Bresenham's Line Algorithm Implemented for the NS32GX32

National Semiconductor
Application Note 611
Rick Pelleg
September 1989

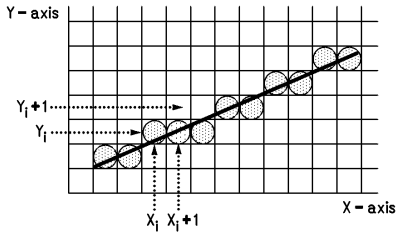
1.0 INTRODUCTION

Even with today's achievements in graphics technology, the resolution of computer graphics systems will never reach that of the real world. A true real line can never be drawn on a laser printer or CRT screen. There is no method of accurately printing all of the points on the continuous line described by the equation $y = mx + b$. Similarly, circles, ellipses and other geometrical shapes cannot truly be implemented by their theoretical definitions because the graphics system itself is discrete, not real or continuous. For that reason, there has been a tremendous amount of research and development in the area of discrete or raster mathematics. Many algorithms have been developed which ''map'' real-world images into the discrete space of as raster device. Bresenham's line-drawing algorithm (and its derivatives) is one of the most commonly used algorithms today for describing a line on a raster device. The agorithm was first published in Bresenham's 1965 article entitled ''Algorithm for Computer Control of a Digital Plotter''. It is now widely used in graphics and electronic printing systems.

This application note describes the fundamental algorithm and shows an implementation specially tuned for the NS32GX32 microprocessor. Although given in the context of this specific application note, the assembly level optimizations are relevant to general programming for the NS32GX32. Timing figures are given in Appendix C.

2.0 DESCRIPTION

Bresenham's line-drawing algorithm uses an iterative scheme. A pixel is plotted at the starting coordinate of the line, and each iteration of the algorithm increments the pixel one unit along the major, or x-axis. The pixel is incremented along the minor, or y-axis, only when a decision variable (based on the slope of the line) changes sign. A key feature of the algorithm is that it requires only integer data and simple arithmetic. This makes the algorithm very efficient and fast.



TL/EE/10434–1

FIGURE 1

The algorithm assumes the line has positive slope less than one, but a simple change of variables can modify the algorithm for any slope value. This will be detailed in Section 2.2.

2.1 Bresenham's Algorithm for 0 < slope < 1

Figure 1 shows a line segment superimposed on a raster grid with horizontal axis X and vertical axis Y. Note that x_i and y_i are the integer abscissa and ordinate respectively of each pixel location on the grid.

Given (x_i, y_i) as the previously plotted pixel location for the line segment, the next pixel to be plotted is either $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$. Bresenham's algorithm determines which of these two pixel locations is nearer to the actual line by calculating the distance from each pixel to the line, and plotting that pixel with the smaller distance. Using the familiar equation of a straight line, $y = mx + b$, the y value corresponding to $x_i + 1$ is

$$y = m(x_i + 1) + b$$

The two distances are then calculated as:

$$d1 = y - y_i$$

$$d1 = m(x_i + 1) + b - y_i$$

$$d2 = (y_i + 1) - y$$

$$d2 = (y_i + 1) - m(x_i + 1) - b$$

and,

$$d1 - d2 = m(x_i + 1) + b - y_i - (y_i + 1) + m(x_i + 1) + b$$

$$d1 - d2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

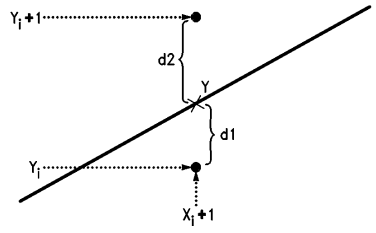
Multiplying this result by the constant dx, defined by the slope of the line $m = dy/dx$, the equation becomes:

$$dx(d1 - d2) = 2dy(x_i) - 2dx(y_i) + c$$

where c is the constant $2dy + 2dxb - dx$. Of course, if $d2 > d1$, then $(d1 - d2) < 0$, or conversely if $d1 > d2$, then $(d1 - d2) > 0$. Therefore, a parameter p_i can be defined such that

$$p_i = dx(d1 - d2)$$

$$p_i = 2dy(x_i) - 2dx(y_i) + c$$



TL/EE/10434–2

Distances d1 and d2 are compared.

The smaller distance marks next pixel to be plotted.

FIGURE 2

RRD-B30M75/Printed in U. S. A.

If $p_i > 0$, then d1 > d2 and y_{i+1} is chosen such that the next plotted pixel is $(x_i + 1, y_i)$. Otherwise, if $p_i < 0$, then d2 > d1 and $(x_i + 1, y_i + 1)$ is plotted. (See *Figure 2*.)

Similarly, for the next iteration, p_{i+1} can be calculated and compared with zero to determine the next pixel to plot. If $p_{i+1} < 0$, then the next plotted pixel is at $(x_{i+1} + 1, y_{i+1})$; if $p_{i+1} > 0$, then the next point is $(x_{i+1} + 1, y_{i+1} + 1)$. Note that in the equation for p_{i+1}, $x_{i+1} = x_i + 1$.

$$p_{i+1} = 2dy(x_i + 1) - 2dx(y_{i+1}) + c$$

Subtracting p_i from p_{i+1}, we get the recursive equation:

$$p_{i+1} = p_i + 2dy - 2dx(y_{i+1} - y_i)$$

Note that the constant c has conveniently dropped out of the formula. And, if $p_i < 0$ then $y_{i+1} = y_i$ in the above equation, so that:

$$p_{i+1} = p_i + 2dy$$

or, if $p_i > 0$ then $y_{i+1} = y_i + 1$, and

$$p_{i+1} = p_i + 2(dy - dx)$$

To further simplify the iterative algorithm, constants c1 and c2 can be initialized at the beginning of the program such that c1 = 2dy and c2 = 2(dy − dx). Thus, the actual meat of the algorithm is a loop of length dx, containing only a few integer additions and two compares *(Figure 3)*.

