MAJC Gives VLIW a New Twist

New Sun Instruction Set Is Powerful But Simpler Than IA-64

by Linley Gwennap

An emerging group of third-generation instruction sets shares some common ideas, in particular an emphasis on the compiler to perform the instruction grouping and branch prediction that is handled in hardware by traditional processors. Sun's MAJC follows the same theme, but with variations that make it uniquely qualified for a range of embedded applications. Most other modern VLIW architectures are designed for specialized multimedia or signal-processing algorithms. The best-known VLIW-based design, IA-64, is intended for high-performance workstations and servers, leaving an opening for MAJC.

With an eye on embedded systems, the MAJC designers focused on reducing code expansion, a bane of most other VLIW architectures. MAJC also includes a more compact register file than IA-64's and uses a few conditional instructions instead of full predication. Because it comes from Sun, MAJC also has features designed to speed execution of Java, which is growing in popularity for embedded software development. In fact, MAJC (pronounced "magic") stands for Microprocessor Architecture for Java Computing.

To achieve these results, Sun left out some of the bells and whistles found in IA-64. Thus, MAJC won't wring as much performance from some applications as the HP/Intel architecture does, although neither company has shipped processors to validate this assumption. But MAJC programs will clearly be more compact than IA-64 code. MAJC's broad range of features also makes it appropriate for many more applications than specialized designs such as the Philips TriMedia or Texas Instruments 'C6xxx instruction sets.

Compact Instruction Encodings

IA-64 (see MPR 5/31/99, p. 1) places three 41-bit instructions plus some header bits into a 128-bit "bundle." These instructions are wider than standard 32-bit RISC instructions for two main reasons. First, they include 7-bit register identifiers to allow access to the 128-entry register files. Second, each instruction includes a 6-bit predicate field to select one of 64 predicate registers. As a result, IA-64 takes an immediate 33% hit in code density compared with a traditional RISC architecture.

Although this expansion may not matter in a high-end server, Sun didn't want MAJC to pay this penalty. Instead, the architecture sticks with 32-bit instructions. This decision required a compromise: instead of a fully predicated instruction set, MAJC has only a few conditional instructions. This choice avoids bloating each instruction with a predicate field. While both methods are effective at eliminating branches, IA-64's full predication requires fewer instructions to simulate a branch. For most applications, however, the increase in instruction width far outweighs this small reduction in instruction count.

Instead of adding header bits, the MAJC designers stole two bits from the first instruction in a "packet" (see MPR 8/23/99, p. 13) to identify groups of parallel instructions. This method limits the maximum number of instructions per cycle to four. In contrast, IA-64 can create and execute arbitrarily large groups of instructions in a single cycle. Applications that have enough instruction-level parallelism to take advantage of IA-64's greater capabilities will have a performance advantage on the Intel/HP design, but most applications are likely to fare just as well on MAJC. More important, Sun's choice keeps the instruction size to 32 bits.

With these features, MAJC should deliver code density similar to that of RISC architectures such as MIPS and ARM. Code density affects cost by reducing the die area of caches and memory and the cost of external DRAM and ROM. MAJC, however, can't match the code density of hybrid RISC architectures such as Hitachi's SH or "compressed" instruction sets such as Arm's Thumb.

Given the maximum packet size of four, a MAJC processor will typically have four function units, each capable of executing any MAJC instruction. The first instruction in a packet always goes to the first function unit, the second to the second, and so on. The versatility of the function units gives the compiler great flexibility in assigning instructions to units. The only limitation is that the first instruction must be a load or store, due to the two bits "stolen" from the opcode.

Unified Register File

MAJC's unified register file, unusual among modern processor implementations, can hold integer, SIMD integer, floating-point, and condition-code values. Implementations can choose either a 32-bit or 64-bit register width. With at least 128 entries, this register file holds twice as much information as the pair of 32-entry integer and FP register files in a standard RISC architecture. Furthermore, the unified design allows the compiler to dynamically allocate registers to integer or FP data as needed; in a program with no FP calculations, for example, the entire register file could be devoted to integer values, providing four times the storage of a standard RISC architecture.

