# Demystifying EPIC and IA-64

## *EPIC Is a Natural Evolution of RISC, Making It Easy to Retrofit Onto RISC*

*by Peter Song*

Using a next-generation architecture technology that Intel and Hewlett-Packard call EPIC (explicitly parallel instruction computing), Merced and future EPIC processors threaten the performance lead held today by RISC processors. EPIC is not entirely new, borrowing many of its ideas from previous RISC and VLIW designs as well as from recent academic research. EPIC has an inherent performance advantage over existing architectures, however, because it is a synergistic assembly of the latest innovations into one architecture. To compete with EPIC processors from Intel, existing RISC architectures are likely to adopt a similar combination of EPIC features in their future versions.

During last year's Microprocessor Forum, Intel and HP gave a high-level, incomplete description of IA-64, for which the companies coined the generic name EPIC (see MPR 10/27/97, p. 1). Nevertheless, we know that EPIC provides a large number of addressable registers, eliminating the need for register renaming and reducing cache accesses. It also provides instruction dependency hints, simplifying instruction issue logic. EPIC uses predicated execution to eliminate some branches, thereby increasing scheduling freedom for the compiler, allowing parallel execution of both paths of branches, and reducing opportunities for misprediction. EPIC uses speculative loads to enable well-behaved accesses to memory as soon as the address can be computed, hiding memory latency.

Intel and HP have revealed only a few details of EPIC and IA-64, but we can project more details than publicly disclosed by considering how these EPIC features can be applied to solve today's performance bottlenecks. IA-64 may impose programming restrictions to accommodate clustering of execution units and registers, greatly simplifying hardware without unduly degrading the processor's throughput. It may also use delayed branches to specify branch target addresses as early as possible, reducing reliance on accurate branch prediction. IA-64 may use load/store instructions that also return the effective address as a result, reducing the overhead of hoisting speculative loads above earlier stores.

At first glance, retrofitting these EPIC features onto an existing instruction set seems to require adding more bits, breaking binary compatibility with existing software. While a few new instructions can be added easily to an instruction set using unused opcodes, adding general-purpose registers and predicated execution seems more difficult, or even impossible, without breaking binary compatibility. For many RISC architectures, however, most—if not all—of the known EPIC features can be added without breaking compatibility. EPIC is a natural evolution of RISC: its fixed-length instruction formats and load/store instructions enable the EPIC features to be added easily.

### IA-64 Likely to Embrace Clustered Designs

IA-64 has 128 integer and 128 floating-point registers, four times as many registers as a typical RISC architecture, allowing the compiler to expose and express an increased amount of ILP (instruction-level parallelism). Merced and future IA-64 processors are expected to have more execution units than today's high-performance processors, taking advantage of the heightened ILP to deliver better performance.

While additional registers and execution units can improve a processor's throughput, they generally degrade the processor's cycle time, since a crossbar is needed between the registers and the execution units in most general-purpose processors. The crossbar enables the execution units to access any register without interfering with each other and is built into the register file. High-performance designs generally use another crossbar for forwarding results from one execution unit to all units that may need the results, saving one or more cycles required for writing the results to the register file and then reading them.

Adding registers or execution units increases the number of switches and wires in the crossbars, as well as the wire lengths and the capacitive loading, resulting in longer delays through the crossbars. Extra metal layers do not reduce a crossbar's size or its propagation delays, since the switches are built using transistors. Because wire delays take

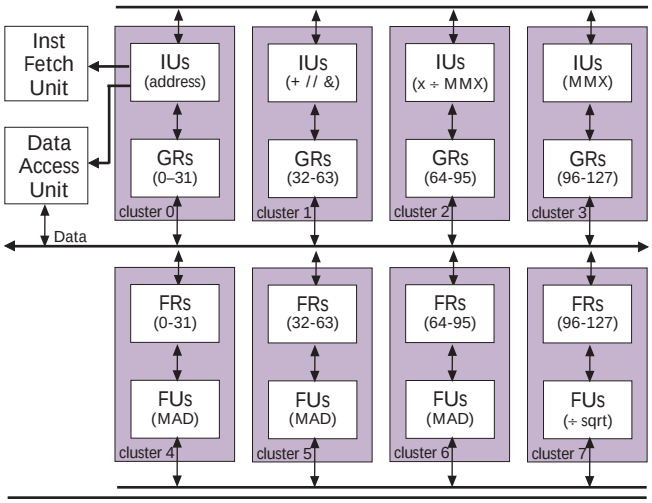


**Figure 1.** IA-64 processors may group registers and function units into execution clusters, allowing implementations to use smaller crossbars and fewer global wires.

