SH7000 Architecture Bends RISC Rules

16-Bit Instructions Give High Code Density but Force Compromises

By Brian Case

As previously reported, Hitachi has introduced a new microprocessor architecture in its SH7000 family of chips for handheld computers (a.k.a., personal digital assistants or PDAs) and other consumer applications *(see **070802.PDF**)*. There are now several architectures competing for PDA-type applications, including ARM, Hobbit, SH7000, NEC's V800 (which will be covered next issue), and, of course, the x86 and 680x0. The range of PDA devices is expected to be broad, and even though many applications will not develop for years, IC vendors who have been locked out of the x86 market are hoping to gain access to high-volume consumer applications by establishing an early presence.

Hitachi has chosen to enter this market with a new architecture that, in its view, is tailored to the needs of PDAs. The major considerations for these devices are cost, power consumption, and performance, and Hitachi has addressed each issue in the architecture.

The cost of a final product is a system issue and therefore is a multifaceted metric, but the SH7000 architecture facilitates low cost in at least two ways: the relatively simple architecture keeps the CPU core implementation small, and the instruction length is 16 bits instead of the RISC-standard 32 bits. Short instructions increase code density, which potentially reduces memory cost.

Performance is addressed in the architecture by adopting some RISC attributes, such as fixed-length instructions, that facilitate a simple, fast pipeline. Even so, the SH7000 has some decidedly non-RISC features.

Power consumption during normal operation is largely determined by implementation technology and logic and circuit design, but to help software take advantage of periods of inactivity, the SH7000 architecture defines two low-power standby modes. These modes, invoked via the SLEEP instruction, allow software to take an active role in reducing the power consumed by SH7000 chips.

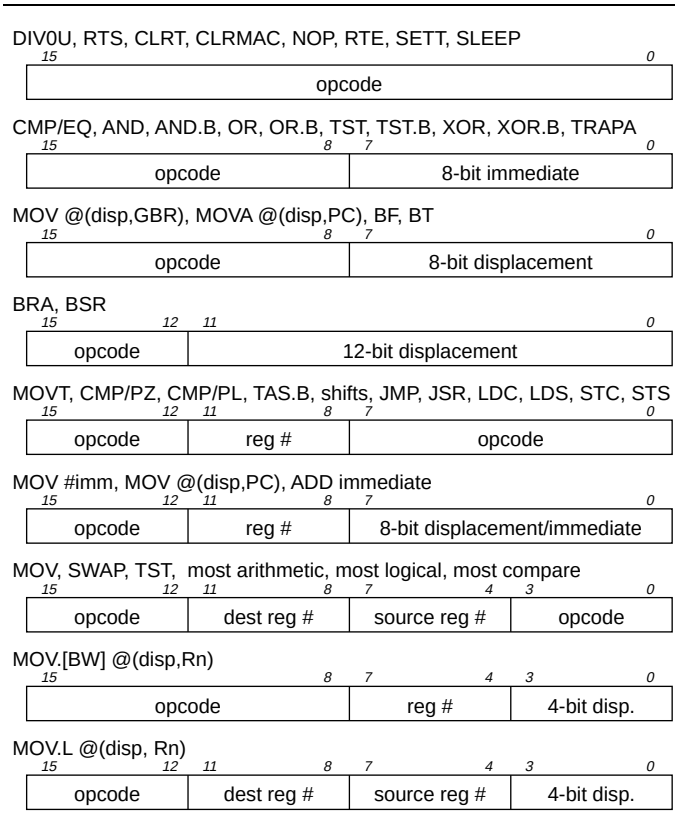

Figure 1. SH7000 instruction formats.

SH7000 Architecture Overview

The SH7000 is RISC-like, but it makes several significant deviations from the commonly acknowledged RISC tenets. The deviations stem primarily from the 16-bit instruction length, which is its most distinguishing feature, at least compared to other commercial RISCs. For example, two important RISC tenets—a large register file and three-address register-to-register operations—are compromised because of the constraints imposed by short instructions. Other non-RISC attributes include saving a stack frame in memory on exceptions, special uses for general registers, and a few instructions that perform read-modify-write memory operations.

The SH7000 lacks a supervisor mode, which is needed to implement modern, protected operating system environments. While it can be argued that small, inexpensive consumer devices have less need for sophisticated operating systems than desktop computers, all of the established competitors of the SH7000 provide a supervisor mode.

SH7000 Instruction Set

As shown in Figure 1, the instruction formats are consistent and easy to decode. Unfortunately, to allow a sufficient number of instructions to be encoded in 16 bits, the formats allow a maximum of only two 4-bit register fields. This has several implications, which are discussed separately below.

