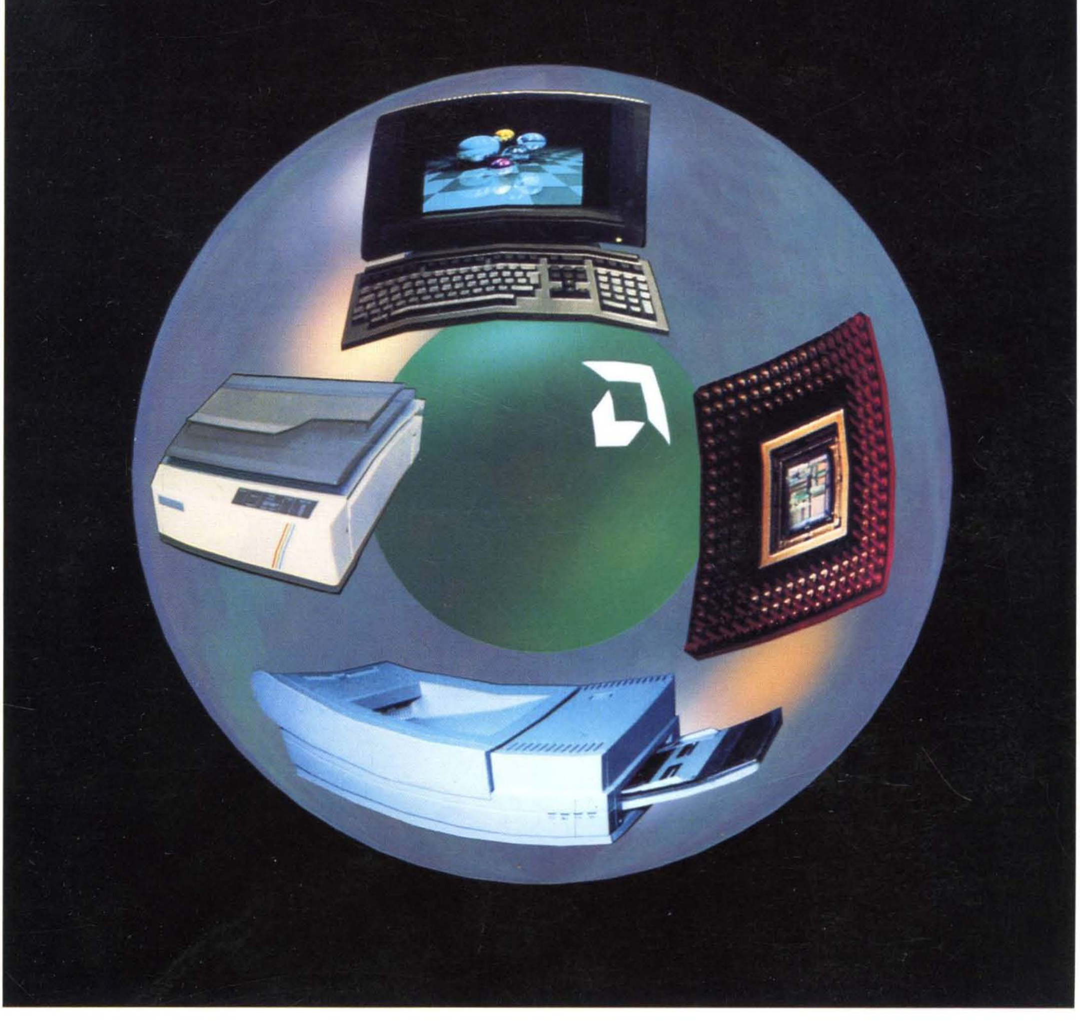
29K Family Graphics Primitives

1990 Handbook

Advanced
Micro
Devices



Advanced Micro Devices

29K Family Graphics Primitives Handbook

by
Tom Crawford

PID# 11011A/0

Figures

Tables

Listings

29K Family Graphics Primitives

by Tom Crawford

INTRODUCTION

This handbook describes how the Am29000™ Streamlined Instruction Processor can be used in graphics applications. It presents primitive graphics functions for 2-D and 3-D rendering of raster displays and evaluates the performance of the Am29000 using standard graphics benchmarks. It also discusses additional hardware and software techniques that can be used to enhance performance. The example programs use low- level routines that can be used to port the standard graphics libraries, such as G.K.S., PHIGS, or X-Windows.

