MIPS16: High-density MIPS for the Embedded Market[1]

Kevin D. KISSELL
Silicon Graphics MIPS Group
kevink@acm.org

Origins of MIPS® RISC Technology

The invention of RISC, or Reduced Instruction Set Computer technology, has been credited to several people. Certainly a great deal of the credit must go to John Cocke of IBM's Yorktown research labs, where a processor called the 801 (named for the building in which Cocke and his team worked) pioneered the concept of designing processors and compilers in tandem. In the days when most applications were coded in assembly language, there was some programmer productivity to be gained in providing powerful machine instructions that handled complex tasks, such as single instructions that search memory for a particular value or even solve polynomial equations. The earliest 16-bit microprocessors, such as the 8086, 68000, Z8000 and NS32016 inherited this legacy from the minicomputers they were intended to replace.

By the late 1970's, however, more and more application code was being written in higher-level languages such as FORTRAN, Pascal, and C. Compilers for these languages rarely if ever used these complex instructions, so the complexity and silicon area expended to implement them was wasted. Cocke and his group at IBM conducted their project by designing a compiler, then designing a CPU that executed only the instructions that the compiler would emit. This resulted in a very simple and very fast CPU. Cocke was later a visitor at the University of California at Berkeley, where his ideas generated a great deal of excitement, particularly with a group of students studying under Professor David Patterson. They initiated a student project which they called the RISC-I, contrasting their reduced instruction set computer with the previous tendency to build Complex Instruction Set Computer (CISC) designs. Many of the students who worked on the Berkeley RISC-I and RISC-II projects went on to Sun Microsystems, where they formed the core of the SPARC design team. Many of the design features of the SPARC, such as register "windows", came directly from the RISC architecture, and RISC became a general term.

The University of California at Berkeley and Stanford University, both in the San Francisco area, are great rivals in many things, from sports to science. Thus, it is hardly surprising that at the same time as Professor Patterson's students were designing the RISC processor, there was a similar project at Stanford, under Professor John Hennessy. This project was called MIPS, which stood for Microprocessor without Interlocked Pipeline Stages. Again, the principle of hardware/software co-design was used, in this case by replacing some CPU pipeline control logic with intelligence in the compiler that would schedule instructions to use the CPUs resources correctly. Hennessy and his team were pleased enough with the results that they founded MIPS Computer in 1984 to commercialize what was then becoming known as RISC technology. MIPS is now a subsidiary of Silicon Graphics, Inc., and the MIPS architecture is perhaps the most widely known and used RISC.

---

1. A version of this paper appears in the proceedings of the RTS97 conference.

The MIPS instruction set has evolved gradually over the 12 years since it was first defined. The original version is known as MIPS-I, and all MIPS processors are capable of correctly executing MIPS-I code. MIPS-II added better multiprocessor synchronization instructions. MIPS-III added support for 64-bit addressing and 64-bit integer data types, as well as a richer set of floating-point instructions. MIPS-IV added a prefetch capability, as well as further refinements for high performance floating-point calculation. MIPS-V was announced in October of 1996, and adds support for SIMD (Single-Instruction, Multiple Data) operations on single precision floating point values. These last enhancements are quite significant at the high end of the MIPS product line, in supercomputing applications and for real-time 3D graphics. Most MIPS processors are MIPS-II or MIPS-III.

Characteristics of RISC Architecture

There has been, and continues to be, some debate as to exactly what constitutes a "RISC" processor. It is generally agreed, however that they can be characterized by having a load-store architecture and a fixed instruction size, both of which substantially simplify the design. A load/store machine can only read memory with a load instruction and can only write memory with a store instruction. They cannot, for example, increment the contents of a memory location with a single instruction: one must load the value into a register, increment the register, and store the new value to memory. This need for intermediary storage makes it highly desirable to have a large number of on-chip registers. A fixed instruction size means that all discrete operation that can be performed by the CPU must be expressed in the same number of bits. Most RISC architectures are designed around a 32-bit instruction word. They were developed at a time when memory technology made 32-bit addresses (thus 32-bit registers, and thus 32-bit memory words) desirable. 32 bits is sufficient to encode a reasonably rich set of operations, and to give them the ability to access a reasonably large register set. The basic MIPS instruction set supports the encodings shown in Figure 1.

