INTRODUCING THE IA-64 ARCHITECTURE

ADVANCES IN MICROPROCESSOR DESIGN, INTEGRATED CIRCUITS, AND COMPILER TECHNOLOGY HAVE INCREASED THE INTEREST IN PARALLEL INSTRUCTION EXECUTION. A JOINT HP-INTEL TEAM DESIGNED THE IA-64 PROCESSOR INSTRUCTION SET ARCHITECTURE WITH PARALLELISM IN MIND.

Jerry Huck
Dale Morris
Jonathan Ross
Hewlett-Packard

Allan Knies
Hans Mulder
Rumi Zahir
Intel

Microprocessors continue on the relentless path to provide more performance. Every new innovation in computing—distributed computing on the Internet, data mining, Java programming, and multimedia data streams—requires more cycles and computing power. Even traditional applications such as databases and numerically intensive codes present increasing problem sizes that drive demand for higher performance.

Design innovations, compiler technology, manufacturing process improvements, and integrated circuit advances have been driving exponential performance increases in microprocessors. To continue this growth in the future, Hewlett-Packard and Intel architects examined barriers in contemporary designs and found that instruction-level parallelism (ILP) can be exploited for further performance increases.

This article examines the motivation, operation, and benefits of the major features of IA-64. Intel's IA-64 manual provides a complete specification of the IA-64 architecture.[1]

Background and objectives

IA-64 is the first architecture to bring ILP features to general-purpose microprocessors. Parallel semantics, predication, data speculation, large register files, register rotation, control speculation, hardware exception deferral, register stack engine, wide floating-point exponents, and other features contribute to IA-64's primary objective. That goal is to expose, enhance, and exploit ILP in today's applications to increase processor performance.

ILP pioneers[2,3] developed many of these concepts to find parallelism beyond traditional architectures. Subsequent industry and academic research[4,5] significantly extended earlier concepts. This led to published works that quantified the benefits of these ILP-enhancing features and substantially improved performance.

Starting in 1994, the joint HP-Intel IA-64 architecture team leveraged this prior work and incorporated feedback from compiler and processor design teams to engineer a powerful initial set of features. They also carefully designed the instruction set to be expandable to address new technologies and future workloads.

Architectural basics

A historical problem facing the designers of computer architectures is the difficulty of building in sufficient flexibility to adapt to changing implementation strategies. For example, the number of available instruction bits, the register file size, the number of address space bits, or even how much paral-

0272-1732/00/$10.00 © 2000 IEEE

lelism a future implementation might employ have limited how well architectures can evolve over time.

The Intel-HP architecture team designed IA-64 to permit future expansion by providing sufficient architectural capacity:

- a full 64-bit address space,
- large directly accessible register files,
- enough instruction bits to communicate information from the compiler to the hardware, and
- the ability to express arbitrarily large amounts of ILP.

Figure 1 summarizes the register state; Figure 2 shows the bundle and instruction formats.

Register resources

IA-64 provides 128 65-bit general registers; 64 of these bits specify data or memory addresses and 1 bit holds a deferred exception token or not-a-thing (NaT) bit. The "Control speculation" section provides more details on the NaT bit.

In addition to the general registers, IA-64 contains

- 128 82-bit floating-point registers,
- space for up to 128 64-bit special-purpose application registers (used to support features such as the register stack and software pipelining),
- eight 64-bit branch registers for function call linkage and return, and
- 64 one-bit predicate registers that hold the result of conditional expression evaluation.

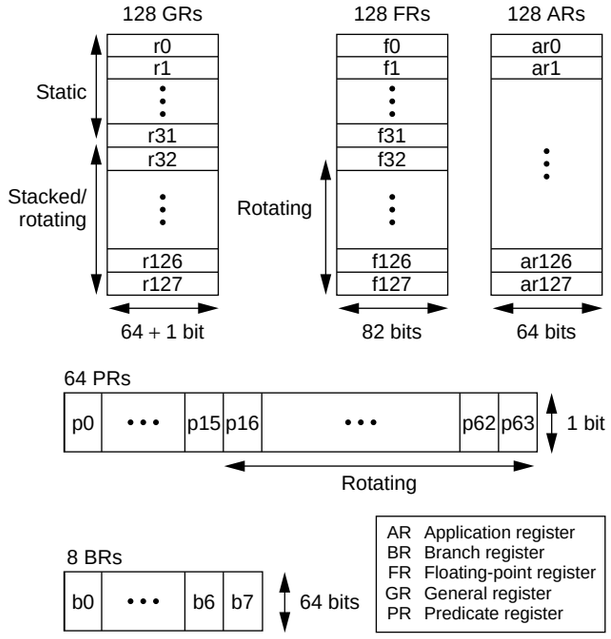


Figure 1. IA-64 application state.

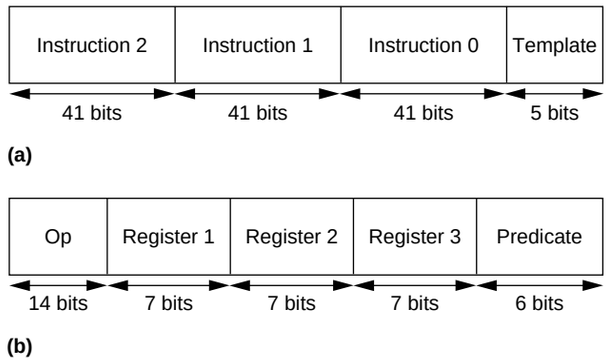


Figure 2. IA-64 bundle (a) and instruction (b) formats.