It is assumed that the reader is familiar with the Am29000's architecture, the calling conventions of the C-language, and the software management of the Am29000's stack cache.

For the convenience of the reader, the two header files that contain the basic definitions for the example programs are given in Appendix A. Excerpts from the listings are used throughout this document to illustrate the explanations.

Listings for all the 29K™ Family Graphics Primitives example programs are available (at no charge) on a single 5.25" DSHD floppy diskette. To obtain this diskette, please contact your local sales office (listed in the back of this publication) or call the 29K Hotline at 1-800-2929AMD in the USA.

Suggested Reference Materials

Consult the following AMD reference materials for more information on the topics covered in this handbook (also see Bibliography):

- *Am29000 User's Manual* (order # 10620). This document contains details on the instruction set and register organization of the Am29000.
- *29K Family Data Book* (order # 12175). This document contains a great deal of technical information about the Am29000, including distinctive characteristics, a general and a functional description, the system diagram, connection diagram, pin designations and descriptions, absolute maximum ratings, operational ranges, DC characteristics, switching characteristics and waveforms, and physical dimensions.
- *Am29000 Memory Design Handbook* (order # 10623). This handbook provides Am29000-memory-system design information and specific examples that will be helpful in determining how to design a memory system for the best cost/performance ratio available to fit your Am29000 application.

The above mentioned reference materials can be obtained by writing or calling:

Advanced Micro Devices, Inc.
901 Thompson Place
P.O. Box 3453
Sunnyvale, CA 94088-3453
1-800-222-9323

Bit Maps

A bit map is a two-dimensional array of pixels used to contain the information presented on the graphics screen. The bit map, or some part of it, is read in sync with the screen raster. The data obtained is used to refresh the screen at a high enough rate to avoid flicker. Figure 1 shows the relationship between a bit map and a CRT screen.

The bit map often contains more than one bit of memory for each location on the screen (pixel), which allows gray scale or color to be displayed. The algorithms described in this handbook are documented for monochrome and 32-bit pixels. They can be easily modified for 8 or 16 bits per pixel.

The words in a monochrome bit map appear as shown in Figure 2. There are 32 pixels per word, with bit numbers decreasing from left to right.

With 2 to 8 bits per pixel, each byte is 1 pixel. The pixel contained in bits 31 through 24 appears to the left of the pixel contained in bits 23 through 16, and so forth. All bits in a pixel have a common byte address. Higher-numbered bits are more significant than numbered bits.

With 9 to 16 bits per pixel, each half-word is one pixel. The pixel contained in bits 31 through 16 appears to the left of the pixel contained in bits 15 through 0. All bits in a pixel have a common half-word address. Again, higher-numbered bits are more significant than lower-numbered bits.

