A decompression core for PowerPC

by T. M. Kemp
R. K. Montoye
J. D. Harper
J. D. Palmer
D. J. Auerbach

Code size efficiency is a critical parameter in the design of computer systems for embedded applications. This paper describes a method for improving code size efficiency involving the use of compression techniques to reduce the size of the stored code, and on-the-fly hardware decompression at full processor speed for execution. A simple frequency-based encoding scheme for PowerPC® code achieves a typical code size reduction to 60% of the original size. A corresponding decompression core has been implemented for an embedded microprocessor, such as the PowerPC 401™. The compression/decompression scheme operates in a manner transparent to the processor and requires no changes to such tools as compilers, linkers, and loaders.

Introduction

The continuing validation of Moore's law [1] with regard to microcontrollers has made the cost of embedded computation smaller and smaller. Today's 32-bit microcontrollers occupy less space than the 8-bit microcontrollers of only five years ago, yet have 10 to 100 times the computational power. This trend has led to newer designs incorporating the more powerful processors in equipment and appliances.

Taking advantage of this increased processor power and increasing application complexity, the sizes of the programs running on these embedded controllers have grown exponentially. The silicon area and cost of the program memory of a common embedded application now overshadow the size and cost of the processor. For example, in an application such as a high-end hard-disk drive, where an embedded processor may occupy a silicon area of about six square millimeters, the program memory for that processor takes 20 to 40 square millimeters. Thus, some design focus has turned again to the size of memory for computer programs.

Different microcontroller manufacturers have dealt with program size problems by modifying their architectures in different ways. One method is to create a new architecture with consistently smaller instructions; however, the programming environment changes, and all new tools must be created. Another way is to add processor front ends that will interpret a set of smaller instructions with limited capabilities. The same programming environment is used, but new instructions mean that the tools change again. Third, new instructions are added to the architecture, usually to combine the functionality of several instructions in the existing instruction set. The requirement for new or modified tools is less in this case, but still remains significant.

These solutions all share a significant problem: They affect the programming development tools. Since each one involves a new set of instructions, any assemblers, compilers, linkers, and debuggers must be reworked for the new architectures. These modifications require money and time to implement, particularly if the tool implementors are separate tool vendors.

Concepts

The goals for reducing the instruction storage space for the PowerPC* [2] were therefore to significantly reduce instruction store space, minimize performance restrictions on the processor, avoid restricting processor capability, and minimize required modifications to development tools. The last restriction was clearly a difficult one. How could an instruction architecture be modified to reduce the

©**Copyright** 1998 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8646/98/$5.00 © 1998 IBM

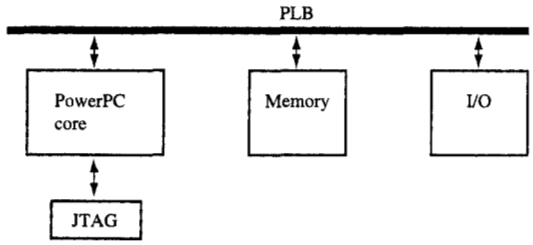


Figure 1

Embedded PowerPC system configuration.

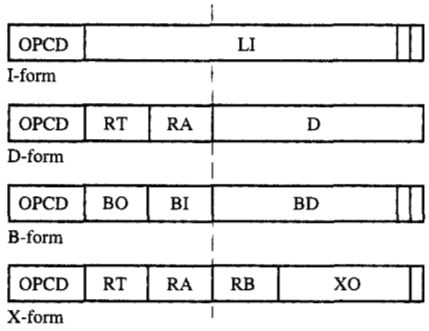


Figure 2

PowerPC instruction formats.

space it takes without making significant tool changes? Clearly we could not modify the instruction format, nor modify how the processor sees its environment.

As shown in **Figure 1**, all embedded PowerPCs consist of a processor core that is attached to its environment via an on-chip bus, the processor local bus (PLB). The processor's memory and I/O devices sit on this bus, providing the processor with its view of the operating environment. The debugging interface is handled by an extension to the JTAG [3] interface, which is connected to a special port to the processor core. Using this port, a special tool allows an external computer to manage program debugging and system control, which are conducted through the processor's PLB interface. Thus, as long as the processor is attached to interfaces providing standard formats, it doesn't matter how the program data are actually stored.

