VOLUME 13, NUMBER 7
MAY 31, 1999

MICROPROCESSOR REPORT

THE INSIDERS' GUIDE TO MICROPROCESSOR HARD WAR

IA-64: A Parallel Instruction Set

Register Frames, x86 Mode, Instruction Set Disclosed

by Linley Gwennap

Finally allowing a full evaluation of their new instruction set, Intel and Hewlett-Packard have released a full description of IA-64's application-level architecture and instruction set. The disclosures address some previous criticisms of the architecture and provide more details concerning how IA-64 processors will execute both x86 and PA-RISC binaries.

The disclosures show a thoroughly modern instruction set with a range of multimedia instructions and prefetch capabilities. Although IA-64 includes many RISC concepts, the architects added some rather complicated and specialized instructions. Concerns remain, however, about code density and just how much of an advantage these new features will provide over a standard RISC architecture.

One criticism had been that the large register file, while effective for compute-intensive routines, would cause excessive overhead on subroutine calls, due to saving and restoring the contents of the registers. The vendors disclosed that IA-64 supports register frames that alleviate much of call/save overhead.

Register Frames Are Dynamically Sized

With IA-64's 128 integer registers plus predicates, saving and restoring the entire register file takes more than four times as long as on a standard RISC processor. To ease this problem, IA-64 implements register frames, which take advantage of the large register file to efficiently handle multiple levels of subroutine calls. Register frames are similar to the fixed-size register windows in SPARC (see MPR 12/26/90, p. 9), except that IA-64 allows software to dynamically specify the size of each frame using the ALLOC instruction.

In the example shown in Figure 1, the top-level routine has specified a register frame with 19 registers, divided as 12 for local use and 7 for parameter passing (output). In addition, the routine can use the first 32 registers, which are designated for global use. When this routine calls a subroutine, the register frame pointer is advanced by 12 (the number of local registers) to create a new register frame. This subroutine uses ALLOC to set up 15 locals and 8 outputs. The first 7 locals overlap the outputs of the previous routine, providing input parameters to the subroutine. The subroutine also has access to the 32 global registers.

From the subroutine's viewpoint, however, the registers in its frame are numbered from 32 to 54, even though they occupy physical registers 44 to 66. Thus, the compiler doesn't have to know which registers are unused by previous routines; it simply arranges each routine within its own virtual register space. This technique simplifies situations in which a subroutine can be called from various places in a program, and it can avoid saving and restoring registers, even when a subroutine is called through a dynamic link.

Register Save Engine Spills and Fills

Even though the IA-64 register file is large, it is finite. The architecture defines a register save engine (RSE) that automatically spills registers to memory when the register file is fully allocated, creating the illusion of an infinite register file. For example, if ALLOC must create a 20-register frame starting at physical register 120, the frame will go to register 127 and then wrap back to register 32. To avoid destroying state, the contents of registers 32–43 are stored to the memory stack. The RSE can save and restore registers before they are needed, minimizing the performance impact. This activity is

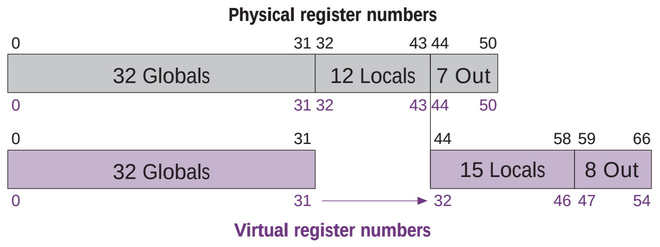

Figure 1 . In this example, a top-level IA-64 routine calls a subroutine and allocates a new frame with 23 registers that overlap the output registers of the original routine. The subroutine uses virtual registers 32–54, but these are mapped to physical registers 44–66 by the hardware.

invisible to software, whereas SPARC software must manually spill and fill registers.

The register frames minimize the number of registers saved and restored during procedure calls, although IA-64's static design will result in more "live" registers than in a traditional dynamic design. Routines with high computational requirements can still take advantage of the full register file by allocating a frame with up to 96 registers (plus globals). Any routine that desires can also use IA-64's register rotation (see MPR 3/8/99, p. 16); to simplify the hardware, the rotating portion of the frame is required to start at GR32 and contain a multiple of eight registers.

With rotation and framing, the translation to physical register addresses requires adding two 7-bit values—the register frame pointer and the rotating register base (RRB)—to the virtual register number and wrapping around from register 127 to 32 if necessary. Although parts of this calculation can be done ahead of time, an IA-64 processor may require an extra pipeline stage to access its registers. Compared with the register renaming found in most modern processors, however, the IA-64 approach requires much less hardware and adds less time to the pipeline.