Instruction encoding

Since IA-64 has 128 general and 128 floating-point registers, instruction encodings use 7 bits to specify each of three register operands. Most instructions also have a predicate register argument that requires another 6 bits. In a normal 32-bit instruction encoding, this would leave only 5 bits to specify the opcode. To provide for sufficient opcode space and to enable flexibility in the encodings, IA-64 uses a 128-bit encoding (called a bundle) that has room for three instructions.

Each of the three instructions has 41 bits with the remaining 5 bits used for the template. The template bits help decode and route instructions and indicate the location of stops that mark the end of groups of instructions that can execute in parallel.

Distributing responsibility

To achieve high performance, most modern microprocessors must determine instruction dependencies, analyze and extract available parallelism, choose where and when to execute instructions, manage all cache and prediction resources, and generally direct all other ongoing activities at runtime. Although intended to reduce the burden on compilers, out-of-order processors still require substantial

```
{ .mii
    add r1 = r2, r3
    sub r4 = r4, r5 ;;
    shr r7 = r4, r12 ;;
}
{ .mmi
    ld8 r2 = [r1] ;;
    st8 [r1] = r23
    tbit p1,p2=r4,5
}
{ .mbb
    ld8 r45 = [r55]
(p3)br.call b1=func1
(p4)br.cond Label1
}
{ .mfi
    st4 [r45]=r6
    fmac f1=f2,f3
    add  r3=r3,8 ;;
}
```

Figure 3. Example instruction groups.

```
if ( (a==0) || (b<=5) ||
   (c!=d) || (f & 0x2) ){
  r3 = 8;
}
```

Figure 4. Compound conditional code.

```
cmp.ne  p1 = r0,r0
add t = -5, b;;

cmp.eq.or p1 = 0,a
cmp.ge.or p1 = 0,t
cmp.ne.or p1 = c,d
tbit.or p1 = 1,f,1 ;;

(p1) mov r3 = 8
```

Figure 5. Example parallel compare. The newly computed predicate is used in the third instruction group, shown here in plain type.

amounts of microarchitecture-specific compiler support to achieve their fastest speeds.

IA-64 strives to make the best trade-offs in dividing responsibility between what the processor must do at runtime and what the compiler can do at compilation time.

ILP

Compilers for all current mainstream microprocessors produce code with the understanding that regardless of how the processor actually executes those instructions, the results will appear to be executed one at a time and in the exact order they were written. We refer to such architectures as having sequential in-order execution semantics, or simply sequential semantics.

Conforming to sequential semantics was easy to achieve when microprocessors executed instructions one at a time and in their program-specified order. However, to achieve acceptable performance improvements, designers have had to design multiple-issue, out-of-order execution processors. The IA-64 instruction set addresses this split between the architecture and its implementations by providing parallel execution semantics so that processors don't need to examine register dependencies to extract parallelism from a serial program specification. Nor do they have to reorder instructions to achieve the shortest code sequence.

IA-64 realizes parallel execution semantics in the form of instruction groups. The compiler creates instruction groups so that all instructions in an instruction group can be safely executed in parallel. While such a grouping may seem like a complex task, current compilers already have all of the information necessary to do this. IA-64 just makes it possible for the compiler to express that parallelism.

The code in Figure 3 shows four instruction groups of various sizes. The gray bars indicate the extent of the instruction groups, which are terminated by double semicolons (;;).

Control flow parallelism

While instruction groups allow independent computational instructions to be placed together, expressing parallelism in computation related to program control flow requires additional support. As an example, many applications execute code that performs complex compound conditionals like the one shown in Figure 4.

In this example, the conditional expression needs to compute the logical Or of four smaller expressions. Normally, such computations can only be done as a sequence of test/branch computations or in a binary tree reduction (if the code meets certain safety requirements). Since such compound conditionals are common, IA-64 provides parallel compares that allow compound And and Or conditions to be computed in parallel.

Figure 5 shows IA-64 assembly code for the C code in Figure 4. Register p1 is initialized to false in the first instruction group; the conditions for each of the Or'd expressions is computed in parallel in the second instruction group; the newly computed predicate is used in the third instruction group, shown here in plain type.

The parallel compare operation (the instructions in boldface) will set p1 to be true if any of the individual conditions are true. Otherwise, the value in p1 remains false.

Control parallelism is also present when a program needs to select one of several possible branch targets, each of which might be controlled by a different conditional expression. Such cases would normally need a sequence of individual conditions and branches. IA-64 provides multiway branches that allow several normal branches to be grouped together and executed in a single instruction group. The example in Figure 6 demonstrates a single multiway branch that either selects one of three possible branch targets or falls through.

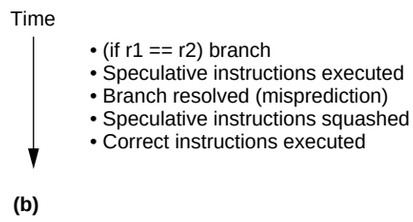
As shown in these examples, the use of parallel compares and multiway branches can

```
{ .mii
       cmp.eq p1,p2 = r1,r2
       cmp.ne p3,p4 = 4, r5
       cmp.lt p5,p6 = r8,r9
}
{ .bbb
(p1) br.cond label1
(p3) br.cond label2
(p5) br.call b4 = label3
}
// fall through code here
```

Figure 6. Multiway branch example.

```
if ( r1 == r2 )
   r9 = r10 - r11;
else
   r5 = r6 + r7;
```

(a)



(b)

Figure 7. Example conditional (a) and conditional branch use (b).

substantially decrease the critical path related to control flow computation and branching.