The simple, direct solution to the code size problem is to make the contents of the memory smaller, leading us to investigate code compression. The most common

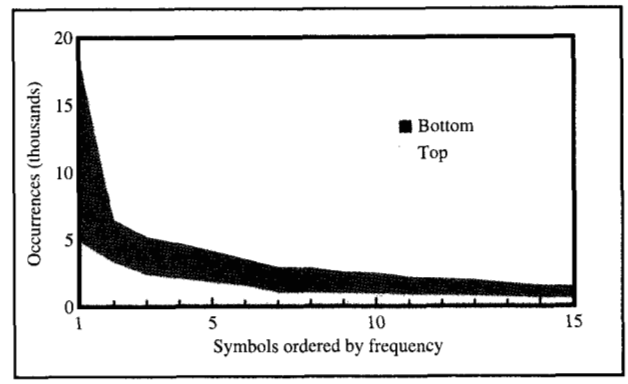


Figure 3

Frequency analysis of instruction half values.

compression algorithms in use today are table lookup, either adaptive, as with Lempel–Ziv [4] or static, e.g., Huffman [5]. An adaptive solution seemed inappropriate for the application, since adaptive lookup schemes depend on a history of the preceding symbols in a sequential file. A static symbol-based lookup mechanism avoids this problem, since decoding any particular symbol requires no knowledge of any preceding symbols. Thus, we elected to use a static encoding scheme. The next task was to choose the symbols and a specific encoding scheme.

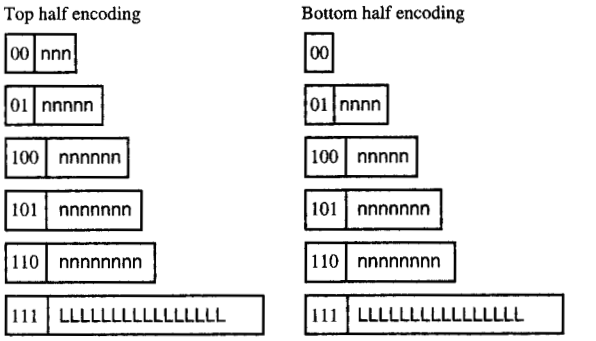
The best known simple static compression mechanism, as described by Huffman, is simple substitution of symbols of a shorter length than the originals for the most frequent symbols, and longer for symbols occurring less frequently. The most obvious symbol choice for PowerPC is the instruction. It is regular and is always 32 bits long. **Figure 2** shows the main instruction formats.

We performed a frequency analysis of full 32-bit-instruction-sized symbols over a large set of PowerPC code, and found that the size of the substitution symbol set was not advantageous. There are too many different common instructions, none of which occur frequently enough to provide an advantage. We needed a smaller set of symbols.

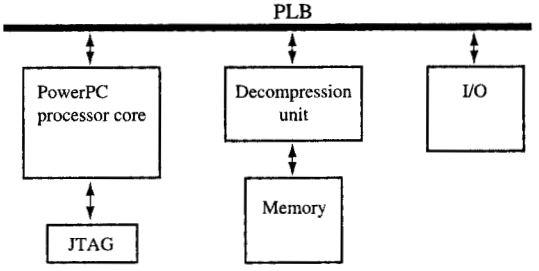
Dividing the instructions in two to provide 16-bit symbols produced a better set. Notice from Figure 2 that the top half of an instruction always contains the instruction opcode (OPCD). In the case of the D-form and X-form instructions, the next two fields, RT and RA, are always registers. The bottom half of the instructions are often immediate values (D) or branch displacements (D, BD). Observing the basic differences in the two halves of the instruction, we did separate frequency analyses on the two halves and discovered that the frequencies of values in the two sets were quite different—so different,

Table 1 Symbol frequencies.

