Fast Ziv-Lempel Decoding using RISC Architecture

David Jaggar

Technical Report COSC 9/1989

Fast Ziv-Lempel decoding using RISC architecture

David Jaggar
Department of Computer Science
University of Canterbury
Christchurch
New Zealand

phone : (+64 3) 642 367
e-mail : ....dave@cosc.canterbury.ac.nz

Abstract

Compression is becoming more important as more information is stored on, and transferred between computers. Some applications of compression require high throughput, such as the access to a hard disk drive or a Local Area Network.

The Acorn RISC Machine (ARM) is a general purpose VLSI microprocessor with a very competitive price/performance ratio. Its architecture is particularly suitable for applications requiring frequent bitwise operations.

It has been used here to construct a fast and inexpensive text decompressor. The algorithm used is a type of Ziv-Lempel compression scheme and has the properties of good compression and a fast decode strategy. The resulting device is very fast, yet inexpensive. It can operate at the speed of Local Area Networks and hard disk drives and lends itself to applications where text is stored once and read many times.

Key Phrases : Compression , Decompressor , Implementation , Reduced Instruction Set Computer , VLSI Microprocessor.

Introduction

The ideal compressor would be the one that runs fast, yet gives good compression. The best compression schemes use complex modeling techniques [4] [6] and arithmetic coding [7], but these are also the slowest. The next best compression class is the Ziv-Lempel approach [8], which can be decoded much faster than the complex modelling schemes. A survey of compression methods is given in [2].

There are many different compression algorithms based on Ziv-Lempel techniques. Here we focus on the algorithm which usually gives the best compression as well as fast decoding, named Ziv-Lempel-Bell (LZB) [1].

Like all Ziv-Lempel based compression schemes, LZB replaces substrings of text with a pointer to a previous occurrence in the text. In LZB, pointers only refer to a recent portion of the text. The pointer consists of two separate parts, the position of the longest match in the window (called the reach) and the actual match length (called the cover). Unlike other Ziv–Lempel compressors, LZB uses two different variable length codes to transmit the reach and cover.

Because the reach and cover consist of a varying number of bits, they can be inefficient to decode on a fixed word width microprocessor, because many shift and logical operations are required to extract bits.

The Acorn RISC (Reduced Instruction Set Computer) Machine, or ARM [3], is ideal to implement the decompressor, for three main reasons :

– most shifts and rotates can be combined with other instructions, and take no extra time to execute

– it is fast

– it is inexpensive.

Local Area Networks, hard disk drives and optical disks, process data at very high speed. Data compression increases both the effective capacity and speed of these applications still further. If data has to be decompressed then the speed of the decompressor should equal that of the application using it. A decompression program which is not based on specific hardware, or is written in a high level language, cannot achieve the decompression speeds reached here.

Fast decoding using the LZB algorithm

LZB distinguishes between lone characters and pointers by preceding each with a single bit flag. If the flag is zero the decompressor just places the next eight bits (an ASCII character) onto the output.

If, however, the flag is a one, the components of a pointer must be decoded. The reach is simply coded as a bit string, representing a binary number, which is used as an index into the buffer of previously seen characters. The length of this bit string (L_r) depends on the number of previously seen characters (N), such that

$$L_r(N) = \lceil \log_2 N \rceil$$

In LZB we restrict the size of N to some maximum size, typically 8192 bytes, which can be stored in a circular buffer.

The cover uses a special coding for integers capable of coding arbitrarily large integers. Each bit in the binary representation of the integer is preceded by a bit flag to indicate if the end of the bit string has been reached yet. Thus small integers take fewer bits to encode than large ones. The length of this bit

string (L_c) depends on the number of characters in the match length (M), such that

$$L_c(M) = 2\lceil \log_2 M \rceil + 1$$

When both the reach and cover have been decoded the selected characters are copied from the buffer (of previously seen characters) onto the output.

Character pointers are decoded in this manner until the end of file marker is reached. For a more detailed description see [Bell, Cleary and Witten 1989].

The ARM architecture

The ARM architecture is particularly suitable for this application because of its bit manipulation instructions. All arithmetic operations in the ARM have three operands, a destination register, and two source registers. The second of these source registers may be passed through a barrel shifter before it is used for the actual arithmetic operation. The barrel shifter can left and right shift (logically and arithmetically) by up to 32 bits, right rotate by up to 31 bits and rotate 1 bit right using the carry flag as a 33rd bit. The magnitude of a shift or rotate may either be an immediate integer, or the contents of any register.

The ARM has 16 registers (although one contains the Program Counter and Status flags), and all arithmetic instructions operate only on registers. Thus the LZB decompressor holds all variables in registers, and only uses the memory for accessing the LZB buffer and the input/output streams.

