold up to eleven doublewords of write data for each bank of the data cache. The store queue uses idle cycles, or cycles when other stores are performing tag lookups, to perform its writes to the data cache. By deferring cache writes to otherwise idle cycles, loads are less likely to be held off from accessing the cache due to contention. Another benefit of the store queue is that stores of less than doubleword size may be merged into a single cache write, thus improving cache utilization. Loads may bypass data directly from the store queue.

Store–to–load dependency checking is implemented through address comparisons performed in the ARB. When a store instruction calculates its effective address, all younger load instructions which have completed their access to the cache compare their address against the store address. If the load detects a match, the load and all younger instructions are flushed from the IRB and re–executed. When a load calculates its effective address, all older stores compare their address against the load address and, if they detect a match, the load waits until the store data is available.

Loads and stores to the I/O address space and semaphore instructions do not issue transactions on the system bus until they are the oldest instructions in the IRB so that they do not issue speculatively.

Support for explicit data cache prefetching is implemented in the PA–8000 via loading to general register zero. This operation may cause a data cache miss which will be issued on the system bus, but the instruction will not cause a trap if the access misses the TLB or if the access fails protection checks. (In these cases, the instruction executes as a NOP.) Unlike ordinary loads, a load to general register zero may retire before a data cache miss it has initiated has been returned from memory. The return data will still be written into the data cache. If a subsequent ordinary load is encountered before the data is returned from memory, the ordinary load is informed that the miss has already been issued, and a second miss to the same address is suppressed.

4.1. TLB

The ninety–six entry Translation Lookaside Buffer (TLB) of the PA–8000 is fully dual–ported in order to support two data cache accesses per cycle without requiring these two accesses to be to the same page. Each entry in the TLB may map any power–of–four sized segment of memory from 4 Kbytes to 16 Mbytes. In addition to the main TLB, the instruction fetch unit maintains a buffer of four translations for its use. Whenever the fetch unit misses its set of translations it sends a request to the main TLB to perform a translation on its behalf. This new translation is then inserted into the fetch unit's buffer. Translations for loads and stores take precedence over translations for instruction fetches. A bypass path is provided so that, in the event a mispredicted branch misses the translation buffer in the instruction fetch unit, the translation from the main TLB is available in time to perform the cache tag compare for the first fetch at the new address.

4.2. Multiprocessing support

The PA–8000 supports a snoopy multiprocessor cache coherency protocol. No external logic is required for up to eight–way multiprocessing. Support is also provided for higher–order multiprocessing using a hierarchical bus structure.

Incoming Cache Coherency Checks (CCCs) take just one cycle from one bank of the data cache to perform their snoop of the data cache. This very low cost for CCCs means that, even on a fully saturated system bus, CCCs consume no more than 10% of the available data cache bandwidth if they all miss. Even in a system where the system bus is saturated and every CCC *hits dirty,* the CCCs consume no more than 50% of the victim processor's data cache bandwidth. No other penalty is paid by the victim processor. Instruction fetching is unaffected, the ALUs are unaffected, and the other data cache cycles are fully available.

CCCs use the same address comparison mechanism to implement strong ordering between processors as is used to detect store–to–load dependencies within a single processor. If an incoming CCC matches a load or store in the ARB, that load or store is flushed and re–executed.

5. Execution units

The PA–8000 integer units implement the new 64–bit functionality in PA2.0 while maintaining full compatibility with existing 32–bit binaries. The new 64–bit operations may be executed even by a program executing with 32–bit addressing, so compilers can take advantage of the wider datapath to improve performance even on 32–bit code. Each integer unit includes shift/merge logic so that it can execute any of the extract or deposit instructions as well as the normal arithmetic operations. A branch adder is also associated with each integer unit.

The floating point hardware in the PA–8000 consists of two multiply–and–accumulate (MAC) units and two divide/square root units. The MAC units perform the very common operation D = A*B+C. These units have a latency of three cycles and are fully pipelined so they may accept a new operation every cycle, giving a maximum throughput of four FLOPs per cycle. Multiply operations and add operations are also handled by the MACs.