Symbol no.	*Top*	*Bottom*
1	4768	12963
2	3782	3101
3	2346	2815
4	2038	2594
5	1764	2315
6	1500	1957
7	1015	1863
8	994	1825
9	963	1532
10	930	1507
11	846	1288
12	794	1221
13	770	1152
14	702	998
15	662	871
16	581	866



Compressed symbol format.



Embedded PowerPC configuration with decompression.

in fact, that a compression by replacing each half of the instruction from a custom substitution table for that half produced substantially better results than using a combined table.

Figure 3 and **Table 1** show a portion of the frequency distribution of values of the top half of instructions compared with the bottom half of instructions from a program for an embedded controller. The two halves differ both in the most frequent values and in the distribution of those values. The bottom half of the instruction has a much higher count for the most frequent symbol than does the top half. The occurrence frequencies drop more sharply for the bottom than the top, crossing over at about symbol number 56. Each of the lists of unique symbols is over 3800 entries long. Many of the symbols occur only once. These great differences in the early shape of the curves led us to decide to encode the two halves of the instructions separately and to pick a near-optimal substitution symbol set for each of the instruction halves, as shown in **Figure 4**.

Each symbol format consists of a tag of two or three bits followed by a symbol value, shown by a number of bits (each bit represented by n). In the top half encoding, the first symbol format, with a tag of 00, can encode eight symbols. The second symbol format, with tag 01, can encode 32 more symbols, and so on. We pick the eight most frequent symbols to be encoded in format 00, the next 32 most frequent to be encoded in symbol format 01, down to those least frequent symbols for which we use the original 16-bit literal value, shown with L representing each bit, encoded in format 111. A notable feature is that the first encoding of the bottom half of the instructions contains only one value: zero. Zero is so prevalent that statistically it deserves an encoding of its own.

The net compression factor with this technique is that the compressed code is typically about 56% of the original size.

Implementation

Since we previously observed that the processor accesses the memory via the PLB, the obvious place to put the decompression unit was on the PLB just in front of the memory. **Figure 5** shows that the decompression unit accepts the processor's addresses and then looks them up in the compressed memory. Thus, the processor uses instructions and addresses them just as though the code were not compressed.

To fetch an instruction, the processor provides the decompression unit the address of the original,

Table 2 Compression factor with matching and mismatching compilers.

	Matching compiler (%)	*Mismatching compiler* (%)
ate	55	96
csh	60	100
disk1	67	101
disk2	55	105
disk3	66	101
disk4	65	102

uncompressed instruction. The decompression unit maps the address to the location at which the compressed instruction is kept, in a presumably smaller space. We define the index table as a two-column map of "uncompressed space" addresses to "compressed space" addresses. Theoretically, for every address the processor presents, the index table has a location of the compressed instruction.

Thus, the basic algorithm for the decompression unit is as follows:

1. Receive memory address of instruction from processor.
2. Look up compressed address in index table.
3. Fetch compressed instruction from memory.
4. Decompress instruction by looking up uncompressed symbols.
5. Return instruction to processor.

The performance of this simple algorithm is somewhat slow. Presuming a one-cycle memory, we must take one cycle to look up the compressed address, one cycle to fetch the compressed instruction, and one cycle to decompress, for a total of three cycles. We have implemented some optimizations to improve this latency.

Optimizations

In the compressed memory space, to optimize memory usage, we pack the compressed symbols as tightly as possible. Since the lengths of the symbols are not a convenient modulo of addressable memory, it is difficult to address every possible instruction. If every instruction were addressed at the bit level, the index table would grow to an unacceptable size, reducing the advantages of compression. Rather than using one index table entry per instruction, we divide the code into fixed-size "compression blocks" and provide an index table entry for each. This approach controls the size of the table. However, it has the disadvantage that since we cannot easily locate individual instructions within a compressed space, we must search for the desired instruction from the beginning of each block of compressed instructions. As compression block size grows, the average random access time increases. A good engineering compromise puts the compression block size at 64 bytes, with 16 instructions in a compression block. Our performance is improved here, since once the block is accessed, the decompression unit can pipeline the stages of the decompression, yielding a three-cycle initial latency, with a rate of one instruction per cycle.

