Jaguar Architecture
Derived from the Motorola 88000

This is a specification of changes to the 88000 architecture required for the Jaguar family of systems. Architectural and implementation issues are divided into the categories of: base architecture, implementation dependent privileged operations, implementation dependent user operations, and implementation mechanisms. The base architecture is the environment seen by most system and user code. It is implementation independent. Implementation dependent privileged operations are performed by a limited, kernel portion of the operating system. They include access to virtual to real address translation hardware, saving state information and launching processes. Implementation dependent user operations are intended for use by a very limited number of low level routines. They may be used, for example, to enhance the performance of certain graphics or digital signal processing operations by making special use of the capabilities of a particular implementation. Implementation mechanisms are the specifications and operating details for a particular implementation. These include performance and hardware configuration specifications. They also include the details of how, for example, a combination of hardware and kernel software is used to handle some cases of IEEE floating point arithmetic. This document covers the base architecture. Implementation dependent features for the first Jaguar microprocessor are presented in the companion Cheetah Specification.

Two approaches for changes to the 88000 architecture are presented. The first approach sacrifices binary compatibility with the existing 88000 in order to optimize the resulting architecture. The second approach is to retain binary compatibility, with some compromise of the additions and changes and some increase in complexity. Apple's proposal is that a detailed architecture, but not a device, specification be developed for each approach, and an effort be made to convince key existing customers to accept the optimized, binary incompatible version.

The compatible version will contain new instructions, most notably for floating point operations, which are intended to supersede the existing mechanisms for similar operations. The existing instruction encodings would be retained. However, it is important that most such instructions in fact trap and are emulated by kernel software (which may make use of the new instructions). Superseded instructions should be emulated in implementations of interest to Apple so that the cost and performance of these parts is not adversely affected. Superseded instructions should be emulated in all implementations, including those not of current interest to Apple, because other customers will want to use one version of their applications software across the 88000 product line. In these cases it is important to eliminate pressure to optimize the performance of the superseded instructions.

If the compatible approach is adopted and superseded instructions are emulated, most current 88000 customers will treat Jaguar implementations as binary incompatible. They will recompile rather than suffer any performance degradation. The conflicts which will arise with the superseded instructions are one of the principal reasons that we propose the binary incompatible approach.

With either the binary compatible or incompatible approaches, its not contrary to Apple's interests if there are versions which implement subsets of the instructions which Jaguar uses. For example, a design might not implement extended precision or any floating point operations in hardware.

The architectural issues and changes are grouped into the categories: integer register operations, floating point register operations, and comparison and program flow operations.

Following these is a summary list of all changes for the binary incompatible and compatible approaches.

The 88000 has (32) 32 bit registers which are used for all types of data. For Jaguar these are considered integer registers and are used for address arithmetic and most non floating point data operations. Separate floating point registers are added, as described later. The floating point registers accommodate 32, 64 and 80-128 bit entities. The floating point units will typically be implemented with the widest data paths. For this reason, and for maximum parallelism, some graphics and data movement operations are defined to use the floating point registers.

Integer Register Operations

Memory Reference

The 88000 architecture is register oriented, with fixed 32 bit instructions and a three register address format for most register to register operations. Register zero is hard wired to be zero. In this context there should be register + displacement and register + register addressing for load and store of bytes, half words and 32 bit words. The displacement should be approximately 16 bits. Byte ordering should be big endian; 0,1,2... Bit ordering can be big endian or little endian. Loading and storing of two or more registers with a single instruction is a minor performance enhancement; its assumed that there are separate floating point loads and stores which are also the most efficient way to move multiple words of data. Address scaling and sign extension are discussed in the following sections.

The 88000 provides register + displacement and register + register addressing for load and store of bytes, half words, words and 64 bit register pairs. The displacement is a 16 bit positive byte displacement. A programmable option allows 88000 byte ordering to be big endian or little endian. This has a minor impact on the current implementation of the 88000; but extra data path multiplexing will be required for versions where the data path between registers and cache is wider than 32 bits. Our position on the little endian option is that it should be included if it really does widen the appeal of the architecture; but multiple word little endian operations should require extra cycles in a future implementation if the cycle time would otherwise be lengthened.

Addressing modes with auto updating of the base register would be useful. In these modes, designated as register + register++ and register + displacement++, the results of the addition are stored into the (first) base register. The (second) index register or displacement is a stride or auto increment value. For array operations its not very important whether the effective memory address is the contents of the base register or the result of the computation. For stack push and pop operations, it would be nice to have both. Motorola proposes that the base register be the effective address if the displacement or index register is positive, and the result of the computation be used if the displacement or index register is negative.

Auto updating addressing modes don't require additional arithmetic, only the storing of the address calculation. However, recovery from exceptions such as virtual to real address translation page faults may be more complex for some implementations. Also, auto address updating instructions perform an (additional) store operation to the integer register file, and for integer memory reference operations, a more complex implementation may be required in order to realize the performance gain which eliminating separate address updating instructions could provide. Our position on these operations is pending further analysis of their near and longer term effects on implementations.

There should be no architectural restrictions on the range or intermingling of code and data within address spaces. Architecturally, a system call should be required before data or self modifying code is executed. A user address space should be mappable into the system address space, without architectural restrictions on location. I/O should be mappable into any address space, without architectural restrictions on location.

The 88000 provides the desired architectural attributes for system, user and I/O address spaces. It also allows the user address space to be a separate 4GB area with system access through special load and store instructions. This mechanism requires data to be specified as system or user at compile time. Jaguar maps a user address space into the system address space, and won't use the special load and store instructions. Motorola and Apple agree that these operations should be eliminated in order to save instruction encoding space. Apparently they haven't been used and can be deleted without causing compatibility problems.