I-Type (Immediate)

Opcode (6 bits)	Source Register (5 bits)	Target. Register (5 bits)	Immediate Value (16 bits)

J-Type (Jump)

Opcode (6 bits)	Jump Target Address (26 bits)

R-Type (Register-to-Register)

Opcode (6 bits)	Source Register (5 bits)	Target Register (5 bits)	Dest. Register (5 bits)	Shift Amount (5 bits)	Function Code (6 bits)

Figure 1. MIPS-I Instruction Encodings

RISC Code Density

As in most engineering decisions, RISC architecture involves certain trade-offs. RISC designs are easier to pipeline, and generally support a higher clock rate and higher performance, but they also generally require more instructions to do the same amount of useful work. This translates to a high instruction bandwidth requirement, which is usually satisfied by an instruction cache. Moreover, since all instructions are the same size, a certain amount of both program memory and instruction bandwidth is wasted for those simple instructions that could in theory be expressed in fewer bits. Indeed, in CISC achitectures, which have a variable instruction size, simple operations have a compact expression. Thus RISC processors have historically, and by and large correctly, had a reputation for having relatively poor code density.

For the manufacturers of workstations who were the first adopters of RISC CPUs, this "code bloat" was a small price to pay for the advantage in performance to be gained. By the late 1980s, every major workstation vendor had converted to one of the RISC CPU families. Largely due to the performance advantage, and in part due to the intrinsically lower cost of a simpler CPU, RISC also made rapid headway in the real-time sector. But for many embedded applications, RISC was not an attractive solution. There was not a perceived need for 32-bit capability, and in mass-production applications the disadvantage in code density implied more memory and a more expensive bill of materials. This situation has now changed

Evolution of the Embedded CPU Market

The embedded uses of microprocessors ranges from simple logic replacement functions to highly complex applications that require what once would have been considered to be mainframe-class performance. Simple 4- and 8-bit CPUs suffice for many logic replacement functions, but the increasing capability and complexity of high-end embedded and real-time applications has started a notable trend toward the use of 16 - and 32-bit CPUs. See Figure 2.

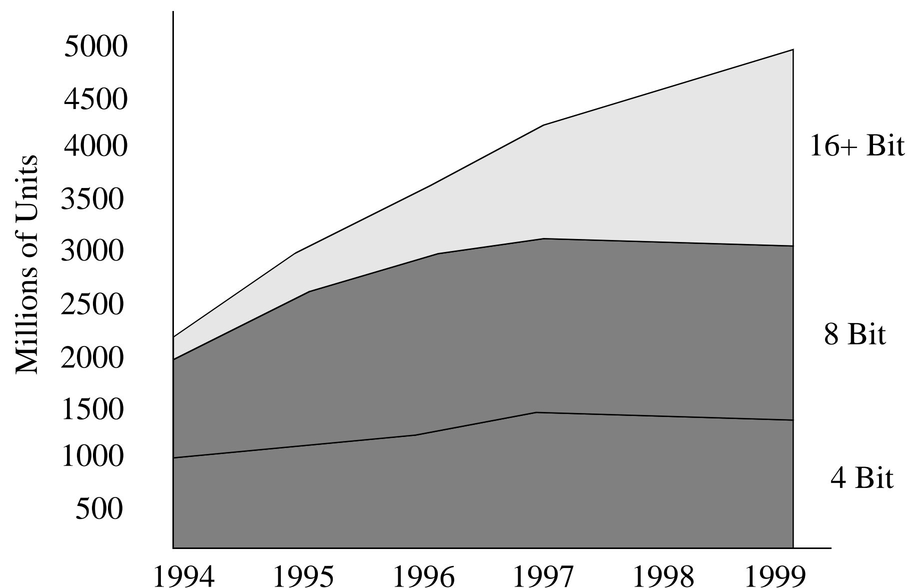

Figure 2: Trends in Embedded Microprocessor Technology
(Source: Dataquest and Semico Research)

This reflects partly the need for more raw computing power, and partly the need to use more advanced software tools and methodologies to achieve reliability and reduce the time to market of complex systems. The share of 16/32/64-bit processor based designs is expected to reach nearly a third of the total by the end of the century.

Another factor in this evolution is the trend toward "system on a chip" solutions. The level of integration attainable with today's semiconductor technology allows most or all of the digital logic required for an embedded product to be placed on a single chip. For cost-sensitive applications with sufficiently high production volumes, this is a very attractive alternative. The premium on silicon "real estate" makes RISC architecture the natural choice for an embeddable processor core. One MIPS RISC core, the LSI Logic CW4003, measures only 1.8 square millimeters. This sort of technology makes possible a huge variety of innovative solutions, but the ASIC environment exacerbates the problems already mentioned with RISC instruction sets: the need for high instruction bandwidth and the increase in code size.

