Fixed-Point Glue Setting

The preparation of this report was supported in part by the National Science Foundation under grants IST-7921977 and MCS-7723728; by Office of Naval Research grant N00014-81-K-0330; and by the IBM Corporation. 'TEX' is a trademark of the American Mathematical Society.

1. Introduction. If TeX is being implemented on a microcomputer that does 32-bit addition and subtraction, but with multiplication and division restricted to multipliers and divisors that are either powers of 2 or positive integers less than 2^{15}, it can still do the computations associated with the setting of glue in a suitable way. This program illustrates one solution to the problem.

Another purpose of this program is to provide the first "short" example of the use of WEB.

2. The program itself is written in standard Pascal. It begins with a normal program header, most of which will be filled in with other parts of this "web" as we are ready to introduce them.

program *GLUE* (*input*, *output*);
  type ⟨ Types in the outer block 6 ⟩
  var ⟨ Globals in the outer block 8 ⟩
  procedure *initialize*; { this procedure gets things started }
    var ⟨ Local variables for initialization 9 ⟩
    begin ⟨ Set initial values 10 ⟩;
    end;

3. Here are two macros for common programming idioms.

  define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
  define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }

4. The problem and a solution. We are concerned here with the "setting of glue" that occurs when a TEX box is being packaged. Let $x_1, \ldots, x_n$ be integers whose sum $s = x_1 + \cdots + x_n$ is positive, and let t be another positive integer. These x_i represent scaled amounts of glue in units of sp (scaled points), where one sp is 2^{-16} of a printer's point. The other quantity t represents the total by which the glue should stretch or shrink. Following the conventions of TEX82, we will assume that the integers we deal with are less than 2^{31} in absolute value.

After the glue has been set, the actual amounts of incremental glue space (in sp) will be the integers $f(x_1), \ldots, f(x_n)$, where f is a function that we wish to compute. We want $f(x)$ to be nearly proportional to x, and we also want the sum $f(x_1) + \cdots + f(x_n)$ to be nearly equal to t. If we were using floating-point arithmetic, we would simply compute $f(x) \equiv (t/s) \cdot x$ and hope for the best; but the goal here is to compute a suitable f using only the fixed-point arithmetic operations of a typical "16-bit microcomputer."

The solution adopted here is to determine integers a, b, c such that

$$f(x) = \lfloor 2^{-b} c \lfloor 2^{-a} x \rfloor \rfloor$$

if x is nonnegative. Thus, we take x and shift it right by a bits, then multiply by c (which is 2^{15} or less), and shift the product right by b bits. The quantities a, b, and c are to be chosen so that this calculation doesn't cause overflow and so that $f(x_1) + \cdots + f(x_n)$ is reasonably close to t.

The following method is used to calculate a and b: Suppose

$$y = \max_{1 \le i \le n} |x_i| .$$

Let d and e be the smallest integers such that $t < 2^d s$ and $y < 2^e$. Since s and t are less than 2^{31}, we have $-30 \le d \le 31$ and $1 \le e \le 31$. An error message is given if $d + e \ge 31$; in such a case some x_m has $|x_m| \ge 2^{e-1}$ and we are trying to change $|x_m|$ to $|(t/s)x_m| \ge 2^{d+e-2} \ge 2^{30}$ sp, which TEX does not permit. (Consider, for example, the "worst case" situation $x_1 = 2^{30} + 1$, $x_2 = -2^{30}$, $t = 2^{31} - 1$; surely we need not bother trying to accommodate such anomalous combinations of values.) On the other hand if $d + e \le 31$, we set $a = e - 16$ and $b = 31 - d - e$. Notice that this choice of a guarantees that $\lfloor 2^{-a} |x_i| \rfloor < 2^{16}$. We will choose c to be at most 2^{15}, so that the product will be less than 2^{31}.