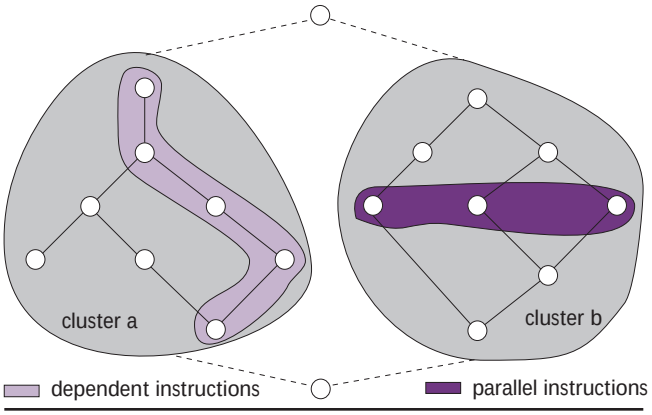


**Figure 2.** A clustered-execution model would map dependent instructions to the same cluster and allow instructions in different clusters, as well as within in each cluster, to execute in parallel.

an increasingly large fraction of cycle times as process geometries shrink—a trend that is unlikely to reverse in the foreseeable future—we expect new architectures, including IA-64, to adopt features that require smaller crossbars and fewer global wires.

IA-64 is likely to embrace partitioning the processor core—registers and execution units—into clusters at the architectural level, reducing the burden of connecting the plethora of registers and execution units. For example, it could partition the 128 registers into four 32-register banks and restrict most instructions to accessing registers from only one bank.

Such a restriction would allow the processor core to be built in clusters, as Figure 1 shows, each consisting of a bank of registers and a set of function units. Since the crossbars in each cluster connect fewer registers and function units—resulting in fewer register-file ports and result-forwarding paths—they are smaller and have shorter propagation delays than a crossbar connecting all registers and all function units. Using smaller crossbars, the processor core can operate at a higher clock speed without taking an extra cycle for the function units to forward their results to each other. There may be paths between the clusters for copying registers from one bank to another, using explicit move instructions. Each path would add a write port to each register and possibly a result-forwarding path to each execution unit.

Digital's 21264 (see MPR 10/28/96, p. 11) uses a clustered design in which each pair of integer and address-generation units has its own copy of the integer registers, reducing the number of read ports from eight to four. Each cluster must maintain a coherent copy of the entire register file, since the Alpha architecture does not restrict register usage for the instructions. Therefore, the results generated in one cluster are sent transparently to the other cluster, resulting the same number of write ports and result-forwarding paths as in a nonclustered design. Since forwarding results to the other cluster incurs an extra cycle, dependent instructions are generally issued to same cluster.

## Clusters Expose Parallelism, Avoid VLIW Flaw

Clearly, the programming model for a clustered design using multiple register banks is not as general or uniform as a nonclustered design. It would be overly restrictive if each bank were to have only a few registers—eight for example—requiring extra instructions to save and restore the registers. Using 32 registers in a bank, however, the programming model could remain general and uniform for most applications. In fact, multiple clusters would present a parallel execution model, since each cluster is free to execute most instructions in parallel—and even out of order—with respect to instructions in the other clusters. Only instructions that transfer registers between the clusters would need to be executed in program order.

Figure 2 shows a flow graph of a hypothetical group of instructions mapped onto two clusters. The instructions mapped to different clusters have no dependency between them and are free to execute in parallel. Many instructions within a cluster also do not have dependency between them and can also be executed in parallel. The two chains of dependent instructions mapped to two clusters may start from or end at the same instruction, as indicated by the dotted lines. In general, mapping chains of dependent instructions onto different clusters would depend on several factors, including the length of the dependency chains.