Register Set Provides Massive Resources

Only the integer registers are framed; all other registers must be saved and restored explicitly by software. This restriction simplifies the register save engine, which doesn't need to keep track of separate regions to save and restore various register types. Furthermore, the predicate registers can be saved quickly, as a single memory access saves all 64 of the 1-bit predicates, and the floating-point registers are not used by most subroutines. To further reduce save overhead, software needs to save FP registers only if the user mask indicates that either the lower or upper half of the FP register file has been written.

As Figure 2 shows, the floating-point registers are 82 bits wide. The native FP format is identical to Intel's 80-bit extended precision (EP) mode, except that the exponent field has 17 bits instead of 15. All results are calculated in the native format and converted to the requested precision. The extra two exponent bits handle common overflows and underflows in EP mode, simplifying some scientific algorithms.

Each of the integer registers (except GR0, which is always zero) has an associated NaT (not a thing) bit to support speculative loads (see MPR 3/8/99, p. 16). The FP registers, however, don't need a NaT bit; instead, they encode NaT as a special value that is unused by the IEEE-754 standard.

The 64-bit instruction pointer (IP) points to the next bundle to be executed. The current frame marker (CFM) holds the register frame pointer and frame size, along with the RRB values for the integer, FP, and predicate register files. The usr mask is the application-visible portion of the processor status register (PSR). It controls the endianness of loads and stores, enables or disables the performance monitors and data-alignment checking, and contains the dirty flags for the upper and lower halves of the FP register file.

Like current Intel products, IA-64 processors will include a serial number and CPU_ID. This information is spread across five 64-bit registers, including two to hold a 16-character ASCII vendor name (e.g., "Genuine Intel").

IA-64 also defines a set of 128 special registers, similar to PA-RISC's control registers, called the application regis-

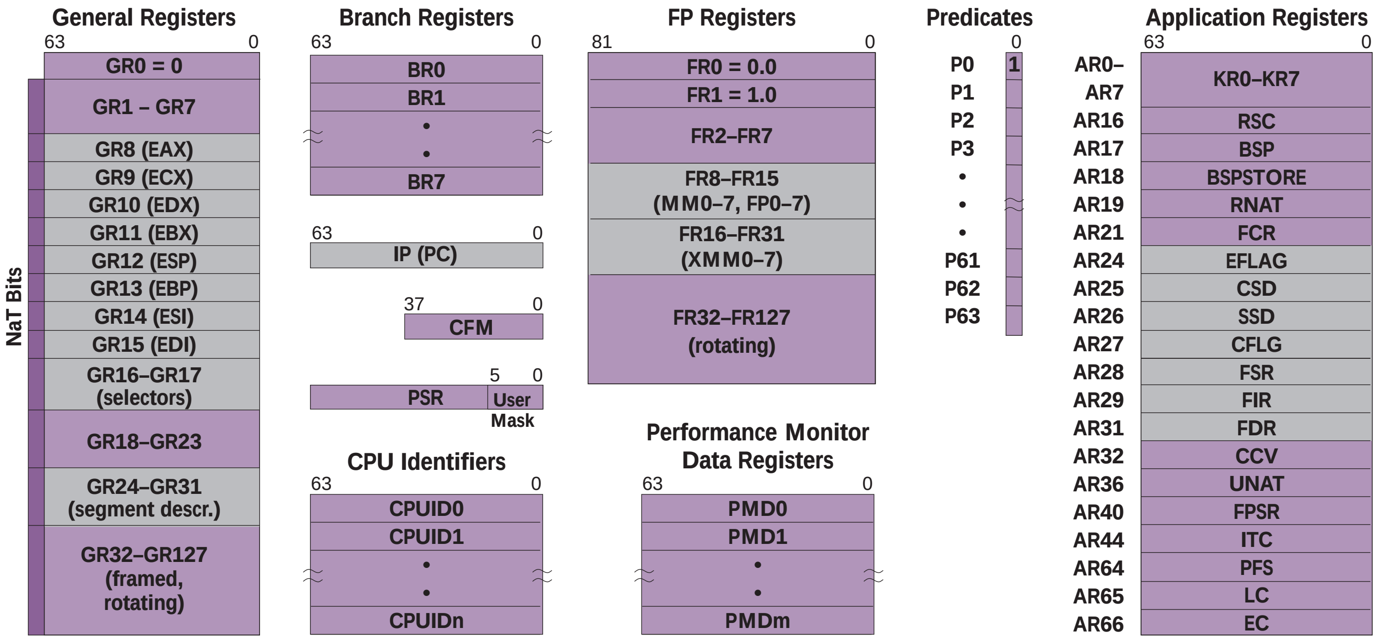

Figure 2. The IA-64 register file consists of 128 integer registers, 128 floating-point registers, and 64 predicate registers, along with an extensive set of special-purpose registers. Registers in gray are shared between IA-64 and x86 modes; x86 functions are in parentheses.

ters. Although many are reserved for future definition, several have important functions. For example, AR65–66 contain the loop count (LC) and epilogue count (EC) used to control loops with rotating registers. AR16–19 provide information for the register save engine, such as the memory address for spilling registers.