Influencing dynamic events

While the compiler can handle some activities, hardware better manages many other areas including branch prediction, instruction caching, data caching, and prefetching. For these cases, IA-64 improves on standard instruction sets by providing an extensive set of hints that the compiler uses to tell the hardware about likely branch behavior (taken or not taken, amount to prefetch at branch target) and memory operations (in what level of the memory hierarchy to cache data). The hardware can then manage these resources more effectively, using a combination of compiler-provided information and histories of runtime behavior.

Finding and creating parallelism

IA-64 not only provides new ways of expressing parallelism in compiled code, it also provides an array of tools for compilers to create additional parallelism.

Predication

Branching is a major cause of lost performance in many applications. To help reduce the negative effects of branches, processors use branch prediction so they can continue to execute instructions while the branch direction and target are being resolved. To achieve this, instructions after a branch are executed speculatively until the branch is resolved. Once the branch is resolved, the processor has determined its branch prediction was correct and the speculative instructions are okay to commit, or that those instructions need to be thrown away and the correct set of instructions fetched and executed.

When the prediction is wrong, the processor will have executed instructions along both paths, but sequentially (first the predicted path, then the correct path). Thus, the cost of incorrect prediction is quite expensive. For example, in the code shown in Figure 7a, if the branch at the beginning of the fragment mispredicts, the flow of events at runtime will proceed, as shown in Figure 7b.

To help reduce the effect of branch mispredictions, IA-64 provides predication, a feature that allows the compiler to execute instructions from multiple conditional paths at the same time, and to eliminate the branches that could have caused mispredictions. For example, the compiler can easily detect when there are sufficient processor resources to execute instructions from both sides of an if-then-else clause. Thus, it's possible to execute both sides of some conditionals in the time it would have taken to execute either one of them alone. The following code shows how to generate code for our example in Figure 7a:

```
      cmp.eq p1, p2 = r1, r2;;
(p1) sub r9 = r10, r11
(p2) add r5 = r6, r7
```

The cmp (compare) generates two predicates that are set to one or zero, based on the result of the comparison (p1 will be set to the opposite of p2). Once these predicates are generated, they can be used to guard execution: the add instruction will only execute if p2 has a true value, and the sub instruction will only

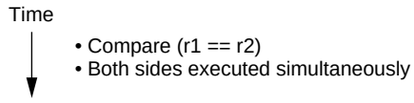


Figure 8. Using predication.

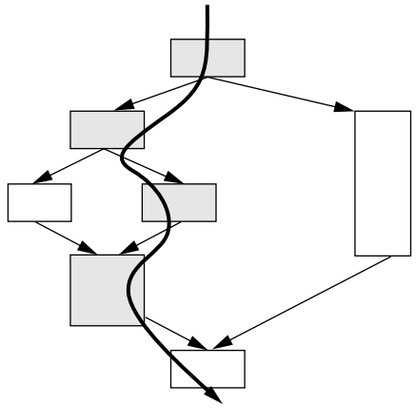


Figure 9. Control path through a function.

execute if p1 has a true value.

Figure 8 shows the time flow for the code with predication after the branch has been removed. By using predication to simplify control flow, the compiler exposes a larger pool of instructions in which to find parallel work. Although some of these instructions will be cancelled during execution, the added parallelism allows the sequence to execute in fewer cycles.[6]

In general, predication is performed in IA-64 by evaluating conditional expressions with compare (cmp) operations and saving the resulting true (1) or false (0) values in a special set of 1-bit predicate registers. Nearly all instructions can be predicated. This simple, clean concept provides a very powerful way to increase the ability of an IA-64 processor to exploit parallelism, reduce the performance penalties of branches (by removing them), and support advanced code motion that would be difficult or impossible in instruction sets without predication.

Scheduling and speculation

Compilers attempt to increase parallelism by scheduling instructions based on predictions about likely control paths. Paths are made of sequences of instructions that are grouped into basic blocks. Basic blocks are groups of instructions with a single entry point and single exit point. The exit point can be a multiway branch.

If a particular sequence of basic blocks is likely to be in the flow of control, the compiler can consider the instructions in these blocks as a single group for the purpose of scheduling code. Figure 9 illustrates a program fragment with multiple basic blocks, and possible control paths. The highlighted blocks indicate those most likely to be executed.

Since these regions of blocks have more instructions than individual basic blocks, there is a greater opportunity to find parallel work. However, to exploit this parallelism, compilers must move instructions past barriers related to control flow and data flow. Instructions that are scheduled before it is known whether their results will be used are called speculative.

Of the code written by a programmer, only a small percentage is actually executed at runtime. The task of choosing important instructions, determining their dependencies, and specifying which instructions should be executed together is algorithmically complex and time-consuming. In non-EPIC architectures, the processor does much of this work at runtime. However, a compiler can perform these tasks more efficiently because it has more time, memory, and a larger view of the program than the hardware.

The compiler will optimize the execution time of the commonly executed blocks by choosing the instructions that are most critical to the execution time of the critical region as a whole. Within these regions, the compiler performs instruction selection, prioritization, and reordering.

Without the IA-64 features, these kinds of transformations would be difficult or impossible for a compiler to perform. The key features enabling these transformations are control speculation, data speculation, and predication.

Control speculation

IA-64 can reduce the dynamic effects of branches by removing them; however, not all branches can or should be removed using predication. Those that remain affect both the processor at runtime and the compiler during compilation.