Execution clusters can be—and should be—defined to be independent of implementation. The multiple-bank partition discussed previously assigns each instruction to a cluster, requiring the instruction to execute in the assigned cluster. It does not map each instruction to an execution unit, however, avoiding the VLIW flaw that breaks binary compatibility between different implementations. It allows different designs to have different combinations of execution units in each cluster. It does not prevent an implementation from merging multiple clusters into one and executing the register-move instructions as NOPs.

Dynamic frequency of instructions also provides a basis for partitioning. For example, divide instructions may be architecturally restricted to only one cluster, due to their infrequent use in most applications. A low-cost design could provide a simple divider in the designated cluster, saving hardware. A design whose target applications make heavy use of divide, however, could provide several fast divide units in the cluster, delivering the necessary divide performance. Multimedia instructions (i.e., MMX)also typically require some special hardware. For simplicity, these could be restricted to one or two clusters.

IA-64 may designate one bank of integer registers as address registers, intended for instruction and data address computations. Load/store instructions could use only the address registers for address operands but any addressable register for the data operand. A bank of address registers allows a single cluster to process all memory accesses more efficiently than a group of integer clusters does, as in the 21264 design.

Checking address dependency and enforcing a desired memory-access order is simpler when all addresses are computed and translated in one place. IA-64 may provide multiple memory-access models, as do the later RISC architectures. In a simple design, the data addresses can be computed and translated in program order, simplifying exception and address-dependency checks. In a weakly-ordered memory model, the accesses that do not have an exception or dependency with earlier accesses could then be allowed to reference memory out of order. Since most exceptions related to load/store instructions are detected by the time the addresses are translated, allowing out-of-order memory access would not unduly complicate the precise-exception mechanism but would improve performance compared with enforcing in-order memory access.

## Template Provides Inter-Instruction Information

According to Intel and HP, the IA-64 instruction format will convey dependency hints using a "template," which will somehow identify instructions that have no dependencies among them and therefore can execute in parallel. The template is expected to specify inter-instruction information, which is readily available in the compiler but is difficult to reconstruct at run time.

The template could use one bit per instruction to indicate if the instruction is the last in a group of instructions that can execute in parallel. It may also specify cluster assignments for the instructions, identifying chains of dependent instructions that should be executed in the same cluster. In this way, it would simplify instruction-issue logic without binding each instruction to a specific execution unit. Each group may have any number of instructions assigned to any number of clusters.

Assuming IA-64 uses eight clusters—four floating-point and four integer, with some clusters providing multimedia and address-computation functions—the template could use three bits per instruction for cluster assignment. It could also use only two bits, since the opcode indicates a floating-point or an integer instruction, but that would also require decoding the opcode. The cluster-assignment information allows most instructions to use 5-bit register numbers, not 7-bit, since it identifies one of the eight register banks. A few exceptions may be the load/store (only for the data operand) and move instructions, which must specify any register. Five-bit register numbers also make four-operand instructions, such as multiply-add, easier to specify.

## Predication Reduces Branches, Misprediction

Predicated execution—also known as conditional or guarded execution—separates execution of an instruction from the decision to commit the result to architectural registers. The prototypical example of predicated execution is the conditional branch, which computes the branch-target address but updates the program counter only if the condition is true. Another example recently added to most architectures is conditional move, which copies the contents of one register to another only if the condition is true. Similarly, any instruction can be made to use predication.

Predicated execution allows if-conversion—an algorithm that converts control dependencies to data dependencies—to eliminate branches. The algorithm replaces a branch instruction—and the instructions that are dependent on it—with a sequence of instructions predicated with the branch condition. Fork-and-merge control structures, such as an IF-THEN-ELSE having few instructions in each of the clauses, are used frequently in typical programs and are generally amenable to if-conversion. If-conversion cannot remove all conditional branches, however, because it cannot be applied to divergent control structures or across function calls.

Eliminating the branches at the fork and merge points creates larger basic blocks, providing the compiler with more instructions from which to extract ILP and produce better scheduling. Eliminating branches—even those that are unconditional or would be predicted correctly—can also eliminate pipeline bubbles, which may otherwise be inserted when fetching the branch target instructions. Branches generally increase cache and TLB miss rates. Since many of the conditional branches are removed, if-conversion also reduces the total number of mispredicted branches, trimming overall execution time.