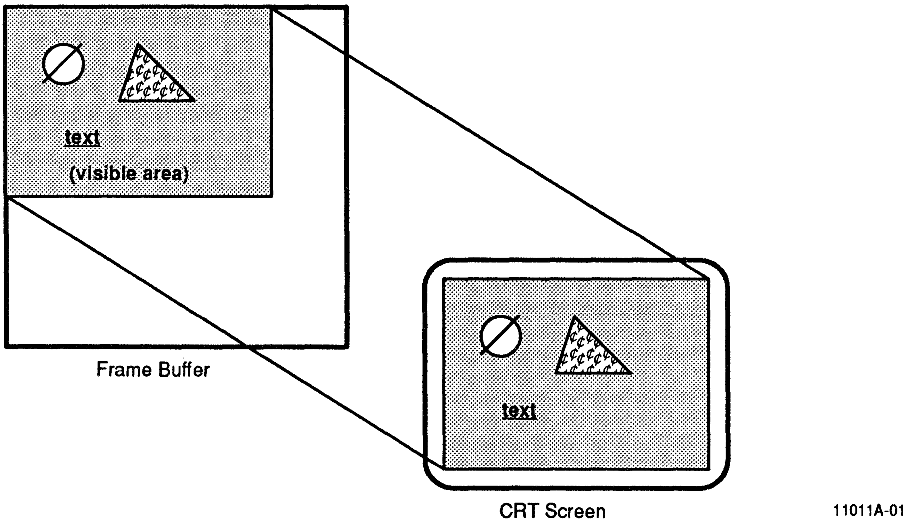


Figure 1. Bit Map

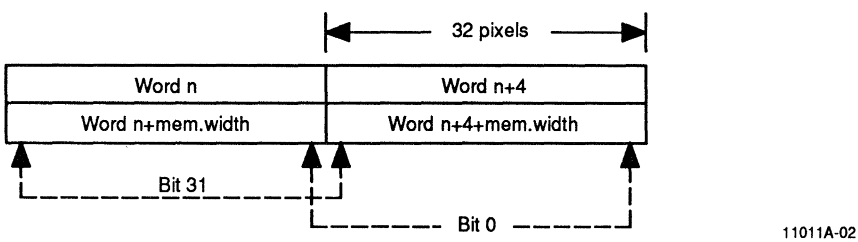


Figure 2. Monochrome Bit Map

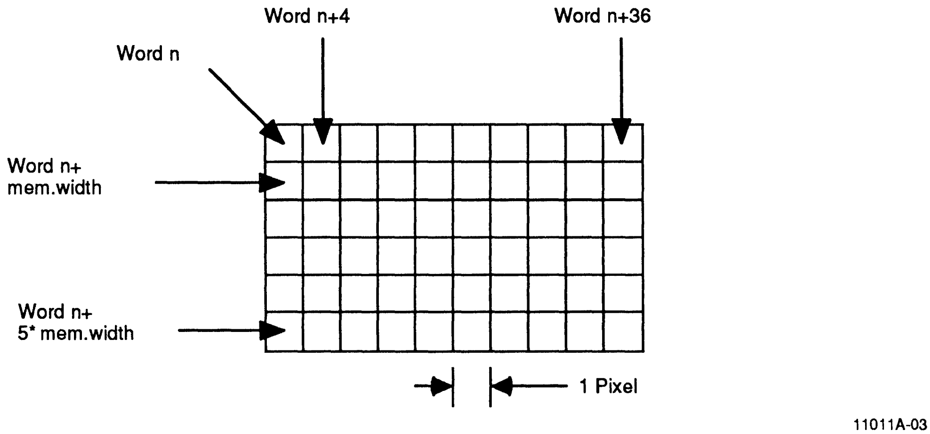


Figure 3. 32 Bits per Pixel

With 17 to 32 bits per pixel, each word contains 1 pixel. All bits in a pixel have a common word address. Again, higher-numbered bits are more significant than lower-numbered bits. A bit map with 32-bit pixels is shown in Figure 3.

Lower-addressed words always appear on the screen to the left of, and above, higher-addressed words. Higher-addressed words appear on the screen to the right of, and below, lower-addressed words.

The algorithms used in this handbook assume that the bit-map memory has byte-write capability. If byte-write is not implemented, the algorithms will need extensive modification for 2 and 4 pixels-per-word and will execute considerably slower.

RENDERING

The sample programs for vector, copy block, string, and filled triangle are C-language callable routines written in ASM29K assembly language.

Common Files

There are two common **.h** files, **G29K_REG.H** and **GRAPH29K.H**, that define registers and structures for the example programs. These files are contained in Appendix A of this document.

G29K_REG.H

G29K_REG.H defines the register names and trap definitions that are used by the example programs. Local register usage is summarized in Table 1.

Table 1. Local Register Assignments in G29K_REG.H

Function	Registers
Input parameters from G29K_Params	lr20 – lr2
S_M1_01 parameters (used by all)	lr24 – lr21
Line internal variables	lr54 – lr25
Block internal variables	lr110 – lr25
Text internal variables	lr35 – lr25
Shading internal variables	lr72 – lr25

The file **G29K_REG.H** also specifies the programmed traps for spill and fill, and two traps for clipping:

```
.equ    V_SPILL          64
.equ    V_FILL,          65
.equ    V_CLIP_SKIP,    100
.equ    V_CLIP_STOP,    101
```

The size of a pixel (in bytes) is defined:

```
.equ    PIXEL_SIZE,       4
```

Five constants define the number of registers that must be claimed by the prologue of each function:

```
.equ    LINE_PRIMITIVE,    53
.equ    BLOCK_PRIMITIVE,  109
.equ    TEXT_PRIMITIVE,    46
.equ    SHADE_PRIMITIVE,   71
.equ    FILL_PRIMITIVE,    71
```

Finally, five macros are defined. The macro ENTER defines three management symbols. The macro PARAM defines a local-register symbol. Neither ENTER nor PARAM generates any code. The macro CLAIM generates the standard C-language calling convention prologue code. The macros RELEASE and LEAVE generate the standard C-language calling convention epilogue code.