Since loads have a longer latency than most computational instructions and they tend to start time-critical chains of instructions, any constraints placed on the compiler's ability to perform code motion on loads can limit the exploitation of parallelism. One such constraint relates to properly handling exceptions. For example, load instructions may attempt to reference data to which the program hasn't been granted access. When a program makes such an illegal access, it usually must be terminated. Additionally, all exceptions must also be delivered as though the program were executed in the order the programmer wrote it. Since moving a load past a branch changes the sequence of memory accesses relative to the control flow of the program, non-EPIC archi-

IA-64 virtual memory model

Virtual memory is the core of an operating system's multitasking and protection mechanisms. Compared to 32-bit virtual memory, management of 64-bit address spaces requires new mechanisms primarily because of the increase in address space size: 32 bits can map 4 Gbytes, while 64 bits can map 16 billion Gbytes of virtual space.

A linear 32-bit page table requires 1 million page table entries (assuming a 4-Kbyte page size), and can reside in physical memory. A linear 64-bit page table would be 4 billion times larger—too big to be physically mapped in its entirety. Additionally, 64-bit applications are likely to populate the virtual address space more sparsely. Due to larger data structures than those in 32-bit applications, these applications may have a larger footprint in physical memory.

All of these effects result in more pressure on the processor's address translation structure: the translation look-aside buffer. While growing the size of on-chip TLBs helps, IA-64 provides several architectural mechanisms that allow operating systems to significantly increase the use of available capacity:

- Regions and protection keys enable much higher degrees of TLB entry sharing.
- Multiple page sizes reduce TLB pressure. IA-64 supports 4-Kbyte to 256-Mbyte pages.
- TLB entries are tagged with address space identifiers (called region IDs) to avoid TLB flushing on context switch.

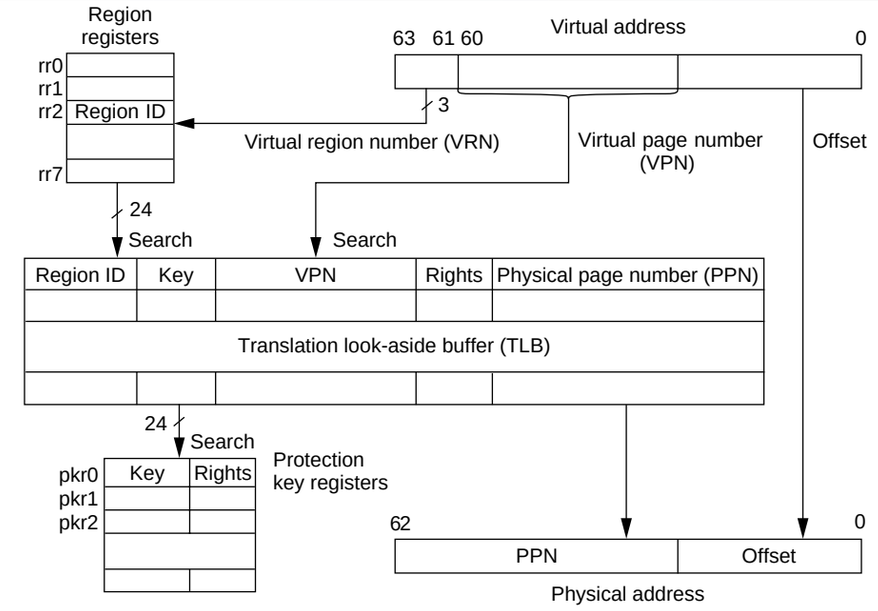


Figure A. Address translation.

Regions

As shown in Figure A, bits 63 to 61 of a virtual address index into eight region registers that contain 24-bit region identifiers (RIDs). The 24-bit RID is concatenated with the virtual page number (VPN) to form a unique lookup into the TLB. The TLB lookup generates two main items: the physical page number and access privileges (keys, access rights, and access bits among others). The region registers allow the operating system to concurrently map 8 out of 2^{24} possible address spaces, each 2^{61} bytes in size. The operating system uses the RID to distinguish shared and private address spaces. Typically, operating systems assign specific regions to specific uses. For example, region 0 may be used for user private application data, region 1 for shared libraries and text images, region 2 for mapping of shared files, and region 7 for mapping of the operating system kernel itself.

On context switch, instead of invalidating the entire TLB, the operating system only rewrites the user's private region registers with the RID of the switched-to process. Shared-region's RIDs remain in place, and the same TLB entries can be shared between different processes, such as shared code or data.

Protection keys

While RIDs provide efficient sharing of region-size objects, software often is interested in sharing objects at a smaller granularity such as in object databases or operating system message queues. IA-64 protection key registers (PKRs) provide page-granular control over access while continuing to share TLB entries among multiple processes.

As shown in Figure A, each TLB entry contains a protection key field that is inserted into the TLB when creating that translation. When a memory reference hits in the TLB, the processor looks up the matching entry's key in the PKR register file. A key match results in additional access rights being consulted to grant or deny the memory reference. If the lookup fails, hardware generates a key miss fault.

The software key miss handler can now manage the PKR contents as a cache of most recently used protection keys on a per-process basis. This allows processes with different permission levels to access shared data structures and use the same TLB entry. Direct address sharing is very useful for multiple process computations that communicate through shared data structures; one example is producer-consumer multithreaded applications.