The 16-bit instruction length also reduces the length of offsets for branches, displacements for loads

and stores, and immediate fields for arithmetic and logical operations. As shown in Figure 1, the maximum unconditional jump offset is 12 bits, while the maximum offset for conditional branches is 8 bits. The largest displacement or immediate field is also 8 bits.

Table 1 shows the SH7000 instruction set. The MOV mnemonic is used for both loads and stores, which are available in byte, word (16-bit), and longword (32-bit) sizes.

The set of memory addressing modes is larger than for a traditional RISC. In addition to the common register-indirect, register+register, and register+displacement, the SH7000 offers both pre-decrement and post-increment addressing. In most cases, all addressing modes are available for byte, word, and longword operand sizes and for both load and store. The notable exception is the PC+8-bit-displacement mode: it is available only for loads of words or longwords.

For addressing modes that use a displacement (including pre-decrement and post-increment, which implicitly use a displacement), the displacement amount is scaled by the operand size (one, two, or four bytes). Thus, even though the register+displacement mode allows only a small 4-bit displacement, it is effectively larger for words and longwords. The displacements for MOVs are zero-extended.

One characteristic common to all RISCs is a load-store architecture: only load and store instructions reference memory, and only one memory reference is allowed per load or store. The SH7000 is a load-store architecture with a couple of curious exceptions. First, there are byte-length versions of the AND, OR, and XOR logical operations that read a byte from memory, perform the logical operation with an 8-bit immediate value, and then store the result to memory. Second, the multiply-accumulate instruction reads two of its three operands from memory (the third, as well as the result, is stored in the special MAC register).

One of the goals of a load-store architecture is to simplify the pipeline by matching instruction semantics to hardware resources: in a pipeline, a single memory interface implies that an instruction can do at most one memory reference. Instructions like the SH7000's byte-length logicals and the MAC are special cases that require dedicated sequencing logic (or even microcode) and reduce instruction throughput. The byte-length logicals violate the load-store model to provide atomic operations for memory-based semaphores. They allow software to set, clear, or invert a single bit in memory that could be used to lock or unlock a data structure. These accesses must be atomic to avoid having one software process be interrupted between the load and the store; if the interrupting process modifies the same semaphore, data corruption could occur. It is less clear why the designers chose to load MAC operands from memory, but it may

DATA TRANSFER	
MOV.[BWL]	Register-register move
MOVA	Move address; R0 gets PC+displacement
MOVT	Get branch condition code bit
SWAP.[BW]	Swap 16-bit words or lower two bytes
XTRCT	Concatenate two registers, extract middle 32 bits
ARITHMETIC	
ADD	Add
ADDC	Add with carry
ADDV	Add, trap on overflow
CMP/EQ	Compare for equal
CMP/HS	Compare for greater or equal, unsigned
CMP/GE	Compare for greater or equal
CMP/HI	Compare for greater than , unsigned
CMP/GT	Compare for greater than
CMP/PZ	Compare for greater or equal zero
CMP/PL	Compare for greater than zero
CMP/STR	Compare bytes
DIV1	Divide step
DIV0S	Initialization step for signed divide
DIV0U	Initialization step for unsigned divide
EXTS.[BW]	Sign extend
EXTU.[BW]	Zero extend
MAC.W	Multiply-accumulate
MULS	Signed multiply
MULU	Unsigned multiply
NEG	Negate
NEGC	Negate with carry
SUB	Subtract
SUBC	Subtract with carry
SUBV	Subtract, trap on underflow
LOGICAL	
AND	AND
NOT	Complement
OR	OR
TAS.B	Test and set memory
TST	AND and set condition code
XOR	Exclusive OR
SHIFT	
ROTL	Rotate left one bit
ROTR	Rotate right one bit
ROTCL	Rotate left one bit through T
ROTCR	Rotate right one bit through T
SHAL	Shift left arithmetic one bit
SHAR	Shift right arithmetic one bit
SHLL	Shift left logical one bit
SHLL2	Shift left logical two bits
SHLL8	Shift left logical eight bits
SHLL16	Shift left logical sixteen bits
SHLR	Shift right logical one bit
SHLR2	Shift right logical two bits
SHLR8	Shift right logical eight bits
SHLR116	Shift right logical sixteen bits
BRANCH	
BF	Relative branch false (T==0), non-delayed
BT	Relative branch true (T==1), non-delayed
BRA	Relative branch, delayed
BSR	Relative branch to subroutine, delayed
JMP	Register indirect jump, delayed
JSR	Register indirect jump to subroutine, delayed
RTS	Return from subroutine, delayed
SYSTEM CONTROL	
CLRT	Clear T
CLRMAC	Clear multiply-accumulate register
LDC	Load to control register
LDS	Load to system register
NOP	No operation
RTE	Return from exception, delayed
SETT	Set T
SLEEP	Enter power-down mode
STC	Store control register
STS	Store system register
TRAPA	Trap always

Table 1. SH7000 instruction set.

