



CodePack[™]: Code Compression for PowerPC[™] Processors

Version 1.0

Mark Game and Alan Booker
PowerPC Embedded Processor Solutions
IBM Microelectronics Division
Research Triangle Park, North Carolina

Overview

Today's increasingly complex embedded designs often require the processing power of a 32 bit RISC processor. Designers are also frequently under immense pressure to keep the fixed cost of these devices as low as possible. Creating a high performance, feature-rich product with low cost can be an arduous task. This paper describes a new technology that in combination with IBM PowerPC™ embedded processor cores, can help design engineers attain these goals.

The IBM PowerPC 400 family of embedded processors meets the price/performance demands of a wide range of embedded applications. Like other traditional RISC processors, the code density (amount of instruction storage space required to accomplish a given function) associated with 32 bit fixed-length instructions can require more memory when compared to other processor architectures. To address this concern, IBM has developed CodePack™, a unique method to store complete PowerPC instructions in memory in a compressed format. With this technique, there is no loss of instruction set functionality or flexibility and little effect to existing software development tools. This code compression technique consists of two parts. The first is a silicon-efficient ASIC core that is placed between the processor and the memory controller in an integrated system-on-a-chip design. The core decompresses instructions on-the-fly, only as needed by the processor. The second part of the solution is a software utility that creates compressed application code images from standard PowerPC Embedded ABI executable files.

Typically, use of the CodePack™ decompression core results in application code that fits in 35-40% less space.

This overview briefly describes both the hardware and software aspects of the CodePack compression scheme.

Decompression

Figure 1 shows a design that includes the CodePack decompression core. Data and instructions are transferred between the PowerPC processor core and the decompression core via the Processor Local Bus (PLB), the IBM standard on-chip, high bandwidth bus. The memory controller, which would normally connect directly to the PLB, is connected to the PLB through the decompression core. A system can also include other peripherals from the IBM Blue Logic (TM) Core Program, other on-chip busses, and/or customer provided logic. The decode lookup tables attached to the decompression core contain the information needed to “decode” (decompress) memory that is compressed. The on-chip decode lookup tables can either be fixed when the chip is fabricated, or be implemented in RAM and loaded as part of system initialization for optimum compression.

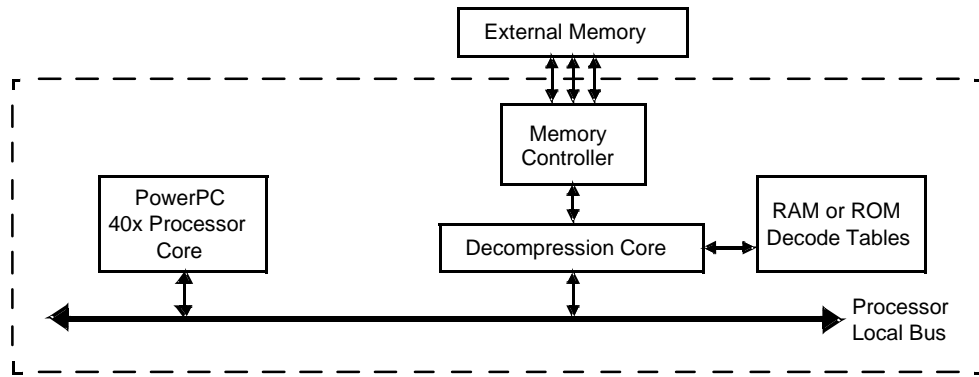


Figure 1. An integrated design including the CodePack decompression core

External memory in a system using the decompression core contains a combination of compressed instructions, uncompressed instructions, data, and potentially other things such as memory mapped peripherals. A signal in the Processor Local Bus (PLB_Compress) allows the processor to control which accesses are subject to decompression processing. The processor does not, for example, drive PLB_Compress during an ordinary data access. In this case, the decompression core passes the request directly to the memory controller.