Memory structures in ASIC processes and methodologies tend to be both slower and less dense than is readily achieved in dedicated memory chips. This puts downward pressure on the size of on-chip instruction caches. In addition, highly integrated designs put a premium on I/O interfaces. A 32-bit memory interface may simply not be practical for a low-cost, highly integrated component, yet making a 32-bit RISC wait two cycles for each instruction fetch from a 16-bit memory interface can negatively impact performance.

MIPS16: An Architecture Extension for Compressed RISC Code

MIPS16 is a solution to the code density and bandwidth issues for MIPS RISC designs. It is classified by MIPS as an "architecture extension", meaning that, while MIPS16 support is not mandatory for all future MIPS implementations, it is the standard mechanism for code compression across all suppliers of MIPS RISC CPUs. As the name suggests, MIPS16 is a 16-bit instruction encoding. It has been designed to be 100% compatible with the existing 32-bit (MIPS-I/II) and 64-bit (MIPS-III) architecture and programming model. It will be available in multiple implementations from multiple vendors of MIPS RISC microprocessors and cores, and supported by multiple compiler and tool-chain vendors.

MIPS16 maps onto the established MIPS architecture by presenting a subset of the CPU to the programmer. Each MIPS16 instruction corresponds to exactly one MIPS-III instruction. MIPS16 instructions can be translated into MIPS-III instructions on-the-fly using relatively simple translation hardware. This can be done serially as a preprocessor between the I-cache and the standard MIPS instruction decode, or in parallel in a 16/32-bit decoder. The reference model from MIPS uses the serial model, as it is modular, portable, and easy to verify. Figure 3 shows the flow of compressed and uncompressed instructions in this model.

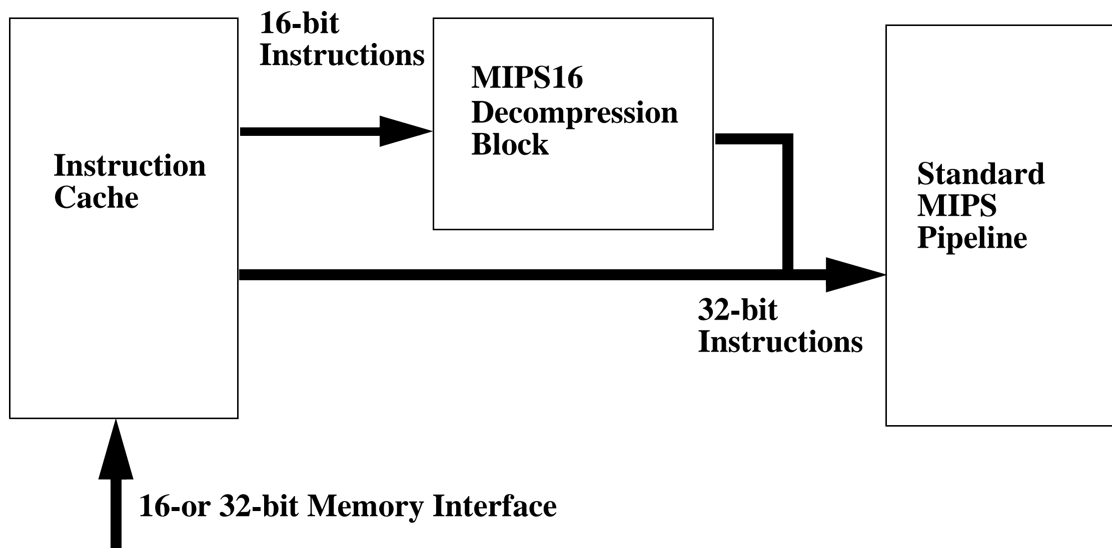

Figure 3. MIPS16 Decompression

The MIPS16 Design Trade-offs

MIPS16 can be thought of as a *reduced* reduced instruction set. To reduce the number of instruction bits by half, one must attack all three components of the instruction word, opcodes, register numbers, and immediate values.