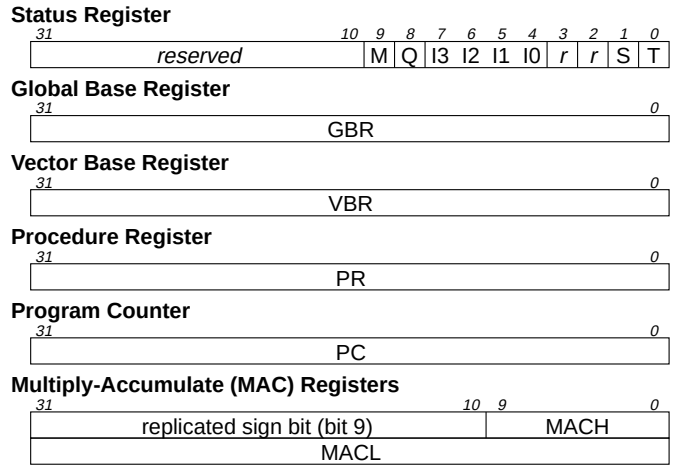

Figure 2. SH7000 special registers.

make little difference since MAC is a multi-cycle operation in any case.

The delayed branch is a feature that was part of all early RISCs but has been left out of some newer architectures, such as Alpha. It was introduced to increase pipeline efficiency, but delayed conditional branches present slight difficulties for superscalar implementations because they are a special case for instruction fetching, grouping, and issue. Instead of leaving delayed branches out of the architecture completely, the SH7000 architects decided to make conditional branches non-delayed and unconditional branches delayed (including return from subroutine and exception). If this architecture succeeds in the marketplace, this foresight may slightly ease the design of future superscalar implementations.

The SH7000 delayed-branch architecture is unique in some additional ways. First, it specifies "illegal slot checking" to prevent a program from having a branch in the delay slot of another branch. Because of the way delayed branches work, the behavior of two consecutive delayed branches can be unexpected. Checking for and preventing this case is a minor simplification of the architecture that frees Hitachi from the bother of specifying the exact behavior and gives some freedom for future implementations.

Second, no interrupts or traps are allowed between a delayed branch and its delay-slot instruction. This simplifies the architecture by eliminating the need to save two return PC values when an interrupt or trap occurs. In other architectures with delayed branches, two return PC values must be saved to properly handle the somewhat-rare case of an interrupt that occurs exactly between a delayed branch and its delay instruction (roughly a one-in-six chance since 16% of all instructions in a typical program are branches).

Beyond delayed branches, another way to increase pipeline utilization is to allow load scheduling so independent instruction(s) can be placed after a load to give the processor useful work to do during memory (or even cache) latency. Like most RISC manuals, the SH7000 architecture manual specifies that load scheduling of one instruction will improve performance. To ease the programming burden, loads are fully interlocked so scheduling is not mandatory.

The SH7000 architecture has atomic multiply instructions and supports divide through one-bit divide step instructions. MULS and MULU provide signed and unsigned multiplication; both instructions multiply two 16-bit register operands to form a 32-bit product. The product is stored in the MACL special register.

Signed and unsigned division are supported with three instructions: DIV0S and DIV0U for division initialization, and DIV1 for forming successive bits of the quotient. A complete division routine requires from 24 to 72 instructions, depending on operand and quotient size. The generation of a remainder is not supported.

The MAC instruction reads two 16-bit integer operands from memory indirectly through the two register pointers specified in the instruction. It then multiplies these two operands, forms a 32-bit product, adds the product to the contents of the MAC registers, and stores the result in the MAC registers. (See Figure 2 for the layout of the MAC registers.) If the S bit in the status register is set to one for a MAC instruction, saturating addition is performed: MACL stores the 32-bit result value and the LSB of MACH is set if MACL saturates. When S is set to zero, a full 42-bit result is saved. The upper 22 bits of MACH are the sign-extension of the lower ten bits.

Multiply-accumulate instructions are also found in the PowerPC, PA-RISC, MIPS-IV, and ARM architectures. Of these, only the ARM has an integer multiply-accumulate; the others are floating-point. The SH7000 MAC is unique because two of its operands are memory-based; all the other architectures have multiply-accumulate instructions that get all operands from registers.