Using if-conversion does not guarantee a performance gain, however, since some of the predicated instructions later become NOPs. In fact, since these annulled instructions compete with instructions doing useful work, they could actually increase the overall execution time if not enough resources are provided to handle these extra instructions. The performance gain diminishes as the number of instructions being predicated increases, limiting the benefit of if-conversion to branches with a few control-dependent instructions.

Processors can be designed to reduce the hindrance, however, allowing compilers to use if-conversion more freely. In today's high-end processors, the annulled instructions would often use otherwise idle resources. These burdensome instructions can also be canceled—as early as in the decode stage—once the predicate condition is known. Such an optimization would require increasing the number of instructions that are fetched and decoded in a cycle but may not require a comparable increase in the execution stages, which generally have significantly more transistors than the fetch and decode stages. The extra decoders need to decipher only the predicate field, not the entire instruction, making early cancellation inexpensive to implement.

## Predication Can Fill Multiple Branch Delay Slots

Predication also remedies the fundamental flaw of delayed branches in superpipelined superscalar designs—finding useful instructions to fill more than a dozen branch delay slots. It enables more of the delay slots to be filled, making the combination of delayed branches and predication an effective and inexpensive way to eliminate pipeline bubbles

### Delayed Branches To Resurrect

Introduced in early RISC processors, delayed branches eliminate pipeline bubbles caused by executing taken-branch instructions. The bubbles exist because the fetch stage has the next sequential instructions—not the branch target instructions—when the branch instruction is processed in the decode stage. In a scalar processor with a modest pipeline, a single delay slot can eliminate the taken branch penalty.

Most of the later RISC architectures do not use delayed branches, however, because superscalar superpipelined designs add many more delay slots than can be filled. Many of today's high-performance processors would need more than a dozen delay slots to completely eliminate the bubbles. In addition, delayed branches complicate the instruction-fetch design—especially in architectures that allow branch instructions in the delay slots—making them undesirable in high-performance designs.

Instead, the newer architectures and processors rely mostly on instruction prefetching and branch prediction to reduce the pipeline bubbles caused by taken branches. Today's high-performance processors use large instruction-fetch buffers, enabling branch instructions to be identified in advance, and branch prediction tables, enabling branch-target instructions to be prefetched. Although prediction accuracy can be high—as much as 95% on SPEC95 benchmarks—even a small percentage of mispredicted branches can significantly degrade performance, as a misprediction typically incurs 3–7 cycles of penalty.

As the name implies, delayed branches are generally viewed as delaying the branching action. They are better viewed as specifying the branch-target addresses earlier in the program than do normal branches, allowing more time to fetch the branch-target instructions. While the delay-slot instructions are being fetched and executed, the extra time enables the branch-target address to be computed, eliminating the need to predict the target address. It also allows access to a large multicycle instruction cache without incurring a branch penalty.

Each delayed-branch instruction can specify the number of delay slots, as in Mitsubishi's D30V (see MPR 12/9/96, p. 1), eliminating the need to use NOPs. More significantly, such an instruction allows the compiler to schedule as many delay-slot instructions as it can, since more delay slots give the processor more time to prefetch the branch target instructions. It also allows an implementation-independent scheduling.

Predication enables more of the delay slots to be filled, making it architecturally synergistic with delayed branches. As architects look for ways to reduce the branch penalty and instruction-fetch latency, we expect to see delayed branches regain popularity.

caused by taken branches (see sidebar). Intel and HP have not revealed the IA-64 branch model, but we expect to see delayed branches in IA-64, as in Philips's TriMedia architecture (see MPR 12/5/94, p. 12).

Predication complements delayed branches by enabling the delay slots to be filled with predicated instructions. As usual, the delay slots are filled first with prior instructions—the instructions that would otherwise appear before the branch instruction. Remaining slots can be filled with instructions from either branch path, since the instructions can be predicated with either the taken or not-taken condition.