The IA-64 region model provides protection and sharing at a large granularity. Protection keys are orthogonal to regions and allow fine-grain page-level sharing. In both cases, TLB entries and page tables for shared objects can be shared, without requiring unnecessary duplication of page tables and TLB entries in the form of virtual aliasing.

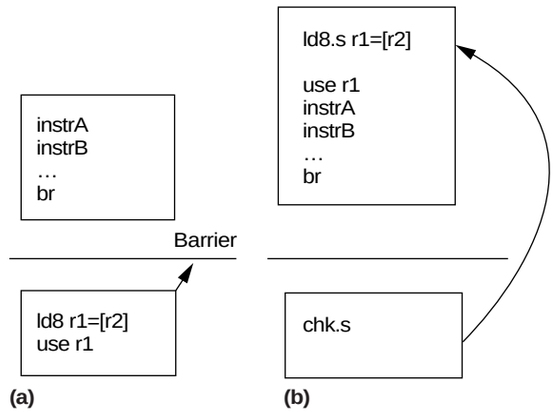


Figure 10. Comparing the scheduling of control speculative computations in traditional (a) and IA-64 (b) architectures. IA-64 allows elevation of loads and their uses above branches.

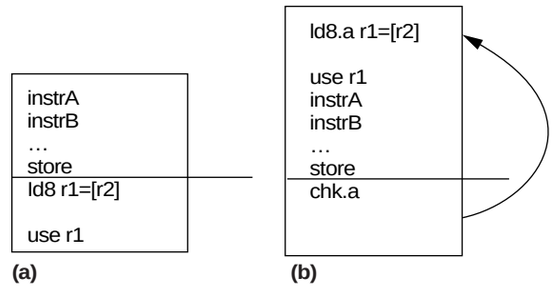


Figure 11. Data speculation example in traditional (a) and IA-64 (b) architectures. IA-64 allows elevation of load and use even above a store.

tectures constrain such code motion.

IA-64 provides a new class of load instructions called speculative loads, which can safely be scheduled before one or more prior branches. In the block where the programmer originally placed the load, the compiler schedules a speculation check (chk.s), as shown in Figure 10. In IA-64, this process is referred to as control speculation. While the example shown is very simple, this type of code motion can be very useful in reducing the execution time of more complex tasks such as searching down a linked list while simultaneously checking for NULL pointers.

At runtime, if a speculative load results in an exception, the exception is deferred, and a deferred exception token (a NaT) is written to the target register. The chk.s instruction checks the target register for a NaT, and if present, branches to special "fix-up" code (which the compiler also generates). If needed, the fix-up code will reexecute the load nonspeculatively, and then branch back to the main program body.

Since almost all instructions in IA-64 will propagate NaTs during execution (rather than raising faults), entire calculation chains may be scheduled speculatively. For example, when one of the operand registers to an add instruction contains a NaT, the add doesn't raise a fault. Rather, it simply writes a NaT to its target register, thus propagating the deferred exception. If the results of two or more speculative loads are eventually used in a common computation, NaT propagation allows the compiler to only insert a single chk.s to check the result of multiple speculative computations.

In the event that a chk.s detects a deferred exception on the result of this calculation chain, the fix-up code simply reexecutes the entire chain, this time resolving exceptions as they're discovered. This mechanism is termed control speculation because it permits the expression of parallelism across a program's control flow. Although the hardware required to support control speculation is simple, this mechanism lets the compiler expose large amounts of parallelism to the hardware to increase performance.

Data speculation

Popular programming languages such as C provide pointer data types for accessing memory. However, pointers often make it impossible for the compiler to determine what location in memory is being referenced. More specifically, such references can prevent the compiler from knowing whether a store and a subsequent load reference the same memory location, preventing the compiler from reordering the instructions.

IA-64 solves this problem with instructions that allow the compiler to schedule a load before one or more prior stores, even when the compiler is not sure if the references overlap. This is called data speculation; its basic usage model is analogous to control speculation.

When the compiler needs to schedule a load ahead of an earlier store, it uses an advanced load (ld.a), then schedules an advanced load check instruction (chk.a) after all the intervening store operations. See the example in Figure 11.

An advanced load works much the same as

a traditional load. However, at runtime it also records information such as the target register, memory address accessed, and access size in the advanced load address table. The ALAT is a cachelike hardware structure with content-addressable memory. Figure 12 shows the structure of the ALAT.

When the store is executed, the hardware compares the store address to all ALAT entries and clears entries with addresses that overlap with the store. Later, when the chk.a is executed, hardware checks the ALAT for the entry installed by its corresponding advanced load. If an entry is found, the speculation has succeeded and chk.a does nothing. If no entry is found, there may have been a collision, and the check instruction branches to fix-up code to reexecute the code (just as was done with control speculation).

Because the fix-up mechanism is general, the compiler can speculate not only the load but also an entire chain of calculations ahead of any number of possibly conflicting stores.

Compared to other structures such as caches, the chip area and effort required to implement the ALAT are smaller and simpler than equivalent structures needed in out-of-order processors. Yet, this feature enables the compiler to aggressively rearrange compiled code to exploit parallelism.

Register model

Most architectures provide a relatively small set of compiler-visible registers (usually 32). However, the need for higher performance has caused chip designers to create larger sets of physical registers (typically around 100), which the processor then manages dynamically even though the compiler only views a subset of those registers.

The IA-64 general-register file provides 128 registers visible to the compiler. This approach is more efficient than a hardware-managed register file because a compiler can tell when the program no longer needs the contents of a specific register. These general registers are partitioned into two subsets: 32 static and 96 stacked, which can be renamed under software control. The 32 static registers (r0 to r31) are managed in much the same way as registers in a standard RISC architecture.