The UNAT register (AR36) temporarily holds 64 NaT bits so they can be saved and restored using the LD8.FILL and ST8.SPILL instructions. The floating-point status register (FPSR) and interval time counter (ITC) are also in the AR file. The CCV register contains the third source operand for the CMPXCHG (compare and exchange) instruction. The eight kernel registers (KR) are user readable but are writable only by privileged software. They can be used to store information such as thread pointers and the CPU number in a multiprocessor system. Other application registers are defined by the system-level architecture, which has not yet been made public.

x86 Compatibility Has Large Overhead

All IA-64 processors allow direct execution of x86 (IA-32) binaries by placing the processor in x86 mode. The architecture supports the full Pentium III instruction set, including the streaming SIMD extensions (SSE). It supports all x86 modes—real, protected, and VM86—as well as self-modifying code, offering full compatibility with existing binaries. Even privileged modes are supported, allowing an IA-64 processor to run an x86 operating system, although not with commercially viable performance.

AR24–31 allow IA-64 programs to access x86 special registers, such as the code segment descriptor (CSD) and floating-point status register (FSR). Other x86 state is mapped onto the IA-64 general and FP registers, as shown in Figure 2. Note that, because the IA-64 FP registers are only 82 bits wide, the 128-bit XMM registers defined by SSE (see MPR 10/5/98, p. 1) must be mapped to pairs of IA-64 registers. Because the integer MMX registers are already mapped onto the FP registers in x86, both of these register sets are mapped onto the same IA-64 FP registers.

Mapping the x86 state onto the IA-64 registers allows simple parameter passing between IA-64 and x86 routines: data placed in one of the shared registers can be accessed directly in either mode. This sharing also slightly reduces the number of registers in an IA-64 processor.

The downside is that switching modes is fairly onerous. The actual mode switch itself is quite simple, as certain branch instructions simply change the processor's mode bit, although this is likely to force the processor to drain its pipeline before switching modes. As Figure 3 shows, the new x86 instruction, JMPE, sets the processor to IA-64 mode, while the IA-64 BR.IA instruction branches to x86 mode.

The overhead comes in preparing for the transition. Because of the register overlap, any shared registers with important data must be explicitly saved to memory before switching modes. Before calling an x86 routine, IA-64 code must properly set up the x86 segment descriptors, PSR, and EFLAG registers.

Furthermore, the architecture allows the processor to overwrite all of the nonshared general, FP, and predicate registers, as well as the ALAT (used for IA-64's speculative stores), during x86 execution. With this flexibility, the processor can use almost any of its resources to emulate the behavior of x86 instructions. But as a result, switching modes generally requires the entire IA-64 register set to be saved or restored.

This mode-switch overhead makes it impossible to mix x86 and IA-64 code at the subroutine level. Any application, operating system, library, or driver must be converted to IA-64 as a complete unit. A native-mode operating system, however, could support x86 applications, libraries, or drivers. Each OS vendor must determine whether and how it will support x86 compatibility.

IA-64 Includes MMX and SSE Instructions

The basic IA-64 instruction set bears a strong resemblance to HP's PA-RISC. The only hardware support specifically for PA-RISC, however, is the ADDP instruction, which shifts bits 31:30 to bits 62:61 before summing. This instruction helps in emulating the PA-RISC segmentation model.

HP is developing a dynamic translator that will convert PA-RISC binaries to IA-64 on the fly. The translator actually recompiles the binaries, taking advantage of as many IA-64 features as necessary. This tool is similar to Digital's FX!32 (see MPR 3/5/96, p. 11) but is simpler; it can deliver 50% of native-mode performance on many applications. Performance will be better for applications that spend more time in the native-mode operating system and libraries.

The integer instructions cover the basics, as Table 1 shows, along with the unusual shift-and-add instructions found in PA-RISC (see MPR 4/3/91, p. 13). The ADD and SUB instructions will optionally offset the result by 1, a feature useful in implementing multiprecision arithmetic. Most of the arithmetic instructions can replace one operand with an immediate value of at least 8 bits. The ADDL instruction can add a 22-bit immediate value to a register, but the source register can be only GR0–GR3. For truly long immediates, the MOVL instruction, which occupies two slots in a bundle, can load a 64-bit constant.