Compressed memory regions are established in the system using the same method that cached, guarded (a PowerPC memory attribute), or memory with other special attributes are established. If an application uses the Memory Management Unit (MMU) in the processor core, part of each page definition includes the memory attributes of the underlying storage, which includes whether the page is compressed or not. If there is an access to a page that has been configured as compressed, the processor will drive the PLB_Compress signal on the PLB as described above. If the MMU is inactive, a series of registers in the processor core are used to define compressed, cached, and guarded memory regions. These storage attribute registers divide the 4GB address space into thirty-two 128MB pieces, each of which are controlled by one bit in each register. The register for configuring compressed memory regions is called the SKR (Storage Kompression Register). If a bit in the SKR is set to a "1", the corresponding 128MB region of memory will be treated as compressed.

The assignment of compressed storage regions is normally done once in the bootstrap code during processor initialization, but may be changed later if needed. Memory regions containing compressed code are configured as compressed while memory regions containing data, stack, memory mapped devices, etc. are configured as uncompressed.

Compression Architecture

The main goal of the CodePack compression scheme was to develop a method to efficiently compress PowerPC application code which at runtime could quickly be decoded with a small amount of logic. Another key design point was to maintain full instruction set capability. The method used in the decompression core is a variable-length encoding technique. An example of this technique is to replace the ASCII characters in an ordinary text file with variable-length bit patterns. Short bit patterns are substituted for characters that appear frequently in the file, and longer bit patterns are substituted for characters that appear less frequently. The table of substitutions (decode lookup table) used to compress the file is also used later by the decompressor to restore the file to its original size and contents.

To compress PowerPC code, the first thought was to substitute variable-length bit patterns for full 32 bit instructions. This unfortunately did not produce appreciable compression results because the distribution of unique instruction patterns in most application programs is too uniform. However, if the PowerPC instructions are split into two 16 bit halves, with each half compressed separately, significant compression occurs with a relatively small decode lookup table. Using the split instruction method just described and two decode tables of 512 entries each, a PowerPC instruction can be compressed to as small as 7 bits in the best case and 38 bits in the worst case (With this type of encoding technique, the worst case will be larger than the original).

Compressed instructions are stored consecutively in memory making sequential execution easy. However, since compressed instructions are no longer fixed in length and all application programs have execution discontinuities in them (such as branches and exceptions), a method is needed to map the original instruction addresses to their corresponding location in compressed memory. If an index table was created in memory that mapped each possible instruction address to its equivalent compressed address, the benefits of the compression would quickly be lost due to the size of the index table. However, if instructions are assembled into groups, an index table entry is only required for each group and the index table shrinks to a reasonable size with only a small loss in efficiency.

The following definitions and their subsequent explanations help describe how compressed memory is organized in the CodePack compression technique:

TIA - Target Instruction Address. An instruction address from the point of view of the processor prior to compression.

Compression Group - A piece of compressed memory representing an aligned 128 byte piece of decompressed memory. A group contains 32 instructions.

Compression Block - A piece of compressed memory representing an aligned 64 byte piece of decompressed memory. Two sequential blocks make up a compression group. A block contains 16 instructions.

Index Table - A table in memory made up of a series of 32 bit entries that map TIAs to their respective addresses in compressed memory. There is one entry for each compression group. The information in an index table entry allows the decompression core to determine the starting address of both compression blocks within the compression group. Since there is one entry per compression group, index table overhead is 4 bytes per 128 bytes of uncompressed instructions (just slightly over 3%). An index table can be up to 2MB in size which would cover an entire 64MB compression region.

ITOR - Index Table Origin Register. A register in the decompression core whose contents point to the beginning of the index table in a compressed region of memory.

Compression Region - A 64MB region of memory that can be referenced using an entire 2MB index table.