Instructions should be aligned on word boundaries. The memory reference instructions should require that data be aligned on its size boundary; half words on half word boundaries, words on word boundaries, etc. However, unaligned accesses will be required in order to use some old data. When memory reference instructions trap on an unaligned access, they should provide the information necessary for kernel software to perform the access transparently to the offending code. But the preferred mechanism for unaligned accesses should be to perform them in line with a short sequence of instructions. In the Jaguar environment, the need to access unaligned data is known at compilation, and special effort should be required to get compilers to create unaligned data items. An application's typical response to encountering old, unaligned data should be to request the user's permission to make a new, reformatted copy.

The 88000 requires instruction and data alignment. It traps on unaligned data accesses. Unaligned data can be efficiently accessed by alignment trap software transparently to the offending code, or with an in line sequence of instructions.

Address Scaling and Relative Addresses

Immediate displacements can be scaled to squeeze out more reach; but it doesn't matter much if they are as large as 16 bits. All pointers should be byte addresses. This means that if there is register + displacement addressing, there should be a form which doesn't scale the register. One register could be scaled in register + register addressing; ideally there is also an unscaled form. With good compilers, scaling registers will not be a big advantage; the benefits and costs are approximately equal.

The 88000 has register + displacement (unscaled), register + register (unscaled) and register + scaled register addressing.

The 88000 has a load relative address instruction. It has the same addressing modes as the memory reference operations. These don't include program counter relative. All of its operations are duplicated by other single instructions except register + scaled register for half words, words and double words. Motorola proposes to delete most of the duplicated load relative address forms, retaining register + scaled register. The deleted forms have not been used by compilers. As a minor point, we propose that the remaining forms of load relative address also be deleted for the binary incompatible approach. They can be replaced by two instructions and are believed to not be used frequently enough to optimize.

Sign Extension of Immediates

For memory reference, sign extension gives more flexibility with little impact if the immediate field is wide enough. It can also be used to efficiently implement a data typing mechanism where the sum of a pointer and the displacement being aligned indicates that the type matches.

The 88000 has 16 bit positive byte displacements. Motorola proposes to add new load and store instructions with signed 10 bit displacements. If auto base register updating instructions are added, the register + displacement++ form would also have 10 bit signed displacements. For the binary incompatible approach, we propose that the existing memory reference instructions be changed to have signed displacements. All memory reference operations should have the same size displacement, which could be somewhat less than 16 bits to make room for added instructions.

We consider having signed displacements for memory reference instructions very desirable. As discussed, we are continuing to investigate the advantages and disadvantages of our proposal for memory reference instructions with auto base register updating. The new loads and stores described in the floating point section are a must. Having more than 10 bit displacements for the added instructions and having simple, uniform and non duplicative memory reference instruction formats are important reasons why we propose the binary incompatible approach. The instruction format issues simplify implementations, and they are partly aesthetics. The latter consideration is significant in winning customers.

For arithmetic, it doesn't matter if there is sign extension or not given that there are a complete set of operations (eg. both add and sub) and the immediates are large (16 bits). More on overflow and carry later. The 88000 doesn't sign extend arithmetic immediates which is consistent with its current handling of memory reference displacements. As a minor point, for the binary incompatible approach we propose that these be changed to be sign extended to maintain consistency.

For boolean operations, the 88000 has "upper" and "lower" application of 16 bit immediates. The immediate operands are extended on the left or right with zeros, and also with ones for and.

Sign Extension of Bytes and Half Words

Bytes and half words are usually being manipulated as characters or other non-integer entities. They should be loaded right justified with zero fill. Then there should be suitably efficient means of sign extension for when integer arithmetic or comparison is to be performed on these 8 or 16 bit quantities.

The 88000 loads bytes and half words as above, and also with sign extension. Its clever extract field and shift right arithmetic instruction can be used to sign extend a byte or half word that didn't come directly from memory. (The shift count can be 0-31 and the field width 1-32. This allows any partial word quantity to be sign extended, in place or shifted right.) For the binary incompatible approach, we propose to eliminate the sign extending versions of byte and half word loads in order to reduce complexity and save instruction encoding space.

Integer Arithmetic

Signed and unsigned 32 bit add, subtract, multiply and divide are required. All of these operations should be available in register x register and register x immediate forms. The 88000 is missing signed multiplies. Motorola proposes to add signed register x register

multiply, but not an immediate form, or at least not one with the same sized immediate as others, because of instruction encoding issues. Our position is that this is an open issue. For the binary incompatible approach, divide immediates would be preferable to delete if something must be omitted.

The exception conditions for integer arithmetic are overflow and divide by zero. Signed operations should affect a sticky overflow flag. There should also be a sticky divide by zero flag, and trap enable flags for overflow and divide by zero. With these, exception handling for integer operations is similar to floating point. This integer status is similar to the equivalent floating point status. It should be transferable to and from an integer register, and it is standard enough that the bit patterns can be made part of the base architecture. This allows the results of an individual operation to be tested without a trap with reasonable efficiency. We have deleted our original proposal to have separate non sticky flags for this.

Like floating point, in normal (parallel) mode, unrelated instructions following a trapping instruction may have been executed (the instruction and operands causing the exception and the correct return location are always provided). There should be a serial mode which guarantees that following instructions have not been executed. This would typically be used when stepping through a program with a debugger. It should be a privileged status bit.

The 88000 currently detects overflow for signed operations and divide by zero, but these traps can't be disabled and they aren't sticky. Parallel/serial mode is a privileged status bit which is defined for floating point, but not explicitly for other operations. Motorola proposes to add all of the desired exception handling, and expand the definition of parallel/serial mode. They also propose that unsigned multiply affect the overflow flag. It should not do this; its purpose is for operations where overflow should be ignored. We initially proposed that unsigned multiply "overflow" into the carry flag for symmetry with other unsigned operations, as described below. But there is no important reason to do this, and future implementations may otherwise be simplified.