All ARM instructions include a condition code, and thus may be conditionally executed, depending on the current value of the processor's status flags. For a more detailed description of the ARM architecture see [5].

LZB decompressor implementation

Appendix A gives the ARM assembly code and Table 1 shows the usage of the CPU registers. Appendix B is a version of the algorithm in the C programming language, given for comparison to the ARM assembly routines.

The input routines for the decompressor are based upon two (macro) routines, *getbit* , which returns a single bit, and *getbits* which returns n bits. Register r0 always holds the last 32 bits loaded from the input, r1 holds the number of bits from r0 that are still unused. The code for these routines is shown in Figure 1.

The next routine called *putchar* , outputs the decoded characters. This routine copies each character into the end of the window of previous text, and puts the character onto the output stream. The number of bits used to encode the reach varies depending on the current size of the window. If adding this new character means that the window size has increased so that another bit is required to encode the reach, then the number of bits used to encode the reach is increased by one. The code for this routine is in Figure 2.

The main loop which uses the above routines is now described. First the bit flag is obtained to determine whether a character or pointer follows. If the flag bit was zero, a character follows, so the next eight bits are loaded from the input. The character is checked against a special end of text character: if it equals this character, and the next bit from the input is zero, the end has been reached. If the character does equal the end of text character, but the next bit from the input is a one, the character is copied onto the output and another pointer/character flag is read.

If the character read is not the end of text character (as is usual) then it is just copied onto the output, and another pointer/character flag is read.

Copying the character onto the output also involves storing it into the buffer of recently seen characters. This buffer is implemented as a circular queue, so that new characters automatically delete old characters by overwriting them.

If the pointer/character flag bit is a one then a pointer must be decoded. Two components are read in, the reach and the cover. The number of bits in the reach varies as the size of the window increases, and each bit of the cover is preceded by a bit flag indicating if further bits follow.

Once the reach and cover have been decoded, the characters are copied from the window to the output (and also into the end of the window) using the ARM's post-indexed auto-increment addressing mode. After this the next pointer/character flag is read and the process repeats. Figure 3 is the code for the main decode routine.

The code given in Appendices A and B assumes that the first 32 bits taken from the input have special meaning. The first sixteen bits indicate that the following data has been compressed with LZB. The failure of this check currently puts the decompressor into an error state, but it could be used to bypass the decompression stage and simply pass the input directly to the output. The next eight bits are treated as the number of bits used for the maximum window pointer. Again this step could be ignored, if the window size is to be fixed. The final eight bits of the header are ignored.

Using the ARM decompressor

There are many applications for this decompressor, so that the resulting hardware is very application dependent. The decompressor could be controlled either manually or via special codes sent as input and output. There are some hardware connections that may be required for a manual control: a Reset line, to start decoding when the power is switched on, and a Decoding status (the decompressor may be passing the input directly to the output). The decompressor could also be controlled by embedding special codes into the input and output. These codes could be used to switch the decompressor on and off, and even to alter the size of the window.

The whole decompressor can be constructed surprisingly cheaply, and could be used just about anywhere, from direct connection to an RS 232 line, to a connection to the read heads of an optical disk. It could be incorporated as a device, on say a mainframe, so that files could be written to the device compressed, and read back decompressed.

Interfacing the decompressor

Input and output for the decompressor is via the memory bus of the ARM. RISC chips like the ARM attempt to read one instruction from memory on every clock cycle. Thus the memory bus is almost 100% utilised, not allowing any sort of cycle stealing direct memory access device to handle the I/O. A possible solution is to use dual port RAM chips, so that the decompressor and I/O controller can access the data buffers at once. But, for high speed RAM, the cost of this alternative is prohibitive.

Thus it was decided to custom design a simple I/O controller. Output is achieved by directly connecting an 8 bit FIFO latch to the lower 8 bits of the data bus and combining the 26 bits of the address bus to provide the chip select. The

unique way in which the address lines are combined (using inverters and AND gates) will create a unique memory location.Writing a byte to this location will transfer that byte to the latch.

The input of data is a little more complicated. It is desirable that the decompressor be fed 32 bit words, minimising the number of memory accesses. If the input arrives in bytes then 4 bytes should be packed into a 32 bit word. A two bit counter can be used to toggle between four 8 bit FIFO buffers, so that successive 8 bits words are mapped into separate buffers. The 32 outputs from the buffers can then be coupled directly to the 32 lines of the memory bus in the same manner as the output latch.

The buffers and the decompressor must be synchronized to ensure that the integrity of the data is maintained. A trap is required to stop the decompressor on two occasions - when the input buffer is empty, and when the output buffer is full. FIFO buffers have outputs which signal when they are full or empty and can be used to "stall" the CPU under these conditions. The ARM chip can be stalled via a pin on the CPU.