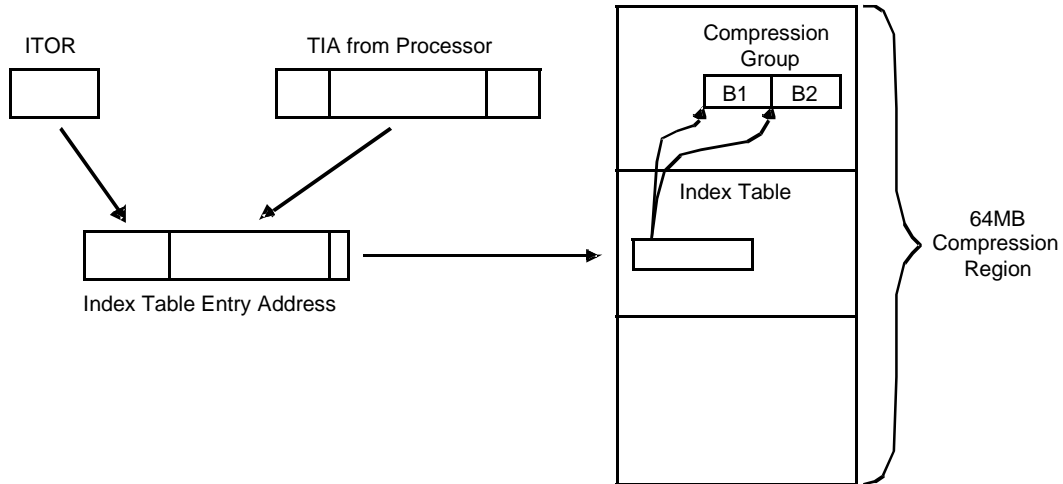


Figure 2. Locating Compression Blocks

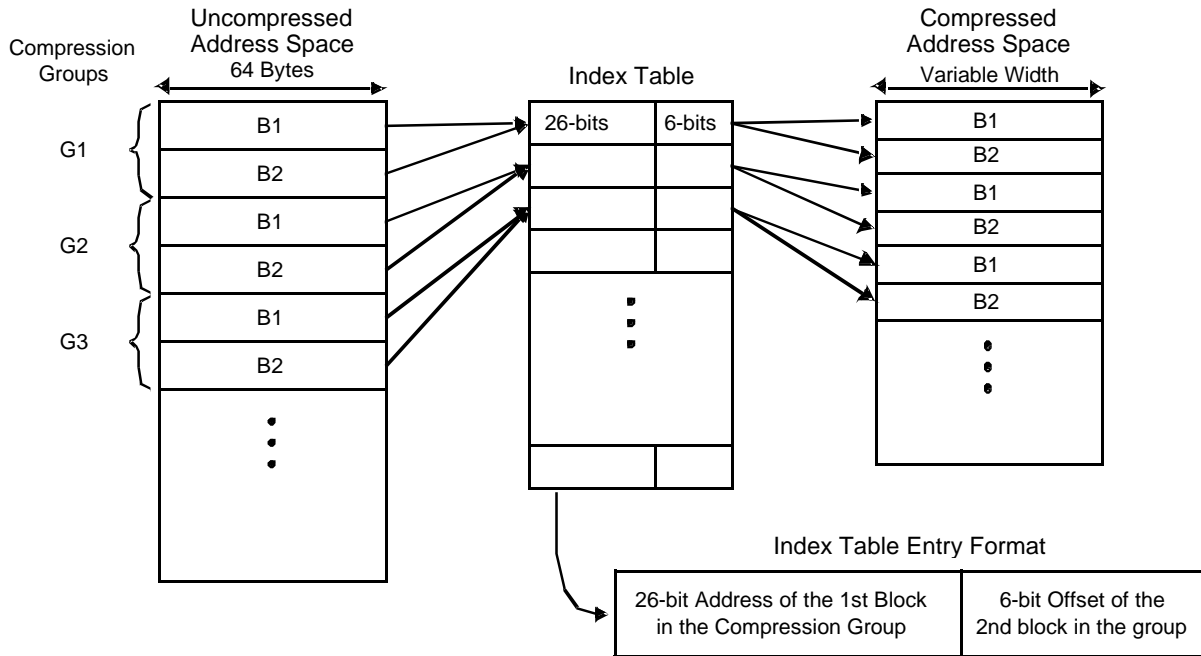


Figure 3. Index Table Mapping of TIAs to Compressed Memory

As the compressed application code runs, the processor constantly fetches instructions. Instructions are located, decompressed, and returned to the processor in the following manner by the decompression core.

