out destroying the 1D C-R property, it is necessary and sufficient that the same record appear in two queries. Thus the maximum number of times a record can be deleted and still preserve the 1D C-R property is $[(m-1)/2]$. There are m records; hence the redundancy is

$$\frac{m(m-1) - m[(m-1)/2]}{m} - 1 = (m-2) - [(m-1)/2].$$

When m is odd, then in a 1D CRWR organization, there will be one record at one end of the organization, which cannot be deleted. Thus, the end correction to be added to the redundancy is $(m - 2[m/2])/m$. Thus, the redundancy of the CRWR organization is

$$(m-2) - [(m-1)/2] + (m - 2[m/2])/m.$$

As there are only two records in each query, it is easy to show that such a 1D CRWR organization can be constructed with any value of m.

This completes the proof.

Received November 1973; revised February 1975

References

1. Ghosh, S.P. On the theory of consecutive storage of relevant records. *J. Inf. Science 16* (1973), 1–9.
2. Ghosh, S.P. File organization: the consecutive retrieval property. *Comm. ACM 15*, 9 (1972), 802–808.
3. Ghosh, S.P. File organization: consecutive storage of relevant records on a drum-type storage. *Inf. Control 25*, 2 (1974), 145–165.
4. Liu, C.L. *Introduction to Combinatorial Mathematics*. McGraw-Hill, New York, 1968.
5. Hall, Jr., M. *Combinatorial Theory*. Blaisdell Pub. Co., Waltham, Mass., 1967.
6. Waksman, A., and Green, M.W. On the consecutive retrieval property in file organization. *IEEE Trans. on Computers C-23* (1974), 173–174.
7. Nakano, Takeo. A characterization of intervals; the consecutive (one's or retrieval) property. *Comment Math. 22*, 1 (1973), 49–59.

Programming Techniques

G. Manacher Editor

Multiple Byte Processing with Full-Word Instructions

Leslie Lamport
Massachusetts Computer Associates, Inc.

A method is described which allows parallel processing of packed data items using only ordinary full-word computer instructions, even though the processing requires operations whose execution is contingent upon the value of a datum. It provides a useful technique for processing small data items such as alphanumeric characters.

Key Words and Phrases: byte processing, character processing, packed data

CR Categories: 4.9

Introduction

One often has the problem of processing many similar data items, each of which is much shorter than a full computer word. One would like to pack several items to a word and process them simultaneously in order to reduce both storage space and processing time.

As a simple example, suppose the data are vectors of the form $\mathbf{a} = (a_1, \ldots, a_m)$, where each a_i is a nonnegative integer with a small range of possible values. We can pack some number k of the elements a_i in a single computer word, so only about m/k words of storage are needed for each vector. Certain operations on these vectors can be done without unpacking them. For example, given another similar vector $\mathbf{b}$, we can form the sum $\mathbf{a} + \mathbf{b} = (a_1 + b_1, \ldots, a_m + b_m)$ using

This research was supported by the Advanced Research Projects Agency under Contract Number DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., 26 Princess St., Wakefield, MA 01880.

full-word binary addition on each word of packed elements. The vector sum is thus computed with only m/k separate add instructions, so we have saved time as well as space by packing the data. (The problem of overflow is considered later.) Other operations such as bit-wise logical operations and multiplication of a vector by a scalar can be done in a similar way.

However, not all operations are so easy to do on packed data. Some involve making decisions on the basis of the values of individual data elements, as in computing maximum $(\mathbf{a},\mathbf{b})$ = (maximum (a_1,b_1), ..., maximum (a_m,b_m)). It is not obvious how such an operation can be performed without unpacking the data.

Sequences of similar data items are also encountered when processing strings of alphanumeric characters. We would like to pack several characters in a single word and process them simultaneously. We will consider the sample problem of comparing two strings of characters and forming an output string indicating where they differ. In particular, we will want the output string to contain a # where the two strings differ, and a blank where they are the same. For example, given the input strings *X Y Z Z Y X* and *X Y U Z V X*, we want to produce the output string △ △ # △ # △ (where △ denotes the blank character). Once again, it is not obvious how an entire word full of characters can be processed simultaneously.