2.2 For Slope < 0 and |Slope| > 1

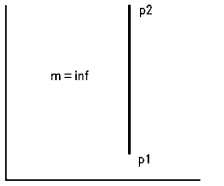
The algorithm fails when the slope is negative or has absolute value greater than one (|dy| > |dx|). The reason for this is that the line will always be plotted with a positive slope if x_i and y_i are always incremented in the positive direction, and the line will always be ''shorted'' if |dx| < |dy| since the algorithm executes once for every x coordinate (i.e., dx times). However, a closer look at the algorithm must be taken to reveal that a few simple changes of variables will take care of these special cases.

For negative slopes, the change is simple. Instead of incrementing the pixel along the positive direction (÷1) for each iteration, the pixel is incremented in the negative direction. The relationship between the starting point and the finishing point of the line determines which axis is followed in the negative direction, and which is in the positive. *Figure 4* shows all the possible combinations for slopes and starting points, and their respective incremental directions along the X and Y axis.

Another change of variables can be performed on the incremental values to accommodate those lines with slopes greater than 1 or less than −1. The coordinate system containing the line is rotated 90 degrees so that the X-axis now becomes the Y-axis and vice versa. The algorithm is then performed on the rotated line according to the sign of its slope, as explained above. Whenever the current position is incremented along the X-axis in the rotated space, it is actually incremented along the Y-axis in the original coordinate space. Similarly, an increment along the Y-axis in the rotated space translates to an increment along the X-axis in the original space. *Figures 4a., g,* and *h.* illustrate this translation process for both positive and negative lines with various starting points.

```
do while count <> dx
   if (p < 0) then p+ = cl
   else
                    p+ = c2
                    next_y = prev_y + y_inc
   next_x = prev_x + x_inc
   plot(next_x,next_y)
   count += l
/* PSEUDO CODE FOR BRESENHAM LOOP */
```

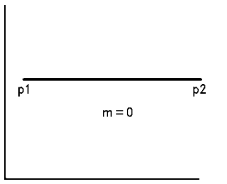
FIGURE 3



TL/EE/10434–3

start p1: x__inc = y′__inc = 0
y__inc = x′__inc = +1
start p2: x__inc = y′__inc = 0
y__inc = x′__inc = −1

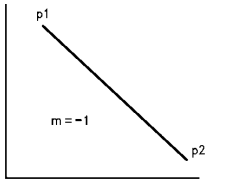
a.



TL/EE/10434–4

start p1: x__inc = +1
y__inc = 0
start p2: x__inc = −1
y__inc = 0

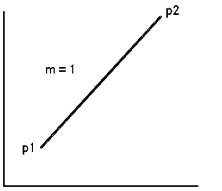
b.



TL/EE/10434–5

start p1: x__inc = +1
y__inc = −1
start p2: x__inc = −1
y__inc = +1

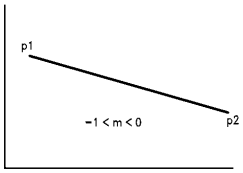
c.



TL/EE/10434–6

start p1: x__inc = +1
y__inc = +1
start p2: x__inc = −1
y__inc = −1

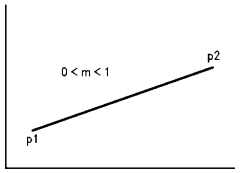
d.



TL/EE/10434–7

start p1: x__inc = +1
y__inc = −1
start p2: x__inc = −1
y__inc = +1

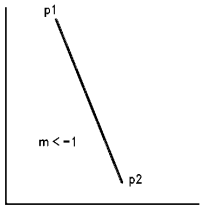
e.



TL/EE/10434–8

start p1: x__inc = −1
y__inc = −1
start p2: x__inc = −1
y__inc = −1

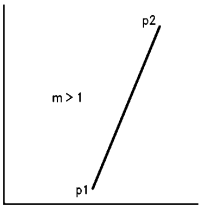
f.



TL/EE/10434–9

start p1: x__inc = y′__inc = −1
y__inc = x′__inc = −1
start p2: x__inc = y′__inc = −1
y__inc = x′__inc = −1

g.



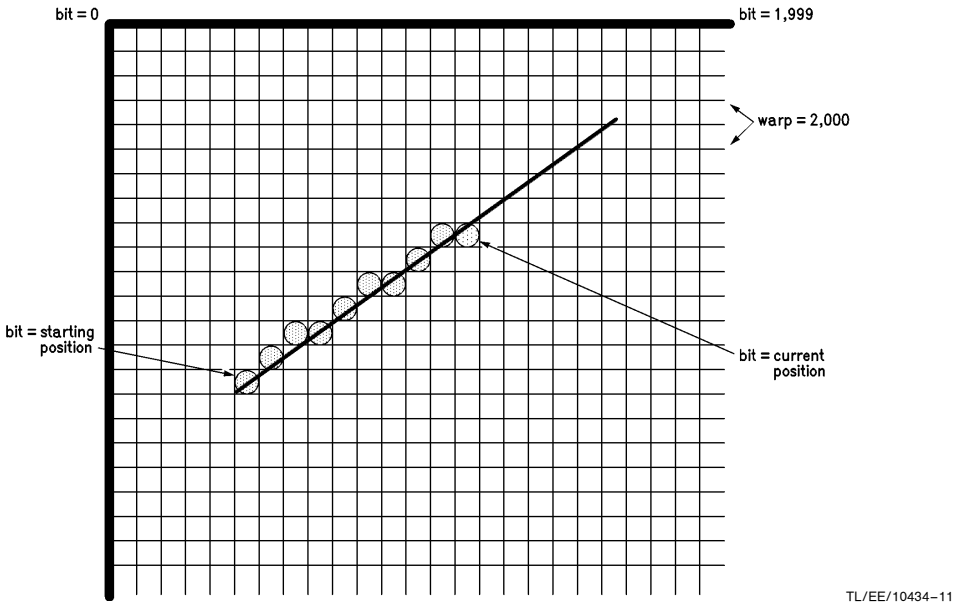
TL/EE/10434–10

start p1: x__inc = y′__incl = −1
y__inc = x′__inc = +1
start p2: x__inc = y′__inc = +1
y__inc = x′__inc = −1

h.

