LSI's TinyRisc Core Shrinks Code Size

Code-Compaction Technique Squeezes MIPS Instructions Into 16 Bits

by Jim Turley

With a nod to embedded designers concerned about code density, LSI Logic and MIPS Technologies have together developed a special MIPS core that uses 16-bit instructions. The new TinyRisc core, which will be available late this year, slashes the MIPS instruction word in half, giving ASIC customers a 32-bit architecture with 16-bit instructions.

TinyRisc is similar in concept to ARM's Thumb option *(see* **090401.PDF***)*. Like Thumb, TinyRisc compromises performance and flexibility in order to improve its code density. The many similarities between Thumb and the TinyRisc instruction set (which the companies call MIPS-16) highlight the technical limitations of such an effort. That two such companies would undertake similar projects demonstrates the increasing concern about code size in many portable embedded applications.

Not Really Compression, Just Different Opcodes

Presenting at last week's Microprocessor Forum, LSI's Paul Cobb described MIPS-16 as "compressed" instructions, but it is actually a completely different instruction set encoded in 16 bits. TinyRisc chips include two instruction decoders, one for conventional MIPS code and another for MIPS-16 instructions. TinyRisc chips can switch between these two instruction formats using special branch instructions. Otherwise, the two instruction sets cannot be intermixed.

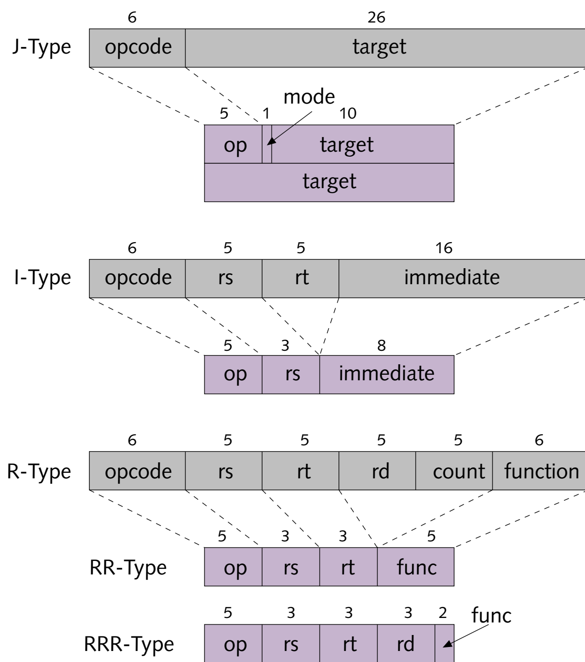

Figure 1. Conventional 32-bit MIPS instructions are simplified and condensed into the new MIPS-16 instruction set. (J, I, and R formats refer to jump, immediate, and register addressing modes.)

Switching between the two instruction sets is necessary because TinyRisc chips cannot operate entirely in 16-bit mode. There are several operations that can only be executed using the conventional 32-bit MIPS instruction set. For example, there are no MIPS-16 instructions for exception handling, system setup, floating-point math, or error recovery. Programmers can decide when to switch to MIPS-16 mode for compactness and when to use 32-bit mode for compatibility and flexibility.

Figure 1 shows how the three types of conventional 32-bit MIPS instructions are broken into four 16-bit types for MIPS-16. In the MIPS-16 instruction set, opcodes are reduced from six bits to five, instantly halving the number of available operations. These operations are encoded entirely differently than their 32-bit versions, not merely shaved from one end of the normal opcodes. The MIPS-16 predecoder maps its 5-bit opcodes to their 6-bit originals using a lookup table. This strategy allowed TinyRisc's designers to select the operations they judged most vital, not necessarily the ones that were missing, say, their most significant bit.

In jettisoning half the instruction map, MIPS-16 forces several compromises. Only eight of the processor's 32 registers are available, most register-to-register operations are destructive, and both immediate data and offsets have been severely truncated. Immediates are one byte, and indirect branch offsets are restricted to 10 bits.