Basic building blocks should be provided for extended precision integer arithmetic. The intent is to provide moderate support for precisions greater than 32 bits, and not 64 bits or any other in particular. A reasonable set of operations is:

> A carry out of 32 bits flag which is set by the unsigned register x register forms of add and subtract. The flag is set with the electrical carry out for all operations; (the compliment of borrow for subtraction).
>
> Register x register forms of signed and unsigned add and subtract which use the carry flag as input. In addition to supporting extended precision arithmetic, these can be used to load the carry flag into a register for testing the results of unsigned operations.
>
> A register x register form of unsigned multiply which leaves a 64 bit result in a register pair.
>
> A register x register form of unsigned divide which divides a 64 bit register pair by a 32 bit register and leaves 64 bits of result/remainder in a register pair.

A less desirable option would be a special register for extended multiply and divide to use. Within reason the speed of extended multiply and all divide operations is not important (eg. one bit per clock for divides and extra clocks for accessing and saving register pairs).

The integer overflow flag is only affected by signed arithmetic. The carry flag is only affected by unsigned arithmetic. The carry flag is not affected by unsigned operations with immediates, which are normally used for address calculations, or by address arithmetic within memory reference instructions.

The 88000 has the extended precision oriented add and subtract operations described above. It also has signed add and subtract which modify the carry flag and ones which input and modify the carry flag. It has unsigned add and subtract which don't modify the carry flag and ones which input but don't modify the carry flag. These additional operations add negligible complexity and consume insignificant instruction encoding space.

Motorola proposes to add signed and unsigned versions of extended precision oriented multiply and divide. However, the extended divide should produce a 32 bit potentially incomplete product, and a 32 bit remainder. Adding a signed version of this divide is a better idea than our original proposal to also have signed and unsigned remainder instructions. If the extended precision signed multiply is added, the cases of either operand being negative should trap to a kernel software implementation. Our position is that adding this instruction is a minor open issue.

We originally proposed that signed and unsigned 32 bit remainder instructions be added. Having both signed and unsigned extended precision division operations is sufficient. The remainder that they return could be chopped (rounded toward zero, rem(-5/2) = -1) or a floor (rounded to $-\infty$, rem(-5/2) = 1). Chop remainder is probably the best choice because it is more common and defined by some languages. Whichever is provided, either can be calculated from the other. We propose to not add the four separate remainder operations suggested by Motorola in response to our initial request.

Comparison operations are entwined with each architecture's mechanisms for conditional branching. These are discussed later.

Boolean Arithmetic

The basic and, or and xor operations are required. There should be register x register, and a complete set of immediate operations. With a fixed 32 bit instruction format, applying a 16 bit immediate to the upper or lower half of a register is desirable. The 1's complement not operation is infrequent, and can be emulated by xor with ones.

The 88000 has the and, or and xor operations described above. It also has register x 1's complement register for all operations, and immediates with both zero and ones fill for and.

Bit and Shift Operations

Left and right with zero fill, arithmetic right, and rotate, of 32 bits are required. The count should be a literal or taken from a register (computed). Its desirable to have efficient extraction of fields for emulating instructions. Other bit manipulations are frequently provided, but are not used enough to justify their existence.

The 88000 provides the left, right, arithmetic right and rotate operations described above. It performs the left, right and arithmetic right operations with very clever extract and shift instructions. Specifying a field width of 32 bits results in a basic shift. The extract and shift arithmetic right operation can be used to sign extend any sized partial word quantity. The 88000 also has set and clear field and scan for first set or clear bit instructions. As a minor

point, we propose that the set and clear instructions be deleted for the binary incompatible approach. We would also delete the scan instruction, but accept it since it is desired by other customers and its cost is small.

Floating Point Register Operations

This section deals with the floating point register and arithmetic operations which are part of the base architecture. There is also a class of special user operations which are implementation dependent. This includes pixel and digital signal processing operations. These are presented in the companion Cheetah specification for the first implementation.

Register Structure

The 88000 has (32) 32 bit registers which are used for all types of data. For Jaguar these are considered integer registers and are used for address arithmetic and most non floating point data operations. Separate floating point registers are added. This increases the amount of execution parallelism that can be achieved with a given level of design complexity. For most Jaguar implementations, the performance of floating point operations will approach that of integer arithmetic. This will result in many more applications using floating point data representations. The benefits of the increased register storage are worth the costs for these applications. The increase in time to save or restore state information is an insignificant part of the overall execution time, and an insignificant part of the time consumed by the operating system related to process switching. Most implementations will minimize save and restore time by only saving or restoring the floating point registers when they have been modified.

Single, double and extended precision floating point data must be supported. The registers should be able to contain at least 16 of any of the data types, with one of the 16 a hard wired zero. The architecture should be able to reference registers containing 32 of any of the data types. In addition it is desirable that single and double operands can make maximum use of the available register storage. In foreseeable implementations, the extended precision will be 80 bits; however it should be transparently extendible up to 128 bits. Double and extended operands should be aligned on appropriate boundaries in registers (and on two and four word boundaries respectively in memory). The instruction encoding of register addresses should allow the same full width operands to be accessed regardless of the sizes that will actually be used.