The computation of c is the tricky part. The "ideal" value for c would be $\rho = 2^{a+b} t/s$, since $f(x)$ should be approximately $(t/s) \cdot x$. Furthermore it is better to have c slightly larger than ρ, instead of slightly smaller, since the other operations in $f(x)$ have a downward bias. Therefore we shall compute $c = \lceil \rho \rceil$. Since $2^{a+b} t/s < 2^{a+b+d} = 2^{15}$, we have $c \le 2^{15}$ as desired.

We want to compute $c = \lceil \rho \rceil$ exactly in all cases. There is no difficulty if $s < 2^{15}$, since c can be computed directly using the formula $c = \lfloor (2^{a+b} t + s - 1)/s \rfloor$; overflow will not occur since $2^{a+b} t < 2^{15} s < 2^{30}$.

Otherwise let $s = s_1 2^l + s_2$, where $2^{14} \le s_1 < 2^{15}$ and $0 \le s_2 < 2^l$. We will essentially carry out a long division. Let t be "normalized" so that $2^{30} \le 2^h t < 2^{31}$ for some h. Then we form the quotient and remainder of $2^h t$ divided by s_1,

$$2^h t = q s_1 + r_0, \qquad 0 \le r_0 < s_1.$$

It follows that $2^{h+l} t - qs = 2^l r_0 - q s_2 = r$, say. If $0 \ge r > -s$ we have $q = \lceil 2^{h+l} t/s \rceil$; otherwise we can replace (q, r) by $(q \pm 1, r \mp s)$ repeatedly until r is in the correct range. It is not difficult to prove that q needs to be increased at most once and decreased at most seven times, since $2^l r_0 - q s_2 < 2^l s_1 \le s$ and since $q s_2 / s \le (2^h t/s_1)(s_2/2^l s_1) < 2^{31}/s_1^2 \le 8$. Finally, we have $a + b - h - l = -1$ or -2, since $2^{28+l} \le 2^{14} s = 2^{a+b+d-1} s \le 2^{a+b} t < 2^{a+b+d} s = 2^{15} s < 2^{30+l}$ and $2^{30} \le 2^h t < 2^{31}$. Hence $c = \lceil 2^{a+b-h-l} q \rceil = \lceil \frac{1}{2} q \rceil$ or $\lceil \frac{1}{4} q \rceil$.

An error analysis shows that these values of a, b, and c work satisfactorily, except in unusual cases where we wouldn't expect them to. When $x \ge 0$ we have

$$\begin{aligned} f(x) &= 2^{-b}(2^{a+b} t/s + \theta_0)(2^{-a} x - \theta_1) - \theta_2 \\ &= (t/s)x + \theta_0 2^{-a-b} x - \theta_1 2^a t/s - 2^{-b} \theta_0 \theta_1 - \theta_2 \end{aligned}$$

where $0 \le \theta_0, \theta_1, \theta_2 < 1$. Now $0 \le \theta_0 2^{-a-b} x < 2^{e-a-b} = 2^{d+e-15}$ and $0 \le \theta_1 2^a t/s < 2^{a+d} = 2^{d+e-16}$, and the other two terms are negligible. Therefore $f(x_1) + \cdots + f(x_n)$ differs from t by at most about $2^{d+e-15} n$. Since 2^{d+e} is larger than $(t/s)y$, which is the largest stretching or shrinking of glue after expansion, the error is at worst about $n/32000$ times as much as this, so it is quite reasonable. For example, even if fill glue is being used to stretch 20 inches, the error will still be less than $\frac{1}{1600}$ of an inch.

5. To sum up: Given the positive integers s, t, and y as above, we set

$$a \leftarrow \lfloor \lg y \rfloor - 15, \qquad b \leftarrow 29 - \lfloor \lg y \rfloor - \lfloor \lg t/s \rfloor, \qquad \text{and} \qquad c \leftarrow \lceil 2^{a+b} t/s \rceil.$$

The implementation below shows how to do the job in Pascal without using large numbers.

6. TeX wants to have the glue-setting information in a 32-bit data type called *glue_ratio*. The Pascal implementation of TeX82 has *glue_ratio* = *real*, but alternative definitions of *glue_ratio* are explicitly allowed.