The compare (CMP) instruction compares two values and stores the result in a predicate register, along with the

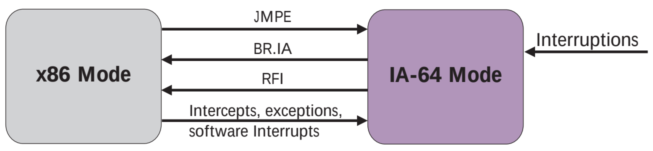

Figure 3. Only a few instructions and events cause an IA-64 processor to switch between IA-64 and x86 (IA-32) modes. The processor can also be configured to direct interrupts to IA-32 mode, allowing compatible execution of an unmodified x86 OS.

Type	Name	Description
Integer Arithmetic		
A	ADD	Addition†
A	SUB	Subtraction†
A	ADD ,1	Three-operand (A+B+1)†
A	SUB ,1	Three-operand (A–B–1)†
A	ADDL	Add long (22-bit immed)†
A	SHLADD	Shift left (1–4) and add
A	AND	Logical AND†
A	OR	Logical OR†
A	ANDCM	Logical AND complement†
A	XOR	Logical XOR†
A	ADDP	32-bit pointer addition†
A	SHLADDP	Shift left and add pointer
A	CMP	Compare GR†
A	CMP4	Compare GR, low 32 bits†
I	TBIT	Test GR bit
I	TNAT	Test GR NaT bit
I	SXTn	Sign extend [n=1,2,4]
I	ZXTn	Zero extend [n=1,2,4]
I	DEP	Deposit†
I	DEP.Z	Zero and deposit†
I	EXTR	Extract signed
I	EXTR.U	Extract unsigned
I	SHL	Shift left
I	SHR	Shift right signed
I	SHR.U	Shift right unsigned
I	SHRP	Shift right pair
I	CZXn	Find first zero [n=1,2]
I	POPCNT	Population count
X	MOVL	Move 64-bit immediate
Parallel Integer Arithmetic		
A	PADD	Parallel modulo addition
A	PADD.SSS	Parallel add, signed sat.
A	PADD.UUU	Parallel add, unsigned sat.
A	PSUB	Parallel modulo subtraction
A	PSUB.SSS	Parallel sub, signed sat.
A	PSUB.UUU	Parallel sub, unsigned sat.
A	PAVG	Parallel arithmetic average
A	PAVG.RAZ	w/round away from zero
A	PAVGSUB	Parallel average of diffs
A	PCMP	Parallel compare
I	PMPY.L	Parallel multiply odd items
I	PMPY.R	Parallel multiply even items
I	PMPYSHR	Parallel multiply, shift right
I	PSAD	Parallel sum of absolute diffs
I	PMIN	Parallel minimum
I	PMAX	Parallel maximum
I	PSHL	Parallel shift left
I	PSHR	Parallel signed shift right
I	PSHR.U	Parallel unsigned shift right
I	PSHLADD	Parallel shift left and add
I	PSHRADD	Parallel shift right and add
I	MIX.L	Interleave odd elements
I	MIX.R	Interleave even elements
I	MUX	Arbitrary copy of elements
I	PACK	Pack into smaller elements
I	UNPACK	Expand into larger elements