The first step that was taken was to perform statistical analysis on a number of MIPS binaries from a variety of applications from embedded, real-time, and workstation environments. This was to determine the frequency distributions of opcode use, of the number of registers simultaneously in use, and of the number of significant bits in immediate values. The results showed that, while the opcode and function code fields could be reduced, and some instructions “thrown away”, the MIPS instruction set was already very lean. While the base MIPS instruction set has 6 bits of major opcode field, sometimes modified by a 6 bit function code, MIPS16 reduces the major opcode field and the function modifier to 5 bits each, and defines a total of 79 instructions, of which 24 are only required for MIPS-III implementations supporting 64-bit data words.

More leverage was to be had in reducing the size and number of register specifier fields in the instructions. The analysis showed that, most of the time, compiler-generated code was using 8 or fewer registers. Restricting MIPS16 to 8 registers allows register specifiers of 3 bits instead of 5. The R-Format standard MIPS instructions support 3 operands, two inputs and an output. In many cases, MIPS16 only permits two register specifiers per arithmetic instruction. One of the input registers must also be used as the result register, overwriting that input.

Perhaps the biggest saving comes from restrictions on the size of immediate values expressible. In the place of the 16-bit immediate field of the MIPS I-Format instructions, most MIPS16 immedi-

ate fields are restricted to 5 bits. Some are restricted to 3 or 4. An example of the resulting mapping is given in Figure 4.

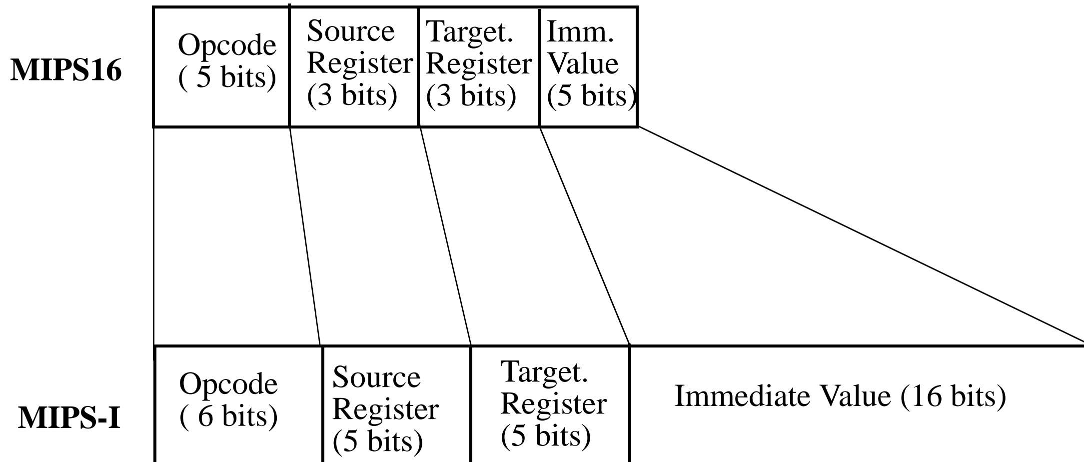

Figure 4. Mapping of Compressed Instructions

Overcoming the Restrictions

The basic MIPS16 encodings restrict the register name space significantly and the immediate operand range severely. Accordingly, MIPS16 contains mechanisms to help overcome these restrictions.

- EXTENDed Instructions
- PC-relative Addressing
- SP-relative Addressing
- Load/Store Offset Shifts

The EXTEND instruction prefix contains only an opcode and an immediate value. It does not generate a MIPS instruction on its own, but instead contributes 11 bits to be concatenated with the 5 bits of immediate data carried in the following MIPS16 instruction. EXTENDing an instruction to 32-bits in this manner allows the same order of immediate value magnitude as is available in the native MIPS instruction set.

For immediate values greater than 16 bits, the standard MIPS instruction set uses the “Load Upper Immediate” (LUI) instruction to load the upper 16 bits of a register, which can be followed with an “Or” of an immediate value into the lower 16 bits. MIPS16 does not support the LUI instruction, but instead introduces program counter relative addressing.

To exploit PC-relative addressing, 32-bit, or even 64-bit, constants can be embedded in the code segment by the programmer or compiler, typically between the bodies of subroutines. Instructions

within the nearby routines can reference this data with a single instruction. Even with the overhead of the constant storage in the code space, this is more compact than the two 32-bit instructions required by the basic MIPS instruction set. It is also possible to do arithmetic using the value of the program counter as an input, which is useful for manipulating strings and data structures embedded in the code space.

The stack pointer is another special register in MIPS16. In the base MIPS instruction set, there is no hardware-designated stack pointer. The program stack pointer is by convention maintained in one of the general purpose registers, $29. MIPS16 refers to it implicitly through special opcodes in order to conserve precious register name space. Loads and stores may be done relative to the stack pointer and the program counter with 8 bits of immediate offset rather than the usual 5, since the base register is implicit in the opcode.

