File compression using variable length encodings

M. Wells*

Department of Computational Science and Electronic Computing Laboratory, The University of Leeds, Leeds LS2 9JT

Within a digital system a symbol is represented as a grouping of digits. If the length of this group is variable, rather than fixed, it is theoretically possible to achieve reductions in the total number of digits required to represent a symbol string. This paper discusses some hardware and software techniques for carrying out the encoding and decoding and gives some observations of the economies that might be achieved by their use. The final conclusions are cautiously optimistic.

(Received January 1972)

Within a digital system information is represented as an encoding of symbols. This representation involves two quite distinct steps, the choiceof a physical representation which has a number of recognisable values, and the choice of a code which maps the symbols on to a particular combination of these values. We shall find it convenient to regard the different values taken up by the physical representation as being referred to as 'digits' 0, 1, 2, ... etc; the decision as to which state of the physical device represents which digit is totally arbitrary. In any system the same digit will have many physical representations, for example:

(*a*) a hole in a punched card
(*b*) a voltage near to +4·5 V
(*c*) a clockwise magnetisation of a ferrite core
(*d*) a flux reversal on a magnetic surface

may all be different representations of a digit '1'. It is a function of the engineering design of the system to ensure that a '1' in one representation transforms in to a '1' in another representation when the program requires it.

The mapping from symbols to combinations of digits is sometimes regarded as being outside the control of the programmer. In some instances this is patently true; a line printer performs the decoding from digit pattern to symbol representation in a way which is fixed by the engineering design of the printer. However, this control can be exercised as long as the symbols continue to be encoded purely as combinations of digits. A transformation from one such encoding to another is a perfectly proper action by the programmer. In particular, the application of such transformation may offer useful economies in certain activities, as we hope to show.

Entropy of a symbol source

Consider a message source, which can emit n distinct symbols, and let $P_i (1 \leqslant i \leqslant n)$ be the probability that the ith symbol occurs. Under certain conditions† we can define the *entropy* of the source as

$$H = -\sum_{i=1}^{n} P_i \log_2 P_i \text{ bits per symbol} \quad (1)$$

Informally, the entropy of a source defines the minimum average number of binary digits per symbol required to encode messages from the source. Notice that entropy is inherently a statistical concept; that it supplies us with a lower bound to the average number of binary digits required per symbol; and that in general this number is not an integer. The number of digits other than binary digits required is simply found by appropriately changing the base of the logarithms in the definition of entropy above, but whatever the number of different digits available, the average symbol length (defining the *length* of a symbol as being the number of digits contained in its encoding) is not in general integral. Obviously the number of digits required to encode any particular symbol must be integral; thus if we are to find an encoding which approaches this lower bound we must be prepared to accept encodings in which the number of digits is variable. For obvious reasons such codes are known as 'variable length' codes, and the most effective such code was devised by Huffman in 1952. A recent paper by Maurer (Maurer, 1969) suggested using such codes; our work, started in ignorance of his, gives observations made on an existing system.

Huffman codes

A minimum redundancy code (MRC) is one which has the minimum average message length, the average being taken over all possible messages. If there is no form of correlation between successive symbols (i.e. if the P_i of equation (1) are not functions of the preceding symbols) the average code length is simply

$$h_{av} = \sum_{i=1}^{n} P_i l_i \quad (2)$$

where l_i is the length of (number of digits in) the encoding of the ith symbol. If we order the P_i in such a way that

$$P_1 > P_2 > \quad > P_{n-1} \geqslant P_n \quad (3)$$

it is obvious that for an MRC

$$l_1 < l_2 \quad < l_{n-1} \leqslant l_n \quad (4)$$

We introduce the concept of a 'prefix'; any code-word‡ has as its kth prefix the first k digits of the code-word. If we are successfully to decode a message encoded in a variable length code, then none of the code-words must be a prefix of any longer code-word. A code with this property is known as a 'prefix code'. Huffman showed how to develop a code which has the property required by equation (4) and in which no code-word is a prefix of any longer code. This is achieved by developing a tree of order p whose leaves are the symbols, the successive nodes being generated by grouping together the p least probable ungrouped nodes (or leaves) and assigning to this node a probability equal to the sum of the nodes dependent from it. When the tree is complete digits 0 to $(p-1)$ are assigned to each branch at each node in an arbitrary way, and the encoding for the symbol is given by listing the digits which lie between the root and corresponding leaf. **Fig. 1** shows this process for an arbitrary set of nine symbols and

*The work described in this paper was carried out under my supervision by four Final year students in Computational Science at Leeds. It is a pleasure to acknowledge the efforts of Messrs. K. B. Shimman, R. G. Stone, J. W. Taylor and D. Viney.