Note: a., g., and h. are rotated 90 degrees left and x′, y′ refer to the original axis.

FIGURE 4



Bit Map is 500,000 bytes, 2000 x 2000 Bits
Base Address of the Bit Map is ''__bit__map''

FIGURE 5

3.0 IMPLEMENTATION IN C

Bresenham's algorithm is easily implemented in most programming languages. Appendix A gives a C routine implementing the algorithm.

The routine accepts as parameters the line's start and end coordinates, (xs, ys) and (xf, yf). It plots the line on a bit-map allocated in memory. The dimensions of the bit map are given in the include file ''bres.h''.

The program uses the variable *bit* to keep track of the current pixel position within the 2000 x 2000 bit map *(Figure 5)*.

Note the macro definition for setting a bit in memory:

```
#define sbit(buffer, pos) (buffer)[(pos)>>3] |= 1<<
                          (((char)pos) & 7)
```

Bit ''pos'' is set by calculating pos MOD 8, which is the same as (pos & 7). Then 1 is shifted by this amount to obtain a mask of the bit to set in byte pos/8 of ''buffer'', which is the same as (pos >> 3).

4.0 IMPLEMENTATION IN SERIES 32000® ASSEMBLY, SPECIALLY TUNED FOR THE NS32GX32

This section demonstrates several kinds of assembly level optimizations for speeding up NS32GX32 programs. These take into account the execution times of different instructions, data dependency between registers, the NS32GX32 on-chip cache, and the NS32GX32 branch prediction method.

4.1 Instruction Execution Time

The NS32GX32 is fully binary compatible to its predecessors from the Series 32000 family of microprocessors. The NS32GX32's pipelined architecture and high frequency internal clock enable programs written for the other members of the processor family to run faster, in general, on the NS32GX32.

However, one of the characteristics of the NS32GX32 is that although the average throughput of its pipeline is 3.5 clock cycles per instruction, there are several instructions whose execution time is much longer than this. Further improvement in execution time may be achieved by avoiding these relatively ''expensive'' instructions.

We will demonstrate here the replacement of the relatively expensive instructions ''sbit'' and ''mul''.

4.1.1 Replacing the ''SBIT'' Instruction

The most straightforward coding of the Bresenham algorithm would use the NS32000 ''sbit'' instruction for setting the current bit of the line (see AN-524: ''Introduction to Bresenham's Line Algorithm using the SBIT Instruction''). ''sbit'', however, takes about 18 cycles to execute. Because the bit setting is a significant part of the routine's main loop, it is important to optimize it.

In our assembly routine, we replace the ''sbitd r1,__bit__map'' instruction with the following sequence, which saves an average of 3.5 cycles per bit set, or about 8% of the main loop time. This sequence is essentially an implementation of the calculation defined in the ''sbit'' macro used in the C language routine.

```
movd   r1,r4
andd   $(7),r4    # r4 = bit mod 8 = which
                  bit to set
movqb  $(1),r6
lshb   r4,r6      # r6 = 1 << (bit mod 8) =
                  mask with set bit
movd   r1,r4
lshd   $(-3),r4   # r4 = bit div 8 = byte
                  where bit is set
orb    r6,_bit_map(r4)
```

Note the use of the ''lsh'' instruction, rather than the ''ash'' one. The former requires 3 cycles for execution, compared to 9 for the latter. Both instructions are equivalent in our case, as r1 (the bit to be set) is an unsigned quantity less than 4,000,000.

4.1.2 Replacing the ''MUL'' Instruction

The setup calculations done before the main loop of the algorithm include two multiplications by the x-dimension of the bit-map.

The ''mul'' instruction takes 37 cycles to multiply by numbers in the order of magnitude of a reasonable bit-map dimension. For any specific application it is usually possible to replace the ''mul'' by one or a few faster instructions. If the dimension is a power of two, one ''lsh'' instruction, executing in 3 cycles, is enough.

In other cases the dimension can be factored into a sum/difference of powers of two. The multiplication is then replaced by a series of shifts and adds. An example of multiplication by 2000 is demonstrated in Appendix D.

4.2 Avoiding Register Interlocks

In certain circumstances the flow of instructions in the NS32GX32 pipeline will be delayed when the result of an instruction is used as the source of the following instruction. One of these interlocks occurs in Section 4.1.1 — the instruction ''orb r6,__bit__map(r4)'' immediately follows the calculation of an r4 in ''lshd $(−3), r4''. To avoid this interlock, we can exchange the order of calculating the byte and the bit, as follows:

```
movd  rl,r0
lshd  $(-3),r0  # r0 = bit div 8 = byte
                where bit is set
movd  rl,r4
andd  $(7),r4   # r4 = bit mod 8 = which
                bit to set
movqb r4,r6
lshb  r4,r6     # r6 = 1 << (bit mod 8) =
                mask with set bit
orb   r6,_bit_map(r0)
```

Moving the calculation of the byte away from its use in the ''orb'' instruction saves about 2.5 cycles per bit, or another 6% of the main loop time. Added to the optimization of 4.1.1, this gives a potential improvement of about 14% compared to the straightforward ''sbit'' instruction. This, however, is not the actual improvement we get: The new sequence requires keeping extra registers free in the loop, forcing us to use memory for some of the algorithm variables (see Section 4.3 below). This has an overhead of about 2% of the main loop time, giving a net improvement of about 12%.

4.3 Data Cache Considerations

The main loop of the Bresenham algorithm uses seven variables: ''c1'' and ''c2'' the loop-constants, ''p'' the decision variable, ''x-increment'' and ''y-increment'', ''bit'' and ''last-bit''. If the ''sbit'' instruction is used, all of these can reside in registers, as the NS32GX32 has eight general-purpose registers.

As mentioned above, the replacement of ''sbit r1,__bit__map'' with the above sequence has the cost of requiring three intermediate temporary values. These values, with the addition of the algorithm's seven loop-used variables, give us the problem of deciding what should be put in registers and what in memory. This decision must be strongly influenced by the NS32GX32 data cache, and its write-through update policy.