In architectures that allow multiple predicates to coexist, such as IA-64 or Trimedia, instructions from multiple basic blocks can be used in branch delay slots, increasing the chances of filling more delay slots. Figure 3 shows a hypothetical example using five basic blocks, *A*–*E*. Using if-conversion, the branch in block *C* is eliminated, and the instructions in blocks *B*–*D* are predicated using the appropriate branch condition. Using a delayed-branch instruction in block *A*, the delay slots can be filled with instructions from blocks *B*–*E*, assuming the branch in block *A* is likely to be taken. Because the first instruction in block *E* is executed in a delay slot, the target of the delayed branch is changed to the second instruction in block *E*.

Some instructions in the delay slots must be predicated using multiple branch conditions. For instance, since the instruction in block *D* is executed only when the branch in block *A* is taken and the branch in block *C* is not taken, it must be predicated with the AND of these two branch conditions. Architectures that keep all predicate bits in a few predicate registers typically provide instructions for generating compound predicates. IA-64 is likely to have *predicate-logical* instructions, which perform bit-wide Boolean operations on the predicate bits, enabling compound predicates to be generated using a single instruction.

Instructions from the paths predicted to execute less frequently can be moved to delay slots, as Figure 3 also shows, reducing the penalty when the branches are mispredicted. To reduce the penalty further, these instructions can be scheduled ahead of those from the predicted paths or even ahead of the prior instructions. If they could be scheduled to use otherwise idle resources, such scheduling may not necessarily delay the overall execution of the prior instructions or those from the predicted paths—when the branches are predicted correctly.

In IA-64, the number of delay slots could be specified in bundles of instructions (i.e., up to the end of the next *n* bundles), reducing the number of bits to specify delay slots while simplifying the instruction-fetch mechanism. Instruction- and data-prefetch instructions can also be used in the delay slots, reducing memory-access latency for later instructions.

## Speculative Load Goes Beyond Predication

IA-64 uses speculative, or nonfaulting, loads to hide memory latency. Instead of causing a trap, a speculative load saves

exception information when it encounters an exception condition and behaves like a normal load otherwise. Software is required to check for the saved exception condition before using the load result.

SPARC V9 has a nonfaulting load similar to IA-64's speculative load; the nonfaulting load returns a zero value when an exception is encountered and behaves like a normal load otherwise. Using a compare instruction, the load result is first checked and is used if not zero. Otherwise, the memory address is read again using a normal load instruction to differentiate between zero-value data and an exception condition, which would then cause an exception.

Many processors implement a data-prefetch instruction, which copies a block of data from memory to cache and becomes a NOP when it encounters an exception condition. It can hide the cache-miss latency but not the load-use penalty, since a load instruction is needed to move the data to a register. An out-of-order design can hide a few load-use penalty cycles, but a wide-issue in-order design such as Merced would require many nondependent instructions between a load and its use to hide even one cycle of load-use penalty.

A speculative load instruction, in contrast, can hide multicycle load-use penalties and some amount of the cache-miss latency without requiring an out-of-order design. It can also prefetch from noncachable locations, while a data-prefetch instruction cannot. Because speculative load instructions are scheduled earlier in the program, however, they cause registers to be unavailable for a longer period, requiring additional registers to avoid register spills and fills. IA-64 may not provide a separate data-prefetch instruction, since a speculative load instruction can be used as a data-prefetch instruction.

Compared with a predicated load instruction, such as those in the ARM instruction set, a speculative load instruction can hide more of the memory-latency cycles. A predicated load instruction cannot be scheduled earlier in the program than the instruction generating the predicate. A speculative load instruction, in contrast, has no such restriction and can be scheduled as early as its address can be computed, allowing the compiler greater scheduling freedom. It requires software to check for an exception, however, adding execution overhead compared with using predicated load instructions.

IA-64 may use both speculative and predicated forms of load instructions, however, providing the compiler an option to use fewer instructions. The compare instructions, which generate the predicates for the predicated loads, could be hoisted above earlier branch instructions. Ideally, they should be scheduled immediately after their operands are available, since generating predicates as early as possible reduces pipeline stalls and promotes early cancellation. As the compare instructions are hoisted earlier in the code, the predicated load instructions can also be scheduled earlier, allowing more of the memory-latency cycles to be hidden. Since predication and speculative loads are already supported in IA-64, adding predicated load instructions is unlikely to add any significant complexity.