One further mechanism for getting the maximum advantage from the limited immediate value range available to MIPS16 code is the promotion of load-store offsets to aligned values. Loads from unaligned addresses are permitted in the base MIPS instruction set, and the immediate value associated with a load or store instruction is simply added to the contents of the base register to derive the effective address. In MIPS16, load and store offsets are shifted left until they are aligned to the data type being loaded or stored. In the case of memory operations on 32-bit words, the shift is two bit positions. In the case of 16-bit "halfwords", the value is shifted by one bit. Taken together with the PC or SP relative addressing modes, this makes possible the direct addressing of relatively large amounts of data despite the constraints on immediate offsets. Word loads relative to the stack pointer can span a range of 1K bytes of memory without the need to EXTEND the instruction. If EXTEND is used, this shifting feature is unneeded and is disabled. Thus unaligned accesses can still be generated if necessary by using the EXTEND prefix.

Switching Between MIPS16 and 32-bit Modes

The MIPS16 architecture provides for the efficient run-time switching between compressed and 32-bit modes of operation through the JALX, or Jump And Link with eXchange, instruction. This is like the MIPS-I JAL instruction in that it transfers control to the specified address while saving the address of the instruction logically following the jump, the return address, in a link register. The JAL is the MIPS mechanism for subroutine calling. The JALX extends the semantics by also toggling the state of the instruction decode logic between 32-bit MIPS mode and MIPS16 mode. A 32-bit subroutine can call a 16-bit subroutine, and vice versa. The previous state is merged with the return address, and restored automatically on return from the subroutine.

Similarly, on traps to the operating system, the instruction mode is merged with the exception program counter, and is restored on return from the exception. Most operating system kernels will require no modification to support MIPS16 CPUs and MIPS16 binaries.

Which Registers Are Visible

Only 8 of the 32 general MIPS registers are normally visible to MIPS16 code. The choice of these registers was made so as to preserve the standard MIPS calling conventions used by compilers. This minimizes the work necessary for compiler writers to support MIPS16. The mapping can be seen in Figure 5.

The 24 general MIPS registers that are not directly visible to the MIPS16 instruction stream can nevertheless be accessed using the MOV32R and MOVR32 instructions, which copy between the specified MIPS general register and the specified MIPS16 register.

The MIPS16 instruction set specifically excludes coprocessor instructions, including floating point instructions and those which reference the “system” coprocessor. These instructions must be in 32-bit code routines, but those routines can be called from compressed MIPS16 code.

