

>: Application Note //

Hardware GZIP Decompression

>: 1 Introduction

This Application Note describes the operation of GZIP compression and decompression, and how this can be implemented in Handel-C using the DK1 design suite. The compression algorithm used by GZIP is explained, and the process of porting it to Handel-C is examined in detail. Results are included for simulation, timing analysis and actual hardware performance.

>: 2 Data Compression

Data compression is the process of encoding data in such a way that the *compressed* data takes up a smaller number of bytes than the *uncompressed* data. Compression algorithms can be lossless or lossy, depending on whether the exact original uncompressed data can be recovered or not. Compression utilities such as GZIP, WinZip, PKZip, arc, compress, etc. all use lossless compression, so that the data resulting from decompression is an exact copy of the original uncompressed data. Formats such as JPEG, MPEG and MP3 use lossy compression, as the decompressed data does not need to be *literally* identical to the original data – it only needs to be *perceived* to be identical.

All forms of compression operate by identifying repeated sequences in data, which can be represented using a more compact code. In addition, lossy compression algorithms also identify data which is effectively redundant, and can be removed entirely, without affecting the perceived quality of the decompressed data.

>: 3 Operation of GNU GZIP

The GZIP (or GNU zip) software is a compression utility which is designed to replace the older *compress* utility. It achieves a higher compression ratio, and is free from patented algorithms. More information on the software is available on the homepage: www.gzip.org.

GZIP takes a single file as its input and produces a single file as its output. GZIP can compress and decompress data using the Deflate algorithm. In addition it can decompress files created using the LZW and LZH algorithms.

>: 4 The Deflate Compression Algorithm

The Deflate algorithm achieves a good compression ratio by using a combination of several techniques: LZ77, Huffman encoding and RLE (Run Length Encoding). The following sections will explain these specific techniques, and then how the compression and decompression operate.

4.1 Compression

The Deflate compression algorithm uses several stages to compress a file:

1. LZ77 compression
2. Huffman encoding (dynamic or static)
3. RLE compression of dynamic Huffman tree descriptions
4. Huffman encoding of RLE compressed tree descriptions

Each of these stages will now be described in more detail.

Hardware GZIP Decompression

LZ77 compression

LZ77 compression reduces redundancy in the data by looking for repeated sequences of bytes. The first occurrence of a sequence is stored literally, and subsequent occurrences refer back to it. This is illustrated in Figure 1, where the first occurrence of 'ABCD' in the input stream is represented by literal values in the compressed stream. The next occurrence of 'ABCD' is represented by a *length* and *distance*, which will be used during decompression to locate the sequence of bytes to copy. In this case, the instruction would be to move back six bytes and copy four bytes from that location.

In order for implementation of LZ77 compression to be practical, repeated sequences can only be searched for within a limited window; for the Deflate algorithm, the window is defined as 32KB. LZ77 achieves good compression on a variety of data, as the 'dictionary' of repeated sequences changes with the data being compressed, as the window 'slides' along the input stream. However, it can be very time-consuming during compression, as many comparisons must be made between the input data and the window. Conversely, the decompression is simple and fast, as the location to copy data from is specified explicitly.

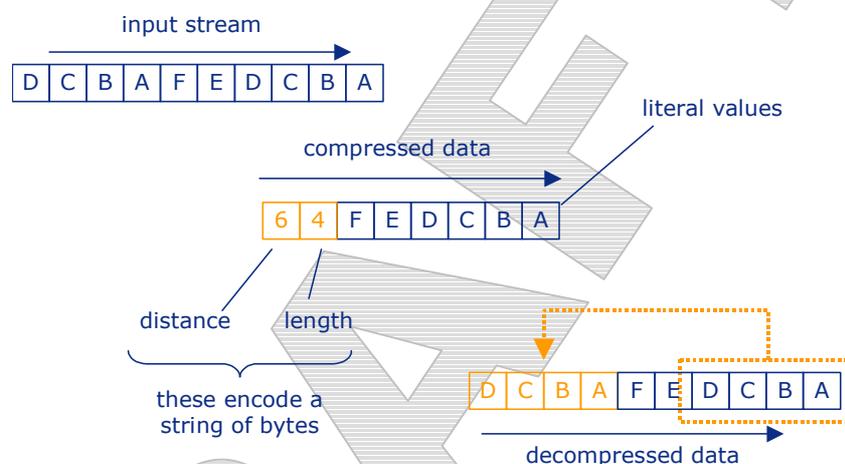


Figure 1. LZ77 compression

Huffman encoding (for LZ77 output)

The output of the LZ77 compression is a mixture of literal, length and distance values. Some of these values occur much more frequently than others, so Huffman encoding can be used to further reduce the size of the compressed data. This is achieved by using codes with fewer bits to represent the common values, and codes with more bits for the rarer values.