The PowerPC processors are cached and usually fetch cache lines rather than single instructions. When a compression block is retrieved from memory and decompressed, the instructions in the compression block sequentially after those specifically requested are kept. We presume that the processor will continue executing instructions sequentially and will therefore request the next cache line's worth of instructions. The decompression unit will be able to provide these instructions with no memory or decompression latency. Pairs of compression blocks are always kept together, so that as the processor sequentially asks for the first address of the second compression block, the decompression engine can locate it directly without use of the index, saving a cycle.

PowerPCs have memory management units as well. A bit (here called the "K" bit) can be placed in the TLB entries for a virtual memory manager and in the storage attribute register for real-mode managers. It can be used to indicate, on a page basis, whether or not compression is implemented. This allows small routines with critical performance requirements to remain uncompressed in the midst of a larger system of compressed code.

Measurements

We evaluated the compression mechanism by compressing a number of programs compiled by a variety of compilers. We found that two factors greatly affect the degree to which a program may be compressed: compiler selection and application execution effects.

The first factor is related to whether the compiler used for the program matches the symbol table in the decompression unit. Various C language compilers use very different methods of allocating registers and generating instruction sequences. The instructions generated by a compiler map to a set of input symbols, particularly for the top half of the instructions, that may have an entirely different frequency spectrum than that for instructions generated by another compiler. The compressed images generated from a program compiled, for example, by the GNU C compiler and compressed with a symbol table generated from a frequency spectrum specific to programs generated by the AIX* XLC compiler show remarkably poor, and sometimes negative, compression. See **Table 2** for some specific examples.

The dynamic properties of the program execution determine how much delay due to decompression is experienced. Two factors affect the execution the most strongly: basic block size and working set size. A basic block is a set of instructions which has no branches and is executed linearly. Since the decompression unit is prefetching and decompressing blocks before the computer asks for them, performance depends on the percentage of time that the processor actually asks for data from a prefetched block. If the average basic block size is large, this prefetching activity is generally successful at hiding the decompression latency. The working set is the set of instructions that are most frequently executed. In a caching system, if this working set is smaller than the cache size, memory is infrequently accessed. For a compressed memory, the less the memory is accessed, the less frequently the decompression latency occurs. Thus, programs that execute a small part of their actual code set will execute proportionately faster.

The IBM Microelectronics Division has implemented the decompression core in IBM CMOS 5S ASIC technology and added it to the PowerPC core catalog. The decompression core consists of 25484 logic cells, and the lookup table ROM for two 512-entry tables is 10814 cells (memory and access logic). Silicon area for the core is about 1 mm^2.

Future work

In the future, we would like to address a couple of issues. One is to improve addressing efficiency further, so that when a cache line near the end of a compression block is requested, the decompression unit, remembering the location of the last decompression block, will prefetch and decompress the next block, further reducing latency.

Another is the problem of latency when a branch is taken to an arbitrary location. The average latency is the basic three cycles, plus half the number of instructions in a compression block, a total of 11 cycles in the present implementation. In a system that experiences a low cache-hit ratio, this latency becomes significant. This may be addressed by keeping a branch address cache in the decompression unit. This would eliminate the index table lookup and the counting of instructions in the compression block.

Conclusions

Overall, it is possible to reduce the size of PowerPC code significantly through compression. The net compression, including the index tables, for average programs is to about 60% of the original program size. This is a more dense code than for any other 32-bit processor, RISC or CISC.

This method of reducing the code size is simple and inexpensive, and it required no changes to compilers, assemblers, or linkers. One step, the compression step, must be added to the code creation chain. The compression program is run on the load image after linking and relocating. The JTAG-based debuggers that use the processor's instruction and data trap capabilities need not be changed either. A debugger that sets program traps by modifying memory must be aware of the compression.

A particular implementation of this compression mechanism is compiler-dependent, but only by differences in the symbols loaded in the symbol tables. Thus, a single design with a separately programmed ROM or flash memory will suffice for a variety of uses.