The most important consideration is that due to the write-through policy, a write to memory is more expensive than a write to a register. Thus, the loop-invariant variables (''c1'', ''c2'', ''last-bit'', ''x-increment'' and ''y-increment'') are better candidates for allocation in memory rather than in a register. The additional temporary values for the ''sbit'' alternative sequence are written, so registers should be allocated to them. Reading one of the variables from a memory location may also result in a data cache miss, because the ''__bit__map'' references may overwrite them. Thus the final choice for memory instead of registers was for ''c2'' and ''x-increment''. ''last__bit'' was not chosen because it is read in every iteration of the loop. ''c1'' and ''x-increment'' are never both read in any of the branches inside the loop, so for each bit set we always have only one read from memory.

4.4 Optimizing Branch Instructions

An additional improvement of 2.4% is achieved in Appendix B's assembly routine by optimizing the flow of branches in the main loop. This is a relatively complicated issue, so its details are given in Appendix D.

4.5 Loop Unrolling

Another method to speed up the main loop of the algorithm is to reduce the overhead of branches in this loop. The idea is to replicate the code in the loop, so that in each iteration two bits will be set, without any conditional branch between them. To ensure that there is an even number of bits to plot when entering the loop, we must add another test after setting the first bit before the loop, and perhaps plot another bit. This slightly lengthens the pre-loop execution time, but is worth doing for the reduction in the loop time.

The general outline of the routine, after the initial calculations, becomes:

1. set first bit
2. if an odd number of bits remains, plot an extra bit.
3. LOOP:
   a. plot bit
   b. plot bit
   c. if not last bit, goto LOOP

Without the code replication, there is a test to check if it is the last bit for each bit plotted. The replication saves one such test and the delay associated with its conditional branch.

The actual code (reverting to the simple ''sbit'' code) is given in *Figure 6.*

In Appendix B, an additional code replication is done, to save the ''br next__bit'' instruction in the code above. The total improvement from the code replication is an additional 5% over the code that sets one bit per iteration.

The register and memory usage in this example are:

```
        +---------------------------------+
        | r1     - current bit to be set  |
        | r3     - last bit to be set     |
        | 32(sp) - c1                     |
        | r2     - c2                     |
        | r7     - p                      |
        | r5     - x-increment            |
        | 24(sp) - y-increment            |
        | 36(sp) - max(dx, dy)            |
        +---------------------------------+

#
# PLOTTING OF THE LINE
#
set_first_bit:

        sbitd   r1,_bit_map     # plot first point by setting bit in bit_map
        cmpd    r1,r3           # is the first bit also the last bit?
        beq     EXIT_SEQUENCE

# Test if no. of bits is odd or even:  if the difference (max(dx,dy))
# is odd, there is a total of an even no. of bits, and vice versa.
# The first bit has been plotted, so we must decide if to plot another
# bit before the loop (which plot two bits in each iteration) or not.

        tbitd   $(0),36(sp)
        bfc     MAIN_LOOP

        .align 4

#
# EXTRA BIT SET FOR EVEN NO. OF BITS
#

PRE_LOOP:
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative

#   if p < 0 then
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map

        cmpd    r1,r3           # have we just plot the last bit?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE

        .align 4
p_negative:
```

TL/EE/10434–13

FIGURE 6

```
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map

        cmpd    r1,r3           # have we just plot the last bit?
        beq     EXIT_SEQUENCE


# ===================  M A I N    L O O P  ======================

MAIN_LOOP:
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative1

#   if p < 0 then
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map

        br      next_bit        # No need to test if last bit

        .align 4
p_negative1:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map
#
#   unrolled repetition
#

next_bit:
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative2
#
#   if p < 0 then
#
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map
        cmpd    r1,r3           # have we just plotted the last bit?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE

        .align 4
p_negative2:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        sbitd   r1,_bit_map     # plot by setting bit in bit_map
        cmpd    r1,r3           # have we just plotted the last bit?
        bne     MAIN_LOOP

# ======================  END OF LINE PLOTTING  ================================
EXIT_SEQUENCE:
```

TL/EE/10434–14

FIGURE 6 (Continued)

5.0 CONCLUSION

An optimized Bresenham line-drawing algorithm has been presented for the NS32GX32 microprocessor. The optimizations used are relevant to general coding for the NS32GX32. Appendix C gives timing results for the implementations given in Appendix A and Appendix B.

Several variations of the Bresenham algorithm have been developed. One particular variation by Bresenham himself relies on ''run-length'' segments of the line for speed optimization. This is explored in Application Note AN-522.

APPENDIX A. IMPLEMENTATION IN C

```
/*
 * File "bres.h"
 *               global definitions and declarations for the
 *               program using Bresenham line drawing algorithm.
 */

#define dimension 2000          /* must be a multiple of 8 */
#define xbytes  (dimension/8)   /* number of bytes along x_axis */
#define warp dimension          /* number of bits along x_axis */
#define maxx  (warp)-1          /* highest x coordinate */
#define maxy  (dimension-1)     /* highest y coordinate */
#define y_lines (maxy+1)        /* number of lines along y axis */

unsigned char bit_map[y_lines*xbytes];  /* array containing bit-map */

/* end of "bres.h"  */

/*
 * File "bres_line.c"
 *                    Implementation of Bresenham's line drawing algorithm
 *                    in the C programming language.
 */

#include "bres.h"

/*-------------------------------------------------------------------------
 *      sbit(pos, buffer)
 *      unsigned int pos;
 *      char buffer[];
 * bit 'pos' is set by calculating pos MOD 8, which is the same as (pos & 7).
 * Then 1 is shifted by this amount to obtain a mask of the bit to set in
 * byte pos/8 of 'buffer', which is the same as (pos >> 3).
 *-------------------------------------------------------------------------
 */

#define sbit(buffer, pos)   (buffer)[(pos)>>3] |= 1 << (((char)pos) & 7)

/*-------------------------------------------------------------------------
 *      line_draw(xs, ys, xf, yf)
 *      int xs, ys, xf, yf;
 * Draws a line on "bit_map", from coordinates (xs, ys) to (xf, yf), using
 * Bresenham's iterative method.
 *-------------------------------------------------------------------------
 */

line_draw(xs, ys, xf, yf)
int xs, ys, xf, yf;
{
    int dx, dy, x_inc, y_inc,           /* deltas and increments */
        p, c1, c2;                      /* decision variable p and constants */
    unsigned bit, last_bit;             /* current and last bit positions */

    dx = xf - xs;
    dy = yf - ys;

    bit = ys * warp + xs;               /* initialize bit to first bit pos */
```