For our purposes we shall let *glue_ratio* be a record that is packed with three fields: The *a_part* will hold the positive integer $a + 16$, the *b_part* will hold the nonnegative integer b, and the *c_part* will hold the nonnegative integer c. When the formulas above tell us to take $b > 30$, we might as well set $c \leftarrow 0$ instead, because $f(x)$ will be zero in all cases when $b > 30$. Note that we have only about 25 bits of information in all, so it should fit in 32 bits with ease.

⟨ Types in the outer block 6 ⟩ ≡
glue_ratio = **packed record** *a_part*: 1 . . 31; { the quantity $e = a + 16$ in our derivation }
b_part: 0 . . 30; { the quantity b in our derivation }
c_part: 0 . . *´100000*; { the quantity c in our derivation }
end;
scaled = *integer*; { this data type is used for quantities in sp units }

This code is used in section 2.

7. The real problem is to define the procedures that TeX needs to deal with such *glue_ratio* values: (a) Given scaled numbers s, t, and y as above, to compute the corresponding *glue_ratio*. (b) Given a nonnegative scaled number x and a *glue_ratio* g, to compute the scaled number $f(x)$. (c) Given a *glue_ratio* g, to print out a decimal equivalent of g for diagnostic purposes.

The procedures below can be incorporated into TeX82 via a change file without great difficulty. A few modifications will be needed, because TeX's *glue_ratio* values can be negative in unusual cases—when the amount of stretchability or shrinkability is less than zero. Negative values in the *c_part* will handle such problems, if proper care is taken. The error message below should either become a warning message or a call to TeX's *print_err* routine; in the latter case, an appropriate help message should be given, stating that glue cannot stretch to more than 18 feet long, but that it's OK to proceed with fingers crossed.

8. Glue multiplication. The easiest procedure of the three just mentioned is the one that is needed most often, namely, the computation of $f(x)$.

Pascal doesn't have built-in binary shift commands or built-in exponentiation, although many computers do have this capability. Therefore our arithmetic routines use an array called '*two_to_the*', containing powers of two. Divisions by powers of two are never done in the programs below when the dividend is negative, so the operations can safely be replaced by right shifts on machines for which this is most appropriate. (Contrary to popular opinion, the operation 'x **div** 2' is not the same as shifting x right one binary place, on a machine with two's complement arithmetic, when x is a negative odd integer. But division *is* equivalent to shifting when x is nonnegative.)

⟨ Globals in the outer block 8 ⟩ ≡
two_to_the: **array** [0 .. 30] **of** *integer*; { *two_to_the*[k] = 2^k }

See also sections 15 and 20.

This code is used in section 2.

9. ⟨ Local variables for initialization 9 ⟩ ≡
k: 1 .. 30; { an index for initializing *two_to_the* }

This code is used in section 2.

10. ⟨ Set initial values 10 ⟩ ≡
two_to_the[0] ← 1;
for k ← 1 **to** 30 **do** *two_to_the*[k] ← *two_to_the*[$k - 1$] + *two_to_the*[$k - 1$];

This code is used in section 2.

11. We will use the abbreviations *ga*, *gb*, and *gc* as convenient alternatives to Pascal's **with** statement. The glue-multiplication function f, which replaces several occurrences of the '*float*' macro in TeX82, is now easy to state:

define *ga* ≡ *g.a_part*
define *gb* ≡ *g.b_part*
define *gc* ≡ *g.c_part*

function *glue_mult*(x : *scaled*; g : *glue_ratio*): *integer*; { returns $f(x)$ as above, assuming that $x \geq 0$ }
begin if *ga* > 16 **then** x ← x **div** *two_to_the*[*ga* − 16] { right shift by a places }
else x ← x ∗ *two_to_the*[16 − *ga*]; { left shift by $-a$ places }
glue_mult ← (x ∗ *gc*) **div** *two_to_the*[*gb*]; { right shift by b places }
end; { note that b may be as large as 30 }

12. Glue setting. The *glue_fix* procedure computes a, b, and c by the method explained above. TEX does not normally compute the quantity y, but it could be made to do so without great difficulty.