The SH7000 architecture has ten shift and four rotate instructions but lacks the "shift-by-n" variable shift instructions—common in other RISCs—that exploit a barrel shifter. Instead, it has eight shift-logical instructions: left and right by one, two, four, and eight bit positions. For arithmetic shifts, only one-bit left and right shifts are provided. The rotate instructions move one bit position, left or right, with or without carry (which is the T bit). All four rotates and the four one-bit shifts affect the T bit.

The lack of a barrel shifter is presumably due to the desire for a small CPU core implementation. A barrel shifter has become standard equipment in nearly every modern architecture and implementation, but relative to an ALU, it is still an expensive execution unit. Given the 16×16 parallel multiplier for the multiply and MAC instructions and the lack of a barrel shifter, it seems the

designers expected SH7000 applications to have a greater need for multiplication than shifts. While leaving out a barrel shifter does save die size, the lack of variable-shift instructions is a weakness of the SH7000 architecture compared to its competition.

The LDC, LDS, STC, and STS instructions are used to load and store the contents of the special registers shown in Figure 2. In most other architectures, data movement for special registers is between general registers only, but the SH7000 also allows special-register data movement directly to memory. When a special register is loaded from or stored to memory, the general register that serves as the memory pointer is automatically incremented or decremented, respectively, by four. This feature speeds saving special registers on the stack. Loading and storing the special registers is likely to be infrequent, however, so it seems like a waste of opcode space to include a set of special-register load and store instructions when the other loads and stores are sufficient.

Conditional-Branch Architecture

The SH7000 has a unique conditional-branch architecture. Most other machines fall clearly into the condition-code (dedicated condition-code register or bits in status register) or relationship-as-data (result of comparison written to general register) categories, but the SH7000 has a combination of the two types. It uses explicit compare instructions to test for a specific relationship, as does the relationship-as-data model, but these compare instructions set the T bit, which resembles a condition code, in the processor status register. The T bit determines the behavior of the BT (branch if T set) and BF (branch if T clear) instructions.

The big benefit of the condition-code model is that condition codes are set as a result of regular arithmetic instructions, thus occasionally eliminating the need for a separate compare instruction. The relationship-as-data model makes compiler code generation and optimization easier since the bottleneck of the single set of condition codes is eliminated (allocating space for branch conditions is handled as register allocation, which is already present in the compiler). The SH7000 model is cleaner than condition codes, since regular arithmetic instructions do not affect the T bit, but not as clean as the relationship-as data model. The T bit is probably a good compromise given the small, 16-register file.

Compare instructions use two register operands except for CMP/EQ, which also has a version that compares a register to an 8-bit constant.

Register Addressing Limitations

As mentioned above, the 16-bit instruction format is limited to at most two 4-bit register fields. There are three main implications of this limitation.

First, 4-bit register fields can address only 16 general registers. Sixteen is probably a sufficient number for many routines, but, compared to the 32-register files of most RISCs, the smaller register file significantly increases the likelihood that extra loads and stores will be required to manage register usage. Also, with the 32 registers found on other RISCs, it is possible to statically allocate some registers for leaf procedures, which can greatly speed a call to a leaf procedure. Realistically, this technique cannot be used on the SH7000.

Second, two-address register-to-register operations, such as additions, must always overwrite one source operand. In code sequences where preserving both operands is required, an extra register-to-register move instruction is needed to copy the source operand that will be overwritten. Similarly, the SH7000's one-address shift operations cannot preserve their source operand, and will sometimes require an extra move.

Third, as mentioned above, many instructions must use R0 as an implicit operand register. Making R0 a special, implied register has two main side effects. First, compiler code-generation and optimization algorithms must be more complex to effectively deal with R0 as a special-case register, although the increase in complexity is probably not too profound. Second, extra register-to-register move instructions will be required in some code sequences to move data into or out of R0.

Simple Register Model

The SH7000 register model is very simple. There are 16 general registers and seven special CPU registers. The SH7000 has only half as many general registers as most other RISCs, but it compensates somewhat in two ways. First, R0 is a true register—not fixed at a value of zero as on some other architectures, such as MIPS and SPARC. Fixing R0 at zero can lead to architectural simplification—comparing to zero and some addressing modes fall out naturally—but it is more important to have an extra real register given the constraints of 4-bit register specifiers. Second, a special, rather than a general, register is used to point to a global data area.

Registers R0 and R15 have special uses. R15 is the stack pointer. In a traditional RISC, any register can be used as the stack pointer because the stack mechanism is defined by software. On the SH7000, interrupts and traps cause an exception frame to be saved on the memory stack, so a stack pointer must be defined by the architecture.