TL/EE/10434–15

```
    last_bit = yf * warp + xf;          /* calculate last bit on line */

    if (abs(dy) > abs(dx)) {            /* abs(slope) > 1 must rotate space. */

        if (dy > 0)
            x_inc = warp;               /* x-axis is now original y-axis */
        else
            x_inc = -warp;

        if (dx > 0)
            y_inc = 1;                  /* y-axis is now original x-axis */
        else
            y_inc = -1;

        /* Calculate Bresenham's constants: */
        c1 = 2 * abs(dx);
        c2 = 2 * (abs(dx) - abs(dy));
        p = 2 * abs(dx) - abs(dy);      /* p is decision variable now rotated *
    }
    else {                              /* abs(slope) < 1 - use original axis *
        if (dy > 0)
            y_inc = warp;               /* y_inc is +/-warp number of bits */
        else
            y_inc = -warp;

        if (dx > 0)
            x_inc = 1;
        else
            x_inc = -1;

        /* Calculate Bresenham's constants: */
        c1 = 2 * abs(dy);
        c2 = 2 * (abs(dy) - abs(dx));
        p = 2 * abs(dy) - abs(dx);
    }

    /*============   Bresenham's Algorithm ==============*/

    sbit(bit_map, bit);                 /* draw the first point */

    while (bit != last_bit) {           /* once for each increment, */
                                        /* i.e., dx times */
        if (p < 0)                      /* no y movement if p < 0 */
            p += c1;
        else {
            p += c2;
            bit += y_inc;
        }
        bit += x_inc;                   /* always increment x */
        sbit(bit_map, (int)bit);
    };
} /* end line_draw() */

/* end of "bres_line.c"  */
```

TL/EE/10434–16

APPENDIX B. IMPLEMENTATION IN ASSEMBLY LANGUAGE

```
#                -- line_draw.s --
# _line_draw routine draws a line in a memory bit-map with dimensions
# DIM x DIM, from coordinates (xs,ys) to (xf,yf).
# Both coordinates are assumed to be inside the bit-map.

# +-----------------------------+       +-------------+
# |     Register and memory     |       |  Parameters |
# |          Use in Loop        |       |             |
# |                             |       | 24(sp) = xs |
# |  32(sp) = c1 constant       |       | 28(sp) = ys |
# |  r2 = c2 constant           |       | 32(sp) = xf |
# |  r7 = p decision variable   |       | 36(sp) = yf |
# |  r1 = current bit position  |       +-------------+
# |  r3 = last bit to be drawn  |
# |  r5 = x-increment           |
# |  24(sp) = y-increment       |
# |  r0 = byte with set bit     |
# |  r4 = bit set in byte       |
# +-----------------------------+

        .file   "line_draw.s"
        .set    DIM, 2000          # change the 500,000 below if you change this
        .comm   _bit_map, 500000   # 2000x2000 = 4,000,000 bits = 500,000 bytes
        .globl  _line_draw         # "export" _line_draw to the world

        .align 4
_line_draw:

# The following sequence replaces 'save [r3,r4,...,r7]'.
# Together with the replacement for 'restore' at the EXIT_SEQUENCE, we get
# a saving of 28 cycles per each call of the routine.
        movd    r3,tos
        movd    r4,tos
        movd    r5,tos
        movd    r6,tos
        movd    r7,tos             # end of 'enter' sequence

        movd    24(sp),r7          # load xs
        movd    28(sp),r2          # load ys
        movd    32(sp),r5          # load xf
        movd    36(sp),r0          # load yf

#
# Calculate first bit to plot = r1 = xs + DIM*ys
#
        movd    r2,r1
        muld    $(DIM),r1
        addd    r7,r1              # first bit to plot = r1 = xs + DIM*ys
#
# Calculate last bit to plot = DIM*yf + xf
#
        movd    r0,r3
        muld    $(DIM),r3
        addd    r5,r3              # r3 = last_bit = DIM*yf + xf

        subd    r2,r0              # r0 = dy = yf - ys
```

TL/EE/10434–17

```
        absd    r0,r6           # r6 = abs(dy)

        movd    r5,r4
        subd    r7,r4           # r4 = dx = xf - xs
        absd    r4,r5           # r5 = abs(dx)

#
# Setting the values for the LOOP:
#
        cmpd    r6,r5           # is  |dy| > |dx|?
        ble     abs_y_less

#
# |dy| > |dx| --- slope > 1; rotate coordinate system
        movd    r6,36(sp)       # save dy, to test even/odd no. bits
#
# calculate the loop constants c1, c2, p
#
        addd    r5,r5           # c1 = 2*|dx| Note: r5 is temporary for c1.
        movd    r5,r2           # r2 = c1
        subd    r6,r2           # r2 = c1-|dy|
        subd    r6,r2           # r2 = c1-2*|dy| = 2*|dx|-2*|dy| = c2

        movd    r5,r7           # r7 = 2*|dx|
        subd    r6,r7           # p = r7 = 2 * |dx| - |dy|
        movd    r5,32(sp)       # 32(sp) is now c1
#
# calculate x-increment, y-increment
#
        cmpqd   $(0),r0         # is  dy > 0?
        bge     negative_dy
        movd    $(DIM),r5       # x-increment = DIM
        br      get_y_increment
        .align 4
negative_dy:
        movd    $(-DIM),r5      # x-increment = -DIM
get_y_increment:
        cmpqd   $(0),r4         # is  dx > 0?
        bge     negative_dx
        movqd   $(1),24(sp)     # y-increment = 1
        br      set_first_bit
        .align 4
negative_dx:
        movqd   $(-1),24(sp)    # y-increment = -1
        br      set_first_bit


        .align 4
#
# |dx| >= |dy|, e.g., slope <= 1; normal coordinate system
#
abs_y_less:
        movd    r5,36(sp)       # save dx, to test even/odd no. bits
```