One potential problem with a unified register file is the need for a large number of read ports. In a traditional four-issue design, a unified file would need up to 12 read ports and 4 write ports, resulting in a large and slow physical implementation. Sun solves this problem by replicating the

register file for each function unit. A four-issue MAJC chip would have four register files, which could each need only three read ports and four write ports. Although this design requires more registers than a single central register file, the physical size of each register file is greatly reduced due to the limited number of ports, resulting in about the same total area. In addition, the smaller individual register files can operate at a higher clock speed. Only a small amount of hardware is needed to synchronize the register files.

Even 128 registers is less than the 256 total registers in IA-64's integer and FP files. To further increase the number of registers, MAJC divides the file into local and global registers. The local registers are unique to each function unit and cannot be accessed by the other units, while the globals are shared. The dividing line is implementation-specific, but an implementation with 96 globals and 32 locals in each of the four function units has a total of 224 registers—nearly as many as an IA-64 processor. With 32 globals and 96 locals, a MAJC processor would have 416 registers.

With a traditional split register file, an architecture essentially uses the opcode as the high-order bit in selecting a register: FP instructions use one register file, integer instructions the other. With local register addressing, MAJC instead extends the nominal 7-bit register address, using the location of an instruction within its packet, which determines the function unit it will be assigned to. This method gives the compiler access to a great number of registers without extending the instruction length with larger register-index fields.

Large register files can be a problem at context-switch time. MAJC includes several dirty bits, each assigned to a block of 32 registers, that indicate whether any registers in that block have been written to. The compiler doesn't need to save blocks of registers that haven't been used. To enforce modest register usage, the processor can trap any accesses beyond the initial 32 registers. This mode can be used to call a subroutine, such as a device driver, without fear that that code will trash important data in the upper registers.

With four-operand (e.g., MULADD) instructions, the 7-bit register identifiers add 8 bits to a traditional RISC encoding. To maintain the same 32-bit instruction width, Sun was forced to cut back on the features of some instructions. Sun feels the performance advantage of the large register file is well worth these compromises.

Fixed Address Branches

The MAJC architects have extensive experience with the physical design of high-performance microprocessors. With a blank slate to play with, they added features to ease critical speed paths in future MAJC processors. For example, the branch and call instructions use a semiabsolute target address, as Figure 1 shows. The 12-bit target offset encoded in the instruction directly selects the lower address bits of the target; in most architectures, this offset is added to the program counter to form a PC-relative target address.

The 12-bit offset is enough to select from 4K instructions, or 16K bytes. Two additional bits (the relative offset, shown as "roff" in Figure 1) are added to the PC to allow branches to point to the previous 16K, the current 16K, or the next 16K. This extra field is required to concisely encode short branches that happen to cross a 16K boundary. Although this approach seems convoluted, the compiler can easily generate the required encodings.

The advantage of semiabsolute addressing is that the target offset can immediately be fed back to the instruction cache to fetch the target instruction. PC-relative addressing inserts an adder into this path, which is often a critical speed path in a modern processor. The MAJC scheme still requires an adder to calculate the upper address bits, but these bits are not needed to access a cache with a set size of 16K or less. This limits the maximum practical instruction-cache size to a four-way set-associative 64K, which should be adequate for embedded processors for the next several years. This limit could have been increased by expanding the target-offset field, but the 32-bit instruction width is a constraint.

Like IA-64, MAJC includes two prediction bits with each branch. One provides the compiler's prediction of the branch's path (taken or not taken), and the other indicates whether hardware or software prediction should be used. With the latter option, the processor relies on the compiler's prediction in all cases, avoiding the need to consume the hardware prediction tables with easy-to-predict branches. Some second-generation instruction sets include a single prediction bit to initialize the hardware predictor, but they cannot override that predictor the way MAJC can.