Type	Name	Description
Memory Transfer		
M	LDn	Load n bytes [n=1,2,4,8]
M	LDFn	Load FP [n=S,D,E]
M	LDF8	Load 64-bit integer to FR
M	LDFPn	Load FP pair [n=S,D]
M	LD.S	Speculative load*
M	LD.A	Advanced load*
M	LD.SA	Speculative advanced load*
M	LD.C	Check load*
M	LD.ACQ	Ordered load
M	LD.BIAS	Load with intent to store
M	LD8.FILL	Load GR with NaT bit
M	LDF.FILL	Load FR with 82 bits
M	STn	Store n bytes [n=1,2,4,8]
M	STFn	Store FP [n=S,D,E]
M	STF8	Store 64-bit integer from FR
M	ST8.SPILL	Store GR with NaT bit
M	STF.SPILL	Store FR with 82 bits
M	CMPXCHG	Atomic compare exchange
M	FETCHADD	Atomic fetch and add
M	XCHG	Exchange memory and GR
M	LFETCH	Prefetch cache line
M	LFETCH.EXCL	Prefetch writable cache line
Register Transfer		
A	MOV	Move from GR to GR†
I	MOV BR	Move between BR and GR†
I	MOV PR	Move between PR and GR†
V	MOV AR	Move between AR and GR†
M	MOV PSR	Move between PSR and GR
M	SUM	Set user mask†
M	RUM	Reset user mask†
M	MOV PMD	Move from PMDR to GR
M	MOV CPUID	Move from CPUID to GR
I	MOV IP	Move from IP to GR
B	CLRRRB	Clear RRB
M	GETF.EXP	Move FP exponent to GR
M	GETF.SIG	Move FP significand to GR
M	GETF.n	Move FP value to GR [n=S,D]
M	SETF.EXP	Move GR to FP exponent
M	SETF.SIG	Move GR to FP significand
M	SETF.n	Move GR value to FP [n=S,D]
M	ALLOC	Allocate register frame
Control Transfer		
B	BR	Unconditional branch
B	BR.COND	Conditional branch
B	BR.CALL	Conditional procedure call
B	BR.RET	Conditional procedure return
B	BR.IA	Branch to IA-32 procedure
B	BR.CLOOP	Counted loop branch
B	BR.CTOP	Counted loop back w/RR
B	BR.CEXIT	Counted loop exit w/RR
B	BR.WTOP	While loop back w/RR
B	BR.WEXIT	While loop exit w/RR
V	CHK.S	Check speculative load
M	CHK.A	Check advanced load
F	FCHKF	Check FP flags
V	BREAK	Break instruction fault

Type	Name	Description
FP Arithmetic		
F	FMA	FP multiply add
F	FMS	FP multiply subtract
F	FNMA	FP negate multiply add
F	FRCPA	FP reciprocal approx
F	FRSQRTA	FP square-root approx
F	FCMP	FP compare
F	FMIN	FP minimum
F	FMAX	FP maximum
F	FAMIN	FP absolute minimum
F	FAMAX	FP absolute maximum
F	FCVT.FX	Convert FP to signed int
F	FCVT.FXU	Convert FP to unsigned int
F	FCVT.XF	Convert signed int to FP
F	FCLASS	Test FP class
F	FMERGE.S	FP merge sign
F	FMERGE.NS	FP merge negative sign
F	FMERGE.SE	FP merge sign and exponent
F	FAND	FP logical AND
F	FANDCM	FP logical AND complement
F	FOR	FP logical OR
F	FXOR	FP logical XOR
F	FCLRF	Clear FP flags
F	FSETC	Set FP control bits
F	XMA.L	Integer multiply add (on FR)
F	XMA.H	return high 64 bits
F	XMA.HU	return high 64 bits unsign.
Parallel FP Arithmetic		
F	FPMA	Parallel FP multiply add
F	FPMS	Parallel FP multiply sub
F	FPNMA	Parallel FP negate mul add
F	FPRCPA	Parallel FP recip approx
F	FPRSQRTA	Parallel FP sq-root approx
F	FPCMP	Parallel FP compare
F	FPMIN	Parallel FP minimum
F	FPMAX	Parallel FP maximum
F	FPAMIN	Parallel FP absolute min
F	FPAMAX	Parallel FP absolute max
F	FPCVT.FX	Parallel FP to signed int
F	FPCVT.FXU	Parallel FP to unsigned int
F	FPMERGE	Parallel FP merge sign
F	FMIX	FP mix elements
F	FSXT	FP unpack and sign ext
F	FPACK	FP pack elements
F	FSWAP	FP swap elements
F	FSWAP.N	FP swap and negate
F	FSELECT	FP select elements
System Control/Miscellaneous		
M	FLUSHRS	Flush register set
M	FC	Flush cache
M	PTC.E	Purge TLB entry
M	INVALA	Invalidate ALAT
M	INVALA.E	Invalidate ALAT entry
M	MF	Memory ordering fence
M	SRLZ.I	Serialize instruction stream
M	SRLZ.D	Serialize data stream
M	SYNC	Synchronize I&D caches

Table 1. The IA-64 user-mode instruction set includes a staggering variety of instructions, each of which is assigned to one of the following categories: A=integer ALU; I=integer non-ALU; M=memory; F=floating point; B=branch; X=extended, V=various. Only major opcodes and some key variations are shown. *These extensions also apply to LDF and LDFP. †These instructions optionally take an immediate operand.

negation of the result in another predicate register. It supports three basic conditions (=, signed <, and unsigned <) but can generate another seven (≠, signed and unsigned ≤, >, and ≥) by swapping the operands or the destination registers. Variations such as CMP.OR and CMP.AND logically combine the comparison result with the value of the destination predicates, creating compound conditions such as A<B OR A<C.