The LZ77 output is grouped into two *alphabets*, one containing literals and lengths, the other containing distances. The number and distribution of the symbols within these alphabets is different, so two separate Huffman encoders are used. Each compressed block is encoded using static and dynamic Huffman trees, and whichever delivers the best results is then used.

The description of the static Huffman trees is built into the Deflate algorithm, so when these are used during decompression only the encoded data is required to be sent. However, the dynamic Huffman trees usually deliver the best compression ratio, and these require a description of the Huffman trees to be sent with the encoded data. Every 32 or 64KB of compressed data, a new set of Huffman encoders is built, so the dynamic Huffman tree descriptions may be sent quite frequently. To reduce this overhead, these tree descriptions are further compressed using RLE and another Huffman encoder, as described below.

Run Length Encoding (RLE)

The Huffman trees used to encode the LZ77 compressed data are described by listing the bit-lengths of their codes, in alphabetical order. These descriptions often contain repeated bit-lengths, so RLE can be used to indicate the bit-length and the number of times to repeat it.

Hardware GZIP Decompression

Huffman encoding (for tree descriptions)

The output from the RLE consists of a small alphabet (19 symbols), which is Huffman encoded again, to further reduce its size. This is done using a dynamic Huffman tree, different to the Huffman trees used for the main compressed data. As before, the tree description is sent ahead of the data, although this time the code bit-lengths are stored in a special order, leaving those likely to be zero until the end, so they can be truncated.

4.2 Decompression

The decompression process is the reverse of compression, although it is simpler and faster. The key aspect of the decompression is that it operates on a 1-bit wide stream of data, requiring that all operations reading from the bitstream are performed sequentially. The first stage in the Inflate algorithm (the inverse of Deflate) is to read one bit from the bitstream to identify the final compressed block, then two bits which indicate the compression type. The different compression types are handled as follows:

Stored: the data was not compressible, so is simply stored as a stream of literal values. The number of bytes is indicated, and the required number are read from the bitstream, then written to the output and the LZ77 sliding window.

Static: the data was LZ77 compressed, then encoded using static Huffman trees. The process to handle a static block is:

1. Build static Huffman decoders for lengths/literals and distances (unless they are kept ready-built)
2. Run the Huffman decoders on the input bitstream until an End Of Block (EOB) symbol is found, sending the resulting literals, lengths and distances to the LZ77 decompressor.
3. Run the LZ77 decompressor on the output of the Huffman decoders, writing the resulting bytes to the output and the LZ77 sliding window.

Dynamic: the data was LZ77 compressed, then encoded using dynamically created Huffman trees. The Huffman tree descriptions for each block are sent ahead of the compressed data. The process to handle a dynamic block is:

1. Read description of initial Huffman tree for decoding descriptions of main Huffman trees.
2. Build initial Huffman decoder from description.
3. Run initial Huffman decoder on the input bitstream, generating tree descriptions for main decoders.
4. Build main Huffman decoders from their descriptions.
5. Run the main Huffman decoders on the input bitstream until an End Of Block (EOB) symbol is found, sending the resulting literals, lengths and distances to the LZ77 decompressor.
6. Run the LZ77 decompressor on the output of the Huffman decoders, writing the resulting bytes to the output and the LZ77 sliding window.

4.3 Software implementation of GZIP Decompression

The software implementation of GZIP decompression aims to minimise memory use, so makes significant use of dynamic memory allocation when creating Huffman decoders. The GZIP source code is highly optimised for execution on a processor, to the point where it can sometimes be difficult to follow. The implementation of features such as bitstream input and byte-stream output is closely related to the data types which can be handled by processors.

These factors make some parts of the software implementation more difficult to port to hardware, although the use of Handel-C for the hardware design allows some sections of the source code to be pasted in with little or no modification. The remainder of this document explains the hardware implementation of GZIP decompression in the Handel-C language using DK1, and gives performance characteristics of the resulting design in simulation and in prototype hardware.