This last compromise would normally limit the extent of MIPS-16 branches to 512 instructions, or 2K bytes, in either direction. (Even though MIPS-16 instructions are 16 bits long, the architecture still requires 32-bit alignment on branch targets.) To sidestep this harsh restriction, branch instructions automatically concatenate the next 16-bit word to the branch offset, enabling the full 26-bit offset MIPS programmers are used to.

New Instruction Set Is Fairly Complete

Table 1 lists the entire MIPS-16 instruction set, including encoding options for many operations. The MIPS-16 predecoder can be grafted onto nearly any MIPS core, so the resulting chip will support whatever instructions it originally did, plus the new 16-bit ones. Thus, R3000-class devices will have the capacity for only 32-bit operations, while R4000-type chips will do both 32-bit and 64-bit math.

Nearly all the general-purpose MIPS-II and MIPS-III instructions have MIPS-16 counterparts, the major omis-

sions being signed add and subtract, unconditional jump, left- and right-justification, and all coprocessor operations. This last restriction eliminates any hope of running floating-point code with the MIPS-16 instruction set.

For arithmetic operations, MIPS-16 supports signed and unsigned multiplication and division, but only unsigned addition and subtraction. For bitwise logical operations, the immediate forms are not supported. For example, AND, OR, and XOR must use two registers (destroying the contents of one source register with the result) rather than an immediate mask. The unconditional jump was replaced with an unconditional (PC-relative) branch.

To switch between instruction-set modes, software must jump or call the target module using JR, JALR, or a new JALX (jump, link, and exchange) instruction. The two least significant bits of the program counter are normally not used. For TinyRisc chips, bit 1 becomes significant, and bit 0 stores the current instruction-set mode. A mode-switching control-transfer toggles bit 0, reversing the current mode and enabling or disabling the MIPS-16 predecoder.

The MIPS-16 designers cheated somewhat, creating a new EXTEND instruction that bypasses the short 5-bit and 8-bit immediate data fields dictated by the MIPS-16 encoding. Placing an EXTEND instruction immediately before another MIPS-16 instruction extends the latter instruction's immediate data field to 16 bits, as in most conventional MIPS instructions. It works by concatenating the 11-bit immediate field of the EXTEND instruction itself with the short immediate field of the subsequent MIPS-16 instruction. The concept is similar to the size-override prefix bytes of x86 processors since the 386.

The extension prefix helps alleviate some of the switching between 16-bit mode and 32-bit mode that would otherwise be required to generate long addresses or large immediate values. It is not compatible with all MIPS-16 instructions; those that can be extended are indicated in Table 1.

Mnemonic	Description	J	I	RR	RRR	Ext
Load/Store						
LB	Load byte, sign extend		●			■
LBU	Load byte, zero extend		●			■
LH	Load halfword, sign extend		●			■
LHU	Load halfword, zero extend		●			■
LW	Load word, sign extend		●			■
LWU	Load word, zero extend		●			■
LI	Load 8-bit immediate data		●			
LD	Load doubleword		●			■
SB	Store byte		●			■
SH	Store halfword		●			■
SW	Store word		●			■
SD	Store doubleword		●			■
MOVE	Move register to register			●		
MFHI	Move from HI		●			
MFLO	Move from LO		●			
Arithmetic						
ADDU	Add, unsigned				●	
ADDIU	Add immediate, unsigned		●			■
DADDU	Add doublewords, unsigned				●	
DADDIU	Add imm. doublewords, unsigned		●			■
SUBU	Subtract, unsigned				●	
DSUBU	Subtract doublewords, unsigned				●	
MULT	Multiply			●		
MULTU	Multiply, unsigned			●		
DMULT	Multiply doublewords			●		
DMULTU	Multiply doublewords, unsigned			●		
DIV	Divide			●		
DIVU	Divide, unsigned			●		
DDIV	Divide doublewords			●		
DDIVU	Divide doublewords, unsigned			●		
Miscellaneous						
EXTEND	Extend next immediate field		●			
BREAK	Breakpoint trap		●			