Bit manipulation is performed by powerful extract and deposit instructions, also based on PA-RISC. Standard shift operations are merely special cases of these instructions. The SHRP instruction extracts a 64-bit value from the middle of two concatenated register values. The MUX instructions rearrange 8-bit and 16-bit elements and are similar to HP's PERMUTE instruction (see MPR 11/18/96, p. 24).

Parallel arithmetic on 8-, 16-, and 32-bit integers is handled by a set of instructions nearly identical to Intel's MMX instruction set (see MPR 3/5/96, p. 1) in function and mnemonic. These include instructions such as parallel add and parallel compare, as well as SSE "new media" instructions such as parallel average and parallel min/max. Because the IA-64 integer registers are all 64 bits wide, these instructions work on the integer registers instead of the special MMX registers, as in x86. The performance of many multimedia algorithms will be greatly enhanced by the availability of 128 registers for parallel operations instead of 8.

Like SSE, IA-64 includes a parallel sum-of-absolute-differences instruction that accelerates video encoding. IA-64 also includes population count, which is useful in cryptographic and other algorithms. The CZX instruction finds the first zero element in a 64-bit register, which assists in string manipulation.

Unlike some 64-bit architectures, IA-64 does not include a 32-bit mode. Compilers can generate 32-bit code by using 32-bit loads and stores. Most arithmetic instructions function identically on 32-bit or 64-bit data, and the IA-64 instruction set includes 32-bit versions of instructions that do not do this. For example, CMP4 is identical in function to CMP except that it looks only at the low 32 bits of the registers. We expect many application vendors will choose to compile in 32-bit mode for compatibility with existing 32-bit programs and data structures.

Floating Point Built Around Fast MAC

IA-64 provides four 13-bit FP status fields, all part of the FP status register (FSR), which contain the IEEE flags and control the rounding modes and the default precision. Each FP instruction specifies one of the four status fields, allowing four independent streams to proceed in parallel while maintaining IEEE compatibility. Traditional architectures can handle only one stream at a time or require complicated hardware to process multiple FP operations in parallel.

In addition to the standard precisions (single, double, and extended), IA-64 instructions can take advantage of the extra exponent bits by setting the WRE bit in the status field. Most FP instructions can also override the default precision by specifying either single or double. All IEEE rounding modes as well as flush-to-zero are supported.

The floating-point instruction set is built around a fused multiply-add (MAC) construct. Simple addition and multiplication are synthesized, using the constants +0.0 and +1.0 stored in FR0 and FR1, respectively. As is becoming common, reciprocal and reciprocal square-root approximations are provided, enabling fast iterative division and square-root calculations to arbitrary precision levels, using Newton-Raphson refinement (see MPR 5/11/98, p. 1).

IA-64 supplies no scalar integer multiply instruction. Instead, integer values can be loaded into the 64-bit significand of FP registers (using LDF8); XMA then performs integer MACs on these values. To avoid moving data between the FP and integer registers, IA-64 supports some logical operations on the FP registers. RSA and other encryption algorithms make heavy use of integer MAC and logical operations. This method leverages the FP MAC unit, reducing hardware overhead, but it could reduce performance if data must be transferred to the integer registers for further processing.

Like SSE, IA-64 includes a full set of parallel operations that compute two single-precision results at once. Although SSE may seem superior in its ability to compute four results at once, the IA-64 compiler can achieve the same result by grouping two parallel FP instructions for execution in the same cycle. SSE's registers are wider, so its eight registers can hold 32 SP values, but the IA-64 register file can hold 252 SP values, even with its narrower registers.

Memory Instructions Control Caching

Load and store instructions support only a single addressing mode (base), with optional postincrement using an immediate value or, for loads only, a register value. These forms also provide a prefetch hint to the hardware, which can suggest that the line pointed to by the postincremented base register should be fetched. Other hints control whether the loaded data should be cached and at what cache level (L1, L2, etc.). Data that is not likely to be reused can bypass the cache hierarchy entirely, improving the cache hit rate.

One, two, four, or eight bytes can be transferred using the *sz* completer (e.g., LD1, LD2, LD4, LD8). Speculation is supported by LD.S, LD.A, and LD.C. The LDFP instruction loads two single- or double-precision values into two FP registers at once, doubling the floating-point bandwidth for many applications. Efficient execution of this instruction will require a 128-bit cache bus.

The compare-and-exchange (CMPXCHG) and fetch-and-add (FETCHADD) instructions perform atomic memory updates that can be used to implement software semaphores.