We will describe a general method for doing such processing of packed data which should prove useful in many applications. It requires only the following full-word operations, which are common to most binary computers: logical *and* ($\wedge$), *or* ($\vee$), and *exclusive or* ($\oplus$); binary addition ($+$); and n-bit left or right shift ($\leftarrow_n$ or $\rightarrow_n$). (Although some of these operations can be programmed using the others, it would be impractical to do so.) Binary multiplication ($*$) and logical complement ($\sim$) are also useful, but not necessary.

The Basic Method

In order to perform different operations on different items in a single word, we must construct bit masks. We need a mask word whose ith item consists of all ones if item i of the data word is to be changed, and all zeros otherwise. In our character string comparison problem, if one string has the substring *X Y Z* stored in a single computer word, and the other string has the corresponding substring *X Y U* stored in a single word, then we want to construct the mask word 0 . . . 0 0 . . . 0 1 . . . 1 containing three mask characters. Given such a mask (and its complement), the rest is easy. We will describe how to construct this mask.

Let the data items consist of n-bit fields. We assume that each item is stored in an $(n + 1)$-bit byte whose leftmost bit equals zero. (A method not requiring the extra bit can often be used. It is described later.) Several bytes are stored in a single computer word. For convenience, the following description is in terms of operations on a single byte. However, since the operations are performed using only the full-word computer instructions listed above, they can be done simultaneously on all the bytes of a word.

The bits of a byte are numbered 0–n from left to right. The ith bit of byte a is denoted by a_i, and a_{j-n} denotes bits j through n of a, for $j \leq n$. Let 0^n and 1^n denote strings of n zeros and n ones, respectively, so 01^n denotes the $(n + 1)$-bit quantity 011 . . . 1.

Let p be a logical function of two data items—i.e. a relation. We want to define a *masking function* m_p to be a function whose value is a mask of ones when p is true and a mask of zeros when p is false. For masking operations on the $(n + 1)$-bit bytes, the value of bit 0 of a mask byte is usually immaterial. We therefore define a masking function m_p as follows:

$$\begin{aligned} m_p(a, b)_{1-n} &= 1^n \quad \text{if} \quad p(a,b) = \textit{true}, \\ &= 0^n \quad \text{if} \quad p(a,b) = \textit{false}, \end{aligned}$$

for all $(n + 1)$-bit data items a and b with $a_0 = b_0 = 0$. (Recall that bit 0 of a data item is assumed to be zero.) For a given relation p, we must write a program to evaluate m_p using only full-word instructions.

The basic idea is to compute m_p in two steps. In Step 1, we compute a *test function* f_p which has the following property:

$$\begin{aligned} f_p(a, b)_0 &= 1 \quad \text{if} \quad p(a,b) = \textit{true}, \\ &= 0 \quad \text{if} \quad p(a,b) = \textit{false}, \end{aligned}$$

for all a, b with $a_0 = b_0 = 0$. In Step 2, we construct a mask byte which depends upon the value of $f_p(a, b)_0$. Thus, Step 1 puts the value of the relation p into bit 0, and Step 2 spreads that value into the mask bits.

Step 2

We describe Step 2 first. Given the value of $f_p(a,b)$, we can construct the mask m_p by the following three operations:

(1) $x := \rightarrow_n (f_p(a,b))$ [shift bit 0 into bit n].
(2) $y := x \wedge 0^n1$ [mask out bits 0 to $n - 1$].
(3) $m_p(a,b) := y * 01^n$ [multiply by a mask of 1's].

To compute $m_{\sim p}$—a masking function for the negation of p—we replace operation (3) by

(3′) $m_{\sim p}(a,b) := y + 01^n$.

If we do not want to use a multiply instruction to compute m_p, we can set $m_p(a,b) := \sim m_{\sim p}(a,b)$. This can be done without a complement instruction, since $\sim z = z \oplus 1^{n+1}$.

Step 1

To describe Step 1, we define test functions for several common relations. Logical combinations of these relations are computed in the obvious way. For example, $f_{p \vee q}(a,b) = f_p(a,b) \vee f_q(a,b)$ for any relations p and q. All of the following definitions only use operations which can be performed on a full word of bytes with the aforementioned computer instructions.

