problem, and states that the 'conventional MAC is not recommended for multicast messages'. It instead recommends the use of a 128-bit *Bidirectional MAC* (BMAC), the means of computation for which is described in Appendix A.2 of Ref. 2.

Yours faithfully
C. MITCHELL
Hewlett-Packard Laboratories,
Filton Road,
Stoke Gifford, Bristol BS12 6QZ

REFERENCES

1. C. Mitchell and M. Walker, Solutions to the multidestination secure electronic mail problem. *Computers and Security* **7**, 483–488 (1988).
2. J. Linn, Privacy enhancement for Internet electronic mail: Part I: Message encipherment and authentication procedures. *Request for comments 1040 (RFC 1040)*, *IAB Internet Privacy Task Force* (1988).

A note on dividing integers by two

Dear Sir,
A number of algorithms require that a two's complement integer be divided by two. The division operation can be performed rapidly by shifting the integer one place to the right. Many microprocessors have the *Arithmetic Shift Right* instruction for this purpose to give, at first sight, binary division by two. However, there is asymmetry between dividing positive and negative integers using this method which can be problematic. This note describes a simple method of removing this problem.

The problem

In two's complement notation, an 8-bit number can be used to represent the integers from -128 to $+127$. A positive integer has the top bit zero and the lower 7 bits indicate the value of the integer: bit r is a 1 if the integer has a component 2^r. After such an integer has been shifted to the right, the bit which indicated if the integer had a component 2^r, now shows if it has a component 2^{r-1}. Hence shift right gives integer divide by two. For example,

3 = 00000011 is shifted right to 00000001 = 1

That is, 3 DIV 2 = 1: this is sufficiently close to the correct answer given that integers only are represented.

To convert to a negative integer, all the bits are inverted (one's complement) and then one is added (two's complement). For example,

3 = 00000011: complemented is 11111100:
+1 gives 11111101 = −3

If a negative integer is shifted arithmetically to the right, the top bit is preserved to maintain the sign, for example

−3 = 11111101 is shifted to 11111110 = −2

Hence −3 DIV 2 = −2.

However, +3 DIV 2 = +1, but −3 DIV 2 = −2: this can be a problem. This result is due to the asymmetry of two's complement notation: one more integer can be represented less than zero than can be represented greater than zero.

Dividing −1 by two in this way gives another potential problem:

−1 = 11111111 is shifted to 11111111 = −1

That is, −1 DIV 2 = −1.

In the *Midpoint Subdivision Clipping algorithm*,[1] integers are processed in a loop of the form:

```
REPEAT
  do some operations
  divide integer by 2
UNTIL integer = 0
```

which will never stop if the integer is negative. A solution to this problem is to treat −1 as a special case and say shift right of −1 is 0. A more general solution is to add 1 to *any* negative integer before shifting:

−1 = 11111111, add 1 = 00000000,
shift right = 00000000 = 0

but this also works for all negative integers, for example

−3 = 11111101, add 1 = 11111110,
shift right = 11111111 = −1

−4 = 11111100, add 1 = 11111101,
shift right = 11111110 = −2

The algorithm

Hence, to maintain symmetry in processing both positive and negative integers, prior to doing the arithmetic shift right, the integer to be processed should be incremented if it is negative. This is equivalent to rounding the answer if it is negative, but not if it is positive. This very simple operation could be performed easily in hardware, although no current microprocessor has such an instruction.

Justification

This action can be justified easily. If it is assumed that shifting right a positive integer divides the integer by two correctly, then shifting right a one's complement integer correctly divides such an integer by two (each bit has been inverted only). Two's complement is one's complement + 1: adding one gives one's complement + 2; shifting this to the right gives one's complement shifted right (which is correct) + 2 shifted right (equals 1), that is the two's complement integer has been divided by 2.

More formally, for n-bit binary integers, a positive integer can be represented by[2]

$$P = \sum_{i=0}^{n-1} (C_i 2^i)$$

{where C_i is 0 or 1, and C_{n-1} is 0}

The two's complement of P is

$$-P = \sum_{i=0}^{n-1} (\bar{C}_i 2^i) + 1$$

{where $\bar{C}_i$ means NOT (C_i)}

P shifted right arithmetically, denoted by ASR (P), is thus given by

$$\mathrm{ASR}(P) = C_{n-1}2^{n-1} + \sum_{i=0}^{n-2} C_{i+1}2^i$$

$$-\mathrm{ASR}(P) = \bar{C}_{n-1}2^{n-1} + \sum_{i=0}^{n-2} (\bar{C}_{i+1}2^i) + 1$$

$$\mathrm{ASR}(-P+1) = \mathrm{ASR}\left(\sum_{i=0}^{n-1} (\bar{C}_i 2^i) + 2\right)$$

$$= \mathrm{ASR}\left(\sum_{i=0}^{n-1} (\bar{C}_i 2^i)\right) + \mathrm{ASR}(2)$$

$$= \bar{C}_{n-1}2^{n-1} + \sum_{i=0}^{n-2} (\bar{C}_{i+1}2^i) + 1$$

Hence $-\mathrm{ASR}(P) = \mathrm{ASR}(-P+1)$.

Conclusion

A simple algorithm is given which provides fast division by two for both positive and negative integers. This algorithm could be implemented easily in hardware.

Yours sincerely
R. J. MITCHELL and
P. R. MINCHINTON
Department of Cybernetics,
University of Reading,
3 Earley Gate,
Whiteknights,
Reading, Berks.

References

1. J. D. Foley and A. Van Dam. 'Fundamentals of Interactive Computer Graphics'. Addison Wesley. 1982.
2. D. L. Lewin, 'Theory and Design of Digital Computers'. Nelson. 1985.

Dear Sir,

In the course of my work I find it necessary to use a computer, and am faced with the choice between using a structured language such as Pascal or an unstructured one such as FORTRAN. Though the structures of Pascal give great clarity, I find that there are certain recurring schemes that can be implemented more concisely using the conditional GOTOs of FORTRAN. This appears to be due to a deficiency of Pascal which I propose could be remedied to some extent by the judicious addition of extra constructions to the language; in particular the addition of an **andif** clause to the existing **if then else** structure.

Any number of **andif** clauses could be appended to an **if** and each would have the effect of transferring control to the **else** if the associated condition was not met (or the next **elseif**, or out of the **if andif** structure in the absence of an **else** or **elseif**).

In the simplest case (using a single **andif**), where it is now necessary to write in Pascal

```
if ⟨condition 1⟩ then
  begin
  ⟨calculation concerning condition 2⟩;
  if ⟨condition 2⟩ then
    ⟨action 1⟩
  else
    ⟨action 2⟩
  end
else
  ⟨action 2⟩
```

or

```
FLAG: = false
if ⟨condition 1⟩ then
  begin
  ⟨calculation concerning condition 2⟩;
  if ⟨condition 2⟩ then FLAG: = TRUE
  end
if FLAG then
  ⟨action 1⟩
else
  ⟨action 2⟩
```