The vendors did not disclose IA-64's virtual address model. We expect the architecture uses segmented addresses of at least 96 bits, for compatibility with PA-RISC. The operation of the ADDP instruction implies that the upper three bits of an IA-64 address are used to select one of eight segment registers (called space registers in PA-RISC), which

For More Information

The "IA-64 Application Instruction Set Architecture Guide" is available at *www.hp.com/go/ia64* and at *http://developer.intel.com/design/ia64/devinfo.htm.* Additional information on IA-32 compatibility in IA-64 can also be found at the Intel URL.

are probably located in AR8–15. This method would provide applications with a linear address space of at least 2^{61} bytes.

Branch Registers Accelerate Control Flow

IA-64 has only a single basic branch instruction (BR), which can be predicated to create a conditional branch. Branch targets must be bundle-aligned and can be specified by a 21-bit displacement (in bundles) from the current instruction pointer or by a 64-bit branch register. Storing branch addresses (particularly return links) in a separate register file allows those registers to be physically located near the branch unit, reducing the overhead of branch processing.

Several branch hint bits tell the hardware whether to use static or dynamic prediction, whether to predict taken or not taken, and whether to deallocate the entry from the branch target cache (BTC). A prefetch hint suggests whether the processor should aggressively prefetch several instruction-cache lines at the target or only a few. If used properly by the compiler, these hints should make the hardware more efficient by not wasting BTC entries and cache entries.

A subroutine call is coded as BR.CALL, while a return is simply BR.RET. A call saves the current frame marker (CFM) in the PFS register and saves the return address in the specified branch register. A return restores the CFM from PFS and branches to the address in the specified branch register.

The BR.CLOOP version tests and decrements the loop counter before branching. BR.CTOP and BR.CEXIT also test the loop counter but update the rotating register base (RRB) before branching. (The latter form simply reverses the loop test.) BR.WTOP and BR.WEXIT update the RRB but test a predicate instead of a loop counter; these instructions are ideal for do…while loops.

The vendors did not disclose the method of handling interrupts. A bit in the PSR selects whether the processor goes into IA-64 or x86 mode on an interrupt; the latter mode would be used only to execute an unmodified x86 operating system. The PSR and IP must be saved, possibly in shadow registers by the hardware. Free register space can easily be created by allocating a new register frame. Presumably, the PSR contains bits to signal the type of interrupt and also to disable particular interrupt types.

Template Provides Additional Decoding

Each IA-64 instruction is placed in one of six categories, as Table 1 shows, which are then mapped onto four types of generic execution units. I-units handle any integer instruction, including the extended MOVL instruction. M-units handle any memory instruction plus integer ALU instructions (using the memory-address adder). F-units handle FP instructions, while B-units handle branch instructions, including the extended multiway branch.

Instructions are combined into 128-bit bundles, each having three 41-bit instructions plus a 5-bit template. The instructions are longer than standard RISC (or x86) instructions, since 7 bits are needed to specify each operand register plus 6 extra bits to specify a predicate register. The compiler then groups instructions that have no dependencies; a group can be as short as a single instruction or can be arbitrarily long, spanning several bundles.

Although instructions are officially 41 bits long, these encodings are unique only to a specified function unit; the same opcodes are interpreted differently by different function units. Ideally, the instructions would have been 43 bits, allowing two extra bits to specify the unit, but there aren't enough bits in a 128-bit bundle to support three 43-bit instructions.

Instead, the template field specifies both the mapping of instructions to execution units as well as any group boundaries, or stops, within the bundle. In a maddening bit of nonorthogonality, only certain combinations of instruction types and group boundaries can be encoded—24 combinations, to be exact, with 8 more reserved for future use.

For example, the MII template specifies an M-unit instruction followed by two I-unit instructions and no stops. The MIB• template specifies M-unit, I-unit, and B-unit instructions, followed by a stop (•). The MLX template encodes the MOVL instruction in the last two instruction slots. Multiway branches are encoded using the MBB and BBB templates. The full list of templates is MII, MI•I, MLX, MMI, M•MI, MFI, MMF, MIB, MBB, BBB, MMB, MFB. All of these forms are also available with a stop at the end.

Because instructions can often be reordered within a group, this arrangement provides more flexibility than it might appear to, but there will still be some situations where the compiler cannot find the right template and must insert a NOP. The architects say they have carefully selected the templates to minimize these occurrences. The limitations are actually an advantage for the hardware, which doesn't have to handle all possible instruction combinations.

Code Density Remains a Concern

One area in which IA-64 will not excel is code density. The average instruction length for IA-64 is nearly 43 bits, versus 32 for a typical RISC and 21–22 for typical x86 code. A few IA-64 features save instructions, such as predication, which eliminates some branches, but other features require more instructions than are used by other architectures.