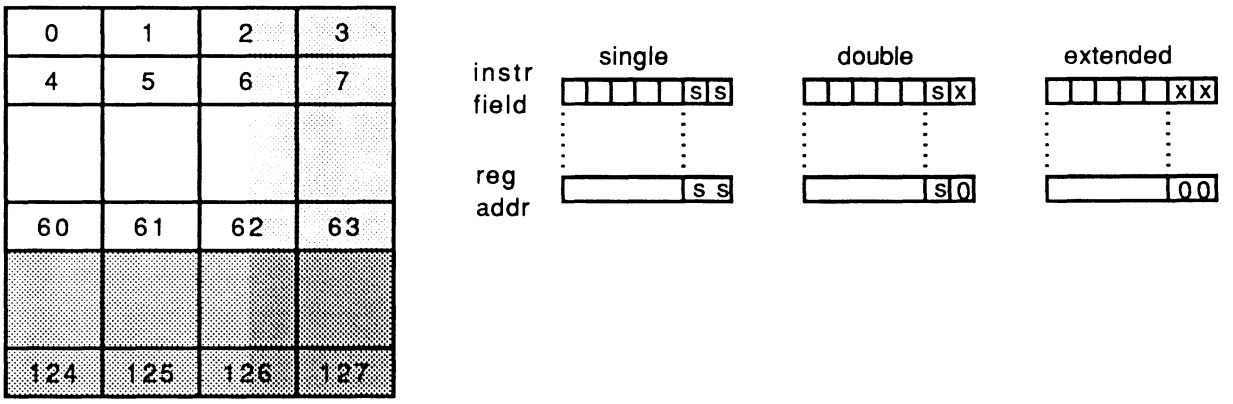
In the following figure, (a) provides for all of the above requirements and desires. The shaded areas are the possible, forward compatible extensions. In the instruction field to register address mapping, (s) bits select a portion of a location and (x) bits are unused but should be zero for future compatibility. The problem is that three 7 bit fields are required for a three register address architecture. This would use too much of the instruction encoding space. A reasonable alternative is a two register address structure. However, the possible extensions are not very likely to be implemented. If they are, access to more than 32 single or double operands would be most useful to graphics and signal processing applications which use special, implementation dependent operations. Thus, access to more than 32 single or double operands could also be implementation dependent. Accepting this as the case makes (b) the favored architecture.

(b) in its basic (b1) form supports all of the requirements and desires. (b2) and (b3) are the possible extensions of (b1) which support all of the requirements. However, (b3) does not allow maximum use of the available registers for single operands and (b2) does not allow maximum use of the available registers for single or double operands. (b2) is the most likely extension. It supports 32 operands of any type; and the extended precision format can separately be extended to 128 bits. (b3) extends the extended precision format to 128 bits, and increases the number of double operands to 32. (b2) and (b3) are forward compatible extensions of (b1), and a full width (b2) is a forward compatible extension of (b3). Of course

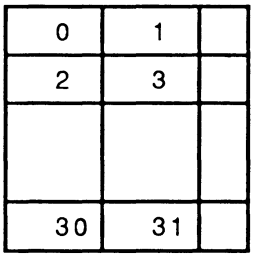
nothing so horrid as loading single r1 and depending on that to replace part of the mantissa of double r0 is allowed.

Floating Point Register Structures

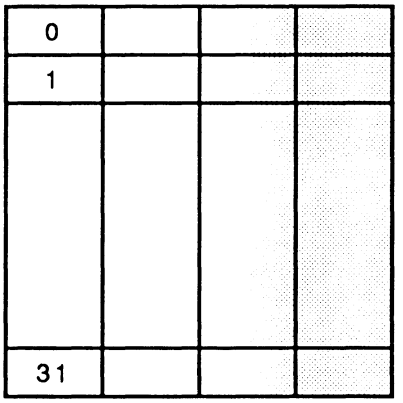
(a)



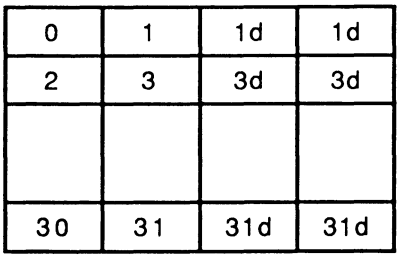
(b1)



(b2)



(b3)



Motorola proposes to add floating point registers similar to (b2). This is certainly acceptable, and advantageous for double and extended precision. During the early design phase we should both confirm that the reduced multiplexing and access time and program efficiencies are worth the additional 1280 bits of storage relative to (b1) for the initial and future implementations.

Execution Model

Typical Standard Apple Numerics Environment (SANE) applications represent primary data as single or double precision quantities, but internal calculations and temporaries are extended precision. Apple's lower level graphics and signal processing operations will only use single or double precision representations and operations. We want the Jaguar architecture to provide the opportunity for the best performance for typical SANE applications, and also provide the opportunity for the best performance when only single or double precision calculations are required. Most other 88000 customers will be primarily interested in single and double precision performance.

Conversion from single or double to extended precision requires exponent arithmetic and simple multiplexing. Conversion from extended to single or double precision requires exponent and rounding arithmetic. The proposed operations are designed to make it as easy as possible to reduce extra cycles for these conversions by eliminating the extra cycles or overlapping them with other operations. For maximum single and double performance, calculations must be able to be performed to these precisions directly. This avoids any conversion time, and the extra time that many implementations will require for operations such as multiply and divide in extended precision.

The following table summarizes the operand precisions and conversions for three possible execution models. Upward conversion includes converting the exponent to the new format. With or without conversion, data is assumed to be multiplexed so as to be aligned in registers as Motorola proposes. With this alignment, mantissa's are left justified for all formats, and though not stated, unused bits are set to zero whenever a register is a destination.

	Load (from memory)	Execute	Store (to memory)
1.	Source specified; no conversion; 3 combinations.	Each source and result specified; result >= source; 14 combinations.	Source and destination specified; destination <= source; 6 combinations.
2.	Source and destination specified; destination >= source; 6 combinations.	One specification for sources and destination; no conversion; 3 combinations.	Source and destination specified; destination <= source; 6 combinations.
3.	Source specified; no conversion; format saved as tag; 3 combinations.	Result specified; each source specified by its tag; result >= source; 3 (14 effective) combinations.	Destination specified; source specified by its tag; destination <= source; 3 (6 effective) combinations.

All of these execution models support direct operations on single and double precision data without conversion of sources or results. For SANE they combine upward conversion with loading or execution and downward conversion with storing. (1) performs upward conversion during execution. (2) combines upward conversion with load operations. (3) is a tagged

version of (1). Tags could also be used with (2) or other models. The advantage of tags is that less instruction encoding space is required. However, we agree with Motorola that the added complexity is undesirable for minimum and very parallel implementations.

Motorola proposes to always convert to extended on loads, perform all execution in extended, and convert downward on stores. We propose either (1) or (2), depending on instruction encoding and implementation issues. Details are discussed in the following sections.

Memory Reference

The following load and store operations are required for execution models (1) and (2).