Table 2 shows the registers used in vector routines. This list can be used as a guide when deleting registers in order to reduce the value of LINE_PRIMITIVE. Before any registers are deleted, the source code to be retained should be searched for any usage of the deleted registers. Note that any local registers that are "above" (higher than) those deleted must be renumbered.

Another candidate for reduction is BLOCK_PRIMITIVE. There are two approaches that might be useful. First, if only the primitives that do moves (as opposed to arbitrary operations) are used, then the dst.array and its pointers can be removed (registers *lr65–lr33*). Renumbering the source array will allow BLOCK_PRIMITIVE to be reduced by 33. Second, the constant MAX_WORDS can be reduced. Doing so will permit BLOCK_PRIMITIVE values as shown in Table 3.

GRAPH29K.H

GRAPH29K.H defines the structure G29K_Params. This structure contains all parameters other than the fundamental items, such as end points or vertices. The structure matches the definitions of the global control parameter registers in **G29K_REG.H**.

Many of the elements in the structure are not used by every routine. For example, the window parameters are not used by routines that do not clip. Table 4 shows the local registers assigned to structure elements and their usage.

If certain functions will never be used, the storage unique to those functions can be removed from the structure, from the register definitions, and from the appropriate _PRIMITIVES definition in **REG29K.H**.

Table 2. Local Registers Used in Vector Routines

Variable	Reg	1_01	1_02	2_01	2_02	3_01	3_02	4_01
GP.wnd_miny	lr2–5		X		X		X	
GP.pxl_value	lr6	X	X	X	X	X		
GP.mem_width	lr7	X	X	X	X	X	X	X
GP.mem_depth	lr8	X	X	X	X	X	X	X
GP.wnd_base	lr9	X	X	X	X	X	X	X
GP.wnd_align	lr10	X	X	X	X	X	X	X
GP.pxl_op_vector	lr11			X	X	X	X	
GP.pxl_in_mask	lr12			X	X	X	X	
GP.pxl_do_mask	lr13			X	X	X	X	
GP.pxl_do_value	lr14			X	X	X	X	
GP.pxl_out _mask	lr15			X	X	X	X	
GP.wid_actual	lr16					X	X	
GP.pxl_op_code	lr17							
GP.mem_base	lr18							
GP.wnd_origin_x/y	lr19,20							
LP.loc.x	lr21	X	X	X	X	X	X	X
LP.loc.y	lr22	X	X	X	X	X	X	X
LP.loc.addr	lr23	X	X	X	X	X	X	X
LP.loc.align	lr24	X	X	X	X	X	X	X
LP.wid.axial	lr25					X	X	
LP.wid.side_1	lr26					X	X	
LP.wid.side_2	lr27					X	X	
LP.gen.cover	lr28					X	X	X
LP.gen.delta_p	lr29	X	X	X	X	X	X	X
LP.gen.delta_s	lr30	X	X	X	X	X	X	X
LP.gen.move_p	lr31	X	X	X	X	X	X	X
LP.gen.move_s	lr32	X	X	X	X	X	X	X
LP.gen.p	lr33		X		X		X	
LP.gen.s	lr34		X		X		X	
LP.gen.min_p	lr35		X		X		X	
LP.gen.max_p	lr36		X		X		X	
LP.gen.min_s	lr37		X		X		X	
LP.gen.max_s	lr38		X		X		X	
LP.gen.slope	lr39	X	X	X	X	X	X	X
LP.gen.x_slope	lr40	X	X	X	X	X	X	X
LP.gen.error	lr41	X	X	X	X	X	X	X
LP.gen.x_error	lr42						X	
LP.gen.addr	lr43			X	X		X	
LP.gen.try_s	lr44				X		X	X
LP.gen.count	lr45	X	X	X	X	X	X	
LP.clp.skip_vec	lr46		X		X		X	
LP.clp.stop_vec	lr47		X		X		X	
LP.clp.others	lr48–54						X	