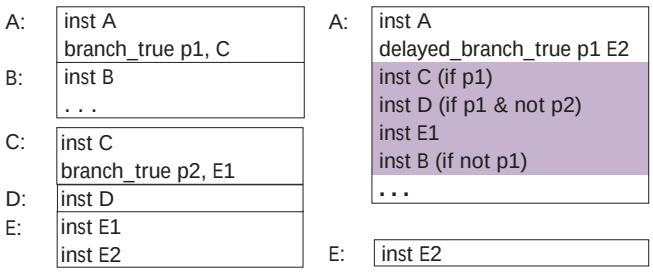


**Figure 3.** Predication allows more branch delay slots to be filled with useful instructions. Example shows four branch delay slots.

## Guessing IA-64's Speculative Load Definition

Intel and HP have not disclosed technical details of IA-64's speculative load, but we can make educated guesses as to how it may be defined in IA-64 and implemented in Merced. As demonstrated by SPARC V9, speculative load is easy to retrofit onto an existing architecture, requiring few new instructions and registers.

In IA-64, a speculative load instruction returns the data if there is no exception and—we expect—clears an invalid bit associated with each of the registers. If an exception occurs, the speculative load would set the invalid bit and return an undefined result, simplifying implementation. Compared with SPARC V9's nonfaulting load, having the invalid bit eliminates a compare instruction, reducing overhead.

IA-64 provides the CHK.S instruction, a branch instruction that would use one of the invalid bits as the branch condition. Since IA-64 has 128 integer and 128 floating-point registers, it would need four 64-bit registers to hold the invalid bits. These registers must be architecturally visible so they can be saved and restored during a context switch. The CHK.S instruction presumably branches to an exception-checking routine, saving the return address in a link register.

In the exception-checking code, a second attempt is made to read from the same address, using a normal load instruction. To facilitate this second attempt, the registers used by the speculative load instruction must be preserved, or the computed address can be saved in one register. For some exception conditions, such as parity error or page fault, the exception may no longer exist for the second access, returning the load result. Otherwise, the persistent error condition would cause an exception. After returning from the exception-handler routine, or when the exception condition no longer exists, an unconditional branch to the address in the link register returns the program to the instruction following the CHK.S instruction.

The overhead of using a speculative load is greater than just the CHK.S instruction. For a speculative load instruction to be hoisted above an earlier store, the compiler must ensure that the two instructions do not access the same memory

```
load    R1, R2, R3    ; R1 ← MEM[R2 + R3]
add     R4, R2, R3    ; R4 ← R2 + R3
. . .
store   R5, R6, R7    ; MEM[R6 + R7] ← R5
add     R8, R6, R7    ; R8 ← R6 + R7
cmp.eq  p1, R4, R8    ; p1 ← (R4 eq R8)
cmov    p1, R1, R5    ; R1 ← (if p1 ) R5
```

**Figure 4.** If a speculative load is hoisted above an earlier store, the store address must be checked against the load address. If they are the same, the store data must be copied to the load data register.

location, a very difficult task. A workaround is to hoist the load before the store and add "fixup" code, which copies the store data to the load instruction's result register if the two have the same address. The overhead of this fixup code can be significant, as Figure 4 shows, requiring up to four instructions for each store. As provided in the PowerPC architecture, IA-64 may provide an address-update option for load and store instructions, returning the computed address in a source-operand register. This would eliminate the two ADD instructions but requires a LOAD-WITH-UPDATE instruction to return two results.

## Conditional Move Is Inadequate

RISC vendors argue that they have already added some of the "EPIC" features to their own architectures. For example, conditional-move instructions allow if-conversion in limited cases. A RISC processor can unconditionally execute the instructions in both paths of a branch, storing their results in disjoint sets of registers. Using what would be the branch condition, the correct set of results is conditionally moved to the set of registers expected by subsequent instructions.