Instruction	Source	Destination
load (1)	single	single
	double	double
	extended	extended
load (2)	single	extended, double, single
	double	extended, double
	extended	extended
store	single	single
	double	single, double
	extended	single, double, extended

Extended operands are stored as 128 bits in memory even though fewer bits are implemented in registers. For future compatibility they must be left justified in the four memory words, which differs from Motorola's proposal, and unused bits should be zero.

Downward conversion is part of the store operations. In SANE most stores will do downward conversion and conversions by themselves are infrequent. This is a partial justification for combining the two. Downward conversion requires exponent and rounding arithmetic, and will take one or more cycles. A simple implementation may only save instruction space and bandwidth and not execution time. Combining conversion and store makes it easier for more complex implementations, including ones with multiple execution units, to overlap the time required for the conversion with following operations.

The general capability to execute one or more stores of integer or floating point data out of sequence is necessary for multiple execution unit implementations to work effectively. The performance gain relative to the cost and complexity is also very favorable for less ambitious implementations. Stores perform address calculation and memory management operations in sequence, but are completed when the data to be stored is ready (following load addresses must be checked against any outstanding store addresses). This significantly increases the parallelism between loads, stores, integer and floating point operations in most cases; and, combined with stores, SANE downward conversion can also be overlapped.

In SANE most data needs to be converted to extended when it is brought into CPU registers. It is proposed that these conversions be combined with either the load or execution operations. In most implementations it is believed that these upward conversions won't lengthen or add execution cycles, and the savings in instruction space and bandwidth is worthwhile. The choice between execution models (1) and (2) depends on implementation and instruction encoding issues.

Floating point loads and stores use integer registers for addresses. These instructions should be consistent with the integer register loads and stores in most ways. They should have the same addressing modes, displacements, byte and bit ordering, address space attributes and data alignment requirements. As with integer register loads and stores, the benefits and costs of address scaling are about equal. The balance between the benefits and costs of auto updating addressing modes is more favorable than for integer register operations because different register files are used for data access and address updating.

Motorola proposes floating point loads and stores with register + register, register + signed 10 bit displacement, register + register++ and register + signed 10 bit displacement++ addressing modes. There are no scaled address modes and the auto base register updating modes are the same as those proposed for integer register operations. With auto updating the base register is the effective address if the displacement or index register is positive, and the result of the computation is used if the displacement or index register is negative.

We would like to have wider displacements. With the binary incompatible approach, the integer and floating point memory reference instructions could be reorganized so that they would all have the same displacement formats. We are neutral on address scaling. Our position on auto base register updating is pending further analysis of the near and longer term effects on implementations.

In our original proposal, we suggested load and store 128 bits between memory and a double floating point register pair (separate from load and store extended to extended). The desirability of these operations as part of the base architecture depends on whether most implementations will have the data paths and register bandwidth to realize increased performance. If they are included, only addressing modes without displacements might be provided in order to save instruction encoding space. Our position is pending further analysis of probable implementations.

If we were starting from scratch, the base architecture would have byte, half word and word integer register loads and stores, and single, double, extended and possibly 128 bit double register pair floating point loads and stores. Double or double register pair floating point would be used for moving blocks of data. Multiple integer registers could be loaded and stored for task switching by privileged implementation dependent operations if the benefit outweighed the cost. The current 88000 has 64 bit integer register pair loads and stores. These instructions were important because all operations including floating point used the same registers. For the binary incompatible approach, we propose to keep them as long as instruction encoding space is not an issue for more important operations.

Floating Point Arithmetic

The 88000 architecture is register oriented, with fixed 32 bit instructions and a three register address format for most register to register operations. Single, double and extended register zero should be hard wired to be zero.

Add, subtract, multiply and divide are required. With a firm hand on the lid of Pandora's box of proliferating instructions, remainder and square root should also be provided. Their performance is important for many applications, they are usually fairly easy to implement in conjunction with the basic operations, and they are specified by the IEEE standard. Floating point remainder needs to be interruptible, or defined as a partial remainder step operation, since a pathological case could take a long time. It should also provide the signed low order

integer quotient bits as a four bit 2's complement number. The remainder quotient bits can be accessed along with the other floating point status information. Remainder and square root would typically be implemented as function calls rather than in line if not defined in the basic architecture. Its intended that all implementations will be able to efficiently trap and decode unimplemented instructions Thus, even if an implementation trapped and emulated remainder and square root in kernel software, their performance would not be significantly less than a function call.

Motorola proposes to add all of the above operations for the new floating point registers, but only for extended operations. Single and double operations must be provided within the framework of execution model (1) or (2). For the binary incompatible approach the integer register floating point operations should be deleted.

In our original proposal we noted that the IEEE standard recommends additional operations which are not part of the official standard, and that arguments are made for some of them to be part of the base architecture. These include: scale (x * 2**n), log (exponent of x as a floating point number), next (floating point number nearest x in the direction of y), and class (a code indicating what type of floating point number x is). We wanted to discuss these operations, but they weren't formally requested. Log and next are the most worthwhile. Scale is easily emulated within the floating point registers, and class is very infrequent. Actually all of these operations can be performed by an amount of software which is reasonable given their low frequency. Motorola proposes to add these operations, and also round floating point to nearest integral valued floating point. Our position is neutral to negative on adding any of these operations to the base architecture, except possibly round to integral value. The latter is also infrequent, but is defined as part of the IEEE standard.

Most implementations of the architecture will have separate data paths for floating point addition/subtraction and multiplication. It may be desirable to make a multiply/accumulate instruction part of the architecture. To do so, it must be suitably general, and offer significant performance advantages over the life of the architecture. A likely alternative is that increasingly parallel implementations will accomplish the same effect with the existing instructions. We agree with Motorola that combined multiply/accumulate instructions should be implementation dependent user operations.