If a hardware status connection is needed for DECODING this can simply be connected in the same way as the output buffer, occupying the lowest bit of a word. The decompressor is started by resetting the CPU, especially at power up. No operating system is necessary since the decompressor can be completely self contained.

The decompressor can be constructed for very little cost. The CPU costs around $100. The price of RAM grows exponentially as the speed increases, but only 10 Kilobytes is required for the decompressor's code and an 8 Kilobyte window. The price for this amount of 120 nanosecond RAM is about $8, for 60 nanosecond RAM, about $40, but increases to over $100 for 30 nanosecond RAM[1]. The RAM that is required will depend on the decompression speed required (see Table 3). The FIFO buffers, counter and gate chip would total around $15, and allowing for the PCB, case and LEDs etc, the total price is likely to be under $200.

Optimisation

The ARM code was optimised by counting the number of CPU cycles used in each part of the code, and using this to weight alternative approaches. The registers were carefully allocated and the resulting code uses all fifteen. Register allocation is important for RISC architectures. For the ARM, loading data from memory takes three cycles compared with one cycle for all data processing instructions, which operate only on data held in registers. Notice also that conditional branch and subroutine instructions have been avoided in the code, as altering the value of the program counter takes three cycles. If a conditional branch does not change the program counter (when the branch is not taken) it only takes one cycle. Thus all branch instructions in the code are set up so that the most infrequent choice takes the branch. A large improvement in the speed was achieved by separating the code into two parts, one for when the window is increasing in size, and one for when the window is at full size. The ARM code was written, debugged, tested and timed on an Acorn Archimedes 310 microcomputer fitted with an 8MHz CPU and 1MByte of RAM. The code will fit easily into a 2K ROM chip.

---

[1]All prices are correct in 1989 New Zealand dollars.

It is difficult to imagine how to decompress LZB faster than has been achieved here. An on chip cache may drop the cycle time for loads and stores to two cycles each, but this only saves 0.1 cycles per bit at most and requires a very fast and rather large on–chip cache (10 Kbytes) so that cache misses never occur. All other instructions take just one clock cycle to execute, except branches, whose cycle count is very difficult to improve on.

The LZB decoding algorithm does not have much potential for implementation on a parallel architecture, as most of the code depends directly on the input so far read. A parallel architecture could not maintain the integrity of the input or output. The code presented here does most operations using the barrel shifter in parallel with other instructions, which is exactly why this architecture was chosen.

It would be possible to have parallel routines for the input (and decode) and the output routines, so that pointers and characters were read in by one processor, and the corresponding output handled by a separate processor. This would reduce the average time for one iteration of the decompressor algorithm from the sum of the time taken by the input and output routines to the maximum time taken by the input and output routines, plus any additional time required for communication between the two processors. Which one of these approaches is faster clearly depends on how long the communication between the two processors takes. This communication must consist of at least a semaphore, to avoid the well known producer-consumer problem, and common memory or registers for the input processor to communicate with the output processor. Clearly this approach would be more expensive, and its effectiveness is dependent on the architecture used.

Average Decompressor Throughput

The performance of the decompressor can be expressed as the average amount of data output in a given time interval (the amount of input data is closely related to the amount of compression achieved). It would be desirable to have an exact figure for the output rate of the decompressor, but this is not possible for one main reason - the program takes different amounts of time to decode pointers and single characters. The ratio of frequencies of these two different decode times affects the average output speed, and depends on the original data compressed. Thus this ratio can only be measured through practical experiments. By compressing several large files it was found that about 75% of the codes were pointers and 25% were explicit characters. The codes used for covers averaged to around four bits, and after the initial part of the file the reach is thirteen bits (for a window of 8192 bytes). Eight bit input is assumed (although only seven may be used for ASCII).

To find the output rate from the decompressor it was necessary to sum the number of cycles the program took, and hence the time taken for a given clock speed could be calculated. The number of cycles for ARM instructions are shown in Table 2

Thus the time spent in each of the macros is as follows :

- readword : always takes 4 cycles
- getbit : 4 cycles if the bit is in r0, 13 cycles if it is not
  Assume the bit is in r0 31/32 of the times, then the expected number of cycles = (31*4 + 13) / 32 = 4.3 cycles
- getbits : does not depend directly on the number of bits required, only

if all the bits are in r0. 6 cycles are used if they are in r0, 15 if not. So on average for n bits, getbits takes (32 - n) / 32 * 6 + n /32 * 15 cycles
- putchar : 8 cycles when window is not at full size, 5 cycles when it is
- initialisation takes 26 cycles

With the assumptions mentioned above we can evaluate the total number of cycles to decode a pointer (reach and cover) and output the corresponding characters to be 167 cycles. The total number of cycles to decode a character is 27.
So it takes 3.48 cycles/bit to decode a pointer (assuming 6 characters) and 3.375 cycles/bit to decode a single character and the average number of cycles/bit is 3.45.