For example, ALLOC and CHK instructions will be frequently used, but they add instruction overhead compared with a traditional architecture. The lack of a multiply instruction for the integer registers will increase instruction counts in some applications. The greatest numerical impact will come from added fix-up code from speculative routines (see MPR 3/8/99, p. 16), but this fix-up code is rarely used and will generally do nothing other than take up disk space.

We expect the savings will roughly balance out the extra instructions, excluding fix-up code. Given the 2:1 ratio in average instruction size and other factors, we expect the code size for IA-64 will be roughly twice as big as an equivalent x86 binary, again excluding fix-up code. The increase in code size over a PA-RISC binary will be more modest, perhaps only 30%.

Comparing code density is always challenging, as it depends on the application and on the level of compiler optimization. Even a 2:1 increase is not significant for most system components, as most large applications involve far more data than instructions. The biggest impacts are in the hit rate and bandwidth of the primary instruction cache. The same size cache will hold half as many IA-64 instructions as x86 instructions, reducing its hit rate. To compensate, IA-64 processors may have larger instruction caches; since modern instruction caches are no more than 5–10% of the total die size, this increase would have only a modest cost impact.

The bandwidth between the instruction cache and the second-level (L2) cache must also be increased to compensate for the larger instructions. In a processor with an external L2 cache, doubling this bandwidth can be expensive. Merced and future IA-64 processors, however, will have at least two levels of cache on the processor, making it simple to build a high-bandwidth interface to the instruction cache. The larger binaries will also require more main memory and more disk space, but most DRAM and disks are consumed by data, not instructions.

A State-of-the-Art Architecture

Given HP's and Intel's desire to develop a new instruction-set architecture, IA-64 delivers a strong feature set that takes advantage of the latest advances in instruction-set design. Among today's major high-end architectures, IA-64 is unique in providing a large amount of explicit information from the compiler to the hardware regarding instruction grouping, branch prediction, speculative execution, and prefetching. The larger register set provides more low-latency storage than competing designs, yet the register framing avoids much of the overhead of saving and restoring the additional registers, except on context switches.

IA-64 is a particularly big step forward for Intel, given its current instruction set. Although the company has managed to deliver competitive integer performance from its x86 processors, this success has come in spite of the foibles of that instruction set. IA-64 offers a full 64-bit address space and much better floating-point performance. The new instruction set also maintains and extends the advantages of MMX and SSE for multimedia applications.

No matter how well they perform, RISC architectures have had limited acceptance because they lack compatibility with popular software. IA-64 solves this problem by offering full compatibility with x86's broad software base. For HP's customers, PA-RISC compatibility will also be available through dynamic binary translation.

In both cases, performance in compatibility mode will be significantly lower than native performance, although applications that spend most of their time in operating-system calls will fare better. Even in compatibility mode, performance will be adequate for many applications.

For IA-64 to succeed, however, there must be a base of performance-hungry native applications. Intel recently announced the creation of a $250 million venture-capital fund to support the development of native IA-64 applications and middleware. The release of the instruction-set details should further spur software development. The broad level of support for IA-64 among system and software vendors nearly guarantees its success, as long as Intel delivers competitive performance.

Compiler Is the Key

The biggest concern about IA-64 performance at this point is not the instruction set but Intel's execution on both the hardware and, just as important, the compiler. Merced, the first IA-64 processor, is nearing tapeout. Intel still expects the first Merced systems to ship in mid-2000, although this schedule seems optimistic. We expect Merced to perform well, but it may not outperform all of its RISC competitors. If the compiler, perhaps the most difficult part of the IA-64 effort, is not as good as expected, Merced will fare worse.

The IA-64 compiler must perform many optimizations that are done in hardware today while juggling several large register files, template limitations, grouping issues, speculative loads, and explicit prefetching. It must also efficiently use register frames, although most architects have shunned register windows because compilers have historically been unable to use them effectively. All of these capabilities are possible in theory, but no compiler has ever been shipped that demonstrates all of them. HP and Intel have spent years working on their IA-64 compilers; the fruits of their labor will be seen soon.

If the compiler writers and chip designers do their jobs well, IA-64 should deliver a performance advantage over RISC instruction sets. The vendors have no performance data to offer at this time, but we estimate this advantage to be modest, perhaps 20–30%, if all other things are equal. RISC vendors will need excellent implementations to overcome this advantage. We don't think IA-64 will be untouchable in the market, but its performance advantage, if achieved, would be enough to justify Intel's decision to develop a new instruction set. M