The basic branch instruction (B) has only two conditional forms: branch if zero and branch if not zero. (An unconditional branch is encoded using R0 as the source, since R0 is always zero.) Again, the limited instruction width prevents the inclusion of additional conditions. For more complex conditions, a set of compare (CMP) instructions tests whether integers or FP values are greater than, equal to,

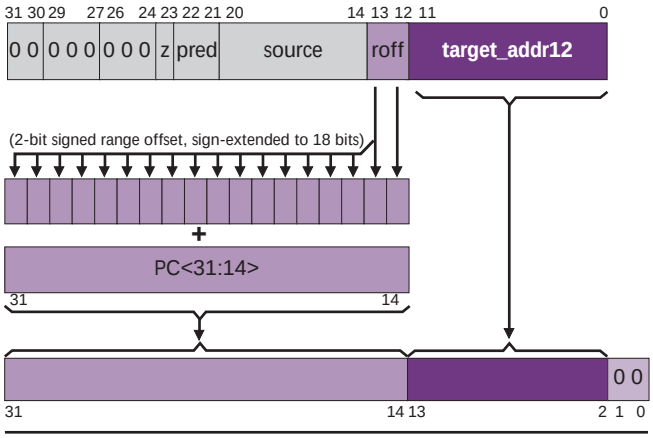

Figure 1. MAJC's semiabsolute branch targets combine a fixed target offset with a small relative offset (roff) that is added to the program counter (PC).

For More Information

Sun has not yet announced any MAJC-based products. For more information on the instruction set, access *www.sun.com/microelectronics/MAJC*.

or less than each other. The processor stores the result of a CMP as a logical value in any of the general registers, which can then be tested using the conditional branch instruction. These compare results are also used by the conditional move, pick, and store instructions. This method consumes general registers to hold logical values, but it avoids the need for a separate set of 1-bit registers, as in IA-64.

Subroutine calls also require a bit of extra overhead. The CALL instruction uses a 26-bit semiabsolute offset that allows access to the full 32-bit address space by forcing all subroutine targets to be 32-byte aligned. To hold this large offset, however, the CALL encoding omits any register specifiers; thus, the return address is always stored in R2. Nested subroutines must save R2 before the next CALL. Alternatively, the jump-and-link (JMPL) instruction can save the return address in any register, but the target address must be placed in a general register before executing this instruction.

For extra performance, MAJC includes an optional prepare-to-jump instruction that starts fetching instructions from the specified address and feeding them to the front of the pipeline. The decoded instructions are held in a temporary buffer until a fast jump (FASTJMPL) initiates execution of the new instruction stream. This instruction sequence permits zero-overhead branches when the target address can be precalculated. In contrast, the normal JMPL will take several cycles to fetch the target address from a register and redirect the instruction-fetch logic.

MAJC includes a unique instruction, BNDCHK, that does three checks at once. The instruction tests whether a value is less than zero or greater than a preset limit while also testing whether a second value is zero. BNDCHK traps if any of the tests are true. This instruction can simultaneously verify that a Java array index is valid and that the object pointer is not null. Other architectures require a series of three conditional branches (or, in the case of IA-64, three predicate-generating comparisons) to perform the same set of tests, requiring at least three instructions.

Flexible Memory Architecture

MAJC's load and store instructions can handle a variety of data types, ranging from bytes to doublewords (64 bits). Data types smaller than the register width are sign extended or zero extended, depending on the opcode, to fill the register. Data types larger than the register width are stored into consecutive registers. The LD.G instruction loads an implementation-dependent number of registers. MAJC provides base+register and base+offset addressing, where the offset is scaled by the size of the data being loaded. This scaling provides greater dynamic range but forces all data to be aligned to its natural size.

A MAJC processor contains two ASI (alternate space identifier) registers that specify memory options such as loading to a specific level of the memory hierarchy, bypassing the cache entirely, flushing cache entries, and accessing I/O devices. Load and store instructions can optionally specify one of these registers to modify the function of the instruction. This tactic avoids using opcode bits to specify these options, which would have expanded the instruction width. The ASI registers must be set up in advance, however, and applications that need to frequently change the ASI options will be burdened.