TL/EE/10434–18

```
#
# calculate the loop constants c1, c2, p
#
        addd    r6,r6           # r6 = c1 = 2*|dy|

        movd    r6,r2           # r2 = 2*|dy|
        subd    r5,r2           # r2 = r2 - |dx| = 2*|dy| - |dx|
        subd    r5,r2           # c2 = 2*|dy| - 2*|dx|

        movd    r6,r7           # r7 = 2*|dy|
        subd    r5,r7           # p = r7 = 2*|dy| - |dx|
        movd    r6,32(sp)       # save c1, free r6 for other use
#
# calculate x-increment, y-increment
#
        cmpqd   $(0),r4         # is  dx > 0?
        bge     dx_negative
        movqd   $(1),r5         # x-increment = 1
        br      y_increment_get
        .align 4
dx_negative:
        movqd   $(-1),r5        # x-increment = -1

y_increment_get:
        cmpqd   $(0),r0         # is  dy > 0?
        bge     dy_negative
        movd    $(DIM),24(sp)   # y-increment = DIM
        br      set_first_bit
dy_negative:
        movd    $(-DIM),24(sp)  # y-increment = -DIM
        .align 4

#
# PLOTTING OF THE LINE
#
set_first_bit:
        # sbitd r1,_bit_map     # plot first point by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8 = byte where bit is set
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8 = which bit to set
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8) = mask with set bit
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # is the first bit also the last bit?
        beq     EXIT_SEQUENCE

#
# Test if no. of bits is odd or even:  if the difference (max(dx,dy))
# is odd, there is a total of an even no. of bits, and vice versa.
# The first bit has been plotted, so we must decide if to plot another
# bit before the loop (which plot two bits in each iteration) or not.
#
        tbitd   $(0),36(sp)
        bfc     MAIN_LOOP
```

TL/EE/10434–19

```
        .align 4
#
# EXTRA BIT SET FOR EVEN NO. OF BITS
#

PRE_LOOP:
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative
#
#   if p < 0 then
#
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment

        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # have we just plotted the last bit?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE
        .align 4
p_negative:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # have we just plotted the last bit?
        beq     EXIT_SEQUENCE

#
# ==================  M A I N    L O O P  ================
#
MAIN_LOOP:
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative2
#
# if p < 0 then
#
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment
```

TL/EE/10434–20

```
        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

#
# unrolled repetition1
#
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative1_2
#
# if p < 0 then
#
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment

        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # have we just plotted the last bit?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE

        .align 4
p_negative1_2:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # have we just plotted the last bit?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE

        .align 4
p_negative2:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
```

TL/EE/10434–21

```
        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)
#
# unrolled repetition2
#
        cmpqd   $(0),r7         # is  p < 0 ?
        bgt     p_negative2_2
#
# if p < 0 then
#
        addd    r2,r7           # p = p + c2
        addd    24(sp),r1       # bit = bit + y-increment
        addd    r5,r1           # bit = bit + x-increment

        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # has the last bit been plotted?
        bne     MAIN_LOOP
        br      EXIT_SEQUENCE

        .align 4
p_negative2_2:
        addd    32(sp),r7       # p = p + c1
        addd    r5,r1           # bit = bit + x-increment
        # sbitd r1,_bit_map     # plot by setting bit in bit_map
        movd    r1,r0
        lshd    $(-3),r0        # r0 = bit div 8
        movd    r1,r4
        andd    $(7),r4         # r4 = bit mod 8
        movqb   $(1),r6
        lshb    r4,r6           # r6 = 1 << (bit mod 8)
        orb     r6,_bit_map(r0)

        cmpd    r1,r3           # has the last bit been plotted?
        bne     MAIN_LOOP
#
# ========  END OF MAIN LOOP ====================
#

EXIT_SEQUENCE:

        # replace the 'restore [r3,....,r7]' instruction:
        movd    tos,r7
        movd    tos,r6
        movd    tos,r5
        movd    tos,r4
        movd    tos,r3
        ret     0
```