The decompressor was tested on an 8MHz ARM, and the average throughput at this clock rate was always between 2 Mbit/sec and 2.5Mbit/sec, which agrees with the theoretical throughput of 2.29 Mbit/sec. Current ARM chips have a maximum clock rate of 24 MHz and thus can decompress at 6.86 Mbits/sec. Faster implementations of the ARM architecture should soon be available with clock speeds up to 48 MHz. Obviously the price of the CPU and memory increases as the speed increases. Table 3 shows the average throughput for various clock speeds and memory access time.

Worst case throughput

Pointers to short blocks take the longest time per bit output to decode, at 3.5 cycles per bit, and this is the worse case, but this is only slightly worse than the typical case of 3.45 cycles per bit. Pointers to large blocks take longer to decode, but result in very fast output of data. Decoding is delayed while the first 26 initialisation cycles are executed, but after that the above output will be maintained. This initial delay equates to the transmission time of under one byte, and thus is practically negligible. The custom I/O controller ensures that data is kept flowing, and if a small delay is acceptable at the start (to fill the FIFO buffer) then constant flow will be maintained.

Conclusion

A device has been designed to decompress LZB encoded files very rapidly. Due to LZB's property of relatively slow encoding, but fast decoding the proposed system has applications wherever files will be compressed once and decompressed many times, or need to be decompressed very quickly. This hardware solution, using a simple general purpose microprocessor as an embedded controller, has a particularly attractive price/performance ratio, and can operate at the speeds required for Local Area Networks, such as Ethernet, or in conjunction with a hard disk drive controller. Mass storage devices such as optical disks, that are written once and read many times, could also benefit from this device. Compressing the stored data not only increases the effective capacity of the drive, but also increases the effective rate of data output.

Acknowledgments

The author would like to thank Dr Timothy Bell, who provided the initial motivation, many useful suggestions and proof reading of both the draft and final copies. The author is also grateful to Dr Michael Maclean for additional suggestions.

Bibliography

[1] T.C Bell, "A unifying theory and improvements for existing approaches to text compression" Ph.D. Thesis, Department of Computer Science, University of Canterbury, New Zealand 1987

[2] T.C.Bell, J.G Cleary and I.H.Witten, "Text compression", Prentice-Hall Englewood Cliffs , New Jersey 1989

[3] R.Cates, "Processor Architecture Considerations for Embedded Controller Applications", IEEE Micro Vol8.No3, pp28-37, June 1988

[4] J.G.Cleary and I.H.Witten, "Data compression using adaptive coding and partial string matching", IEEE Transactions on Communications, COM-32 , Vol4 pp396-402, April 1984

[5] P.Cockerell, "ARM Assembly language programming" M.T.C Publishing, Hertfordshire, England, 1987

[6] G.V.Cormack and R.N.Horspool, "Data compression using dynamic Markov modelling", Computer Journal, Vol 30 No 6, pp541-550, December 1987

[7] I.H.Witten, R.Neal and J.G.Cleary, "Arithmetic coding for data compression" Communication of the Association for Computing Machinery, Vol29, No6, pp520-540, June. Reprinted in *C Gazzette 2* (3) 4-25, December 1987

[8] J.Ziv and A.Lempel, "A universal algorithm for sequential data compression", IEEE Transaction on Information Theory, Vol23, No3, pp337-343, May 1977

Figure 1 : Input Routines

Register 0 is used as a 32 bit input buffer
Register 1 holds the number of bits in Register 0

Get the next bit from the input

On Exit : The carry flag holds the bit read

```
getbit      cmp r1,#0              \ time to read a new word of bits ?
            blmi readword          \ subroutine to get a new word
            sub r1,r1,#1           \ increment the counter
            mov r0,r0,lsr#1        \ shift top bit into carry
```

Read another 32 bits from the input into Register 0

```
readword    mov r1,#32             \ reset the bit counter
            ldr r0,inputdata       \ load a new word
            mov pc,r14             \ return from subroutine
```

Gets n bits from the input

On Entry : Register 2 holds the number of bits required
On Exit : Register 3 holds the bits read, shifted to the least significant end

```
getbits     cmp r1,r2              \ see if current word holds enough bits
            bmi getnext            \ if not, get another word
            rsb r14,1,1,lsl r2     \ r14 = (1<<r2)-1 (a mask)
            and r3,r0,r14          \ mask out rubbish (top bits of word)
            sub r1,r1,r2           \ and update the counter
gotbits     mov r0,r0,lsr r2       \ update new word
```