(a) *Equality*. For the equality relation, observe that $a = b$ if and only if $a \oplus b = 0^{n+1}$, which is true if and only if $a \oplus b \oplus 01^n = 01^n$. It is then easy to see that if $f_=$ is defined by

$$f_=(a,b) = (a \oplus b \oplus 01^n) + 0^n1,$$

then $f_=(a,b)_0 = 1$ if and only if $a = b$. Hence, $f_=$ is a test function for the equality relation.

(b) *Comparison of unsigned integers*. Let $a < b$ mean that a is numerically less than b if a and b are interpreted as nonnegative (unsigned) binary integers. Then $a < b$ if and only if $2^n \leq b + 2^n - a - 1$. But $2^n - a - 1$ is just the n-bit logical complement of a, which equals $a \oplus 01^n$, and $2^n \leq z$ if and only if $z_0 = 1$. We can therefore define the test function $f_<$ by

$$f_<(a,b) = b + (a \oplus 01^n).$$

Let $\leq$ be the relation $<$ or $=$. Since $a \leq b$ if and only if $a < b + 1$, we can define the test function $f_\leq$ by

$$f_\leq(a,b) = b + 0^n1 + (a \oplus 01^n).$$

(The sum is always less than 2^{n+1}, so overflow out of bit 0 is impossible.)

(c) *Comparison of signed integers*. Let $a \ll b$ mean that a is numerically less than b when a and b are interpreted as two's complement signed binary integers, with bit 1 as the sign bit. (Two's complement arithmetic seems the most natural for byte computations.) Observe that

(i) If $a_1 = b_1$ then $a \ll b$ if and only if $a < b$.

(ii) If $a_1 \neq b_1$ then $a \ll b$ if and only if $a > b$.

It is then easy to see that the test function $f_{<<}$ can be defined by

$$f_{<<}(a, b) = f_<(a, b) \oplus \leftarrow_1 (a \oplus b).$$

The test function for the relation $\lll$ ($\ll$ or $=$) is similarly defined by

$$f_{\underline{<<}}(a, b) = f_\leq(a, b) \oplus \leftarrow_1 (a \oplus b).$$

(d) *Overflow*. We now consider the problem of overflow on an addition operation. (Subtraction presents a similar problem, and is left to the reader.) When performing vector addition, we usually have to test for overflow and take some special action if it occurs. Let us define the overflow relation ov^+ such that $ov^+(a,b)$ is true if the sum of a and b is outside the range of representable numbers. The problem is trivial for unsigned integers, since we can simply let $f_{ov^+}(a, b)$ equal $a + b$—i.e. bit 0 of the $(n + 1)$-bit byte acts as an overflow indicator. For two's complement signed integers, the following program is one of several ways to compute both $f_{ov^+}(a, b)$ and the sum of a and b.

```
temp := $a + b$;
sum$(a, b)$ := temp $\wedge 01^n$;
$f_{ov^+}(a, b)$ := temp $\oplus \leftarrow_1$ [sum $(a, b) \oplus a \oplus b$];
```

We can use f_{ov^+} to construct the necessary mask to perform some special processing for each byte in which overflow occurred. We could also use it to determine if overflow occurred in any byte within a word. This is discussed later.

Programming Techniques

It is a straightforward task to program parallel byte operations using this mask generation technique. However, the following observations may prove useful.

(a) No overflow occurs from bit 0 of a byte. Thus, either one's complement or two's complement full-word arithmetic operations can be used. However, if bit 0 of the leftmost byte is the leftmost bit of the word, then an add instruction can generate an overflow condition—bit 0 of both operand words equal to zero and bit 0 of the result equal to one. Hence, any overflow interrupt must be inhibited. It may be impossible to use the sign bit of a computer with sign/magnitude arithmetic, depending upon what happens when addition of positive numbers generates an overflow.

(b) On a shift operation, the values of the bits shifted into the end of the word are immaterial. Hence, any type of shift or rotate instruction may be used.