Table 3. Permitted Values for MAX_WORDS and Associated BLOCK_PRIMITIVE Values

MAX_WORDS	BLOCK_PRIMITIVE
16	109–32
8	109–48
4	109–56

Table 4. Local-Register Usage in GRAPH29K.H

Name	Reg	What	Where (not) used
wnd_min/max_x/y	lr2–5	Clipping window	Used only for clipping
pxl_value	lr6	Current color	Used by all but bitblt
mem_width	lr7	Scan line size	Used by all
mem_depth	lr8	Encoded bits per pix	Used by all
wnd_base	lr9	See S_M1_01.S	Used by all
wnd_align	lr10	See S_M1_01.S	Used by all
pxl_op_vector	lr11	Address of routine	Used by arbitrary op
pxl_in_mask	lr12	Source input mask	Used by arbitrary op
pxl_do_mask	lr13	Source accept mask	Used by arbitrary op
pxl_do_value	lr14	Source accept value	Used by arbitrary op
pxl_out _mask	lr15	Dest output mask	Used by arbitrary op
wid_actual	lr16	Line width	Used by AA, wide lines
pxl_op_code	lr17	Unused	
mem_base	lr18	Unused	
wnd_origin_x/y	lr19,20	Unused	

Vector Routines

There are seven complete vector routines, as well as some common subroutines. The reason for having several routines for a type of function (e.g., for drawing lines) is that each routine can be optimized for a specific set of circumstances. It is strongly recommended that the reader study the simplest vector routine (**P_L1_01.S**) first. All the vector routines are listed in Table 5.

Table 5. Vector Routines

Routine	Function
P_L1_01.S	Single width, set only, not clipped
P_L1_02.S	Single width, set only, clipped
P_L2_01.S	Single width, general operation, not clipped
P_L2_02.S	Single width, general operation, clipped
P_L3_01.S	Anti-aliased, wide lines, not clipped
P_L3_02.S	Anti-aliased, wide lines, clipped
P_L4_01.S	Monochrome, single width, general operation, not clipped
P_L5_01.S	Single width, set only, not clipped, fixed-width map

All line-drawing routines begin with the normal global functions. The function name is declared to be global, the ENTER macro is used to specify that 54 general registers are required, and the routine label occurs:

```
  .global_P_L1_01
  ENTER  LINE_PRIMITIVE
_P_L1_01:
```

The four parameter register names are declared with PARAM macros. These assign local register numbers above (higher than) the local registers previously defined. These parameters are passed in the local registers shown below:

Macro	Register Name	Register Number
PARAM	Start.x	lr54
PARAM	Start.y	lr55
PARAM	Finish.x	lr56
PARAM	Finish.y	lr57

The CLAIM macro is the function prologue. If a spill operation is not necessary, this consists of five instructions. If a spill is necessary, the standard **SPILL** routine is used, which may involve a Load/Store Multiple operation.

P_L1_01.S

This section begins with a single-width, unclipped vector with set. "With set" means that the current drawing color will be deposited into each pixel location, without regard to the current contents of the bit map. This function assumes 32-bit pixels.

An example of a subroutine that calls this function is given in the file **TEST_L1.C**, which is contained in Appendix A.

Five parameters are loaded from the structure G29K_Params:

```
GP.pxl.value    lr6
GP.mem.width    lr7
GP.mem.depth    lr8
GP.wnd.base     lr9
GP.wnd.align    lr10
```

```
const   Temp0,_G29K_Params + 4 * 4
consth  Temp0,_G29K_Params + 4 * 4
mtsrim  cr,(5 - 1)
loadm   0,0,GP.pxl.value,Temp0
```

Routine **S_M1_01** (which is contained in Appendix A) is called to convert the *Start.x*, *Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr*.

```
add     LP.loc.x,Start.x,0
call    ret,S_M1_01
add     LP.loc.y,Start.y,0
```