Hardware GZIP Decompression

>: 5 Hardware implementation of GZIP decompression

This section covers the implementation of GZIP decompression in the Handel-C language, using DK1. The goal is to decompress GZIP files using an FPGA, which could be mounted on a PCI card, allowing higher performance and reduced load on the host computer.

5.1 Porting GZIP source code to Handel-C

To reduce the time required to implement the GZIP decompression, the software C source code was used as the basis for the Handel-C design. Some aspects of the algorithm had to be implemented differently, to take account of the differences between hardware and software, while other parts could be used with little or no modification. The source code was also enhanced to make use of Handel-C's support for parallelism.

The process for porting the application to Handel-C was to write the code for one 'module' of the system at a time, and generate test data sets from the GZIP software to use with the GZIP Handel-C in simulation. These modules do not necessarily correspond to functions in the GZIP software, but rather to stages in the decompression algorithm. The following section describes these modules in more detail.

5.2 Modules for Handel-C GZIP decompression

The GZIP decompression process was split into several modules, which pass data to each other directly through channels or through shared RAMs. The following list briefly describes the modules' functions, and their input and output.

Top-level function

Function: Detect the type of a compressed data block, and call the appropriate modules to decompress it. Co-ordinate operation of all other modules in the application.

Input: Compressed data from input GZIP file, received through 16 bit channel.

Output: Decompressed data output, sent through 8 bit channel.

Block memory access

Function: Provide a simple interface for pipelined access to Block RAM on Xilinx FPGAs. Required to achieve high clock rates. Used to provide RAMs for Huffman tables and LZ77 sliding window.

Input: Address, and data for a RAM write.

Output: Data from a RAM read

Bitstream reader

Function: Read a variable number of bits from the input bitstream (up to 15 in this implementation). Initiate a read from the input bitstream channel if required. Also allow non-destructive reads of the input bitstream.

Input: Compressed data from input GZIP file, received through 16 bit channel from top-level function.

Output: Bits read from bitstream returned to caller.

Builder for tree description Huffman decoder

Function: For dynamic compressed blocks. Use the tree description generated from the input bitstream to build a Huffman decoder table for the tree descriptions for the main Huffman decoders.

Input: Tree description from input bitstream, accessed through bitstream reader.

Output: Huffman table written to shared Block RAM.

Hardware GZIP Decompression

Tree description Huffman decoder

Function: For dynamic compressed blocks. Decode the Huffman codes from the input bitstream, generating tree descriptions for the 'literal/length' and 'distance' Huffman decoders.

Input: Compressed data from input GZIP file, accessed through bitstream reader. Relevant Huffman decoder table generated by 'builder' module, stored in shared Block RAM.

Output: Tree descriptions for 'literal/length' and 'distance' Huffman decoders, sent through channels.

Builder for 'literal/length' Huffman decoder

Function: Use the tree description generated from the input bitstream or the static bit-length codes to build a Huffman decoder table for the 'literal/length' Huffman decoder.

Input: Tree description from Huffman decoder, or top-level function, received through channel.

Output: Huffman table written to shared Block RAM.

Builder for 'distance' Huffman decoder

Function: Use the tree description generated from the input bitstream or the static bit-length codes to build a Huffman decoder table for the 'distance' Huffman decoder.

Input: Tree description from Huffman decoder, or top-level function, received through channel.

Output: Huffman table written to shared Block RAM.

'Literal/length' and 'distance' Huffman decoders

Function: Decode the Huffman codes from the input bitstream, generating 'literal' values and 'length/distance' pairs.

Input: Compressed data from input GZIP file, accessed through bitstream reader. Relevant Huffman decoder tables generated by 'builder' modules, stored in shared Block RAMs.

Output: LZ77 'literals' and 'length/distance' pairs

LZ77 decompress

Function: Use the LZ77 algorithm on the stream of 'literal' values and length/distance pairs to reconstruct the original uncompressed data.

Input: 'Literal' values and length/distance pairs from the main Huffman decoders, received through three channels.

Output: Decompressed data, sent through 8 bit channel to top-level function.

5.3 GZIP module implementation in Handel-C

Block memory access

Xilinx FPGAs include **Block RAM**, which is more efficient than distributed RAM for storing larger quantities of data. The Handel-C language supports the use of Block RAM by adding the specification `with {block=1}` after a RAM declaration. This results in DK1 using Block memory to build the RAM, and generating a RAM clock at double the program clock rate, which is used to implement single-cycle access to the Block RAM. The disadvantage of the single-cycle Block RAM access is that the clock rate of the program is limited to half of the maximum clock rate of the RAM.