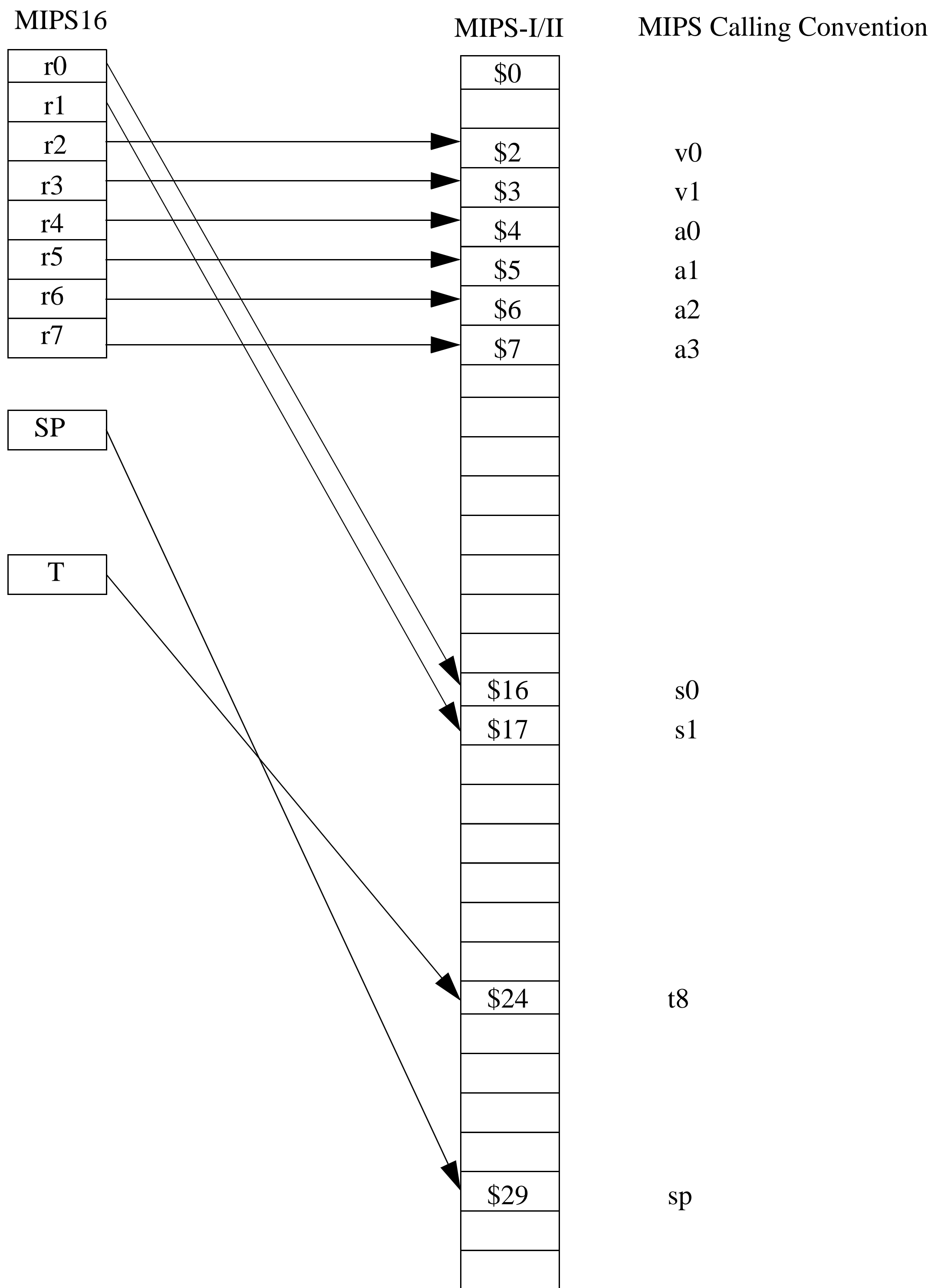

Figure 5. MIPS16 Register Mapping

Handling of Conditional Execution

The MIPS instruction set has no designated condition code bits or condition registers. The execution of conditional branches is determined by the state of general registers. Since general registers are a scarce resource in the compressed instruction set, MIPS16 has a more restricted repertoire for conditional branching. Branches may always be made conditionally on any MIPS16 register being equal to or not equal to zero. In addition, one of the MIPS general registers ($24) serves as a special condition register, called "T", for handling inequalities. "Set-on-less-than" instructions, which in the base MIPS instruction set may specify any general register to hold the result, implicitly use the T register as the destination. The base MIPS instruction set has no "compare" instruction as such. Programmers are expected to perform subtraction explicitly and test for a zero result. In MIPS16, the shortage of available registers makes a non-destructive compare more desirable, and a non-destructive compare-immediate instruction is provided, which maps into an exclusive-or of the specified register with an 8-bit immediate field, again using the T register as the implicit destination. The BTEQZ and BTNEZ then allow branches based on the zero or nonzero state of the T register.

Effectiveness of MIPS16

MIPS16 instructions are, except when EXTENDed, half the size of their standard MIPS counterparts, but also somewhat less expressive. The careful design of the compressed instruction set has minimized the impact of this loss of expression. More instructions are required to perform some operations, but with compilers optimized for MIPS16, a net code saving of 40% has been achieved across a range of embedded and desktop codes. This makes MIPS16 code density better than any conventional RISC or CISC instruction set.

The performance effects of MIPS16 are complex and variable. While the larger absolute number of instructions can have a negative impact, the higher code density improves the hit ratio of the instruction cache and reduces the off-chip instruction bandwidth requirements, which can, depending on the application, more than make up for the increased instruction count.

Conclusion

MIPS16 is a very efficient code compression mechanism that preserves architectural and binary compatibility with the long established MIPS RISC architecture. While other high-density instruction sets have been proposed, no other scheme provides for 64-bit data or 16-bit EXTENDed immediate fields.

Acknowledgments

I would like to thank Earl Killian at MIPS and Hartvig Ekner of LSI Logic for defining the MIPS16 instruction set architecture and for their assistance in preparing this paper

References

For more complete information on MIPS16 see **http://www.mips.com/arch/MIPS16**