1. The processor fetches an instruction at a TIA in memory configured as compressed.
2. Activated by the processor assertion of PLB_Compress, the decompression core calculates the address of the index table entry of the compression group containing the TIA using part of the TIA and the ITOR register. Figure 2 shows how the index table entry address is determined.
3. The decompression core fetches the index table entry from external memory.

4. The decompression core calculates the starting address of the compression block containing the target instruction using the information in the index table entry. Figure 2 shows how the compression block is located using the index table entry.
5. The decompression core reads (burst mode) the compression block containing the target instruction from memory. As the compression block is read in, the decompression core uses the contents of the decode lookup tables to decompress the block.
6. The uncompressed target instruction is returned to the processor. Other than the possible increased latency, the processor is generally unaware of compression.

In Figure 3, several compression groups are shown. The index table entry for each group helps the decompression core locate the appropriate compressed memory block for a particular TIA. Since the index table entries contain information about the location of compressed memory blocks, the compressed groups on the right side of the figure do not necessarily have to be in the order shown.

Performance

There are several features in the decompression core that reduce the impact of decompression latency in the system.

The first feature is a 64 byte output buffer in the decompression core. This buffer holds the last block of 16 decompressed instructions until the processor requests an instruction outside of the block. Requests for instructions in this buffered block can be returned immediately. The output buffer acts like a prefetch buffer in this case.

The second feature is the ability of the decompression core to return an instruction to the processor core immediately after it has been decoded. Consider the scenario where the processor requests an instruction that is not currently in the output buffer of the decompression core. This causes the decompression core to access the compression block that contains the instruction. The decompression core must read and decode the block serially, that is, it starts with the first instruction of the block and continues until all 16 instructions have been read and decompressed. If the target instruction is the 4th one in the block, the decompression core returns this instruction to the processor immediately after decompressing it, continuing to read and decompress the remainder of the compression block in parallel with the processor execution.

Just as the decompression core buffers the decompressed 16 instructions of the last block that was accessed by the processor, it also saves the index table information from the last access. If the processor then requests an instruction that is not in the same compression block as the one in the output buffer but is in the same compression group, the decompression core can calculate the starting address of the new block without having to refetch the index table entry for the group.

Lastly, since instructions have already been decompressed when they reach the processor core, the contents of the L1 instruction cache will contain decompressed instructions ready for execution. This significantly reduces decompression latency in cache friendly applications.

The flow chart in figure 4 illustrates the various performance features described above.