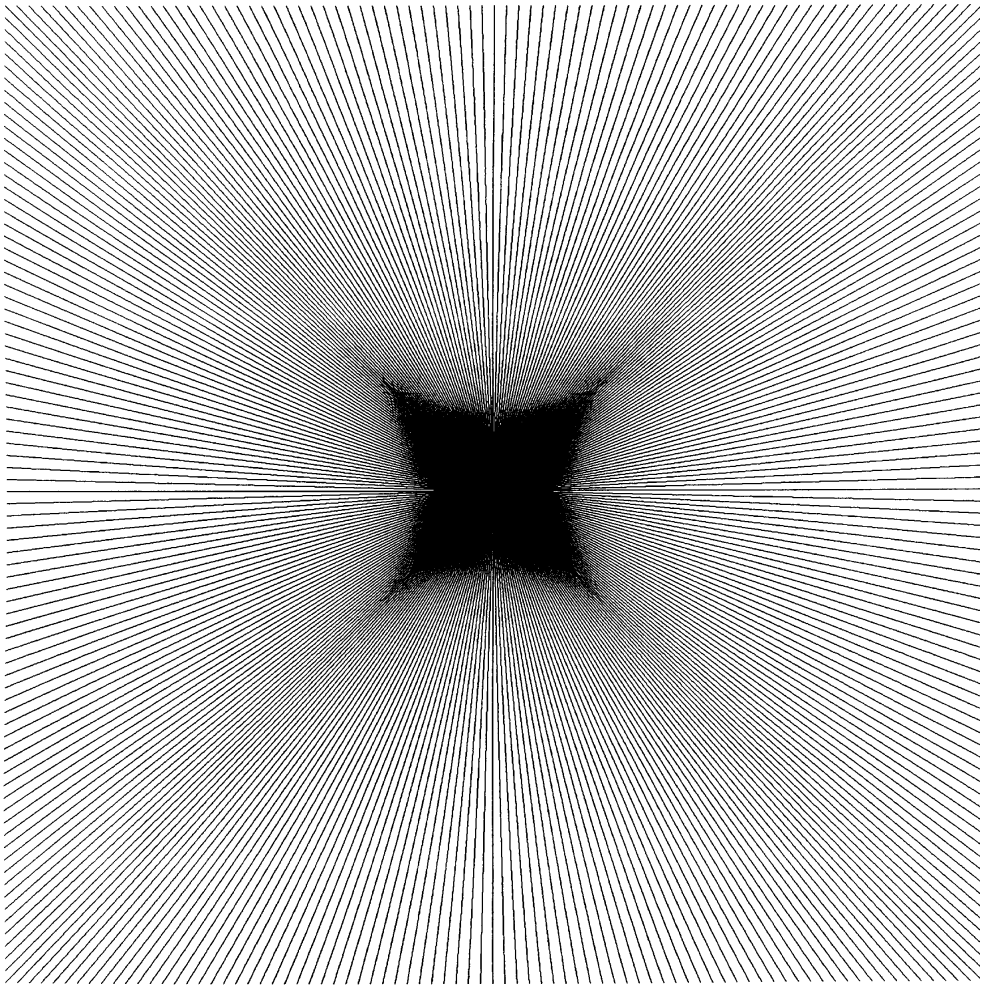
TL/EE/10434–22

APPENDIX C. TIMING PERFORMANCE OF THE NS32GX32 MICROPROCESSOR

Timing was measured on the Star-Burst image of *Figure 7*. The driving ''main'' program in C (given below) was used to call the assembly ''line_draw'' routine. For greater accuracy, time was measured for 100 iterations of the Star-Burst image, and divided by 100 for the results given in *Figure 8*. File ''bres.h'' is in Appendix A.

```
#include "bres.h"

/*
 *      main()
 * Generates the Star-Burst image
 *
 */

main()
{
    int i, count;

    for (count = 1; count <= 100; count++) {
        for (i = 0; i <= maxy; i += 25)
            line_draw(0, i, maxx, maxy-i);
        for (i = 0; i <= maxx; i += 25)
            line_draw(i, maxy, maxx-i, 0);
    }
}

/* end of "main.c"  */
```

TL/EE/10434–23



TL/EE/10434–12

Star-Burst Benchmark—This Star-Burst image was drawn on a 2000 x 2000 pixel bit-map. Each line is 2000 pixels in length and passes through the center of the image, bisecting the square.

The lines are 25 pixels apart, and were drawn using the ''line__draw.s'' routine. There is a total of 160 lines.

The total drawing time for this image was 0.44 seconds on a 25 MHz NS32GX32.

FIGURE 7. Graphics Image (2000 x 2000 Pixels), 300 DPI

Parameter	Time on 25 MHz NS32GX32
Setup Time per Line	4.82 μs
Lines per Second	364.3
Pixels per Second	728,531
Total Time of Star-Burst Image	439.5 ms

FIGURE 8. Timing Performance for the Star-Burst Image of *Figure 7.* The Whole Image Consists of 160 Lines of 2000 Pixels Each.

The setup time was measured from the start of the ''__line__draw'' routine, up to the setting of the first bit. The overhead of the ''main'' routine and of calling the ''__line__draw'' routine are not included.

The number lines-per-second includes only net time in the ''__line__draw'' routine itself, including setup time. It was calculated as follows: The overhead of the ''main'' driving routine, including its call of ''__line__draw'', was subtracted from the time measured for the whole image. The difference was divided by 100 (the amount of iterations of the whole image) and then by 160 (the amount of lines per image). This gives the time for one line. The reciprocal of this time is the number of lines per second drawn.

The numbers of Pixels-per-second is Lines-per-second multiplied by 2000 (the number of pixels-per-line).

The total time for the image includes the overhead of the ''main'' routine.

Note: The total time for the C version of the ''line__draw'' routine (Appendix A), as compiled by the GNX version 3 C optimizing compiler, was 850 ms.

APPENDIX D. REPLACING THE "MUL" INSTRUCTION

As mentioned in Section 4.1.2, two multiplications are needed in the setup calculations before the algorithm's main loop. A ''mul'' instruction takes 37 cycles to execute—much more than the average of 3.5 cycles per instruction.

In this appendix, we show a specific example, suitable for a bit-map with a x-dimension of 2000 pixels. We represent 2000 as 16 * (128 − 3). ''muld $(2000),r0'' is replaced by the following sequence, saving about 24 cycles per multiplication (a 3X speedup):

```
movd  r0,r3
lshd  $(7),r3  # r3 = 128 * r0
movd  r0,r6
addd  r6,r6    # r6 = r0 + r0 = 2 * r0
addd  r0,r0    # r6 = r6 + r0 = 3 * r0
subd  r6,r3    # r3 = r3 - r6 = (128 - 3)
               *  r0 = 125 * r0
lshd  $(4),r3  # r3 - 16 * r3 - 16 * 125
               *  r0 = 2000 * r0
```

APPENDIX E. OPTIMIZING BRANCH INSTRUCTIONS

One of the features of the NS32GX32 that enables it to achieve its high performance is the incorporation of a pipelined instruction processor.

The flow of instructions through the pipeline is delayed when the address from which to fetch an instruction depends on a previous instruction, such as when a conditional branch is executed. The loader includes special circuitry to handle branch instructions, which calculates the destination address and selects between the sequential and non-sequential streams.

An incorrect prediction of the branch causes a breakage in the instruction pipeline, resulting in a delay of 4 cycles.