Like IA-64 (and SPARC v9), MAJC includes a nonfaulting load. This option lets the compiler hoist loads above conditional branches, hiding the latency of the load. If a fault occurs, this load does not trap but, unlike SPARC v9's nonfaulting load, it does save the fault status. Sun would not disclose how this status is saved or checked, but the MAJC method is probably similar to IA-64's "NaT" bits (see MPR 3/8/99, p. 16), which are attached to each register to indicate whether its data is valid.

MAJC includes atomic swap and compare-and-swap memory operations that can be used to manage software semaphores. These are particularly critical in Java environments, which typically have many concurrent threads synchronized via semaphores. MAJC permits these semaphore instructions to coexist with other load and store operations in the memory pipeline; the processor ensures there are no address conflicts. Most processors drain the memory pipeline before executing an atomic operation, reducing performance when many semaphores are used.

To efficiently support dynamic code generation for just-in-time (JIT) Java compilers, MAJC has a store-instruction (STI.W) instruction. This special store automatically bypasses the data cache and invalidates the corresponding entry in the instruction cache, ensuring that the new instruction will be executed properly when referenced. Most other processors require a lengthy series of cache-flush and synchronization operations to correctly execute self-modifying code.

Basic and Extended Math

In MAJC assembly code, basic math operations—such as add, subtract, multiply, and divide—use the same mnemonics for integer and floating-point values, as Table 1 shows. Mnemonic extensions specify whether the data type is 32-bit integer (.I), 64-bit integer (.L), single-precision FP (.F) or double-precision FP (.D). An extensive set of convert (CVT) instructions transforms register values from one format to another. Add and subtract also sport a saturating option (.S) that is useful in multimedia algorithms. MAJC is unusual in supporting multiply-add and divide on the integer side, but this support is almost free, due to the unified register file and function units.

MAJC includes some math operations solely for floating-point values. These include minimum, maximum, square root, and reciprocal square root. The latter two deliver full-precision results rather than the partial estimates generated by some recent instruction sets. The CLIP.F instruction is similar to BNDCHK in that it checks whether a source value is within the bounds of ±RS2, where RS2 is a preset limit stored in a register. This instruction performs two comparisons at once and is useful in some graphics functions.

In addition to the usual logical and shift instructions, MAJC provides trendy new instructions such as byte shuffle, count leading zeroes (CCCB), and count ones (POPCOUNT). These accelerate cryptography, table lookups, and the variable-length decoding portion of video decompression.

MAJC has extensive support for parallel 16-bit integer arithmetic. A 32-bit implementation can hold two such values per register; a 64-bit implementation, four. In addition to basic arithmetic and shift instructions for this data type, MAJC offers average (MEANP) and exponential (POWERP) calculations. The PDIST instruction, similar to that in SPARC's VIS (see MPR 12/5/94, p. 16), is useful in video compression. Two DOT instructions accelerate the dot-product calculations used in many signal-processing algorithms. Sun's long experience with VIS was critical in the definition of these parallel integer instructions.

In contrast, MAJC has no support for parallel single-precision FP operations, such as those in Intel's SSE (see MPR 3/8/99, p. 1) and IA-64. One instruction, RECSQRTP, calculates two reciprocal square-root estimates in an unusual 16-bit (S2.13) fixed-point format. Because the initial MAJC chip uses 32-bit registers, parallel single-precision FP has no benefit. Sun may add instructions of this type for future 64-bit MAJC chips. As Intel's experience shows, such instructions can deliver sizable performance increases on 3D graphics and other applications.