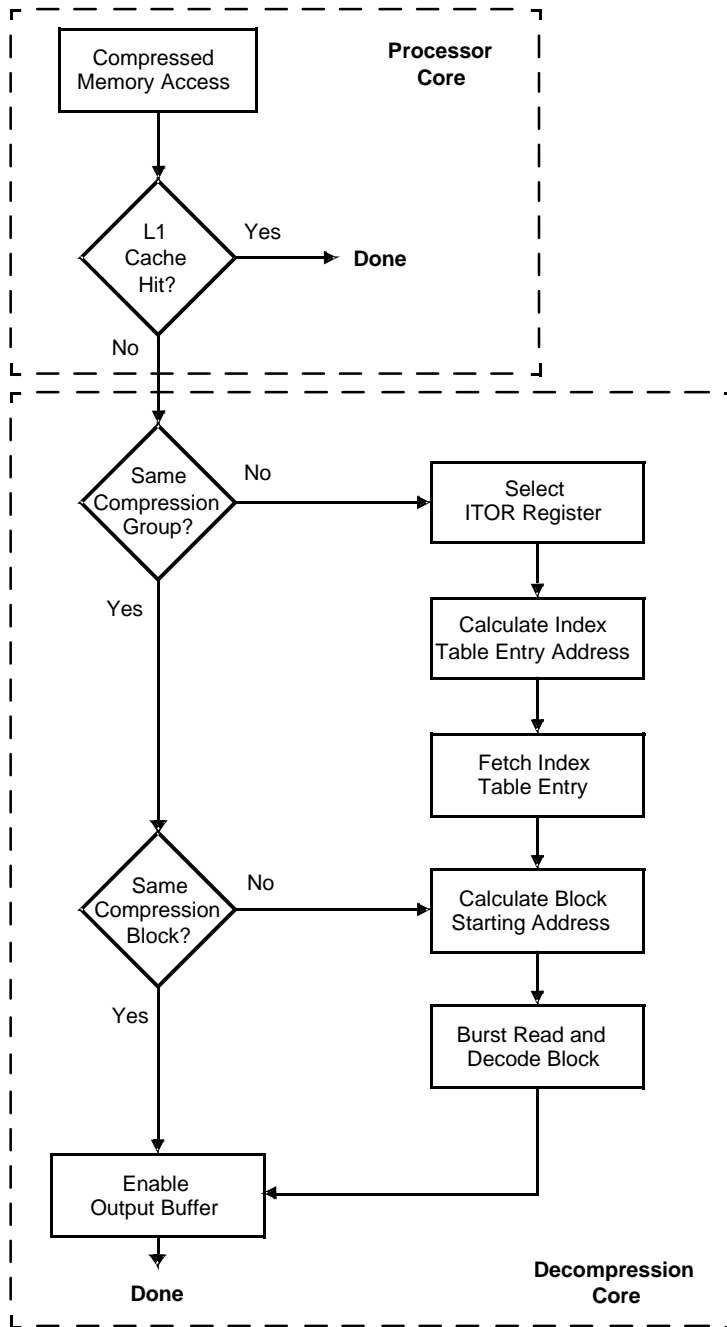


Figure 4. Decompression Flow

The combination of all of these performance features in the decompression core helps to hide any latencies associated with the decompression process. The performance of compressed applications evaluated so far have been within +/- 10% of the performance of uncompressed versions of the same application. In some benchmarks that were run with slower memory (such as FLASH), the compressed code actually runs faster because compressed instructions are smaller and can be fetched from slow memory in less time.

Software Development Tools

Unlike other competitive code compression solutions, the CodePack compression technique has little impact on software development tools. This is because the original uncompressed instructions are supplied to the processor by the decompression core. The tools used to create compressed application programs are the same ones used to create regular 32 bit uncompressed PowerPC application programs. There is only one additional step that must be performed at the end of the process using a software utility provided by IBM. IBM will also make the compression technology accessible to its third party PowerPC tool development partners for integration into their software generation tools. The utility operates on widely accepted PowerPC EABI (Embedded Application Binary Interface) ELF executable files created by the linker. All details of the compression process are managed by this utility which means that special versions of compilers, assemblers, linkers, runtime libraries, real-time operating system libraries, etc. are not required to create compressed applications. Figure 5 illustrates the application build process. The decode lookup table file can be either an input to or an output of the compression utility. The decode lookup table file is used with the compression utility as follows:

- Output - For decompression cores with RAM decode tables. Decompression values in this file are created by the compression utility based on instruction patterns in the application code. The contents of the output table file are copied into the decompression core hardware tables before enabling the decompression core.
- Input - For decompression cores with fixed decode lookup tables. The contents of the input file match the contents of the hardware tables.

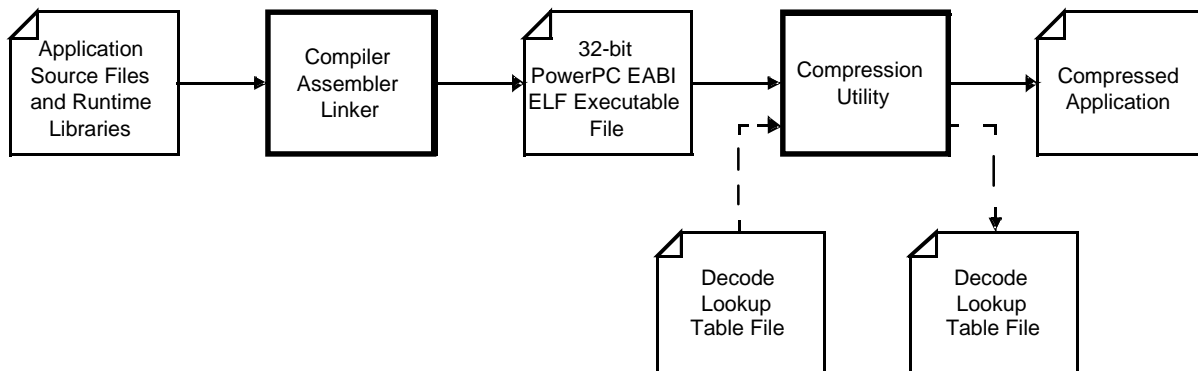


Figure 5. Compressed Application Build Process

The compression utility performs the following functions for decompression cores with RAM decode tables.