†These conditions are summarised by saying that the source must be 'ergodic'. A full treatment can be found in any book on information theory, e.g. 'Information Theory' by J. F. Young. Butterworth 1971.

‡By code-word we understand the encoding of a particular symbol.

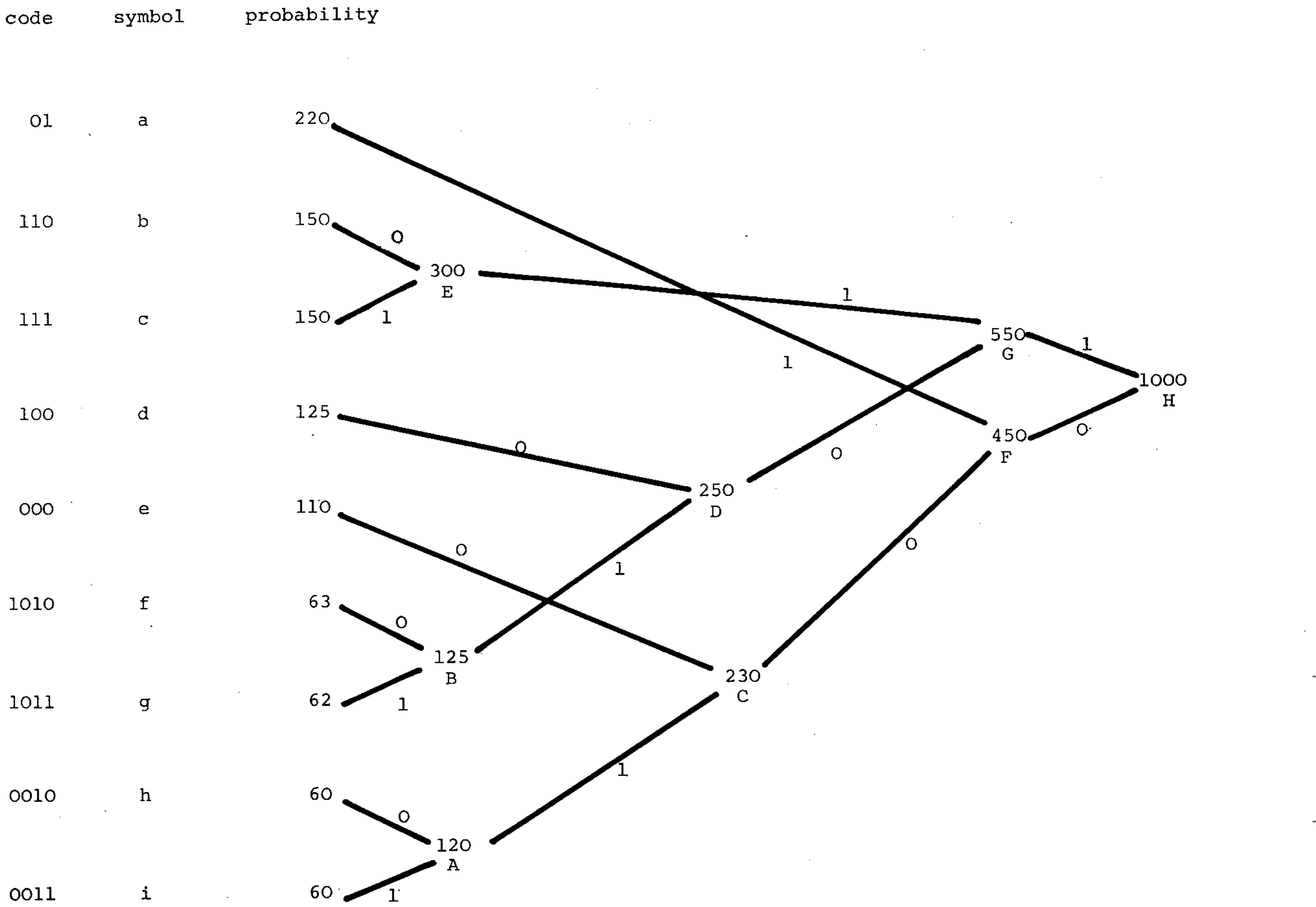

Fig. 1. Generating a Huffman code. The upper case letters by each node show the order in which the nodes are generated. This code has entropy 2·994 bits/symbol and a mean code length of 3·025 binary digits/symbol.

$p = 2$. Binary Huffman codes approach the lower limit of number of binary digits per symbol at least as closely as any other code. It is thus possible to estimate the likely compression that can be achieved by encoding using a Huffman code, or one derived from it, by estimating the entropy of the message source and comparing this with the mean length of the encoding currently in use.

Results for the Eldon 2 file store

In this section we examine some results for a particular file store, that associated with the Eldon 2 operating system in use at Leeds. This used an encoding with eight binary digits per symbol, a symbol being either a single character (e.g. a lower case letter or a digit) or a multi-character symbol (e.g. an ALGOL underlined word, or a 'tab' on a device which has no hardware tabbing mechanism). The number of distinct symbols is about 130, but some of these have very special contexts; for example it is arguable whether the 'end of file' symbol is really a symbol at all.

If we take all the files within the system, and treat this collection as a single message we can deduce a frequency distribution for the symbols, and hence an entropy and a lower bound to the mean code length per symbol. However, this collection of files contains some which *a priori* have very different frequency distributions from others, and from the mean. For example, a file containing the text of a FORTRAN program is unlikely to contain many of the ALGOL underlined words. We therefore collected data for three classes of textual material which might be expected to show the widest deviations:

1. ALGOL program.
2. FORTRAN program.
3. Assembly Language.

The basic raw data for all subsequent work are simply the frequency distributions of the symbols for each class of material; it is a trivial task to compute from this the entropies as being:

ALGOL 5·58 bits/symbol.
FORTRAN 4·45 bits/symbol.
Assembly Language 5·19 bits/symbol.

Since our original symbols were encoded on eight binary digits per symbol there is a potential compression here to between 55 and 70% of the original volume, a mouth-watering prospect!

We have achieved this dramatic saving by classifying our text into one of three classes, and encoding it correspondingly. We now examine the effect of using a single coding, which covers *all* text, but is not optimal for any one class. If we use primed quantities to denote this generalised class, we have the following results. The entropy of a particular class is given by

$$H = -\sum_i p_i \log_2 p_i \text{ bits/symbol}$$

and its mean symbol length by

$$h_{av} = \sum_i p_i l_i \text{ digits/symbol}$$

Encoding the same messages with the 'wrong' code will give a mean symbol length of

$$h_{av} = \sum_i p_i l_i \text{ digits/symbol}$$

and the increase in mean symbol length will then be

$$h_{av} = \sum_i p_i (l'_i - l_i) \text{ digits/symbol} \quad (6)$$

Now the l_i are uniquely determined by the p_i; in particular only if p_i, p'_i are so different that they cause changes in the l_i do we see an increase in the mean symbol length by using the 'wrong' code. Further it is only amongst the more frequently occurring symbols that the changes are relevant. As is the case with natural languages, the more common symbols account for a very high proportion of the total (as a rough guide the 10 most frequent symbols account for half the total) and these appear to be remarkably consistent as between the various classes of material. This consistency can be improved still further if one is prepared to regard the upper case letters in which FORTRAN and assembly language are usually presented as being a graphic representation of the lower case letters favoured by users of ALGOL. Other points emerging are:

1. The anomalous status of semicolon (;) used by ALGOL and KDF9 assembly language as a statement separator with a correspondingly high probability, but occurring very infrequently in 'FORTRAN' files.
2. The anomalous status of the 'tab' symbol, used quite frequently ($p \simeq 0{\cdot}02$) in ALGOL and assembly language but very little in FORTRAN. The effect is twofold, in that the virtual absence of 'tab' symbols in FORTRAN causes a dramatic increase in the frequency with which spaces occur.
3. The astonishing resemblance of the frequency distributions for these artificial languages to that of English text viz.:

English text	ALGOL	FORTRAN
space	space	space
e	i	I
t	e	O
a	t	T
o	a	N
i	s	E
n	d	A
s	n	R
h	o	C
r	c	S
d	f	M

Bearing in mind programmers' predilections for integers to be known as 'i' and for the presence in FORTRAN of DO, GOTO as potent sources of letter 'O's, the resemblance is quite startling.

The quantity defined by (6) is easily computed for encodings of FORTRAN using a Huffman code corresponding to ALGOL, but with the case of the letters reversed as described; the increase in the average length per symbol is about 0·50 binary digits, if one assumes a binary Huffman code is in use. We shall see in the next section that there are some advantages in using codes which have a radix other than two, and in such cases the expansion as defined by (6) is even smaller.

Other economic codes

We consider two extensions of the ideas discussed so far:

(a) A further examination of Huffman codes based on a branching ratio other than two.

(b) The development of codes which are self-synchronising.

1. *Non-binary Huffman codes*

Huffman's original paper considered the production of codes in which there are more than two distinct digits, i.e. which have more than two branches from each node of the associated tree. When generating such a code, it is essential to augment the symbol set with dummy nodes, each with a zero probability of occurrence before attempting to construct the tree. If this is not done the process described earlier leads to a tree whose root has less than the maximum allowable number of branches leaving it. Such a code does not have the smallest mean code length. For a tree with constant branching ratio p, and containing n non-terminal nodes the number of leaves is readily shown to be $p \times (n-1)/(p-1)$. (Notice that for a binary tree *any* number of leaves can be produced.) For a complete radix-p code with S symbols the number of non-terminal nodes n is obviously fixed by finding the least 'n' such that

$$\frac{p(n'-1)}{(p-1)} > S$$

The number of zero probability terminal symbols to be added to the S existing symbols is then given by $S - p(n'-1)/(p-1)$. The tree is then constructed, and the zero probability symbols ignored.

2. *Self-synchronising codes*

If, while decoding a stream of symbols encoded in a variable length prefix code the decoder misses the end of a code word, and hence starts subsequent decoding from within code words, the decoder is said to be out of synchronisation. This can arise if a digit is missing, or corrupted. With some variable length codes there is a possibility that the decoder will automatically resynchronise.

A code may be described as fully, partially or never self-synchronising according as it will always, sometimes or never come into synchronisation. A necessary and sufficient condition for a code to be fully self-synchronising is the existence of a 'universal synchronising sequence'; such a sequence whenever it is encountered will always cause the decoder to resynchronise. Appendix 1 contains an (informally stated) algorithm for determining whether or not such a sequence exists; the algorithm is based on a proof by Gilbert and Moore (1959) of the statement above. Not all Huffman codes are fully self-synchronising. The binary Huffman code shown in Fig. 1 is an example. Bobrow and Hakimi (1969) define a class of 'inclusion codes' which are equivalent to any given Huffman code, in the sense that they have the same code-lengths (and hence the same mean code length) as the Huffman code. They prove that every inclusion code whose code lengths are mutually prime is fully self-synchronising. Their proof appears to contain an error, but we know of no counter example, and an algorithm based on their work appears always to generate codes which are fully self-synchronising.

Transformations between fixed and variable length codes

We have seen that variable length codes may have some economic advantages over fixed length codes; however, for many purposes, a fixed length code is much superior. In this section we investigate the transformations between a fixed length and a variable length code.

The transformation from a fixed length code to a variable length code is economically realised by a simple table look up; two entries are required, one giving the length of the code word and the other the actual digits of the code. These are used to control shifting and packing operations, and the entire operation is simply realised using either hardware or software. It may be convenient to represent code words as non-standard floating point numbers, with the length as exponent and the digits of the code as the mantissa.