Handle a request for bits that is split across two words

```
getnext     mov r3,r0              \ get everything from this word
            ldr r0,inputdata       \ get a new word
            orr r3,r3,r0,lslr1     \ shift new word up & OR it in
            rsb r14,1,1,lsl r2     \ r14 = (1<<r2)-1 (a mask)
            and r3,r3,r14          \ mask out rubbish (top bits of word)
            rsb r1,r2,#32          \ reset the bit counter
            b gotbits              \ return to the main stream code
```

Figure 2 : Output routine

Puts the decoded character into the window and onto the output stream
On Entry : Register 3 holds the byte to be stored

```
putchar     and r14,r6,r7          \ mod the index into the window
            strb r3,[r10,r14]      \ store the byte into the window
            add r6,r6,#1           \ increment the index
            strb r3,outputdata     \ store the byte into the output buffer
            cmp r6,r7
            bgt fullsize           \ window full, don't increase pointer size
            cmp r6,r8
            addgt r5,r5,#1         \ increment the pointer size if at limit
            movgt r8,r8,lsl#1      \ double up the limit also
```

The label fullsize is the entry point to the code used when the window has reached its maximum size.

Figure 3 : Main decode routine

See Table 1 for the register allocation in this routine.

```
start       mov r5,#1              \ initial pointer size
            mov r6,#0              \ initialise the queue (window) pointer
            mov r8,#2              \ initial pointer size limit
            mov r7,#8191           \ the maximum windowsize - 1
            mov r1,#0              \ force a new word to be read
getflag     getbit                 \ get the flag bit
            bcc getchar            \ flag was 0, so get a char
            mov r2,r5
            getbits                \ get the pointer
            mov r4,r3              \ get the result into the pointer
            mov r9,#1              \ the initial count of characters
getnum      getbit
            bcs fixnum             \ while first bit in bit pair is not 1
            getbit                 \ get the next bit
            adc r9,r9,r9           \ get the carry into the bottom bit
            b getnum               \ go get another
fixnum      add r2,r5,#3           \ fix up the number of chars to read,
            add r9,r9,r2,lsr#3     \ add 3 to pointersize and div 8
            mov r2,#0              \ initialise the index into the phrase
putphrase   add r14,r4,r2          \ add the pointer into the index
            and r14,r14,r7         \ mod the answer by the window size
            ldrb r3,[r10,r14]      \ get a byte from the window
            putchar                \ add in the char
            add r2,r2,#1           \ increment the phrase index
            cmp r2,r9              \ at the end of the phrase yet ?
            bne putphrase          \ keep processing the phrase
            b getflag              \ or go round the loop again
getchar     mov r2,#charbits       \ number of bits to a char
            getbits                \ get the char
            cmp r3,#endchar        \ is it a possible end
            beq checkend           \ check it is the end
            putchar                \ add the new char
            b getflag              \ go round the loop again
checkend    getbit                 \ check the next bit
            bcc start              \ finished if it was the end
            putchar                \ or store the char...
            b getflag              \ ...and get a phrase
```

Table 1 : Register and Memory Usage

Global Register Usage

r0 : The last word of input , from which bits are being removed
r1 : Counter of how many bits have been read from r0
r2 : Scratch register and counter
r3 : Scratch register for building a word of bits
r4 : The reach part of the pointer
r5 : Number of bits in the current reach
r6 : Index into the window
r7 : Size in bytes minus 1 of the full size window
r8 : Maximum window size before the reach needs an extra bit
r9 : The cover part of the pointer
r10 : Start address of the window
r11 : Address of the status bits
r12 : Address of the input word
r13 : Address of the output word
r14 : The link register and a scratch register

Important Memory Locations

inputdata : data is read from this location
outputdata : data is written to this location
status : bit 0 maps to the "Decoding" LED

Constants

charbits : the number of bits in a character, set to 7 or 8 for ASCII
maxwindow : the maximum window size minus one, 8191 is typical
wordsize : the size of input words, 32 is the maximum and the best
endchar : the character that represents the end of text

Table 2 : ARM instruction timings

ldr	load from memory	3 cycles
str	store to memory	2 cycles
b (bl)	branch (branch with link)	3 cycles
mov pc,r14	return from subroutine	3 cycles
all other instructions (used in the program)		1 cycle

Table 3 : Decompression speed for different CPU/memory speeds

Clock Speed (MHz)	Memory Speed (nanosecs)	Bits Output (Mbits per sec)
8	125	2.29
12	80	3.43
16	60	4.57
24	40	6.86
32	30	9.14
48	20	13.71