Mnemonic	Description	J	I	RR	RRR	Ext
Logical						
AND	Logical AND			●		
OR	Logical OR			●		
XOR	Logical exclusive-OR			●		
NOT	Logical invert			●		
NEG	Subtract without overflow			●		
SLT	Set on less-than				●	
SLTU	Set on less-than, unsigned				●	
SLTI	Set on less-than, immediate		●			■
SLTIU	Set on less-than, immediate, unsigned		●			■
CMPI	Compare with immediate		●			■
Jump/Branch						
JAL	Jump and link	●				
JALX	Jump and toggle MIPS-16 mode	●				
JR	Jump indirect via register			●		
JALR	Jump and link via register			●		
BEQZ	Branch if equal immediate		●			■
BNEZ	Branch if not equal immediate		●			■
BTEQZ	Branch if target equal immediate		●			■
BTNEZ	Branch if target not equal immediate		●			■
B	Branch unconditionally	●				■
Shift						
SLL	Logical shift left		●			■
SRL	Logical shift right		●			■
SRA	Arithmetic shift right		●			■
SLLV	Logical shift left, variable			●		
SRLV	Logical shift right, variable			●		
SRAV	Arithmetic shift right, variable			●		
DSLL	Logical shift left doublewords		●			■
DSRL	Logical shift right doublewords		●			■
DSRA	Arithmetic shift right doublewords		●			■
DSLLV	Logical shift left doublewords, variable			●		
DSRLV	Logical shift right doublewords, variable			●		
DSRAV	Arithmetic shift right doublewords, variable			●		

Table 1. The complete MIPS-16 instruction set includes most standard MIPS functions outside of system-control operations. The new instruction set uses four formats, listed as J (jump), I (immediate), RR (two-register), and RRR (three-register). Several immediate-type instructions can be prefixed with the special EXTEND opcode that can be used to extend the tiny MIPS-16 immediate field.

Code Density Surpasses Even CISC

The effects of the MIPS-16 instruction set on performance, code density, and cache efficiency are complex and interrelated. Far from applying a simple 0.5× multiplier, code tends to shrink by a variable amount. The shorter instructions are less flexible than their 32-bit counterparts, necessitating more instructions for the same work. The net shrinkage also depends on the proportion of code that can be run in 16-bit mode. MIPS claims a 40% shrink on average for code that is compressible. In other words, instruction size decreases by 50% while instruction quantity increases by 20% compared with conventional MIPS code.

Obviously, data is not compressed at all. Neither are system-level functions such as OS calls and interrupt handlers. Applications with a low ratio of code to data, such as high-end network routers, will see very little improvement. On the other hand, systems with tiny data sets may see a substantial improvement. The 40% overall goal is certainly unattainable no matter what the configuration, but double-digit percentages are possible, given the right system.

MIPS ran a half-million lines of C code through its MIPS-16 compiler and compared the object-code density to that of 68K, ARM, Thumb, x86, and conventional MIPS processors. The results are shown in Figure 2. The test code included GSM, fiber-channel, and networking routines.

Not surprisingly, MIPS-16 code was smaller than any of the others and dead even with Thumb. This latter statistic suggests that a fundamental limit has been reached for this method of compaction and that both MIPS and ARM have reached this limit.

MICHAEL MUSTACCHI

At the Microprocessor Forum, LSI Logic's Paul Cobb explains how TinyRisc compacts MIPS code.

Hitachi's SuperH, which also uses 16-bit instructions, was not tested (or the results were not released), but its density should be nearly as good as TinyRisc's or Thumb's. SuperH can't fall back on a 32-bit instruction set and must use precious opcodes for system functions, so its code density suffers a bit compared with that of the dual-mode chips.