The transformation from a variable length to a fixed length encoding is not straightforward, and if implemented entirely by software may be disastrously slow. As before the output from the process must yield the length of the code word, allowing the corresponding number of digits to be discarded from the input sequence after each transformation. We describe below three techniques.

The first two are a software and a hardware implementation of a tree search, based on a stored representation of the tree from which the code is derived. The third is a decision circuit which is in effect a direct representation of the tree.

Suppose the code is developed on a p-ary tree, the number of terminal symbols being n. The total number of non-terminal nodes is $(n - 1)/(p - 1)$, and the total number of nodes is thus $n + (n - 1)/(p - 1)$. Each node contains either the addresses of p nodes (if it is a non-terminal node) or the digits of a particular code word (if it is a leaf). The algorithm for searching such a tree is:

1. Extract the root of the tree from the table, put it in W.
2. Extract the next digit, q, from the message, $(0 \leqslant q \leqslant p)$.
3. Extract the qth field from W, say W_q.
4. Extract the W_qth word from the table
   if this is a leaf FINISH
   if not, call this word W, return to (2).

Obviously at the conclusion the number of digits extracted in step (2) is simply the code-length for this code. This decoder works for any code which will fit into the table, and the code is in fact defined by this table. The average number of read operations in performing the decoding is one greater than the average code-length. This algorithm can be implemented in software, using as its store part of the main store of the central computer. In the case where p is an exact power of 2 the algorithm can economically be implemented by hardware. (It can of course be realised in hardware for any p, but it is less economic.) The decoder uses a shift register S to hold the head of the message being decoded; associated with S is some form of reloading to ensure that the shift register recharges as necessary. S shifts in steps of $\log_2 p$ binary digits per shift. The leftmost $\log_2 p$ digits drive a one-out-of-p decoder whose outputs gate a field from the store buffer register SBR into the store address register SAR. The store must hold $n + (n - 1)/(p - 1)$ words, each capable of holding either p addresses in the range 0 to $n + (n - 1)/(p - 1)$, or the fixed length pattern corresponding to a terminal symbol, together with a flag to indicate which of these is held.

A cycle of the decoder functions as follows. SAR is loaded with the address of the root node, and the store stimulated, loading SBR with a node-word. If the flag bit is 0, the node-