Appendix A : Macro Expanded Listing

Here is the complete, macro expanded code for the decompressor. The code is split into two separate halves, and each has the same label names, except for an 'a' or 'b' as a suffix. The first half is used until the window buffer is full, then execution transfers to the second half.

```
\ the start of a new run
start       adr r11,status          \ load the base registers ...
            adr r12,inputdata       \ ... with the IO channels
            adr r13,outputdata
            strb r0,r11             \ indicate the decompressor is ready
            mov r5,#1               \ initial pointer size
            mov r6,#0               \ initialise the queue (window) pointer
            mov r8,#2               \ initial pointer size limit
            mov r7,#8191            \ the maximum windowsize - 1
            mov r3,#0
            strb r3,r11             \ indicate we are decoding
            mov r1,#0               \ force a new word to be read
getflaga    \ get the flag bit
            cmp r1,#0               \ time to read a new word of bits ?
            blmi read               \ subroutine to get a new word
            sub r1,r1,#1            \ increment the counter
            mov r0,r0,lsr#1         \ shift top bit into carry
            bcc getchara            \ flag was 0,so get a char
            \ get the reach
            cmp r1,r5               \ see if current word holds enough bits
            bmi getnext1a           \ if not,get another word
            rsb r14,1,1,lsl r5      \ r14 = (1<<r5)-1 (a mask)
            and r4,r0,r14           \ mask out all unwanted bits
            sub r1,r1,r5            \ update the counter
            mov r0,r0,lsr r5        \ update new word
gotbits1a   mov r9,#1               \ the initial count of characters
getnuma     \ get the cover
            cmp r1,#0               \ time to read a new word of bits ?
            blmi read               \ subroutine to get a new word
            sub r1,r1,#1            \ increment the counter
            mov r0,r0,lsr#1         \ shift top bit into carry
            bcs fixnuma             \ while first bit not 1,get next bit
            cmp r1,#0               \ time to read a new word of bits ?
            blmi read               \ subroutine to get a new word
            sub r1,r1,#1            \ increment the counter
            mov r0,r0,lsr#1         \ shift top bit into carry
            adc r9,r9,r9            \ get the carry into the bottom bit
            b getnuma               \ go get another
fixnuma     add r2,r5,#3            \ fix up the number of chars to read,
            add r9,r9,r2,lsr#3      \ add 3 to pointersize and div 8
            mov r2,#0               \ initialise the index into the phrase
putphrasea  add r14,r4,r2           \ add the pointer into the index
            and r14,r14,r7          \ mod the answer by the window size
            ldrb r3,[r10,r14]       \ get a byte from the window
            \ add in this char
            and r14,r6,r7           \ mod the index into the window
            strb r3,[r10,r14]       \ store the byte into the window
            add r6,r6,#1            \ increment the index
            strb r3,r13             \ store the byte into the output buffer
            cmp r6,r7
            bgt clonecode           \ window full size, jump to other half
            cmp r6,r8
            addgt r5,r5,#1          \ increment the pointer size if at limit
            movgt r8,r8,lsl#1       \ double up the limit also
```

```
            add r2,r2,#1              \ increment the phrase index
            cmp r2,r9                 \ at the end of the phrase yet ?
            bne putphrasea            \ keep processing the phrase
            b getflaga                \ or go round the loop again
getchara    \ get the char
            cmp r1,#charbits          \ see if current word holds enough bits
            bmi getnext2a             \ if not,get another word
            rsb r14,1,1,lsl#charbits  \ r14 = (1<<charbits)-1 (a mask)
            and r3,r0,r14             \ mask out all unwanted bits
            sub r1,r1,#charbits       \ update the counter
            mov r0,r0,lsr#charbits    \ update new word
gotbits2a   cmp r3,#endchar           \ is it a possible end
            beq checkenda             \ check it is the end
            \ output the char
            sub r14,r8,#1             \ build a mod mask
            and r14,r6,r14            \ mod the index into the window
            strb r3,[r10,r14]         \ store the byte into the window
            add r6,r6,#1              \ increment the index
            strb r3,r13               \ store the byte into the output buffer
            cmp r6,r7
            bgt getflagb              \ window full size, jump to other half
            cmp r6,r8
            addgt r5,r5,#1            \ increment the pointer size if at limit
            movgt r8,r8,lsl#1         \ double up the limit also
            b getflaga                \ go round the loop again
checkenda   \ check the next bit
            cmp r1,#0                 \ time to read a new word of bits ?
            blmi read                 \ subroutine to get a new word
            sub r1,r1,#1              \ increment the counter
            mov r0,r0,lsr#1           \ shift top bit into carry
            bcc start                 \ finished if it was the end
            \ or store the char
            sub r14,r8,#1             \ build a mod mask
            and r14,r6,r14            \ mod the index into the window
            strb r3,[r10,r14]         \ store the byte into the window
            add r6,r6,#1              \ increment the index
            strb r3,r13               \ store the byte into the output buffer
            cmp r6,r7
            bgt getflagb              \ window full size, jump to other half
            cmp r6,r8
            addgt r5,r5,#1            \ increment the pointer size if at limit
            movgt r8,r8,lsl#1         \ double up the limit also ...
            b getflaga                \ ... and get a phrase

getnext1a   mov r4,r0                 \ get everything from this word
            ldr r0,r12                \ load a word
            orr r4,r4,r0,lsl r1       \ shift new word up & OR it in
            rsb r14,1,1,lsl r5        \ r14 = (1<<r5-1) (a mask)
            and r4,r4,r14             \ mask out rubbish (top bits of word)
            sub r14,r5,r1             \ find the bits used from this word
            rsb r1,r14,#32            \ update the counter