Conversion from single, double and extended to 32 bit integer are required. There should be conversion from integer to double or to double and extended with execution model (1). Converting to double and then to extended is identical to a direct conversion. There should be conversion from integer to double and extended with execution model (2). Conversion from integer directly to single is not necessary. It is more complex than converting from integer to double or extended, and conversions either direction between integer and floating point formats are usually infrequent. Converting integer to double or extended and then to single is identical to a direct conversion. Performing conversions between floating point formats with memory operations is acceptable. However specific instructions for these conversions will be aesthetically desirable to many customers.

From a conceptual viewpoint the source of an integer to floating point conversion should be an integer register and the destination should be a floating point register. From a conceptual viewpoint the source of a floating point to integer conversion should be a floating point register and the destination should be an integer register. The implementation is easier if all sources and destinations are floating point registers. If this is the definition then there must be a way to transfer 32 bit integers between memory and the floating point registers. The standard single to single floating point loads and stores are fine as long as the conversion instructions use the appropriate fields within register file entries.

Motorola proposes conversions from all floating point formats to integer, integer to double and extended, and between all floating point formats. The source and destination are floating point registers for all operations. However an integer as a source or result is a 32 bit quantity sign extended to 64 bits. 64 bit integer floating point loads and stores are provided for these. We want to load and store these as 32 bit integers. The Jaguar architecture addresses greater than 32 bit integer arithmetic with the building blocks described in the integer register section. Conversion between greater than 32 bit integers and floating point is done in software. To support 64 bit integers more aggressively we would add a complete set of 64 bit integer operations which would use the integer registers, and conversion operations between floating point and 64 bit integers with all operands in the floating registers for implementation convenience.

Our original proposal suggested instructions to move data both directions between floating point and integer registers. These, and conversions either direction between floating point and integer formats, are relatively infrequent operations. Moving data through memory and using floating point registers as the source and destination for conversions is completely acceptable. This eliminates the need for data path connections between the integer and floating point registers.

Floating point status must include the rounding mode and the five trap on exception enables and associated sticky exception flags specified by the IEEE standard. It should also include the low order quotient bits from floating point remainder. The floating point status should be a special register which can be transferred to or from an integer register, and it is standard enough that the bit patterns can be made part of the base architecture. This allows the results of an individual operation to be tested without a trap with reasonable efficiency. We have deleted our original proposal to have separate non sticky flags for this.

On the current 88000 floating point status is accessed by loading, storing or exchanging integer and control registers, and user code can not read (or write) implementation dependent status. Motorola's proposal for Jaguar is to continue this desirable arrangement, except that an implementation dependent floating point registers modified bit would be accessible by users. We would like this flag to be in a privileged status register.

The architecture must provide complete support of the IEEE standard for floating point and integer data. Any reasonable implementation will use a combination of hardware and kernel software to provide the basic operations. What's in hardware and what's accomplished with the modern equivalent of firmware depends on the cost and performance benefits for a given implementation. As a minimum, most implementations will handle the trap enable and sticky exception flags and almost all cases in round to nearest mode with traps disabled in hardware.

Some time critical procedures may knowingly generate overflows or very small results which underflow or should be denormalized. The normal environment for these routines is round to nearest and all traps disabled. Ideally the hardware will be able to generate the correct IEEE results for these situations without software assistance. This includes Nans, precluding the flexibility of letting the format of these be specified by kernel software as we originally proposed. If the correct IEEE result can't be delivered, then there must be a special mode where something arithmetically acceptable is produced without software involvement. This includes accepting Nans as input (denormalized numbers don't need to be accepted as long as they aren't produced while in the special mode), producing infinities and Nans as specified by the IEEE standard, and producing zero or the smallest magnitude normalized numbers when tininess

occurs. If a special mode is necessary, it should be considered an implementation dependent user feature, and non privileged routines should be able to enable and disable it directly.

Architecturally, during normal (parallel) execution, unrelated instructions following a trapping instruction may have been executed. All information required to implement the IEEE standard, and the correct return location, will always be available. There should be a serial mode which guarantees that following instructions have not been executed. This parallel/serial mode should apply to floating point and other operations. It would typically be used when stepping through a program with a debugger. It should be a privileged status bit.

The current 88000 supports the desired parallel/serial mode for floating point instructions. The issue has otherwise been mute. Motorola will explicitly extend the architectural definition of this capability to cover floating point and other operations.

Comparison operations are entwined with each architecture's mechanisms for conditional branching. Floating point and integer comparisons are discussed later.

Comparison and Program Flow Operations

Conditional and unconditional branches, calls and returns should have an optional delay slot. If the delay slot is specified, then the next sequential instruction is executed regardless of whether or not the branch, call or return is taken. For implementation flexibility, there should be the architectural restriction that the delay slot instruction cannot be a program flow altering instruction or the target of such an instruction. If the delay slot is not specified, then the next sequential instruction is executed only if the (conditional) branch is not taken. There shouldn't be any restrictions on the type of operation performed by the next sequential instruction in this case.

Conditional branches should have static prediction. That is they come in two flavors; predicted to branch and predicted not to branch. Implementations can use or not use this information depending on their prefetching capabilities. This is a valuable feature with only a set of rules for prediction at compilation. Its even more effective if operating system or application code is executed on a simulator and the results are used to adjust the prediction before release.

Data comparisons may be combined with branches or be separate instructions. If separate, the results may be saved in an integer register or as condition code status information. The following are preferred. Full arithmetic comparisons of integer register or floating point data are separate compare instructions. They leave their results in an integer register. Conditions which take a small portion of a cycle to evaluate, such as testing individual bits and zeroness of an integer register, are combined with branch operations.

In order to most efficiently implement the IEEE standard, there should be two forms of floating point comparison. One signals invalid if either operand being compared is a Nan, and an invalid trap happens if enabled. The other does not signal invalid. In either case, invalid is one of the possible results of the comparison.

Condition codes may be a small implementation convenience, but they limit code placement flexibility and parallelism. Condition code architectures need a complete set of arithmetic instructions which don't alter the condition codes as well as selected operations which do. Generally, the arithmetic results of a comparison can be saved as well as setting the condition codes. However, this will rarely save cycles over the preferred combination of operations described above.