But conditional-move instructions alone are inadequate to take full advantage of if-conversion or predication. First, because software expects instructions in only one of the two branch paths to execute, the instructions in the other path should not generate an exception. This forces the compiler to use only instructions that can never generate an exception, limiting the cases in which if-conversion can be applied. Second, the conditional-move instructions add overhead that is not needed if predicated instructions are used. These extra instructions increase the conversion overhead, reducing the benefits of if-conversion. Third, since both sets of results are stored in addressable registers, more registers are needed for if-conversion using conditional-move instructions. Last, instructions generating results that will be discarded cannot be identified without predication and therefore cannot be canceled early.

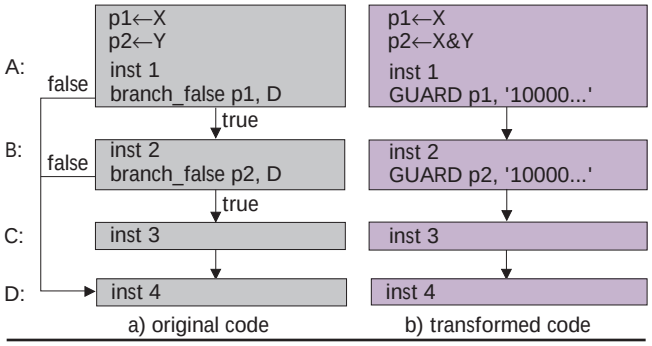


**Figure 5.** GUARD instruction can specify predication for instructions that follow it.

## Retrofitting Predication Using GUARD Instruction

Except for the ARM architecture (see MPR 12/18/91, p. 11), which supports predication in all instructions, most existing architectures support predication in only a few instructions, generally limited to the branch and move instructions. In almost all cases, existing instruction sets do not have spare bits to add features, such as predication, to all instructions in the set. Most instruction sets, however, don't use all of the opcodes, allowing some new instructions to be added. Fortunately, predication can be added to existing instruction sets using a few instructions.

At the 1994 International Symposium on Computer Architecture, Pnevmatikatos and Sohi described a way to retrofit predication onto an existing architecture by adding the GUARD instruction and a set of predicate registers. A GUARD instruction specifies a 1-bit predicate register and a mask that identifies the instructions to be predicated. Assuming the GUARD opcode and the predicate register can be specified in 12–14 bits, the mask would have enough bits to identify 18–20 of the next instructions to be predicated on the register. By using additional GUARD instructions specifying the same predicate register, any number of predicated instructions can be specified.

Figure 5 shows an example of the GUARD instruction. The instructions in blocks *A* and *D* are to execute unconditionally, those in block *B* are guarded by the condition *X*, and those in block *C* are guarded by the condition *X&Y*. Using if-conversion, the branch instructions in blocks *A* and *B* are replaced with GUARD instructions. The first GUARD instruction specifies that the next instruction is guarded by the predicate p1 being true and others are not guarded by p1. Instruction 2 is therefore executed only if p1 is true, and the others are executed unconditionally, unless they are guarded by other predicates. To correctly execute the instructions in block *C*, the predicate p2 is generated using the condition *X&Y*.

To implement the GUARD instruction, the processor maintains a mask of instructions to be nullified. The mask is simply a shift register, initialized to all zeros so no instruction is to be nullified. When a GUARD instruction specifies a predicate register and a list of predicated instructions, the bits corresponding to the identified instructions are set—to be nullified—if the predicate is false. Otherwise the bits in the mask are not altered. In each cycle, the mask is shifted by the number of instructions retired, with zeroes filling in.

Other than requiring an extra instruction, using the GUARD instruction to specify predication has advantages compared with using a predicate field in each instruction. The GUARD instruction can specify an arbitrary sequence of

predicated instructions, offering the same degree of flexibility a predicate field provides. In addition, it makes early cancelation easier to implement, since the processor knows which instructions to nullify before it fetches the instructions. Best of all, it allows predication to be retrofitted without overhauling existing instruction sets.

## 32-Bit Modifiers Can Extend RISC ISA

IA-64 introduces a new concept in instruction-set design; it encodes an instruction using two disjoint groups of bits—the instruction field and template—in a bundle. VLIW instruction formats use a similar bundle to encode multiple operations, but the encodings are not of the same length, and therefore each encoding occupies a fixed position within the bundle. The template allows three equal-length instructions to fit in a 128-bit bundle, avoiding VLIW's fixed-position format that requires NOPs.