            mov r0,r0,lsr r14         \ update new word
            b gotbits1a               \ return to the main stream code

getnext2a   mov r3,r0                 \ get everything from this word
            ldr r0,r12                \ load a word
            orr r3,r3,r0,lsl r1       \ shift new word up & OR it in
            rsb r14,1,1,lsl#charbits  \ r14 = 1<<charbits-1 (a mask)
            and r3,r3,r14             \ mask out rubbish (top bits of word)
            rsb r2,r1,#charbits       \ how many bits removed from new buffer
            sub r1,r1,r2              \ update the buffer pointer
            mov r0,r0,r2              \ update new word
```

```
            b gotbits2a                 \ return to the main stream code

getflagb    \ get the flag bit
            cmp r1,#0                   \ time to read a new word of bits ?
            blmi read                   \ subroutine to get a new word
            sub r1,r1,#1                \ increment the counter
            mov r0,r0,lsr#1             \ shift top bit into carry
            bcc getcharb                \ flag was 0,so get a char
            \ get the reach
            cmp r1,r5                   \ see if current word holds enough bits
            bmi getnext1b               \ if not,get another word
            rsb r14,1,1,lsl r5          \ r14 = (1<<r5) -1 (a mask)
            and r4,r0,r14               \ mask out all unwanted bits
            sub r1,r1,r5                \ update the counter
            mov r0,r0,lsr r5            \ update new word
gotbits1b   mov r9,#1                   \ the initial count of characters
getnumb     \ get the cover
            cmp r1,#0                   \ time to read a new word of bits ?
            blmi read                   \ subroutine to get a new word
            sub r1,r1,#1                \ increment the counter
            mov r0,r0,lsr#1             \ shift top bit into carry
            bcs fixnumb                 \ while first bit not 1,get next bit
            cmp r1,#0                   \ time to read a new word of bits ?
            blmi read                   \ subroutine to get a new word
            sub r1,r1,#1                \ increment the counter
            mov r0,r0,lsr#1             \ shift top bit into carry
            adc r9,r9,r9                \ get the carry into the bottom bit
            b getnumb                   \ go get another
fixnumb     add r2,r5,#3                \ fix up the number of chars to read,
            add r9,r9,r2,lsr#3          \ add 3 to pointersize and div 8
            mov r2,#0                   \ initialise the index into the phrase
putphraseb  add r14,r4,r2               \ add the pointer into the index
            and r14,r14,r7              \ mod the answer by the window size
            ldrb r3,[r10,r14]           \ get a byte from the window
            \ add in this char
            and r14,r6,r7               \ mod the index into the window
            strb r3,[r10,r14]           \ store the byte into the window
            add r6,r6,#1                \ increment the index
            strb r3,r13                 \ store the byte into the output buffer
clonecode   add r2,r2,#1                \ increment the phrase index
            cmp r2,r9                   \ at the end of the phrase yet ?
            bne putphraseb              \ keep processing the phrase
            b getflagb                  \ or go round the loop again
getcharb    \ get the char
            cmp r1,#charbits            \ see if current word holds enough bits
            bmi getnext2b               \ if not,get another word
            rsb r14,1,1,lsl#charbits    \ r14 = (1<<charbits)-1 (a mask)
            and r3,r0,r14               \ mask out all unwanted bits
            sub r1,r1,#charbits         \ update the counter
            mov r0,r0,lsr#charbits      \ update new word
gotbits2b   cmp r3,#endchar             \ is it a possible end
            beq checkendb               \ check it is the end
            \ output the char
            sub r14,r8,#1               \ build a mod mask
            and r14,r6,r14              \ mod the index into the window
            strb r3,[r10,r14]           \ store the byte into the window
            add r6,r6,#1                \ increment the index
            strb r3,r13                 \ store the byte into the output buffer
            b getflagb                  \ go round the loop again
checkendb   \ check the next bit
            cmp r1,#0                   \ time to read a new word of bits ?
            blmi read                   \ subroutine to get a new word
```

```
            sub r1,r1,#1              \ increment the counter
            mov r0,r0,lsr#1           \ shift top bit into carry
            bcc start                 \ finished if it was the end
            \ or store the char ....
            sub r14,r8,#1             \ build a mod mask
            and r14,r6,r14            \ mod the index into the window
            strb r3,[r10,r14]         \ store the byte into the window
            add r6,r6,#1              \ increment the index
            strb r3,r13               \ store the byte into the output buffer
            b getflagb                \ ... and get a phrase

getnext1b   mov r4,r0                 \ get everything from this word
            ldr r0,r12                \ load a word
            orr r4,r4,r0,lsl r1       \ shift new word up & OR it in
            rsb r14,1,1,lsl r5        \ r14 = (1<<r5-1) (a mask)
            and r4,r4,r14             \ mask out rubbish (top bits of word)
            sub r14,r5,r1             \ find the bits used from this word
            rsb r1,r14,#32            \ update the counter
            mov r0,r0,lsr r14         \ update new word
            b gotbits1b               \ return to the main stream code

getnext2b   mov r3,r0                 \ get everything from this word
            ldr r0,r12                \ load a word
            orr r3,r3,r0,lsl r1       \ shift new word up & OR it in
            rsb r14,1,1,lsl#charbits  \ r14 = 1<<charbits-1 (a mask)
            and r3,r3,r14             \ mask out rubbish (top bits of word)
            rsb r2,r1,#charbits       \ how many bits removed from new buffer
            sub r1,r1,r2              \ update the buffer pointer
            mov r0,r0,r2              \ update new word
            b gotbits2b               \ return to the main stream code

read        mov r1,#32                \ reset the bit counter
            ldr r0,r12                \ load a new word
            mov pc,14                 \ return
```