This procedure replaces several occurrences of the '*unfloat*' macro in TEX82. It would be written as a function that returns a *glue_ratio*, if Pascal would allow functions to produce records as values.

```
procedure glue_fix(s, t, y : scaled; var g : glue_ratio);
  var a, b, c: integer;  { components of the desired ratio }
    k, h: integer;  { 30 − ⌊lg s⌋, 30 − ⌊lg t⌋ }
    s0: integer;  { original (unnormalized) value of s }
    q, r, s1: integer;  { quotient, remainder, divisor }
    w: integer;  { 2^l, where l = 16 − k }
  begin ⟨Normalize s, t, and y, computing a, k, and h 13⟩;
  if t < s then b ← 15 − a − k + h else b ← 14 − a − k + h;
  if (b < 0) ∨ (b > 30) then
    begin if b < 0 then write_ln(´!␣Excessive␣glue.´);  { error message }
    b ← 0; c ← 0;  { make f(x) identically zero }
    end
  else begin if k ≥ 16 then  { easy case, s_0 < 2^15 }
      c ← (t div two_to_the[h − a − b] + s0 − 1) div s0  { here 1 ≤ h − a − b ≤ k − 14 ≤ 16 }
    else ⟨Compute c by long division 14⟩;
    end;
  ga ← a + 16; gb ← b; gc ← c;
  end;
```

13. ⟨Normalize s, t, and y, computing a, k, and h 13⟩ ≡

```
  begin a ← 15; k ← 0; h ← 0; s0 ← s;
  while y < ´10000000000 do  { y is known to be positive }
    begin decr(a); y ← y + y;
    end;
  while s < ´10000000000 do  { s is known to be positive }
    begin incr(k); s ← s + s;
    end;
  while t < ´10000000000 do  { t is known to be positive }
    begin incr(h); t ← t + t;
    end;
  end  { now 2^30 ≤ t = 2^h t_0 < 2^31 and 2^30 ≤ s = 2^k s_0 < 2^31, hence d = k − h if t/s < 1 }
```

This code is used in section 12.

14. ⟨Compute c by long division 14⟩ ≡

```
  begin w ← two_to_the[16 − k]; s1 ← s0 div w; q ← t div s1; r ← ((t mod s1) ∗ w) − ((s0 mod w) ∗ q);
  if r > 0 then
    begin incr(q); r ← r − s0;
    end
  else while r ≤ −s0 do
      begin decr(q); r ← r + s0;
      end;
  if a + b + k − h = 15 then c ← (q + 1) div 2 else c ← (q + 3) div 4;
  end
```

This code is used in section 12.

15. Glue-set printing. The last of the three procedures we need is *print_gr*, which displays a *glue_ratio* in symbolic decimal form. Before constructing such a procedure, we shall consider some simpler routines, copying them from an early draft of the program TEX82.

```
define unity ≡ ´200000   { 2^16, represents 1.0000 }
⟨ Globals in the outer block 8 ⟩ +≡
dig: array [0 .. 15] of 0 .. 9;   { for storing digits }
```

16. An array of digits is printed out by *print_digs*.

```
procedure print_digs(k : integer);   { prints dig[k − 1] . . . dig[0] }
  begin while k > 0 do
    begin decr(k); write(chr(ord(´0´) + dig[k]));
    end;
  end;
```

17. A nonnegative integer is printed out by *print_int*.

```
procedure print_int(n : integer);   { prints an integer in decimal form }
  var k: 0 .. 12;   { index to current digit; we assume that 0 ≤ n < 10^12 }
  begin k ← 0;
  repeat dig[k] ← n mod 10; n ← n div 10; incr(k);
  until n = 0;
  print_digs(k);
  end;
```

18. And here is a procedure to print a nonnegative *scaled* number.

```
procedure print_scaled(s : scaled);   { prints a scaled real, truncated to four digits }
  var k: 0 .. 3;   { index to current digit of the fraction part }
  begin print_int(s div unity);   { print the integer part }
  s ← ((s mod unity) ∗ 10000) div unity;
  for k ← 0 to 3 do
    begin dig[k] ← s mod 10; s ← s div 10;
    end;
  write(´.´); print_digs(4);
  end;
```