The delta in the x direction is computed and put into *LP.gen.delta.p*. The suffix 'p' stands for primary. It is unknown at this point whether the primary direction will be x or y. The initial error is set correctly for reversibly retraceable lines and put into *LP.gen.error*. This can be forced to zero if such lines are not desired. The movement value for the primary direction is set to the distance (in bytes) between pixels in the x direction and put into *LP.gen.move_p*. If *Finish.x* is not greater than *Start.x*, the delta is complemented (made positive) and the movement value is set to move in the negative direction.

```
sub     LP.gen.delta_p,Finish.x,Start.x
sra     LP.gen.error,LP.gen.delta_p,31
jmpf    LP.gen.delta_p,L_01
const   LP.gen.move_p,PIXEL_SIZE
subr    LP.gen.delta_p,LP.gen.delta_p,0
constn  LP.gen.move_p,-PIXEL_SIZE
```

At label L_01, a similar set of calculations is done assuming that y will be the secondary direction. The delta in the y direction is calculated and loaded into *LP.gen.delta.s*. The suffix 's' stands for secondary. The complement of the linear distance between pixels in the y direction is loaded into *LP.gen.move_s*. This is the width of the bit map in bytes. This will be combined with the primary movement to obtain a combined movement. If *Finish.y* is not greater than *Start.y*, the delta is complemented and the true value of the memory width is loaded into the secondary-movement variable, *LP.gen.move_s*.

```
L_01:
  sub     LP.gen.delta_s,Finish.y,Start.y
  jmpf    LP.gen.delta_s,L_02
  subr    LP.gen.move_s,GP.mem.width,0
  subr    LP.gen.delta_s,LP.gen.delta_s,0
  add     LP.gen.move_s,GP.mem.width,0
```

At label L_02, the code decides whether the primary movement is, in fact, in the x direction. The secondary movement value is combined with the primary movement value, and the result is put into *LP.gen.move_s*. If delta_p is not greater than or equal to delta_s, then the primary direction is y. The values in *LP.gen.delta_p* and *LP.gen.delta_s* are swapped with three exclusive-or (XOR) operations, and a corrected primary-movement value is computed.

```
L_02:
  cpge    Temp0,LP.gen.delta_p,
          LP.gen.delta_s
  jmpt    Temp0,L_03
  add     LP.gen.move_s,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.delta_p,LP.gen.delta_p,
          LP.gen.delta_s
  xor     LP.gen.delta_s,LP.gen.delta_p,
          LP.gen.delta_s
  xor     LP.gen.delta_p,LP.gen.delta_p,
          LP.gen.delta_s
  sub     LP.gen.move_p,LP.gen.move_s,
          LP.gen.move_p
```

At label L_03, the code determines which half of the first octant contains the line to be drawn. This is shown in Figure 4.

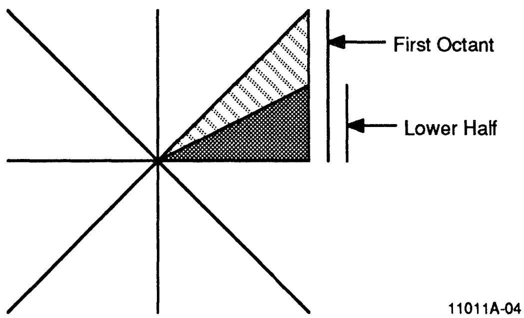


Figure 4. Partial Octant

```
L_03:
  sub     Temp1,LP.gen.delta_p,
          LP.gen.delta_s
  cpgeu   Temp0, Temp1,LP.gen.delta_s
  jmpt    Temp0,L_04
  sub     LP.gen.count,LP.gen.delta_p,1
  add     LP.gen.delta_s,Temp1,0
  xor     LP.gen.move_p,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.move_s,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.move_p,LP.gen.move_p,
          LP.gen.move_s
  nor     LP.gen.error,LP.gen.error,0
```

By drawing the vector as if it were always in the same half of an octant (that is, the bottom half), it is possible to maximize the use of the Branch Target Cache™ (BTC).

If the first jump in the generation loop is usually taken, the entire routine will often execute entirely out of the BTC.