Cache and Bus Performance Changed

As Thumb users have discovered, processor performance may actually increase in systems with a 16-bit path to program memory. TinyRisc chips will be able to fetch an entire instruction in one cycle across a 16-bit bus, rather than needing more cycles or a wider data path. Before TinyRisc, designing a MIPS-based system with a 16-bit bus was foolhardy.

Cache effectiveness also changes in nonintuitive ways. Twice as many 16-bit instructions will fit in the instruction cache, yet because the number of instructions generally increases, the effective capacity of the instruction cache is not doubled. Depending on the cache implementation, the CPU may be able to fetch two or more instructions in a single cache-line access, improving performance by increasing bandwidth; fewer cache transactions may also reduce power consumption slightly. Finally, because the cache tags and associativity are not altered but the number of instructions they map doubles, hit rates may suffer slightly.

Creative Register Mapping

With access to only eight registers, programmers (or their compilers) are forced to be creative with register allocation. That task is complicated somewhat by MIPS tradition and architecture, which relies on certain registers for special functions. Register r0, for example, is read-only and always returns the value zero; r31 holds return addresses, and r29 holds the stack pointer. Unlike ARM's Thumb, MIPS-16 cannot simply restrict addressing to the first eight registers, or stack addressing would collapse.

Instead, the program-visible registers are mapped to the conventional register set. Registers r1–r7 are mapped straight across, but rather than try to use r0, the MIPS-16 logic applies a little modulo arithmetic, mapping r0 onto r16 to provide another usable register.

This solution still does not solve the problem of stack and pointer addressing through the higher registers. The MOVE instruction has access to all 32 general-purpose registers, so 16-bit programs can copy data from invisible to visible registers. There are also four new forms of SP-relative load, store, and add, making it easier to access constants and addresses of constants.

MIPS-16 Richer Than Thumb

With only 16 bits for encoding, there aren't many opcodes left for frivolous instructions once the basics are out of the way. MIPS and ARM were forced to make many of the same choices when designing their minimalist instruction sets. However, the MIPS-16 set includes a richer set of operations, including 64-bit arithmetic, integer division, and a different approach to conditional branching.

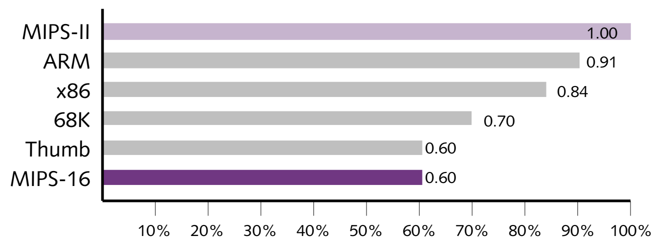

Figure 2. TinyRisc object code is more compact than that of other RISCs and the leading CISCs. Code density was equal to an ARM7 with the Thumb code-compaction module, suggesting that both MIPS and ARM have reached similar conclusions about compaction techniques. Relative sizes have been normalized for the MIPS-II instruction set. (Source: MIPS Technologies)

Even in 32-bit mode, ARM chips have never supported integer division, relying instead on software to execute an iterative loop. ARM's approach to conditional execution and condition codes often makes conventional conditional branches superfluous. With no condition codes, MIPS-16 has only a limited set of compare-and-branch instructions, all of which compare a register to an immediate value.

Where the MIPS-16 instruction set really differs from Thumb is in its support for 64-bit add, subtract, multiply, divide, and shift operations. These instructions all take their operands from two contiguous registers, an option that requires no additional addressing bits. In addition, the MIPS-16 EXTEND instruction eliminates a lot of unnecessary mode switching.

The ARM/Thumb combination has an advantage in memory-reference instructions. It can push and pop stack operands and load or store multiple registers with one instruction, a singularly non-RISC operation that MIPS chips do not support.