Some portions of the GZIP decompression are mostly sequential in nature, so a high clock rate is desirable to get the maximum performance. Single cycle Block RAM access would limit the clock rate, so for this application a simple interface is created to allow pipelined access to the Block RAMs. For a write to the RAM, the user provides the address and data, and the memory contents will be updated in the following clock cycle. For a read from the RAM, the user provides the address, and the data will be available in the following clock cycle.

>: Application Note //

Hardware GZIP Decompression

cycle. Internally, the Block RAM is interfaced to as an EDIF component, but this complexity is hidden from the user, who only sees the macros shown below. The RAM_Driver macro manages the transfer of data and addresses to and from the Block RAM, and runs forever once called. To perform a read, the RAM_ReadAddr macro must be called first, and RAM_ReadData in the following clock cycle. Pipelined operation is possible by calling the macros in parallel, though a location which is written to should not be read from for the next two clock cycles.

```
macro proc RAM_Driver();  
macro proc RAM_Write(Addr,Data);  
macro proc RAM_ReadAddr(Addr);  
macro proc RAM_ReadData(Data);
```

The use of EDIF Block RAM components will not operate in simulation using DK1, so a compile-time switch is used to use an emulation of the Block RAM during debugging, and then the real component for an EDIF build. A further complication is that different FPGA families have different size Block RAMs, but this is again catered for using a compile-time switch in the GZIP decompression core.

Bitstream Reader

The input to the GZIP decompression core is a bit stream only one bit wide. Several of the GZIP decompression modules read from the bitstream, and often read a variable number of bits. The bitstream reader provides a single interface for all modules to use when accessing the bitstream. It allows up to 15 bits to be read, and will automatically re-fill the bitstream buffer when required. Due to the number of calls made to the bitstream reader, the hardware built by DK1 can become complex and adversely affect the clock rate. An easy method of avoiding this is to store variables in a register before using them, in this case by using one macro to set the number of bits to read from the bitstream, and another to actually perform the read and return the result. To avoid increased latency, these macros can safely be called in parallel.

Huffman Decoders

GZIP decompression uses three different Huffman decoders: one for lengths and literals, another for distances, and another to decode the *descriptions* of the first two. A new set of Huffman decoders needs to be generated every few 10K of compressed data, so the implementation of the algorithm must try to minimise the time spent building them, as no actual decompression can be performed until the decoders are ready. The software version of GZIP decompression builds Huffman decoders which decode a set number of bits in a small table, and decode further bits in "Level 2" tables when required. A simple example of this is shown in Figure 2, where if all the codes were stored in a single table, 8 entries would be required, but by using two levels, only 6 entries are needed. By reducing the number of entries required, the use of multi-level Huffman tables minimises the overhead of building the decoders.