The stacked registers implement the IA-64 register stack. This mechanism automatically provides a compiler with a set of up to 96 fresh registers (r32 to r127) upon procedure entry. While the register stack provides the compiler with the illusion of unlimited register space across procedure calls, the hardware actually saves and restores on-chip physical registers to and from memory.

By explicitly managing registers using the register allocation instruction (alloc), the compiler controls the way the physical register space is used. Figure 13's example shows a register stack configured to have eight local registers and three output registers.

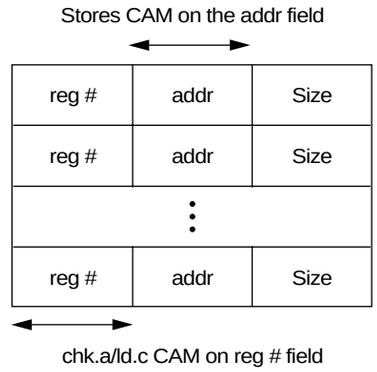


Figure 12. ALAT organization.

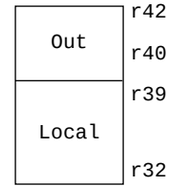


Figure 13. Initial register stack frame after using the alloc instruction: eight local and three output.

The compiler specifies the number of registers that a routine requires by using the alloc instruction. Alloc can also specify how many of these registers are local (which are used for computation within the procedure), and how many are output (which are used to pass parameters when this procedure calls another). The stacked registers in a procedure always start at r32.

On a call, the registers are renamed such that the local registers from the previous stack frame are hidden, and what were the output registers of the calling routine now have register numbers starting at r32 in the called routine. The freshly called procedure would then perform its own alloc, setting up its own local registers (which include the parameter registers it was called with), and its own output registers (for when it, in turn, makes a call). Figure 14a (next page) shows this process.

On a return, this renaming is reversed, and the stack frame of the calling procedure is restored (see Figure 14b).

The register stack really only has a finite number of registers. When procedures request more registers than are currently available, an automatic register stack engine (RSE) stores registers of preceding procedures into memory in parallel with the execution of the called procedure. Similarly, on return from a call, the RSE can restore registers from memory.

As described here, RSE behavior is synchronous; however, IA-64 allows processors to be built with asynchronous RSEs that can

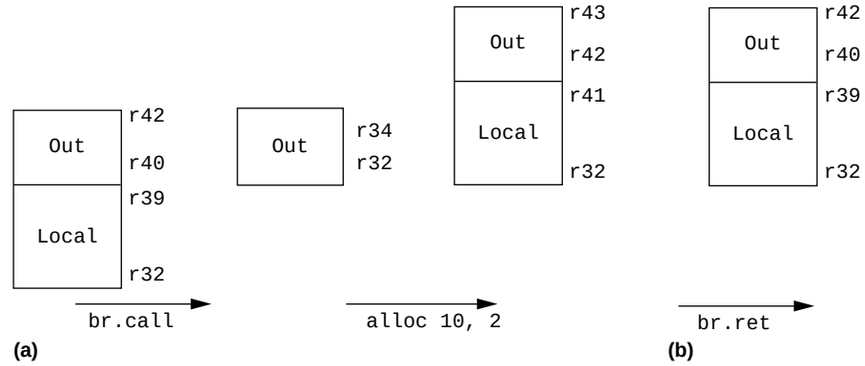


Figure 14. Procedure call (a) and return (b).

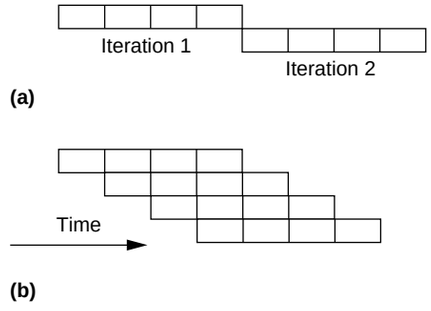


Figure 15. Sequential (a) versus pipelined (b) execution.

speculatively spill and fill registers in the background while the processor core continues normal execution. This allows spills and fills to be performed on otherwise unused memory ports before the spills and fills are actually needed.

Compared to conventional architectures, IA-64's register stack removes all the save and restore instructions, reduces data traffic during a call or return, and shortens the critical path around calls and returns. In one simulation performed on a PA-RISC-hosted database code, adding RSE functionality to PA-RISC removed 30% of the loads and stores, while consuming only 5% of the execution ports dynamically.

Software pipelining

Computers are very good at performing iterative tasks, and for this reason many programs include loop constructs that repeatedly perform the same operations. Since these loops generally encompass a large portion of a program's execution time, it's important to expose as much loop-level parallelism as possible.

Although instructions in a loop are executed frequently, they may not offer a sufficient degree of parallel work to take advantage of all of a processor's execution resources. Conceptually, overlapping one loop iteration with the next can often increase the parallelism, as shown in Figure 15. This is called software pipelining, since a given loop iteration is started before the previous iteration has finished. It's analogous to the way hardware pipelining works.

While this approach sounds simple, without sufficient architectural support a number of issues limit the effectiveness of software pipelining because they require many additional instructions:

- managing the loop count,
- handling the renaming of registers for the pipeline,
- finishing the work in progress when the loop ends,
- starting the pipeline when the loop is entered, and
- unrolling to expose cross-iteration parallelism.