The divide/square root units have latencies of 17 for single–precision operations and 31 for double–precision operations. These units are not pipelined, but other FLOPs may execute on the MACs while the divide/square root units are busy. The combination of multiple execution units, a dual–ported data cache, support for up to ten pending data cache misses, and explicit data prefetching support provides exceptional floating point performance, even on workloads whose working sets are larger than the data cache.

6. System interface (Runway)

The PA–8000 interfaces directly to the Runway bus, which is the same bus used by the PA–7200[3]. This is a 64–bit multiplexed address/data split transaction bus. The bus supports the full 40–bit physical address space of the PA–8000, allowing access to as much as 960 gigabytes of RAM. Arbitration takes place on separate wires, so it does not consume any bus cycles.

The bus interface logic of the PA–8000 allows up to ten data cache misses, one instruction cache miss, and one instruction cache prefetch to be pending at the same time for the local processor. These transactions may return in any order, allowing improved memory system performance in the presence of bank contention. Instruction cache prefetches are initiated by the bus interface itself by fetching the next sequential line whenever a cache miss is received from the instruction fetch unit.

The bus interface supports cpu:bus frequency ratios of 1:1, 4:3, 3:2, 5:3, 2:1, 7:3, 5:2, and 3:1. That is, the processor uses a clock which is at least as fast as the system bus clock and may be up to three times as fast as the system bus. This flexibility allows the design of a wide range of products combining various processor speeds and bus frequencies to produce the highest possible system performance from the available subsystems.

7. Other performance features

In addition to those features already outlined, the PA–8000 implements a number of new features added to PA–RISC in PA2.0 to improve performance:

- A 22–bit displacement instruction address relative branch to reduce the cost of procedure calls to distant procedures
- A short pointer external branch to reduce the overhead of branching between spaces
- A fast TLB insert mechanism to reduce the cost of TLB misses
- Longer (16–bit) displacement load and store operations
- New variants of the Floating–Point Compare and Floating–Point Test instructions to allow multiple independent conditions to be tested

7.1. Performance monitoring

Perhaps one of the most important performance features of the PA–8000 is the hardware that has been included for performance monitoring and debug support. Hardware has been included that can match specified patterns of instruction opcode, address, cache hit/miss status, and branch mispredictions. These match signals can be combined using an on–chip state machine to detect specific sequences of events. Finally, the state machine can cause one of four event counters to increment. The event counters may also be programmed to increment based on a number of other control signals that indicate what the processor is doing at any given time. This hardware will be used by HP's performance analysis group to identify opportunities for compiler improvements to achieve even higher performance with the PA–8000. It will also be used to evaluate additional features which could be of benefit in future processor designs.

8. Conclusions

Hewlett–Packard's PA–8000 CPU is designed to deliver industry–leading performance on both commercial and technical applications and provide a growth path for future 64–bit applications. It achieves its high performance through a combination of high clock frequency and sustainable superscalar operation. Sustainable superscalar operation is accomplished by matching dual integer, floating point multiply/accumulate, and divide/square root functional units with a very deep instruction reorder buffer, a high performance instruction fetch unit, dual ported data cache and support for multiple pending cache misses.

Acknowledgements

Many people have been involved in the development of the PA–8000 and the author would like to thank all of them for the tremendous effort they have put in to make this processor possible. Special recognition is warranted for those individuals who worked on the project from its earliest days and who set the direction of the design: Gregg LeSartre, Jon Lotz, Don Kipp, Darius Tanksalvala, and Steve Mangelsdorf.

References

[1] E. DeLano, et al, "A High Speed Superscalar PA–RISC Processor", Compcon Digest of Papers, February 1992, pp. 116–121.

[2] P. Knebel, et al, "HP's PA7100LC: A Low–Cost Superscalar PA–RISC Processor", Compcon Digest of Papers, February 1993, pp. 441–447.

[3] G. Kurpanek, et al, "PA7200: A PA–RISC Processor with Integrated High Performance MP Bus Interface", Compcon Digest of Papers, February 1994, pp. 375–382.

[4] R. B. Lee, "Precision Architecture", IEEE Computer, Vol. 22, No. 1, January 1989, pp. 78–91.