The granularity of the compression is based entirely on memory, rather than on a mode of the processor. Thus, a single thread of execution may switch in and out of compressed code effortlessly. The performance penalty of this implementation is minor if the program's cache hit ratio is reasonable. That is, if the processor is reading instructions from its instruction cache most of the time, the compressed memory access delays do not occur often enough to have a significant impact.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. *http://www.intel.com/intel/museum/25anniv/html/hof/moore.htm* .
2. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, C. May, Ed., ISBN 1-55860-316-6, Morgan Kaufmann Publishers, San Francisco, 1994.
3. Joint Testing Architecture Group, *IEEE Standard No. 1149.1-1990*, American National Standards Institute, Washington, DC, 1990.
4. Willard L. Eastman, Abraham Lempel, Jacob Ziv, and Martin Cohn, "Apparatus and Method for Compressing Data Signals and Restoring the Compressed Data Signals," U.S. Patent 4,464,650, August 7, 1984.
5. D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proc. IRE* **40,** No. 9, 1098–1101 (1952).

Received December 9, 1997; accepted for publication June 10, 1998

Timothy M. Kemp *IBM Microelectronics Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (kemp@vnet.ibm.com).* Mr. Kemp is a Storage Platform Architect for the IBM Microelectronics Division. He received a B.S. degree in electrical engineering and computer science from the University of California at Berkeley in 1971 and joined the IBM Advanced Systems Development Division in 1972. Mr. Kemp has worked on a wide variety of systems, including industrial process control, operating systems, network databases, compilers, and electronic computer-aided design. Most recently, he has concentrated on developing electronics architectures for hard-disk drives.

Robert K. Montoye *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (montoye@us.ibm.com).* Dr. Montoye is a Research Staff Member in the Experimental Systems group. He received his B.S. in 1977 in physics and his M.S. in 1981 and Ph.D. in 1983 in computer science from the University of Illinois. Joining IBM in 1983, he designed and implemented the RS/6000 floating-point unit. After pursuing interests outside IBM from 1990 to 1995, he returned to IBM to focus on high-performance and cost-effective memory systems. Dr. Montoye is a member of the IBM Academy of Technology; he has published a number of technical papers and holds twelve patents.

Jeffrey D. Harper *IBM Microelectronics Division, 11400 Burnet Road, Austin, Texas 78758 (jdharper@us.ibm.com).* Mr. Harper received the B.S. and M.S. degrees in electrical engineering from Auburn University in 1986 and 1988 respectively. He joined the Systems Technology Division at IBM in Austin in 1988 and served as technical lead of the PowerMite decompression macro hardware development. Mr. Harper is currently working in the IBM Microelectronics World Wide Field Design Center developing custom ASICs utilizing the IBM Blue Logic technology.

John D. Palmer *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (jpalmer@almaden.ibm.com).* After receiving a B.S. in applied mathematics from Clemson University, Mr. Palmer joined IBM in 1969 at the Poughkeepsie, New York, Development Laboratory, where he worked in TSO development. Following a two-year military leave of absence, he was assigned to the IBM Advanced Development Laboratory in Los Gatos, California, working on IMS and imaging projects. From 1974 to 1982 Mr. Palmer was an MVS system programmer at several different IBM research facilities and development laboratories. He has since worked in research and research management on multisystem IMS and DB2 projects, and his current assignment is in disk-drive microcode and electronics at the IBM Almaden Research Center in San Jose, California. His current interests are in error analysis, disk-drive performance, processor integration, and microcode structure.

Daniel J. Auerbach *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (dja@almaden.ibm.com).* Dr. Auerbach is Manager of the DASD Controller Architecture and Electronics Department at the IBM Almaden Research Center. He received a Ph.D. degree in physics from the University of Chicago; before joining IBM in 1978, Dr. Auerbach served on the faculty of Johns Hopkins University. His research interests include information storage systems, the design of parallel computers, and the dynamics of gas–surface interactions. His work on gas–surface interactions involves the use of molecular-beam and laser-spectroscopic techniques to allow quantum-state-specific studies of the microscopic details of fundamental gas–surface-interaction processes underlying materials processing. Before assuming his present management post, Dr. Auerbach was Department Group Manager of Science and Technology, Almaden Research Center.