Name	Description	Name	Description	Name	Description
Integer Arithmetic		**Parallel Integer Arithmetic**		**Memory Transfer**	
ADD.I/.L	Add integer/long integer	ADDP(.S)	Parallel add (with saturation)	LD.B/.H/.W/.D	Load byte/half/word/double
ADD.S	Saturating add	SUBP(.S)	Parallel subtract (w/sat.)	LD.UB/.UH/.UW	Load unsigned byte/half/word
SUB.I/.L	Subtract	MULHI.S	Multiply high halfwords	LD.G	Load group
SUB.S	Saturating subtract	MULLO.S	Multiply low halfwords	LD.*sz1*.A1/A2	Load from alternate space 1/2
MUL.I	Multiply	MULADDP	Parallel multiply-add	LD.*sz1*.NF	Nonfaulting load
MUL.H/.UH	Multiply low/high halfwords	MULADDP.S	Parallel mul-add with sat.	PREFETCH	Prefetch data from memory
MULADD.I	Multiply and add	DOTADDP	Parallel dot-product w/add	ST.B/.H/.W/.D	Store byte/half/word/double
MULSUB.I	Multiply and subtract	DOTSUBP	Parallel dot-product w/sub.	ST.*sz2*.A1/A2	Store to alternate space 1/2
DIV.I/.L(.U)	Divide (unsigned)	POWERP	Parallel exponent: A^B	STC .*sz2*(.A1/.A2)	Conditional store (to alt space)
ADDLO	Add 16-bit immediate	MEANP	Parallel average: A/2+B/2	STCP	Parallel conditional store
SETLO	Store 16-bit immediate	PDIST	Pixel distance: Σ|A-B|	STI.W	Store instruction word
SETHI	Store immed to high 16 bits	SH.LL(.D)	Shift logical left (64-bit)	BLKZERO	Zero one cache line
CMP.*cond2*	Compare integers	SH.RL(.D)	Shift logical right (64-bit)	SWAPW	Atomic swap with memory
CMPL.*cond2*	Compare long integers	SH.RA(.D)	Shift arithmetic right (64-bit)	CASW	Atomic compare-and-swap
MOVC.*cond1*	Conditional move	CMPP.*cond2*	Parallel compare	**Data Conversion**	
PICKC	D=A if C=0; else D=B	MOVCP	Parallel move conditional	CVT.DI/.FI	Convert float (DP/SP) to int
AND	Logical AND	PACK	Pack into parallel halfwords	CVT.DL/.FL	Convert DP/SP to long int
OR	Logical OR	**FP Arithmetic**		CVT.IF/.LF.*rnd*	Convert int/long to SP float
XOR	Logical XOR	ADD.F/.D.*rnd*	FP add (SP/DP)	CVT.ID	Convert int to DP float
SH.LL(.D)	Shift logical left (64-bit)	SUB.F/.D.*rnd*	FP subtract (SP/DP)	CVT.LD.*rnd*	Convert long int to DP float
SH.RL(.D)	Shift logical right (64-bit)	MUL.F/.D.*rnd*	FP multiply (SP/DP)	CVT.DF.*rnd*	Convert DP float to SP
SH.RA(.D)	Shift arithmetic right (64-bit)	DIV.F/.D.*rnd*	FP divide (SP/DP)	CVT.FD	Convert SP float to DP
BITEXT	Extract bits from register pair	MULADD.F/.D	FP multiply add (SP/DP)	CVT.FX	Convert float to fixed-point
BYTESHUFFLE	Rearrange bytes	MULSUB.F/.D	FP multiply subtract (SP/DP)	CVT.XF	Convert fixed-point to float
CCCB	Count leading zeroes	SQRT.F/.D.*rnd*	FP square root (SP/DP)	CVTP.IF.*rnd*	Parallel convert int to float
POPCOUNT	Count ones	RECSQRT.F	FP reciprocal square root	**System Control/Miscellaneous**	
Control Transfer		NEG.F/.D	FP negate (SP/DP)	FLUSHI	Flush instruction pipeline
B.*cond1*	Conditional branch	ABS.F/.D	FP absolute value (SP/DP)	MEMBAR	Flush memory pipeline
CALL	Subroutine call, link in R2	MAX.F/.D	FP maximum (SP/DP)	GETIR	Read internal register
JMPL	Jump indirect and link	MIN.F/.D	FP minimum (SP/DP)	SETIR	Write internal register
PREJMPL	Prepare to jump	CMP.F/.D.*cond3*	FP compare (SP/DP)	SOFTTRAP	Software-initiated trap
FASTJMPL	Fast jump-and-link	CLIP.F	FP clip test: -B<A<B?	SIR	Software-initiated reset
BNDCHK	Bounds check and trap	**Parallel FP Arithmetic**		TRAPC	Trap if necessary
DONE	Return from exception	RECSQRTP	Parallel recip sqrt 16-bit FP	YIELD	Yield to another thread
RETRY	Return to excepting instr	CMPP.F.*cond3*	Parallel compare SP float	PAUSE	Wait for interrupt