Normal addressing for branches and call operations is PC + displacement. Call instructions and unconditional branches should have the widest possible displacement. The displacement should be approximately 16 bits for conditional branches. There should be an unconditional branch whose target address is the contents of an integer register. This is typically used for return operations since call instructions leave their return address in an integer register. There should also be a call operation whose target address is in a register. This is important for dynamic binding of procedures, and a convenience in several other situations.

There must be some kind of trap or system call operation. Ideally this is an instruction with a literal parameter which is used to vector to one of many (in the range of 256) entry points. A register (computed) parameter form of system call is not required.

Special loop branches are not worth including. Typically rx = rx + ry and branch if the original rx was negative, positive, zero, etc. Often this special branch doesn't fit a compiler's optimization of count and addressing indices for a particular loop. When it does, cycles won't be saved unless there aren't enough branch and load delay cycles in which the arithmetic can be performed with regular instructions.

The 88000 has an integer register compare which puts a 10 bit result string into an integer register. Each bit indicates one of the possible relationships between the registers compared as if signed and unsigned. Five of the bits are the complement of the others, eg. equal and not equal.

Floating point compare puts a 12 bit result string into an integer register. 8 of the bits indicate greater than, less than, equal, unordered, and their complements. The other 4 bits are a range check. Its not stated, but its assumed that invalid is signaled if unordered is true. The range check seems too specialized to include in the base architecture, but it has negligible cost. We would propose to delete it for the binary incompatible approach unless its desired by other customers. We also propose that another floating point compare be added which delivers results similar to the current one, but it doesn't signal invalid on unordered. The one which signals invalid is used if the program source specifies predicates which should signal invalid if unordered: ie, > ,<=, <, >=, <> (= is used as the complement of <> for the static branch prediction mechanism proposed below, and not for the equality predicate), and <=> (ordered, the unordered bit is its complement for branch prediction, and not the unordered predicate). If unordered is true than all other conditions are false. The other compare is used if invalid should not be signaled, ie, ?>, ?<=, ?<, ?>=, ?<>, ?=, ? (the ordered bit is its complement for branch prediction, and not the <=> predicate), and = (and its complement which is used for branch prediction, and not the <> predicate). If unordered is true than all other conditions are true except = and its complement.

All branches, calls and returns have delay slots with the desired attributes. Conditional branching is provided for a specified integer register bit set or clear and all relationships of an integer register compared with zero. Addressing is PC + displacement *4 with a 16 bit sign extended displacement. Unconditional branch addressing is an integer register (for return operations) or PC + displacement * 4 with a 26 bit sign extended displacement. Call addressing is also an integer register or PC + displacement * 4 with a 26 bit sign extended displacement. If the delay slot instruction is executed, the return address is correctly incremented by 4.

System call instructions on the 88000 are conditional, with the same tests as conditional branches. They trap to one of 512 entry points, 8 bytes apart. The first 128 entries cannot be entered in user mode. Testing r0 for zero guarantees a trap. Details of the privileged aspects of system calls and returns (launching processes) are discussed in the companion Cheetah specification for the first implementation.

The 88000 lacks static prediction. Otherwise it has a nearly ideal set of comparison and program flow operations. Motorola proposes to add static prediction by the convention that branch on bit is predicted to be taken and branch on bit clear is predicted to not be taken. The current branch on comparison with zero will be predicted to be taken and and a new branch on comparison with zero will be predicted to not be taken.

The 88000 also has a trap on bounds check instruction. It performs an unsigned register x register or register x 16 bit zero extended displacement comparison and traps if the first register is larger than the second register or displacement. This instruction is in effect an arithmetic compare and branch, to a particular trap entry point, and predicted not to branch. The model for the architecture is otherwise separate compare instructions and branch instructions. With static prediction, the branch instructions can result in zero cycles if predicted correctly. With this degree of parallelism, the normal architectural model can equal the performance of the special bounds check instruction; and provide more flexibility in testing, and in handling violations within the executing process. As a minor point, we propose to

delete the trap on bounds check instruction for the binary incompatible approach in order to reduce complexity and save instruction encoding space.

Binary Incompatible Summary

	Proposed Changes	Apple	Motorola
1.	Drop little endian support unless others really want it.	minor issue	?
2.	Add auto base register updating integer register loads and stores; register + register++, register + displacement++ Displacements signed and size consistent with other memory reference instructions.	depends on implementation	will do with 10 bit signed displacement, larger is compatibility issue
3.	Delete the separate user space addressing mode.	minor issue	yes
4a.	Delete unscaled forms of load relative address.	minor issue	yes
4b.	Delete all of load relative address.	minor issue	compatibility issue
5.	Make address displacements signed. Shorten size slightly if necessary so that all memory reference instructions are consistent, including new ones.	very important	compatibility issue, add new 10 bit signed
6.	Make arithmetic immediates signed for consistency with (5).	minor issue	see (5)
7.	Delete sign extending byte and half word loads.	minor issue	compatibility and ? issue
8a.	Add signed register x register multiply.	important	yes
8b.	Add signed register x immediate multiply	important	no
9a.	Add sticky integer overflow and divide by zero flags and enables similar to floating point.	important	yes
9b.	Add non sticky integer overflow and divide by zero flags.	no	agree
10.	Define the existing serial/parallel mode to apply to floating point and other operations.	important	yes
11a.	Unsigned multiply should affect the carry flag.	no	agree
11b	Unsigned multiply should not affect the overflow flag.	important	missunder-standing?
12a.	Add unsigned extended multiply.	important	yes
12b	Don't add signed extended multiply.	minor issue	?