(c) It is sometimes necessary to test each byte against a single value. To do this, the value can be spread to all the bytes of a word by multiplying it by a word of 0^n1 bytes.

(d) Common subexpressions can often be found when several test functions must be evaluated. This is aided by the fact that the $\oplus$ operation is associative and commutative.

(e) One must sometimes determine if the relation $p(a, b)$ holds for any or for all pairs of data items a, b. For example, we may want to generate an error message if overflow occurred in any of the sums of a vector addition operation. Testing for a zero word after operation (2) of Step 2 provides a *for any* test. A *for all* test is done with a *for any* test of $\sim p$. It may pay to make a *for any* test in order to skip a calculation if no bytes in the word are to be modified.

An Example Programmed

Let us now consider an actual program to solve our character string comparison problem. Assume a simple computer with one accumulator, an index register, single address instructions, and the following branching instructions: *increment index and branch* (IXB) which increments the index register and branches if its value is not zero, *branch if accumulator is zero* (BRZ), and *unconditional branch* (BR). The following is a typical program for our problem when one character is stored per word. (We assume that the index register is properly initialized.)

```
α: LOAD          a, indexed
   EXC·OR        b, indexed
   BRZ           β
   LOAD          "#"
   STORE         c, indexed
   IXB           α
   BR            γ
β: LOAD          "△"
   STORE         c, indexed
   IXB           α
γ: . . .
```

This program executes six instructions per character.

Multiple character per word processing can be done with the following program. The bracketed expressions indicate the contents of one byte of the accumulator after executing the instruction, and $\{z\}$ denotes a word each byte of which equals z.

```
α: LOAD          a, indexed
   EXC.OR        b, indexed    [a ⊕ b]
   EXC.OR        "{01^n}"      [a ⊕ b ⊕ 01^n]
   ADD           "{0^n1}"      [f_=(a, b)]
   R.SHIFT       n
   AND           "{0^n1}"
   ADD           "{01^n}"      [m_≠(a, b)]
   STORE         temp
   AND           "{#}"
   STORE         c, indexed
   LOAD          temp
   COMPLEMENT                  [m_=(a,b)]
   AND           "{△}"
   OR            c, indexed
   STORE         c, indexed
   IXB           α
```

This program executes 16 instructions per full word of characters.

Given any problem, multiple byte processing will be faster than single item per word processing if enough bytes can be put into a single word. For a given problem and a given computer, let the *break-even number* be the number of bytes per word which makes the two processing times equal. If we assume that all instructions take the same amount of time to execute, then the break-even number for our example is $16/6 \approx 2.7$. Multiple byte processing is thus faster if three or more bytes fit in a single word.

General Observations

Having found the break-even number for one problem and one computer, we now make some general observations about how it should vary when the problem or the computer is changed.

(a) Most large computers have several registers instead of a single accumulator. Using two registers, we can remove one instruction from each of the above programs, increasing the break-even number to $15/5 = 3$. However, if the constants are initially placed in registers, then the extra instructions required for multiple byte processing become register to register operations. These are usually faster than operations which reference memory. This will tend to lower the break-even number.

(b) It is difficult to make any general statement about the effect of a larger instruction set. However, observe that if there is a *masked store* instruction on a multiple register computer, then four more instructions can be eliminated from the multiple byte processing loop, reducing the break-even number to 2.2 (assuming enough characters in a string so initialization times can be ignored).

(c) Conditional branches slow down execution on a high-speed pipelined computer like the CDC 7600. Since multiple byte processing does not require conditional branching to test individual characters, this will lower the break-even number for such a computer.

(d) On an array computer like the Illiac-IV, testing must be done by masking (i.e. disabling individual processors) even for one item per word processing. Multiple byte processing will usually be faster on such a computer if there are more data items than processors.

(e) When only one of several operations is to be performed on each data item, multiple byte processing may perform all the operations for each word, while single item per word processing does just one operation for each item. Increasing the number of different operations will tend to increase the break-even number.