For conditional branches the branch is predicted taken if it is backward or if the tested condition is NE or LE. A branch predicted incorrectly, whether taken or not, causes the delay of 4 cycles. On the other hand, a branch predicted correctly causes a delay of 1 cycle if it is taken, and no delay if not taken. Thus in the main loop we use

```
cmpqd   $(0),r7        # is p < 0 ?
bgt     p_negative2
```

rather than

```
cmpqd   $(0),r7        # is p < 0 ?
ble     p_non_negative2
```

On the average, half of the branches are taken. For the half incorrectly predicted there is no difference between the two. With ble, however, the prediction is ''branch taken'', with a delay of 1 cycle when correct, while with bgt the prediction is ''not taken'', with no delay when correct. This gives an additional 2.4% improvement in loop time.

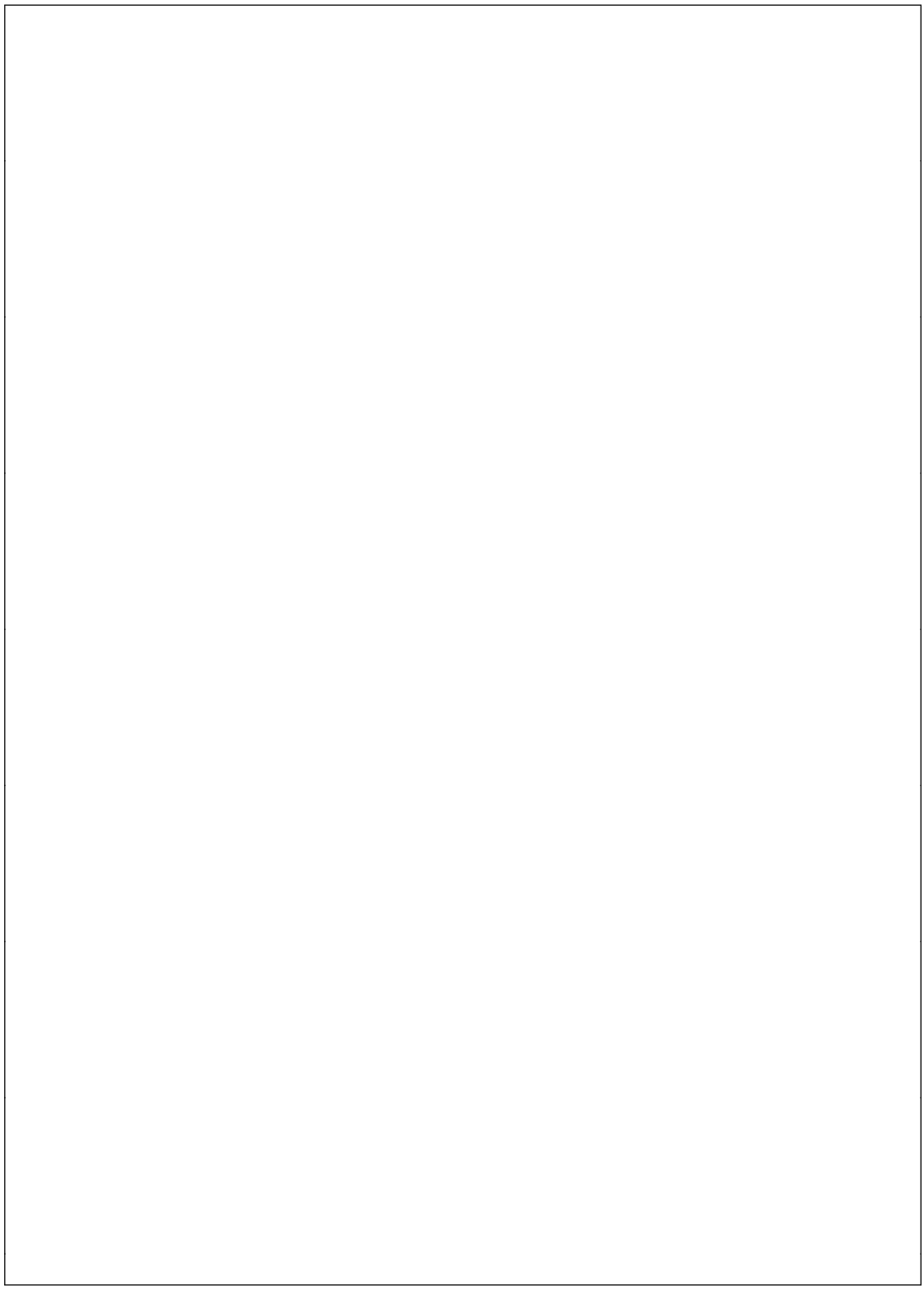
Similarly, we use

```
bne   MAIN_LOOP
br    EXIT_SEQUENCE
```

near the end of the loop, rather than

```
beq   EXIT_SEQUENCE
br    MAIN_LOOP
```

because the second sequence would result mostly in a wrong prediction.



Lit. # 100611

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-8090
Tel: 1(800) 272-9959
TWX: (910) 339-9240

National Semiconductor GmbH
Livry-Gargan-Str. 10
D-82256 Fürstenfeldbruck
Germany
Tel: (81-41) 35-0
Telex: 527649
Fax: (81-41) 35-1

National Semiconductor Japan Ltd.
Sumitomo Chemical
Engineering Center
Bldg. 7F
1-7-1, Nakase, Mihama-Ku
Chiba-City,
Ciba Prefecture 261
Tel: (043) 299-2300
Fax: (043) 299-2500

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductores Do Brazil Ltda.
Rue Deputado Lacorda Franco
120-3A
Sao Paulo-SP
Brazil 05418-000
Tel: (55-11) 212-5066
Telex: 391-1131931 NSBR BR
Fax: (55-11) 212-1181

National Semiconductor (Australia) Pty, Ltd.
Building 16
Business Park Drive
Monash Business Park
Nottinghill, Melbourne
Victoria 3168 Australia
Tel: (3) 558-9999
Fax: (3) 558-9998