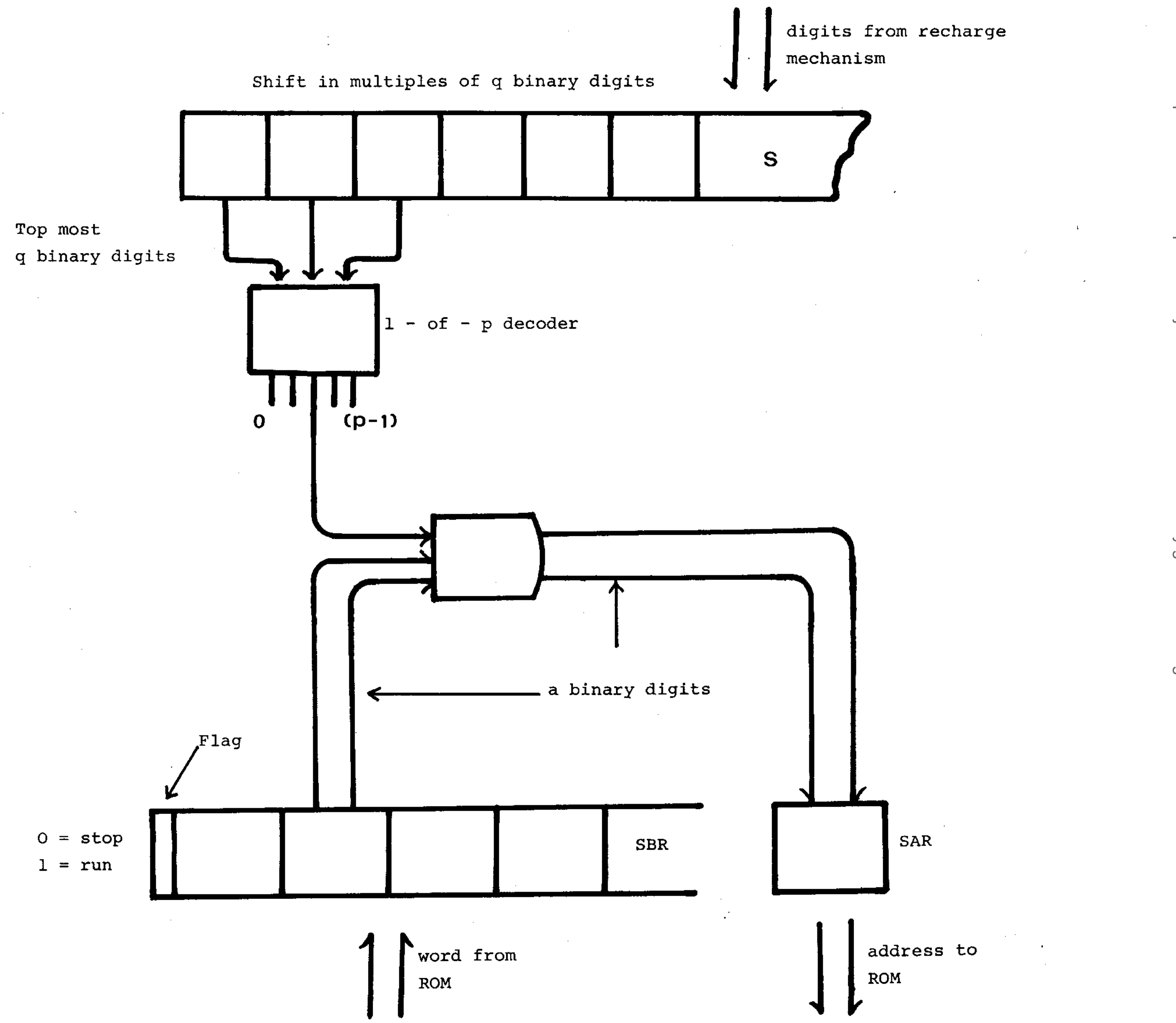

Fig. 2. (*a*) **Stored table variable-to-fixed length decoder.**
Number of characters $= n$ **Branching ration** $= p$
$q = \lceil \log_2 p \rceil$ = **number of binary digits per code digit.**

$a = \lceil \log_2 (n + (n - 1)/(p - 1)) \rceil$ = **number of binary digits per tree address.**