(f) The above example is perhaps unrealistically simple. A more complicated example is provided by a problem taken from an actual compiler optimization algorithm. For each i, we must perform the following calculation:

```
if a[i] = 0
  then a[i] := b[i]
  else if a[i] ≠ x and a[i] ≠ b[i]
         then begin
                a[i] := x;
                if y then flag := true
              end,
```

For the simple computer described above, packing the arrays a and b gives a break-even number which varies from 2.9 to 4.4, depending upon the relative execution frequencies of the different conditional clauses. Reasonable values for these give a break-even number of about 4.

(g) The two examples we have given just require selecting the value of a data item—no real processing of the items is done. This is typical of problems involving small data items. The way in which the break-even number will vary with the amount of processing depends upon the efficiency of doing the operations on a full word of bytes and the probability that an operation will be performed on any single item. It is hard to make any general statements about this. However, we would expect that if the operations are easily performed simultaneously on all bytes of a word, then the break-even number for our simple computer should be in the 3 to 4 range for most real problems.

(h) Storage space restrictions may require that data items be packed several to a word. In this case, the cost of single item per word processing must include the unpacking and repacking operations. This will dramatically lower the break-even number unless a great deal of processing must be done for each item.

Eliminating the Extra Bit

It may be desirable to eliminate the $(n+1)$-th bit (bit 0) for either of two reasons: (i) to allow more bytes per word, or (ii) because other factors make it awkward to obtain the extra bit.[1] We now show how to eliminate the extra bit if testing only requires the $=$ or $\neq$ relations.

Using $(n+1)$-bit bytes, the computation of $f_=(a, b)$ is performed as follows:

$x := a \oplus b \oplus 01^n,$
$f_=(a,b) := x + 0^n1.$

Since $x_0 = 0$, $f_=(a, b)_0$ will equal 1 if and only if there is a carry out of bit 1 in forming the sum $x_{1-n} + 1$. But this will happen if and only if bit 1 of $x_{2-n} + 1$ and bit 1 of x both equal 1. This suggests that we define the function $g_=$ by the following operations on n-bit bytes.

$x := a \oplus b \oplus 1^n \qquad [=(\sim a) \oplus b]$
$y := x \wedge 01^{n-1} \qquad [= x_{2-n}]$
$g_=(a, b) := (y + 0^{n-1}1) \wedge x$

Then $g_=(a, b)_1 = 1$ if and only if $a = b$, so $g_=$ can be used as a test function, with the obvious modification of Step 2.

The computation of $g_=$ is more complicated than that of $f_=$. On our single accumulator computer, it requires three extra operations. For the original sample problem, this adds about 20 percent to the computing time, and raises the break-even number to about 3.2. For a computer with multiple registers, only two extra operations are needed.

This should be a practical method of testing for equality despite the extra computation. Testing for inequality is, of course, done with $\sim g_=$.

A similar trick can be used for the relations $<$, $\leq$, $\ll$ and $\leqq$. However, it is more complicated, and requires so much more computation that it seems impractical to test for these relations without using the extra bit. For example, computing the analogous function $g_<$ would involve the following operations:

$x := (\sim a \wedge 01^{n-1}) \oplus (b \wedge 01^{n-1})$
$g_<(a,b) := (x \wedge \sim a) \vee (x \wedge b) \vee (\sim a \wedge b).$

[1] Conversely, these other factors sometimes provide the extra bit. For example, if ASCII characters must be stores in 8-bit bytes, then the parity bit position can be used as bit 0.

Even on a multiple register computer, this requires about three times as many instructions as the computation of $f_<$.

Note that if arithmetic operations are to be performed on the bytes, then the extra bit is probably needed anyway to prevent overflow out of one byte from propagating to the next byte.

Conclusion

Parallel processing of packed data items with ordinary full-word computer instructions is possible even if the computation includes operations contingent upon the value of an item. It will be at least as fast as one item per word processing if enough items can be packed into a word, allowing one extra bit per item. (If the tests just require the relations $=$ and $\neq$, then the extra bit is not necessary.) The required minimum number of items per word depends upon the problem and the choice of computer, but three or four items per word is probably sufficient in many cases. The technique should therefore be useful for a large number of problems involving small data items like alphanumeric characters.

Acknowledgment. The method of eliminating the extra bit was suggested by Donald Knuth.