This same concept can be used to extend RISC instruction sets, allowing almost any new feature to be added. We can define a 32-bit instruction modifier, which must be the first word in every quadword, as Figure 6 shows. The modifier provides up to 10 additional instruction bits for each of the next three instructions in the quadword.

The modifier can be used to add predication to the entire instruction set. It can also extend register-number width for most instructions, allowing additional registers to be added. It can specify cluster assignments and the grouping hints that IA-64's template is likely to provide. Adding these types of functions is likely to cause minor changes to existing decode logic. As in IA-64, when the program branches to one of the three instructions, the modifier instruction must also be fetched and decoded, but this requires only that the instruction-fetch mechanism uses quadword-aligned addresses.

Switching between the normal mode and the extended mode—which recognizes the modifier—can be done by setting or clearing a mode bit. To enter the extended mode, the instruction that alters the mode bit must be the last instruction in a quadword. In comparison, IA-64 is expected to use special branch instructions to switch between x86 and IA-64 modes (see MPR 3/31/97, p. 16).

Another way to add the modifier is to use an unused opcode. The new opcode indicates the remaining bits in the instruction provide additional encoding for the next instructions. Unlike in the previous approach, the modifier is not required in every quadword, reducing code-size expansion. For example, the modifier can specify predication for some number of instructions that follow it. If the modifier is absent, the same instructions would execute unconditionally.

This second approach restricts the kind of changes that can be made to an existing instruction set, since the modifier provides fewer bits and can only specify optional features. It allows both backward and forward binary compatibility, however. Existing binaries will execute without a change on a processor that implements the extended instruction set. The modifiers in new binaries would cause traps, and therefore can be emulated, on a processor that does not implement the newer instruction set. In contrast, IA-64 binaries could not be emulated—except entirely in software at an unacceptable performance level—on an x86 processor.

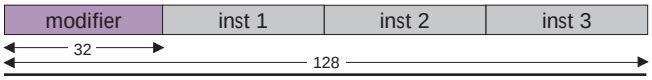


**Figure 6.** A 32-bit modifier can encode additional instruction bits for the next three instructions in every quadword.

## EPIC Is a Natural Evolution of RISC

EPIC has an inherent advantage over most existing RISC architectures, allowing simpler implementations to achieve a higher clock speed and deliver better performance. The large number of registers allows partitioning the execution core into clusters without burdening the programming model. Execution clusters can be defined to be independent of implementations, providing binary compatibility while enabling different designs for different price/performance targets.

Predication reduces the number of branches, enabling the compiler to create bigger basic blocks to extract parallelism. It also complements delayed branches, which specify branch target addresses earlier in the program than do normal branches. Using predicated instructions to fill multiple delay slots, delayed branches can hide many of the pipeline bubbles caused by taken branches. Delayed branches that specify the number of delay slots eliminate the need to use NOPs and enable implementation-independent scheduling.

All of the features that are currently known to be in EPIC can be retrofitted onto most existing RISC architectures, given available opcodes. For adding predication as well as additional registers, unused opcodes can be defined as instruction modifiers, altering the semantics of instructions that follow them. This approach may not support all of the EPIC features together, but it provides better binary compatibility than does Merced's dual-instruction-set approach.

Using Merced's approach, all of the EPIC features can be added to any processor. Retrofitting the EPIC features onto RISC should incur much less overhead than on x86, since EPIC is a natural evolution of RISC. The RISC architectures' fixed-length instruction format and decoupled memory and register operations allow most of the EPIC features to be added easily. In many cases, the RISC-to-EPIC translator could be a simple extension to the decoders.

The transition to IA-64 gives the few remaining RISC vendors an opportunity to deliver EPIC processors that outperform those from Intel. An IA-64 processor could deliver better performance than most RISC processors, but Merced is more than just an IA-64 processor. It must deliver reasonable x86 performance as well, and therein lies a fetter that could slow Merced's IA-64 performance. Its x86 hardware can only be a burden on cycle time, die size, and design time. Merced is expected in systems in 1999, however, putting Intel ahead of most RISC vendors in the race to ship an EPIC processor. M