The special use for R0 is as an implicit operand for some instructions that need more register fields than the 16-bit instruction length will allow. Most of the special uses of R0 are for MOV instructions that specify register+offset or register+register addressing. The notable exceptions, undoubtedly made because of frequent use in real programs, are loading and storing a longword with the register+offset mode. These two instructions alone,

which allow any register as the source or target, take up one-eighth of the architecture's opcode space.

Figure 2 shows the SH7000 special register set. The status register contains: the branch condition code (T), which is set by compares and tested by BT/BF; the saturate bit (S) for multiply-accumulate operations; the interrupt mask level (I3–I0); and bits to hold intermediate state for divide-step instructions (M, Q). The T bit has three other main purposes: it is used by some shift and rotate instructions, it serves as the carry bit for add and subtract with carry, and it indicates the overflow condition for ADDV and SUBV.

The global base register (GBR) points to a global data area. In other RISCs, a general register might be set aside by software for this purpose, but with only 16 registers, using a special register provides a significant savings. The architecture provides MOV instructions that allow operands of all three sizes to be loaded or stored with the GBR+8-bit-offset addressing mode; the source or destination register, however, must be R0.

The vector base register (VBR) points to an area where the addresses of exception handling routines are stored. For power-on and manual reset, the vector contains both a routine pointer (PC value) and an SP value.

The PR register serves as a one-element stack for subroutine calls. The BSR and JSR instructions save the return PC in PR, and the RTS instruction restores PC from PR. If the called routine itself needs to do a BSR or JSR, then it must explicitly save and restore the contents of PR. For the common case of calling a leaf routine (a routine that itself calls no routines), however, this one-element stack approach is of benefit because a general register is not tied up. Another benefit is the fact that the BSR and JSR instructions do not have to specify a register for the return address either implicitly or explicitly (which would use four precious branch-displacement bits).

The MACL and MACH registers, which are the destination of the MULS, MULU, and MAC instructions, were described in the previous section.

Power Saving Modes

The SLEEP instruction is used to invoke one of two possible power-down modes: sleep mode and software-standby mode. The mode entered is determined by the state of the SSBY (software standby bit) in the SBYCR special register (SBYCR is a memory-mapped peripheral register, not a CPU special register like those in Figure 2). The major difference is that sleep mode leaves the on-chip clock running to drive peripheral functions, while software-standby mode stops all clocks. Software-standby mode saves the most power, but most peripherals are reset to initial states. Both modes preserve CPU registers and on-chip RAM.

Evaluation and Conclusions

The SH7000 is a simple, small architecture with some important RISC characteristics. The architecture facilitates a small implementation, and the instruction-set semantics (load-store architecture, few complex instructions, etc.) are a good match to a basic pipeline, which should ensure high performance at reasonable cost. The 16-bit instruction length probably results in small code size for most applications, but it also results in a performance loss relative to other architectures. Compared to the purer RISCs like ARM and V800, extra instructions will be required because of the shortcomings of the two-address, register-to-register architecture, fewer registers (although ARM also has only 16), and special uses of R0. The lack of variable-shift instructions means that in-line code or a subroutine call will be required when an algorithm requires shifting by a dynamically determined amount.

Based on the characteristics of the architecture and the features of the first implementations, it is clear that Hitachi intends the SH7000 to be a high-end version of the traditional integrated microcontroller. This makes it unique among its competitors, which are emphasizing more "big-system" features. Hobbit and x86 implementations have a user/supervisor mode bit, floating-point instruction set, cache, and an MMU. ARM has a user/supervisor mode bit, cache, and an MMU. The V800 has cache and floating-point.

The SH7000 has some strange architectural artifacts. It has two distinct instructions and opcodes—SHAL and SHLL—that perform exactly the same function. This does not present any problems, but at the same time there is no need for a duplicate instruction. Perhaps the SHAL instruction fell out of the instruction encodings, and it was easier to leave the instruction in than take it out. The load and store instructions for the system and control registers are another feature that seems difficult to justify.

Despite any complaints about the architecture, the SH7000 is undoubtedly adequate for use in PDAs and similar devices. While it's true that the SH7000's competitors are superior in some ways, none of the processors aimed at the PDA market are perfect; also, some of the differences between the processors are unimportant to the success of the final product. The important determinants of success will be the cost of the processor and the system and whether the SH7000 can garner the software support and design wins that will create high volume markets. On the other hand, Hitachi may have internal markets that are large enough to justify its investment in the SH7000 and guarantee high volume production. ♦