Appendix B : Decompressor written in C

```
FILE *input , *output ;
static char *window ;
int windowptr=0 , windowsize , reachsize=0 , sizelimit=1 ;

main(argc,argv)
int argc ;
char *argv[] ;
{
register int reach , cover , ch , endchar ;

if (getbits(16) != TXTLZBED) {
   fprintf(stderr,"This program is only for LZB compressed ASCII\n") ;
   exit(2) ;
   }
windowsize = (1 << getbits(8)) - 1 ;      /* get the maximum window size */
window = (char*)malloc(windowsize+1) ;
(void)getbits(8) ;
endchar = (1 << charbits) - 1 ;               /* the end of text marker */

for (;;) {
   if (getbit()) {                          /* decode a reach,cover pair */
      reach = getbits(reachsize) ;                    /* get the reach */
      for (cover=1 ; !getbit() ; cover=(cover<<1) + getbit()) ;
      cover  = cover + (3+reachsize) / 8  ;           /* get the cover */
      for (ch = 0 ; ch < cover ; ch++)           /* add these characters */
         putchar(*(window + ((reach+ch) & windowsize))) ;
      }
   else {                                   /* decode a single character */
      if ((ch=getbits(charbits))==endchar) && !getbits(1)) break ;/* end */
      else putchar(ch) ;           /* if not end condition, add the char */
      }
   }
}
```

```
unsigned buffer , bitsinbuf = 0 ;

/* This routine returns the next bit from the input */
getbit()
{
 if (bitsinbuf == 0) {
    readint(buffer) ;                /* time to read a new 32 bit buffer */
    bitsinbuf = 31 ;
    }
 else
    bitsinbuf -- ;
 retval = buffer & 1 ;                                  /* return the bit */
 buffer >>= 1 ;
 return(retval) ;
 }


/* This routine returns the next num bits from the input */
getbits(num)
unsigned int num ;
{register unsigned retval ;

 if (num==0) return(0) ;
 if (bitsinbuf < num) {              /* time to read a new 32 bit buffer */
    unsigned int oldbuf

    oldbuf = buffer ;
    readint(buffer) ;
    retval = (oldbuf | (buffer << bitsinbuf)) & (1 << num - 1) ;
    bitsinbuf = 32 - (num - bitsinbuf) ;
    }
 else {
    retval = buffer & (1 << num - 1) ;
    buffer >>= num ;
    bitsinbuf -= num ;
    }
 return(retval) ;                                       /* return the bit */
 }


/* This routine puts into the window and outputs a character */
putchar(ch)
char ch ;
{
 fprintf(output,"%c",ch) ;
 *(window + (windowptr & windowsize)) = ch ;
 windowptr++ ;
 if (windowptr>sizelimit && windowptr<=windowsize) {
    reachsize ++ ;          /* check if the pointer size needs increasing */
    sizelimit = sizelimit << 1 ;
    }
 }
```