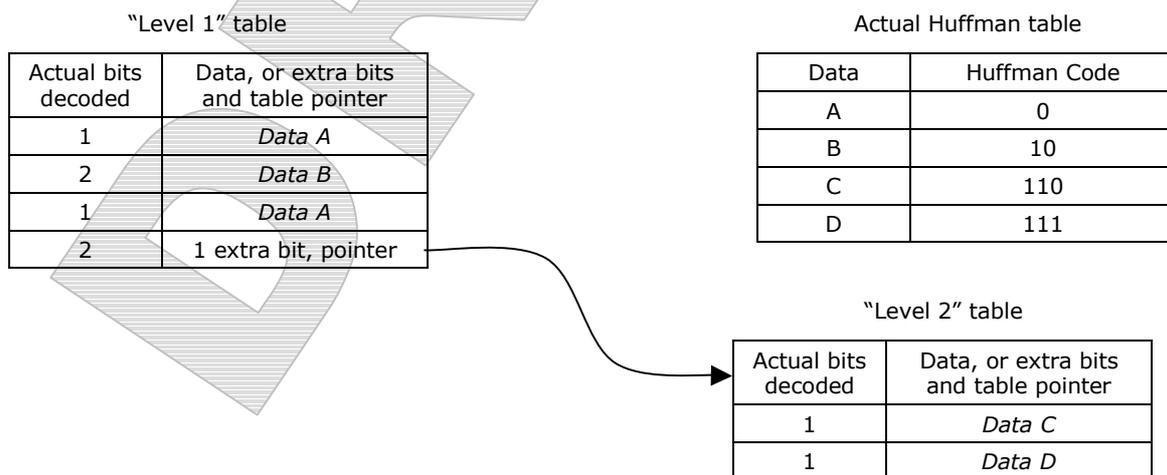


Figure 2. Multi-level tables for Huffman decoding

Hardware GZIP Decompression

The Handel-C implementation of the Huffman decoders is similar to that used in software, using Block RAMs to represent the Level 1 and Level 2 tables. The main difference is that the software uses dynamic memory allocation to create the Level 2 tables, while the Handel-C implementation stores them contiguously in a single Block RAM. Another difference is in the way in which the decoders are built; the software GZIP builds a Level 2 table every time it discovers a need for one, but the Handel-C GZIP builds the entire Level 1 table, then all the Level 2 tables. This is more efficient for a hardware implementation, as the control logic is simplified.

Some trade-offs can be made for area against clock rate for the Handel-C GZIP; for example, the code for building the Huffman decoders is shared in the software version, but the Handel-C version uses three separate builders, each optimised for its particular decoder. This takes up more area on the FPGA, but results in a higher clock rate. A further optimisation which can be made to improve clock rate is to make use of both ports on the Block RAMs, as they are "built in" as standard, and do not require any extra hardware to be built. In this case, one port is used by the procedure which builds a Huffman table (a write-only process), and the other port is used by the Huffman decoder itself (a read-only process). The Block RAMs used for the Huffman tables are implemented using the pipelined memory access macros described above.

Huffman decoder performance

The performance of the Huffman decoders has a significant impact on the overall performance of the GZIP decompression, particularly for files that are only slightly compressed. This is because files which are not compressed very much will have a higher proportion of literal LZ77 codes, compared to the length/distance LZ77 pairs. The literal values can be processed in only one or two clock cycles by the LZ77 module, but the Huffman decoder takes at least eight cycles to produce each one. It would be possible to reduce the number of cycles required by the Huffman decoder, but the trade-off would be a lower clock rate.

LZ77 decompressor

The principle of LZ77 decompression was explained above in Section 4.1. The main component in the hardware implementation is the 'sliding window', which is 32KB with byte-wide access. The sliding window is built using the pipelined Block RAM access macros, although due to its size it actually uses several Block RAMs in parallel. Copy operations, resulting from a length/distance pair, are performed using a pipelined loop, which reads from one port of the sliding window Block RAM and writes back to the other port. This results in fast operation, processing one byte every clock cycle once the pipeline is full. The pipeline does not work if the distance for the copy operation is too small, as writes to the sliding window can not be completed before the same location is read from. This is solved by keeping a short history of the last few bytes written into the sliding window, and using this when the distance is too short.

5.4 DK1 Simulation testing of GZIP modules

A significant advantage of developing a GZIP decompression core in Handel-C is that the DK1 Simulator can be used to test the functionality of the design before implementing it in hardware. These simulations are performed rapidly – a model of the entire GZIP decompression core can process 100KB of compressed data in 25 seconds or less. This equates to a simulation speed of approximately 35,000 cycles per second, on a Pentium III 850.

During development of each GZIP module, simulations were conducted to verify their operation. The software GZIP source code was modified to produce files containing intermediate results at different stages during decompression. Test harnesses can then be written for the Handel-C GZIP modules, using external C++ functions to read data from the intermediate files and send it to the module's inputs. C++ functions were also used to write results back to files, for comparison with those produced by the software GZIP. Whenever any problems are encountered in a module, the ability to debug Handel-C code using DK1 in the same way as C/C++ code is debugged then becomes invaluable. Breakpoints can be set and all variables monitored, allowing problems to be quickly located and solved.