13.	Add signed and unsigned extended divide with 32 bit (partial) quotient and 32 bit remainder as result.	important	yes, but missunder-standing on remainder?
14.	Add signed and unsigned remainder instructions.	no, use (13)	added per original request
15.	Don't add signed and unsigned extended remainder instructions.	not required	missunder-standing?
16.	Delete set and clear bit instructions.	minor issue	compatibility issue
17.	Delete scan instruction.	no, desired by others	agree
18.	Delete existing floating point instructions.	very important	compatibility issue
19.	Add floating point registers. Proposed architecture is (32) 80 bit, holding single, double or extended. Register zero is hard wired to zero.	very important	yes
20.	Add floating point loads and stores which support single, double and extended execution. Displacements signed and size consistent with other memory reference instructions.	very important	always convert to and from extended; 10 bit signed displacement, larger is compatibility issue
21.	Extended is left justified in memory in four words with zero fill.	important	not left justified
22.	Add auto base register updating floating point loads and stores; register + register++, register + displacement++ Displacements signed and size consistent with other memory reference instructions.	depends on implemen-tation	will do with 10 bit signed displacement, larger is compatibility issue
23.	Add 128 bit double floating point register pair loads and stores.	depends on implemen-tation	no
24.	Delete double integer register pair loads and stores if there isn't enough instruction encoding space.	keep if encoding space	compatibility issue

25.	Add floating point add, subtract, multiply, divide, remainder and square root supporting single, double and extended execution.	very important	extended only
26.	Neutral- on adding floating point scale, log, next and class instructions.	not required	?
27.	Neutral+ on adding floating point round to integral value instruction.	not required	?
28.	Don't add multiply/accumulate instructions to the base architecture.	implementation dependent	agree
29.	Add conversion from single, double and extended floating point to 32 bit integer instructions.	very important	yes, but 64 bit sign extended 32 bit result
30.	Add conversion from 32 bit integer to double and (or) extended floating point instructions.	very important	yes, but 64 bit sign extended 32 bit source
31.	Add instructions for conversion between floating point formats without going through memory.	minor issue	yes
32.	Add move instructions between floating point and integer registers.	no	agree
33.	Add non sticky floating point exception flags.	no	agree
34.	Add floating point registers modified bit, as privileged status.	important	yes, but user status
35.	Provide enough hardware support for the IEEE standard so that round to nearest with traps disabled does not need software assistance, or provide a mode, as an implementation dependent user feature, which doesn't require software assistance, but has some exceptions to the IEEE standard.	very important	?
36.	Add a floating point compare similar to the current one which doesn't signal invalid.	very important	?
37.	Delete the range check bits from floating point compare.	keep if desired by others	compatibility issue
38.	Add static prediction to conditional branches.	important	yes
39.	Delete the trap on bounds check instruction.	minor issue	compatibility issue

Binary Compatible Summary

	Proposed Changes	Apple	Motorola
1.	Drop little endian support unless others really want it.	minor issue	?
2.	Add auto base register updating integer register loads and stores; register + register++, register + displacement++ Displacements signed.	depends on implementation	will do with 10 bit signed displacement
3.	Delete the separate user space addressing mode.	minor issue	yes
4.	Delete unscaled forms of load relative address.	minor issue	yes
5.	Add signed displacement load and store byte, half word and word instructions.	very important	will do with 10 bit displacements
6a.	Add signed register x register multiply.	important	yes
6b.	Add signed register x immediate multiply	important	no
7a.	Add sticky integer overflow and divide by zero flags and enables similar to floating point.	important	yes
7b.	Add non sticky integer overflow and divide by zero flags.	no	agree
8.	Define the existing serial/parallel mode to apply to floating point and other operations.	important	yes
9a.	Unsigned multiply should affect the carry flag.	no	agree
9b.	Unsigned multiply should not affect the overflow flag.	important	missunderstanding?
10a.	Add unsigned extended multiply.	important	yes
10b	Don't add signed extended multiply.	minor issue	?
11.	Add signed and unsigned extended divide with 32 bit (partial) quotient and 32 bit remainder as result.	important	yes, but missunderstanding on remainder?
12.	Add signed and unsigned remainder instructions.	no, use (11)	added per original req
13.	Don't add signed and unsigned extended remainder instructions.	not required	missunderstanding?
14.	Implement existing floating point instructions in software.	very important	agree

15.	Add floating point registers. Proposed architecture is (32) 80 bit, holding single, double or extended. Register zero is hard wired to zero.	very important	yes
16.	Add floating point loads and stores which support single, double and extended execution. Displacements signed.	very important	always convert to and from extended; will do with 10 bit signed displacement
17.	Extended is left justified in memory in four words with zero fill.	important	not left justified
18.	Add auto base register updating floating point loads and stores; register + register++, register + displacement++ Displacements signed.	depends on implementation	will do with 10 bit signed displacement
19.	Add 128 bit double floating point register pair loads and stores.	depends on implementation	no
20.	Add floating point add, subtract, multiply, divide, remainder and square root supporting single, double and extended execution.	very important	extended only
21.	Neutral- on adding floating point scale, log, next and class instructions.	not required	?
22.	Neutral+ on adding floating point round to integral value instruction.	not required	?
23.	Don't add multiply/accumulate instructions to the base architecture.	implementation dependent	agree
24.	Add conversion from single, double and extended floating point to 32 bit integer instructions.	very important	yes, but 64 bit sign extended 32 bit result
25.	Add conversion from 32 bit integer to double and (or) extended floating point instructions.	very important	yes, but 64 bit sign extended 32 bit source
26.	Add instructions for conversion between floating point formats without going through memory.	minor issue	yes
27.	Add move instructions between floating point and integer registers.	no	agree
28.	Add non sticky floating point exception flags.	no	agree

29.	Add floating point registers modified bit, as privileged status.	important	yes, but user status
30.	Provide enough hardware support for the IEEE standard so that round to nearest with traps disabled does not need software assistance, or provide a mode, as an implementation dependent user feature, which doesn't require software assistance, but has some exceptions to the IEEE standard.	very important	?
31.	Add a floating point compare similar to the current one which doesn't signal invalid.	very important	?
32.	Add static prediction to conditional branches.	important	yes