In some cases this overhead could increase code size by as much as 10 times the original loop code. Because of this, software pipelining is typically only used in special technical computing applications in which loop counts are large and the overheads can be amortized.

With IA-64, most of the overhead associated with software pipelining can be eliminated. Special application registers to maintain the loop count (LC) and the pipeline length for draining the software pipeline (the epilog count, or EC) help reduce overhead from loop counting and testing for loop termination in the body of the loop.

In conjunction with the loop registers, special loop-type branches perform several activities depending on the type of branch (see Figure 16). They

- automatically decrement the loop counters after each iteration,
- test the loop count values to determine if the loop should continue, and
- cause the subset of the general, floating, and predicate registers to be automatically renamed after each iteration by decrementing a register rename base (rrb) register.

For each rotation, all the rotating registers appear to move up one higher register position, with the last rotating register wrapping back around to the bottom. Each rotation effectively advances the software pipeline by one stage.

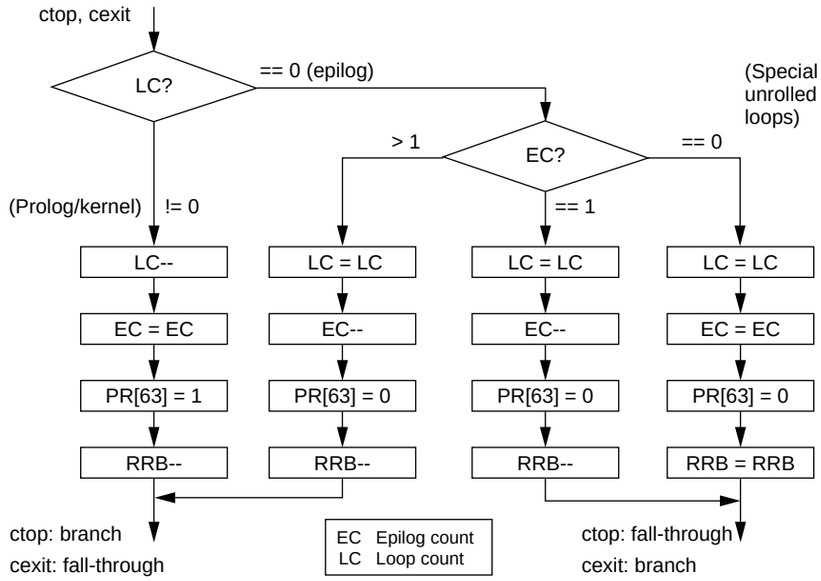


Figure 16. Loop-type branch behavior.

The set of general registers that rotate are programmable using the alloc instruction. The set of predicate (p16 to p63) and floating (f32 to f127) registers that rotate is fixed. Instructions br.ctop and br.cexit provide support for counted loops (similar instructions exist to support pipelining of while-type loops).

The rotating predicates are important because they serve as pipeline stage valid bits, allowing the hardware to automatically drain the software pipeline by turning instructions on or off depending on whether the pipeline is starting up, executing, or draining. Mahlke et al. provide some highly optimized specific examples of how software pipelining and rotating registers can be used.[7]

The combination of these loop features and predication enables the compiler to generate compact code, which performs the essential work of the loop in a highly parallel form. All of this can be done with the same amount of code as would be needed for a non-software-pipelined loop. Since there is little or no code expansion required to software-pipeline loops in IA-64, the compiler can use software pipelining much more aggressively as a general loop optimization, providing increased parallelism for a broad set of applications.

Although out-of-order hardware approaches can approximate a software-pipelined approach, they require much more complex hardware, and do not deal as well with problems such as recurrence (where one loop iteration creates a value consumed by a later loop iteration). Full examples of software-pipelined loops are provided elsewhere in this issue.[8]

Summary of parallelism features

These parallelism tools work in a synergistic fashion, each supporting the other. For example, program loops may contain loads and stores through pointers. Data speculation allows the compiler to use the software-pipelining mechanism to fully overlap the execution, even when the loop uses pointers that may be aliased. Also, scheduling a load early often requires scheduling it out of its basic block and ahead of an earlier store. Speculative advanced loads allow both control and data speculation mechanisms to be used at once. This increased ILP keeps parallel hardware functional units busier, executing a program's critical path in less time.

While designers and architects have a model for how IA-64 features will be implemented and used, we anticipate new ways to use the IA-64 architecture as software and hardware designs mature. Each day brings discoveries of new code-generation

IA-64 floating-point architecture

The IA-64 FP architecture is a unique combination of features targeted at graphical and scientific applications. It supports both high computation throughput and high-precision formats. The inclusion of integer and logical operations allows extra flexibility to manipulate FP numbers and use the FP functional units for complex integer operations.

The primary computation workhorse of the FP architecture is the FMAC instruction, which computes A * B + C with a single rounding. Traditional FP add and subtract operations are variants of this general instruction. Divide and square root is supported using a sequence of FMAC instructions that produce correctly rounded results. Using primitives for divide and square root simplifies the hardware and allows overlapping with other operations. For example, a group of divides can be software pipelined to provide much higher throughput than a dedicated nonpipelined divider.

The XMA instruction computes A * B + C with the FP registers interpreted as 64-bit integers. This reuses the FP functional units for integer computation. XMA greatly accelerates the wide integer computations common to cryptography and computer security. Logical and field manipulation instructions are also included to simplify math libraries and special-case handling.