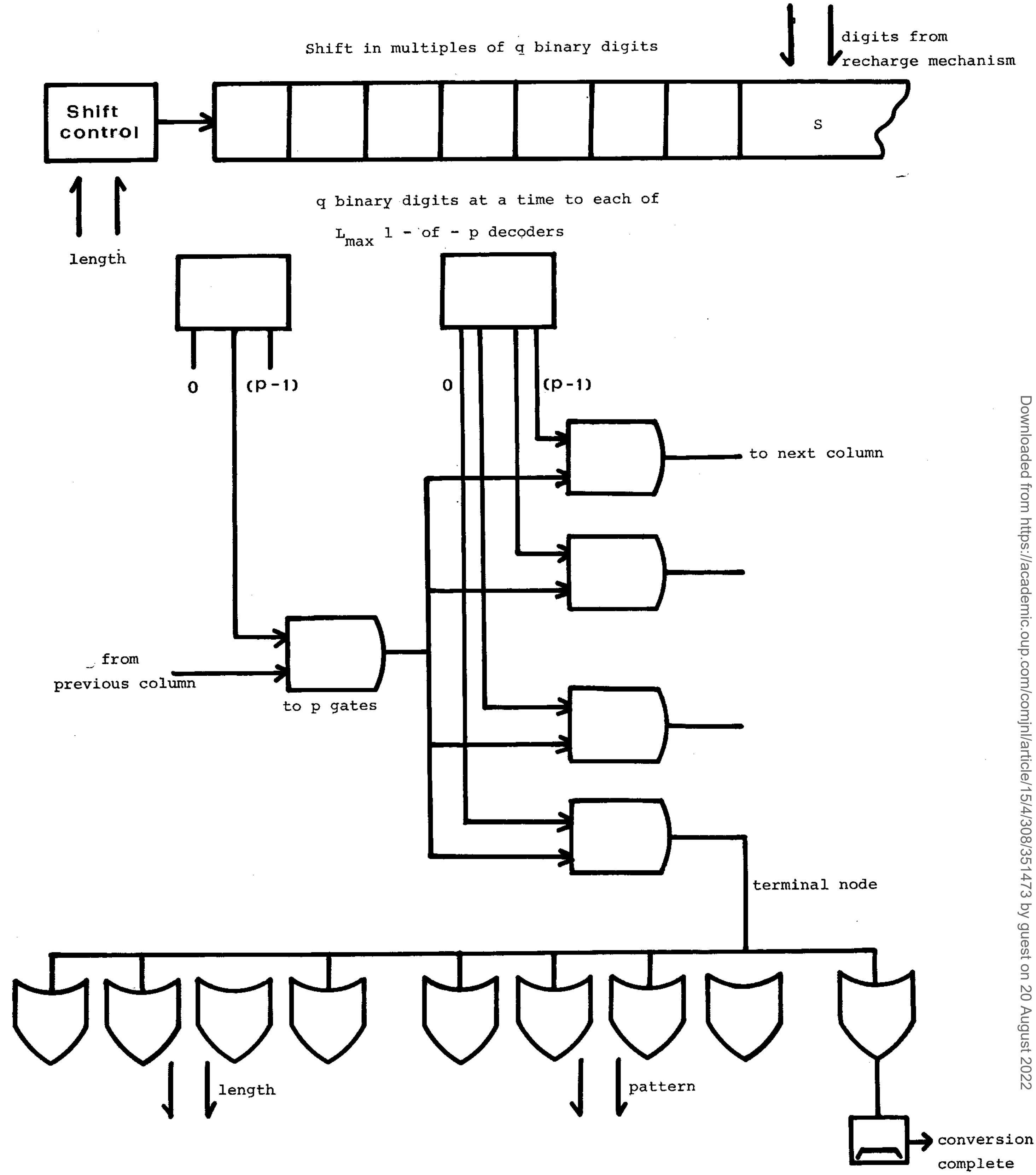

Fig. 2. (*b*) **Decision circuit variable-to-fixed length decoder**

word corresponds to a leaf, and the corresponding fixed length code is in *SBR*. If the flag bit is 1, the leftmost digit of *S* will select the corresponding field of *SBR* for transfer to *SAR* and the cycle will repeat. The store here can be a read only store, if one is prepared to sacrifice flexibility for speed. The mean time for a conversion is dominated by a term $(1 + h_{av})\tau$ where τ is the cycle time for the store; the worst case time $(1 + L_{max})\tau$. For a typical *MOS* read-only memory, where τ is in the order of 100 ns, and for a code with $p = 2$, $L_{av} = 6$ $L_{max} = 24$ these times are about 700 ns and 2·5 μs respectively, indicating that conversion by this technique for such codes can run in real time for existing file store devices.

If still faster transformation is required, it can be achieved by a decision circuit technique, which implements the tree directly. We again use a register *S*, holding the digits of the message for some radix *p*, and having a recharge mechanism. The contents of the leftmost L_{max} positions in *S* are supplied to L_{max} one-of-*p* decoders whose outputs enable an associated column of two input AND gates, arranged in clusters of p elements to correspond with the nodes of the original tree. It is then obvious that starting from the left, one and only one AND gate in each column will raise its output. This output corresponds to the branch which is selected at this level of the tree, and corresponds either to a leaf, or an entry to a further node, i.e. to a cluster of p AND gates in the next column. The outputs corresponding to leaves could be used to access a line in a store containing the associated fixed length code and the number of places by which S is to be shifted, i.e. the number of digits in the variable length code. It is equally simple to construct decision circuits to generate this information directly. These will consist in principle of two clusters of OR gates, one for the fixed length code, one for the length of the variable length code, with inputs from the selected leaf to those positions where a '1' is required. (In practice the very high fan-in will require more than one level of gating.) The circuit is of course asynchronous, and completion of a decoding is signalled by an output appearing at an AND corresponding to a leaf. To cover delays in the length and pattern generating circuitry, it would be possible to generate the conversion complete signal by taking an OR via a delay. If we assume gates with a mean delay of τ, and a shift register with a shift-time σ, the average and maximum conversion times are approximately $2\tau + L_{av}(\tau + \sigma)$ and $2\tau + L_{max}(\tau + \sigma)$, which for typical IC logic where $\tau \simeq \sigma \simeq 10$ ns are about 160 ns and 500 ns respectively.