TinyRisc Core Takes a Clock-Speed Hit

Embedded MIPS chips are noteworthy for their high clock speeds, supposedly one of the seminal advantages of a RISC architecture. This high clock rate is possible because each of the pipeline stages is relatively simple.

But TinyRisc's code compaction takes its toll. Inserting the MIPS-16 predecoder into the first pipeline stage, along with the usual decode logic, adds enough complexity to compromise clock speed. LSI's first TinyRisc processors are expected to run at 80 MHz—or 70 MHz in 16-bit mode. The increased decoding overhead adds more latency than the 12.5-ns period of an 80-MHz pipeline can handle. TinyRisc customers will have to clock their parts at 70 MHz or below, or else implement some elaborate scheme to slow the clock when entering 16-bit mode.

ARM's Thumb does not incur this penalty, but in fairness, Thumb-equipped chips have never achieved 80-MHz operation to begin with. Currently, ARM7 designs are limited to about 40 MHz; the ARM8 and StrongArm cores run substantially faster but do not use Thumb.

ARM7's first pipeline stage comprises little more than a latch, so adding the Thumb predecoder does not complicate that portion of the pipeline enough to create a bottleneck. Thus, while Thumb and TinyRisc add similar amounts of complexity to their respective host CPUs, Thumb manages to hide its effects with a longer cycle time.

First TinyRisc Core Ready in December

LSI's first TinyRisc core with the MIPS-16 predecoder is the TR4101 (not to be confused with NEC's VR4101 MIPS microprocessor), which should be available for ASIC design at the end of this year. The core measures just 2 mm^2 in LSI's 0.35-micron three-layer-metal process. The company expects the core to dissipate about 1 mW/MHz at 3.3 V, a negligible factor in the total power budget of most ASICs.

Price & Availability

The TinyRisc core with the MIPS-16 instruction-set predecoder will be available from LSI Logic for new ASIC designs beginning in December. For more information, contact LSI Logic (Milpitas, Calif.) at 800.574.4286 or 408.433.7700, or visit the Web at *www.lsilogic.com.*

Two derivatives are planned for 1998, one faster and one smaller. The TR4120 will add two-way superscalar execution to the basic MIPS core but leave the MIPS-16 predecoder in place. The 4120's core will measure about 5 mm^2 in a newer process that lowers the supply voltage to 2.5 V. In the same process, the 4102 should drop power dissipation to about 30% below that of the 4101.

Size Matters

Obviously, the market pressure to reduce code size is increasing across many embedded applications, or two important microprocessor vendors would not have undertaken the task of shrinking their instructions. The advantages long touted by RISC adherents seem to be a mixed blessing in this environment: embedded RISC chips have consistently led the pack in performance, but they've also brought the burden of bloated binaries.

RISC was designed with the general-purpose computing and workstation markets in mind, with the understanding that memory is cheap. But while memory may be cheap, less memory is always cheaper. For makers of cellular telephones and other cost-constrained consumer items, a $5 difference in RAM or ROM space can make a big difference in volume profits. Often, memory size is fixed and the product's feature set is variable. Tighter object code means more autodial features, better voice recognition, or perhaps clearer images on the screen.

The older CISC architectures have a new competitor here. Their code density has always been the yardstick by which others are measured, but that rule of thumb has now been superseded by two different RISC designs. Now that two major RISC vendors have taken the plunge, who's left? PowerPC and SPARC are possible future candidates; SuperH is already there; Intel's strategic plans make such changes to the i960 family unlikely. Recent emphasis on embedding PowerPC processors may push either IBM or Motorola to make the move. Hopefully, such changes would be mediated and compatible across PowerPC implementations, but a breakaway attempt is also a possibility.

TinyRisc and Thumb show that for some users, code size is an important factor in selecting an embedded microprocessor. They also demonstrate some vendors' willingness to change their instruction sets to meet customer demands. The years ahead will show just how far those demands will push the boundaries of traditional CPU architecture. M