The large 128-element FP register file allows very fast access to a large number of FP (or sometimes integer) variables. Each register is 82-bits wide, which extends a double-extended format with two additional exponent bits. These extra-exponent bits enable simpler math library routines that avoid special-case testing. A register's contents can be treated as a single (32-bit), double (64-bit), or double-extended (80-bit) formatted floating-point number that complies with the IEEE/ANSI 754 standard. Additionally, a pair of single-precision numbers can be packed into an FP register. Most FP operations can operate on these packed pairs to double the operation rate of single-precision computation. This feature is especially useful for graphics applications in which graphic transforms are nearly doubled in performance over a traditional approach.

All of the parallel features of IA-64—predication, speculation, and register rotation—are available to FP instructions. Their capabilities are especially valuable in loops. For example, regular data access patterns, such as recurrences, are very efficient with rotation. The needed value can be retained for as many iterations as necessary without traditional copy operations. Also, If statements in the middle of software-pipelined loops are simply handled with predication.

To improve the exposed parallelism in FP programs, the IEEE standard-mandated flags can be maintained in any of four different status fields. The flag values are later committed with an instruction similar to the speculative check. This allows full conformance to the standard without loss of parallelism and performance.

techniques and new approaches to old algorithms. These discoveries are validating that ILP does exist in programs, and the more you look, the more you find.

ILP is one level of parallelism that IA-64 is exploiting, but we continue to pursue other sources of parallelism through on-chip and multichip multiprocessing approaches. To achieve best performance, it is always best to start with the highest performance uniprocessor, then combine those processors into multiprocessor systems.

In the future, as software and hardware technologies evolve, and as the size and computation demands of workloads continue to grow, ILP features will be vital to allow processors' continued increases in performance and scalability. The Intel-HP architecture team designed the IA-64 from the ground up to be ready for these changes and to provide excellent performance over a wide range of applications. MICRO

References

1. *Intel IA-64 Architecture Software Developer's Manual,* Vols. I-IV, Rev. 1.1, Intel Corp., July 2000; http://developer.intel.com.
2. R.P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Computers,* Aug. 1988, pp. 967-979.
3. B.R. Rau et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs," *Computer,* Jan. 1989, pp. 12-35.
4. S.A. Mahlke et al., "Sentinel Scheduling for Superscalar and VLIW Processors," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, Oct. 1992, pp. 238-247.
5. D.M. Gallagher et al., "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, Oct. 1994, pp. 183-193.
6. J. Worley et al., "AES Finalists on PA-RISC and IA-64: Implementations & Performance," *Proc. The Third Advanced Encryption Standard Candidate Conf.,* NIST, Washington, D.C., Apr. 2000, pp. 57-74.
7. S.A. Mahlke et al., "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," *Proc. 22nd Int'l Symp. Computer Architecture,* IEEE Computer Society Press, Los Alamitos, Calif., June 1995, pp. 138-150.
8. J. Bharadwaj et al., "The Intel IA-64 Compiler Code Generator," Special Issue: Microprocessors of the 21st Century, Part 2, Intel IA-64 Architecture, *IEEE Micro,* this issue.

Jerry Huck is a chief architect in HP's computer systems organization. He was responsible for managing the HP side of the joint HP-Intel IA-64 architecture development group. His team continues to evolve the PA-RISC and IA-64 architecture definition, while

Jerry currently works on new application development environments for HP's e-services initiatives. His interests include instruction set design, virtual memory architecture, code optimization, and floating-point architecture. Huck received a PhD degree from Stanford University.

Dale Morris is currently HP's chief IA-64 processor architect. As a technical contributor at Hewlett-Packard, he has worked on enterprise-class computer system design, development, and extension of the PA-RISC architecture, and the development and realization of the IA-64 architecture. Morris earned BSEE and MSEE degrees from the University of Missouri-Columbia.

Jonathan Ross is a senior scientist/software at Hewlett-Packard, where he was the HP-UX kernel development team technical lead. His first responsibilities were implementing and improving multiprocessor capabilities of HP-UX/PA-RISC kernel virtual memory, process management, and I/O services. He worked as a processor architect for IA-64 privileged-mode definition and other instruction set definition tasks. Presently, he is working on tools for IA-64 system performance characterization. Ross obtained a bachelor's degree in electrical engineering/computers from Stanford University.

Allan Knies is a senior computer architect on Intel's IA-64 architecture team, responsible for the IA-64 application architecture. He currently leads a small development team that concentrates on understanding and improving IA-64 application performance through architecture, microarchitecture, and compiler technology improvements. Knies received a BS degree in computer science and mathematics from Ohio University and MS and PhD degrees from Purdue University.

Hans Mulder is a principal engineer in the Enterprise Processor Division of Intel. He is responsible for the EPIC-technology-based architecture enhancements in IA-64 and the performance projections and design support related to all Intel IA-64 microprocessors under development. Earlier, he was an architect at Intel in the 64-bit program and held an academic position at Delft University in the Netherlands. Mulder holds a PhD degree in electrical engineering from Stanford University.

Rumi Zahir is a principal engineer at Intel Corporation. He is one of Intel's IA-64 instruction set architects, one of the main authors of the *IA-64 Architecture Software Developer's Manual,* and has worked on the Itanium processor, Intel's first IA-64 CPU. Apart from processor design, his professional interests also include computer systems design, operating system technology, and dynamic code optimization techniques. Zahir received a PhD degree in electrical engineering from ETH Zurich in Switzerland.

Direct comments about this article to Allan Knies, Intel Corp., M/S SC12-304, 2200 Mission College Blvd., Santa Clara, CA 95052; allan.knies@intel.com.