Conclusions

We have seen that substantial savings in the total number of digits required to represent a file may be possible by using a single variable length code. For ALGOL, which has the richest symbol set of the languages we have considered, a fixed length code would need at least seven binary digits; the corresponding variable length code would require rather less than 5·6 binary digits per symbol on average. For FORTRAN and assembly language as normally represented the corresponding savings are from six binary digits to again about 5·6 binary digits per symbol. However, there seems no good reason why one should not have 'FORTRAN compound basic symbols' just as one does for ALGOL; other benefits apart, their existence would eliminate or simplify the lexicographic scan present in most compilers. If this technique were adopted, a compression of about 20% might be achieved for free format files—of course no such compression is possible for fixed format files. The economic effect lies not so much in the reduction in backing store *volume* as in the reduction of backing store *traffic*; most backing store devices have a total volume which is gross in comparison with their long term average transfer rate and an effective 20% increase in this rate would often be welcome. Against this one must set the cost of extra equipment, hardware or software, required to carry out the conversion. It is our view that the economies may well be worthwhile.

Appendix 1

This algorithm is based on the proof by Gilbert and Moore that the existence of a universal synchronising sequence is a necessary and sufficient condition for a code to be fully self-synchronising.

1. From among all the code words choose the one with the longest sequence of zeros. If there is a choice choose the code word with the longest sequence of digits after the sequence of zeros. Denote by A the sequence of zeros and the remaining digits of the code word, and by a the number of digits in A
(e.g. if the two code words with the longest sequence of zeros are

10100110100000010
101001101000000110

then A is 000000110 and a is 9).
2. For $i = 1(1)\ a - 1$ attempt to decode A starting from the beginning of A and from the ith digit in A. After each attempt write down the digits left over, calling them B_1, B_2 etc. in ascending order of length. Find the shortest sequence of code words S such that B, S is a valid message. Add S to the end of A and repeat this whole step until the set B_i is empty, i.e. until A decodes completely wherever one starts to decode it. A is now a universal synchronising sequence.

Proof:
A decodes completely from the beginning and after the ith digits for $i = 1(1)a$. The code is exhaustive and hence contains an 'all zeros' code word of length A or less. Consider decoding any sequence GA: three possibilities exist:

1. G was completely decoded and thus A is decoded from the beginning which by construction can be done completely.
2. The digits remaining after decoding G are those discarded in step 1 of constructing A. A is decodable after the a-th digit by construction.
3. No other digits remaining after decoding G can force decoding of A to begin further in than the a-th bit. By construction A is decodable from every position up to the a-th.

A recent book by J. J. Stiffler, *Theory of Synchronous Communications*, Prentice-Hall 1971, gives a very thorough treatment of self-syncronising codes,

References

Bobrow, L. S., and Hakimi, S. L. (1969). Graph Theoretic Prefix Codes and their synchronising Properties, *Information and Control*, Vol. 15, pp. 70-94.

Gilbert, E. N., and Moore, E. F. (1959). Variable Length Binary Encodings, *Bell System Technical Journal*, Vol. 38, pp. 933-967.

Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes, *Proc. I.R.E.*, Vol. 4D, No. 9, pp. 1098-1101.

Maurer, W. C. (1969). File Compression Using Huffman Codes, *Computing Methods in Optimisation Problems*, Vol. 2, pp. 247-256.