>: Application Note //

Hardware GZIP Decompression

>: 6 Clock rate and area testing

During development of the Handel-C GZIP modules, each was put through a test Place And Route (PAR), to verify that a consistently high clock rate was being achieved. Whenever this was not the case, the technology mapper and estimation tool provided with DK1 simplified the process of optimising the Handel-C code. Once all the modules were complete and assembled to form the GZIP decompression core, a final test PAR was performed for two different Xilinx FPGAs which are available on boards we have access to. The PAR results are shown for a Virtex-II 1000 grade 5 in Figure 3, and for a Virtex-E 2000 grade 6 in Figure 4. They indicate a clock rate in excess of 133 MHz for the Virtex-II, which is Xilinx's most recent FPGA, and 66 MHz for the older Virtex-E FPGA. The GZIP decompression core uses around 45% of the resources of the Virtex-II, and around 15% for the Virtex-E (which is a higher capacity device).

```
Design Summary:
Number of errors:      0
Number of warnings:   78
Number of Slices:      2,375 out of 5,120 46%
Number of Slices containing
unrelated logic:      0 out of 2,375 0%
Number of Slice Flip Flops: 1,785 out of 10,240 17%
Total Number 4 input LUTs: 3,734 out of 10,240 36%
  Number used as LUTs:      3,162
  Number used as a route-thru: 173
  Number used for 32x1 RAMs: 274
  (Two LUTs used per 32x1 RAM)
  Number used as 16x1 RAMs: 49
  Number used as 16x1 ROMs: 74
  Number used as Shift registers: 2
Number of bonded IOBs: 25 out of 324 7%
  IOB Flip Flops:          16
Number of Block RAMs: 21 out of 40 52%
Number of GCLKs:      1 out of 16 6%
Total equivalent gate count for design: 1,457,912
Additional JTAG gate count for IOBs: 1,200
```

Constraint	Requested	Actual	Logic Levels
NET "ClockInput_main_55_WGT1" PERIOD = 7.519 nS HIGH 50.000000 %	7.519ns	7.505ns	7

```
Timing summary:
-----
Timing errors: 0 Score: 0
Constraints cover 56769 paths, 0 nets, and 19433 connections (100.0% coverage)
Design statistics:
  Minimum period: 7.505ns (Maximum frequency: 133.245MHz)
```

Figure 3. Test GZIP PAR results for Virtex-II 1000 grade 5

>: Application Note //

Hardware GZIP Decompression

Design Summary:

```

Number of errors:      0
Number of warnings:   12
Number of Slices:     2,450 out of 19,200   12%
Number of Slices containing
  unrelated logic:    0 out of 2,450   0%
Number of Slice Flip Flops:  1,812 out of 38,400   4%
Total Number 4 input LUTs:  3,790 out of 38,400   9%
  Number used as LUTs:      3,212
  Number used as a route-thru:  179
  Number used for 32x1 RAMs:    274
  (Two LUTs used per 32x1 RAM)
  Number used as 16x1 RAMs:    49
  Number used as 16x1 ROMs:    74
  Number used as Shift registers:  2
Number of bonded IOBs:     24 out of 404   5%
  IOB Flip Flops:          16
Number of Block RAMs:     77 out of 160   48%
Number of GCLKs:          1 out of 4   25%
Number of GCLKIOBs:       1 out of 4   25%
Total equivalent gate count for design: 1,343,862
Additional JTAG gate count for IOBs: 1,200
  
```

Constraint	Requested	Actual	Logic Levels
NET "ClockInput_virtex_55_WGT1" PERIOD = 15.152 nS HIGH 50.000000 %	15.152ns	15.000ns	5

Timing summary:

Timing errors: 0 Score: 0

Constraints cover 59745 paths, 0 nets, and 18437 connections (100.0% coverage)

Design statistics:
Minimum period: 15.000ns (Maximum frequency: 66.667MHz)

Figure 4. Test GZIP PAR results for Virtex-E 2000 grade 6

>: 7 Data throughput

Having measured the maximum clock rate using a test PAR, the data throughput of the GZIP decompression core can be calculated from simulation results, which indicate the number of clock cycles required to decompress given files. The table below shows the theoretical peak decompressed data output rate from the GZIP decompression core, and also the minimum and maximum sustained rates. The actual data output rates observed depend on the level of compression present in the input GZIP file, with higher compression resulting in a higher rate.

Type	Rate (MB/s)
Theoretical peak	130
Minimum sustained	20
Maximum sustained	116

Hardware GZIP Decompression

The large range between the minimum and maximum sustained data rates is due to the mismatch in performance between the Huffman decoders and the LZ77 decompressor, which was mentioned in Section 5.3. When the compression ratio of the input is high, there is more 'work' for the LZ77 module to do in comparison with the Huffman decoders. For lower compression ratios, the Huffman decoders have a much higher workload, and the performance begins to fall. Modifications can be made to the Huffman decoders to improve their performance, but at the expense of the maximum clock rate; these are described in Section 8.