19. Now we're ready to print a *glue_ratio*. Since the effective multiplier is $2^{-a-b}c$, we will display the scaled integer $2^{16-a-b}c$, taking care to print something special if this quantity is terribly large.

```
procedure print_gr(g : glue_ratio);   { prints a glue multiplier }
  var j: −29 .. 31;   { the amount to shift c }
  begin j ← 32 − ga − gb;
  while j > 15 do
    begin write(´2x´); decr(j);   { indicate multiples of 2 for BIG cases }
    end;
  if j < 0 then print_scaled(gc div two_to_the[−j])   { shift right }
  else print_scaled(gc ∗ two_to_the[j]);   { shift left }
  end;
```

20. The driver program. In order to test these routines, we will assume that the *input* file contains a sequence of test cases, where each test case consists of the integer numbers t, x_1, …, x_n, 0. The final test case should be followed by an additional zero.

```
⟨ Globals in the outer block 8 ⟩ +≡
x: array [1 .. 1000] of scaled;   { the x_i }
t: scaled;   { the desired total }
m: integer;   { the test case number }
```

21. Each case will be processed by the following routine, which assumes that t has already been read.

```
procedure test;   { processes the next data set, given t and m }
  var n: 0 .. 1000;   { the number of items }
    k: 0 .. 1000;   { runs through the items }
    y: scaled;   { max_{1≤i≤n} |x_i| }
    g: glue_ratio;   { the computed glue multiplier }
    s: scaled;   { the sum x_1 + ··· + x_n }
    ts: scaled;   { the sum f(x_1) + ··· + f(x_n) }
  begin write_ln('Test␣data␣set␣number␣', m : 1, ':'); ⟨ Read x_1, ..., x_n 22 ⟩;
  ⟨ Compute s and y 23 ⟩;
  if s ≤ 0 then write_ln('Invalid␣data␣(nonpositive␣sum);␣this␣set␣rejected.')
  else begin ⟨ Compute g and print it 24 ⟩;
    ⟨ Print the values of x_i, f(x_i), and the totals 25 ⟩;
    end;
  end;
```

22.

```
⟨ Read x_1, ..., x_n 22 ⟩ ≡
  begin n ← 0;
  repeat incr(n); read(x[n]);
  until x[n] = 0;
  decr(n);
  end
```

This code is used in section 21.

23.

```
⟨ Compute s and y 23 ⟩ ≡
  begin s ← 0; y ← 0;
  for k ← 1 to n do
    begin s ← s + x[k];
    if y < abs(x[k]) then y ← abs(x[k]);
    end;
  end
```

This code is used in section 21.

24.

```
⟨ Compute g and print it 24 ⟩ ≡
  begin glue_fix(s, t, y, g);   { set g, perhaps print an error message }
  write('␣␣Glue␣ratio␣is␣'); print_gr(g); write_ln('␣(', ga − 16 : 1, ',', gb : 1, ',', gc : 1, ')');
  end
```

This code is used in section 21.

25. ⟨ Print the values of x_i, $f(x_i)$, and the totals 25 ⟩ ≡

begin *ts* ← 0;
for k ← 1 **to** n **do**
  begin *write*($x[k]$: 20);
  if $x[k] \ge 0$ **then** y ← *glue_mult*($x[k], g$)
  else y ← −*glue_mult*($-x[k], g$);
  write_ln(y : 15); *ts* ← *ts* + y;
  end;
write_ln(´␣Totals´, s : 13, *ts* : 15, ´␣(versus␣´, t : 1, ´)´);
end

This code is used in section 21.

26. Here is the main program.

begin *initialize*; m ← 1; *read*(t);
while $t > 0$ **do**
  begin *test*; *incr*(m); *read*(t);
  end;
end.

27. Index. Here are the section numbers where various identifiers are used in the program, and where various topics are discussed.