1. Counts the frequency of the unique high 16 bit and low 16 bit instruction patterns that exist in the text sections to be compressed.
2. Sorts the two lists according to frequency, and builds two decode tables that the utility will use to compress the code and the decompression core will later use to decompress it. The utility can also accept a table as input for those implementations of the decompression core that fix the decode tables at the time of fabrication.

3. Replaces the high and low 16 bit patterns in the text sections with the appropriate entry from the decode tables. Since the tables have a fixed number of entries, a special tag is used to mark those entries that do not appear in the table.
4. Creates index table entries for each compression group.
5. Creates a binary file or S-record file containing the compressed code and index table entries.

Creating an application that has a mixture of compressed and uncompressed code is quite easy since the utility compresses the application on a text section basis. Text sections are the portions of the application code that contain executable instructions. A mixed application has two or more text sections and the user specifies which text sections are to be compressed and which sections are not. At link time, the user decides which object modules are placed in each text section.

The utility also has options to include uncompressed data sections, symbol sections, and string sections in the output file it produces. This allows for the creation of a file that is ready to load into memory and run.

Debug Tools

When debugging compressed applications, the powerful on-chip debug and trace capabilities of the PowerPC 40x processor cores remain available. Since instructions are already restored to their full 32-bit length when they reach the processor core, the debugger is presented with an uncompressed view of the system when the decompression core is enabled. If the user wishes to use the debugger to view the true contents of memory, the decompression core can be easily disabled. The hardware breakpoint capabilities of the processor cores remain available allowing debuggers to use processor resources to set breakpoints at; instruction addresses, on data accesses, on exceptions, etc. Additional support has been added to the IBM RISCWatch debugger to allow software trap breakpoints and instruction modification in compressed code stored in RAM.

The real-time trace function of the processor core also remains available when executing compressed applications. This capability allows the creation of instruction flow traces with caches enabled providing critical information for software debug and/or performance tuning.

Conclusions

The PowerPC Code Compression scheme is a unique method of improving code density in a 32-bit RISC architecture. It makes it possible for normal 32-bit PowerPC application programs, created with existing development tools, to be stored in memory in a compressed format. Storage requirements and, ultimately, overall system cost is reduced as a result.

CodePack Highlights

- Decompression core occupies a small silicon area.
- Designed to work with IBM's Blue Logic on-chip bus architecture that allows the development of highly integrated, low power, high performance, low cost system-on-silicon solutions.
- Typical improvement in code density is 35-40% over uncompressed 32 bit PowerPC applications.
- Special versions of software generation tools (compiler, assembler, linker, real-time operating system) are not required to build compressed applications.
- All instructions supported by the PowerPC processor core can be used in compressed code, not just a subset.
- The decompression core is designed to minimize performance impacts.
- RAM decode tables allow the compression scheme to be optimized for YOUR application each time it is recompiled.

References

Robert Sedgewick, *Algorithms in C*, Addison-Wesley Publishing Company, Inc. New York, New York, 1990.

Alan Booker, "PowerPC Compression Programming Model", Revision 1.2

T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, D.J. Auerbach, "A Decompression Core for PowerPC", IBM Journal of Research and Development, Volume 42 Number 5/6, September, 1998.

Cathy May, Ed Silha, Rick Simpson, Hank Warren, editors, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

Stephen Sobek and Kevin Burke, "PowerPC Embedded Application Binary Interface", Version 1.0, 1995.

System V Application Binary Interface, Third Edition, UNIX System Laboratories Inc., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

PowerPC Code Compression Utility User's Manual, Third Edition, September 1998.

Steve Mihalik, Steve Sobek, and Steve Zucker, *Programming PowerPC Embedded Applications*, Embedded Systems Programming, December 1995.