It should be noted that software GZIP has a data output rate of approximately 25 MB/s on an average Pentium III-based computer, although this is highly dependent on the system configuration.

>: 8 Hardware implementation: RC1000

As a test case, the GZIP decompression core has been implemented on the Celoxica RC1000 rapid prototyping board. The RC1000 is a PCI board with a Virtex-E 2000 (grade 6) FPGA and 8 MB asynchronous SRAM in four banks. The maximum clock rate for the board is 100 MHz, but to use the SRAM banks the clock must be divided at least by three, resulting in a core clock of 33 MHz.

Implementation on the RC1000 was straightforward, as it has Handel-C support libraries for the hardware, and also a MS Windows® and Linux® driver to allow software on the host computer to communicate with the board. Approximately 80 lines of Handel-C code was written to allow the GZIP decompression core to get data in and out of the SRAM banks, and synchronise its operation with that of the host computer. The host software used an existing template C++ application which uses DMA to transfer the compressed data to one SRAM bank on the RC1000, then instruct the FPGA to start working. When the FPGA signals it has completed processing, the host software uses DMA to retrieve the decompressed data from a different SRAM bank on the RC1000. Overall, the RC1000 implementation took approximately one day.

Because the RC1000 implementation of the GZIP decompression core had to run at 33 MHz, rather than its maximum of 66 MHz (for the Virtex-E), there was an opportunity to modify the Huffman decoders to improve their performance. Some operations which had to be multi-cycle to achieve 133MHz for the Virtex-II could now be merged into single cycles, so increasing the rate at which the Huffman decoders could operate. These modifications reduced the number of clock cycles required to process GZIP files by approximately 33%, and did not reduce the clock rate theoretically achievable on a Virtex-E. There was no reduction because the Block RAM access on the Virtex-E parts is much slower, and so the LZ77 module becomes the most critical, leaving headroom to modify the Huffman decoder.

The host program for the RC1000 GZIP decompression was modified to include timing measurement, producing the following results:

DMA to card:	4.3 ms
FPGA processing:	22 ms
DMA from card:	0.4 ms
Entire program:	900 ms

It should be noted that the time for FPGA processing includes an RC1000 STATUS and CONTROL 'handshake' to mark the start and end of processing. The actual FPGA time can be calculated to be approximately 17.6 ms. It is clear from these results that the efficiency of the host software can have a significant impact on the performance of the overall design - in this case no effort had been made to optimise the host software, as is immediately apparent.

>: Application Note //

Hardware GZIP Decompression

>: 9 Conclusion

GZIP Decompression is not an ideal candidate for hardware implementation, as the algorithm is largely sequential in nature, leaving few opportunities for exploiting the ability of Handel-C to execute operations in parallel. Nevertheless, the Handel-C GZIP core is able to deliver performance ranging from equal to four times faster than the software version, when implemented on a Xilinx Virtex-II (grade 5). The use of Handel-C simplified the porting of the algorithm, as it is similar to the C language in which the GZIP source code is freely available. In combination with this, the use of DK1 allowed interactive simulation and debugging of the GZIP decompression core, resulting in a development time of only two man-months.

>: 10 Further Information

For information on Celoxica products contact Sales@celoxica.com. Regional office locations are given below.

Customer Support at Support@celoxica.com and +44 (0)1344 663649.

Celoxica Ltd.
20 Park Gate
Milton Park
Abingdon
Oxfordshire OX14 4SH
United Kingdom
Tel: +44 (0) 1235 863 656
Fax: +44 (0) 1235 863 648

Celoxica, Inc
900 East Hamilton Avenue
Campbell, CA 95008
USA
Tel: +1 800 570 7004
Tel: +1 408 626 9070
Fax: +1 408 626 9079

Celoxica Japan KK
YBP West Tower 11F
134 Godo-cho, Hodogaya-ku
Yokohama 240-0005
Japan
Tel: +81 (0) 45 331 0218
Fax: +81 (0) 45 331 0433

Celoxica Pte Ltd
Unit #05-03
31 Int'l Business Park
Singapore
609921
Tel: (65) 896 4838
Fax: (65) 566 9213

Copyright © 2001 Celoxica Ltd. All rights reserved. Celoxica and the Celoxica logo are trademarks of Celoxica Ltd.

www.celoxica.com