Table 1. The MAJC instruction set. *cond1*=Z or NZ (zero, not zero); *cond2*=EQ, LT, LE, or ULT (equal, less than, less than or equal, or unsigned less than); *cond3*=EQ, LT, LE; *rnd*=RN, RZ, RH, RL (round toward nearest, round toward zero, round high, round low); *sz1*=B, H, W, D, UB, UH, UW, or G (byte, halfword, word, doubleword, unsigned byte, unsigned halfword, unsigned word, group); *sz2*=B, H, W, D.

OS Support for Multiple Threads

MAJC has the usual privileged instructions to flush caches, force memory ordering, access internal registers, and reset the processor. Exception handlers can return to the main routine using DONE or RETRY. The latter backs up the return address by one packet, reexecuting the packet that failed. This feature could be used if, for example, an I/O device had an error that was later cleared.

The YIELD instruction provides support for thread switching. MAJC processors can optionally store the context of up to four threads. Upon encountering a YIELD instruction, the processor can switch to another thread if one is available. This method can overlap thread execution to take advantage of situations that the compiler thinks may cause a long stall in the hardware.

MAJC Delivers Embedded Power

There is no apparent reason that MAJC would not be appropriate for high-end servers as well as embedded systems. Although it does not have all of the high-performance features of IA-64, MAJC appears superior to the aging SPARC architecture (and other RISCs) for high-end applications. MAJC has more registers, more conditional instructions, and better speculative loads than SPARC as well as other innovative features SPARC lacks. MAJC lacks one crucial thing: the extensive software base that SPARC has today and IA-64 will have in the near future. Thus, Sun remains committed to SPARC CPUs in its workstations and servers.

Sun's intent is to move MAJC into the embedded space, where having a software base is far less important. In this space, SPARC has had little success, and MAJC should do better. The new instruction set should deliver better performance—particularly for graphics, signal processing, and Java bytecode—than SPARC, without a significant increase in code size or die size. MAJC is well suited for applications such as set-top boxes, cellular base stations, and digital cameras that demand powerful processors.

MAJC is not as well suited for embedded applications that demand minimum cost or minimum power. With its large register file and general-purpose function units, a MAJC core is likely to be larger than a simple RISC core, increasing die cost and power consumption. In addition, MAJC code sizes will be bigger than those for many of the popular embedded architectures that use instructions of less than 32 bits, increasing memory costs. In some applications, however, the differences will be small enough to justify the higher performance.

In devices that primarily execute Java bytecode, these performance differences may be substantial. MAJC's support for fast semaphores and fast thread switching improve performance on Java code, which typically has many threads. The BNDCHK instruction accelerates array-index checking, which is done in real time in Java. The STI.W instruction speeds JIT compilers, which are popular Java tools. For these reasons, a MAJC chip should outrun a comparable traditional processor on Java code. MAJC's benefit will be reduced, however, if programmers compile their Java programs directly to the traditional processor's assembly code.

Sun's new architecture takes the basic concepts of the most modern instruction sets and modifies them to better suit a range of embedded applications, adding a few new tricks along the way. Among VLIW architectures, it is the most appropriate for these embedded systems. MAJC provides a strong tool that will help Sun expand its presence in the embedded-processor market. M