The difference in the deltas is computed into *Temp1* and is compared to the delta in the secondary direction. If the difference is less than the secondary direction, the line is in the top half of the first quadrant. This is equivalent to saying that more combined moves will be required than moves in the primary direction only. If this is the case, the code will adjust the parameters so that the line will be generated as though it were in the bottom half. In either case, the number of iterations necessary in the generation loop is computed and placed into *LP.gen.count.*

If the vector is in the top half of the first octant, the secondary delta is set to the difference of the deltas previously computed. The primary and secondary move values are swapped, and the initial error term is reversed.

At label L_04, the primary and secondary error increments are calculated. These error increments are exactly those described in Bresenham's algorithm, namely "2*delta_s" and "2*delta_s – 2*delta_p." The initial error term is calculated. It is now possible to begin to draw the vector.

```
L_04:
   sll    LP.gen.slope,LP.gen.delta_s,1
   sll    LP.gen.x_slope,LP.gen.delta_p,1
   sub    LP.gen.x_slope,LP.gen.slope,
          LP.gen.x_slope
   add    LP.gen.error,LP.gen.error,
          LP.gen.slope
   sub    LP.gen.error,LP.gen.error,
          LP.gen.delta_p
   jmpt   LP.gen.count,L_07
   sub    LP.gen.count,LP.gen.count,1
```

The variable *LP.gen.count* is tested to determine whether to enter the actual generation loop or to draw a single pixel; at least one pixel is always drawn.

In the generation loop, for each pixel in the primary direction, a pixel is drawn and a direction for the next pixel is chosen. There are two basic paths through the loop, not including the end point. The path usually taken involves movement in the primary direction only.

```
L_05:
   jmpt   (primary) to L_06
   store  pixel

L_06:
   add    primary inc to error
   dec    count, jump L_05
   add    primary move to addr
```

This loop is five instructions per pixel executing entirely out of the Branch Target Cache.

The alternate path is taken when movement in both directions is required.

```
L_05:
   do not jump (secondary)
   store pixel
   add secondary inc to error
   dec count, jump L_05
   add combined move to addr
```

In this case (movement in both directions), there are also five instructions per pixel, but the routine does not execute entirely out of the BTC. In a system without single-cycle instruction burst or with five or more cycles for the first instruction, executing entirely from the BTC is advantageous. This is because the memory cannot keep up with the processor. In fact, in a system with fast access and single-cycle burst, the routine is slightly slower due to the extra calculations necessary to place the line into the proper half of the octant.

```
L_05:
   jmpt     LP.gen.error,L_06
   store    0,0,GP.pxl.value,
            LP.loc.addr
   add      LP.gen.error,LP.gen.error,
            LP.gen.x_slope
   jmpfdec  LP.gen.count,L_05
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_s
   jmp      L_08
   store    0,0,GP.pxl.value,
            LP.loc.addr
L_06:
   add      LP.gen.error,LP.gen.error,
            LP.gen.slope
   jmpfdec  LP.gen.count,L_05
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_p

L_07:
   store    0,0,GP.pxl.value,
            LP.loc.addr
L_08:
```

In either case, when the count has expired, the last pixel is drawn and the routine exits.

```
   RELEASE
   nop
   LEAVE
```

P_L1_02.S

This C-language callable routine draws single-width, set vectors with clipping. The routine begins with the normal global functions.

Nine parameters are loaded from the structure G29K_Params.

```
GP.wnd.min_x                  lr2
GP.wnd.max_x                  lr3
GP.wnd.min_y                  lr4
GP.wnd.max_y                  lr5
GP.pxl.value                  lr6
GP.mem.width                  lr7
GP.mem.depth                  lr8
GP.wnd.base                   lr9
GP.wnd.align                  lr10
const   Temp0,_G29K_Params
consth  Temp0,_G29K_Params
mtsrim  cr,(9 - 1)
loadm   0,0,GP.wnd.min_x,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x*, *Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr.*

```
add     LP.loc.x,Start.x,0
call    ret,S_M1_01
add     LP.loc.y,Start.y,0
```

The routine assumes that x will be the primary direction and y will be the secondary direction. The value delta_p is accordingly calculated from the x end points, and delta_s is calculated from the y end points. The routine calculates delta_p by subtracting *Start.x* from *Start.y.* The initial error is set to the sign bit of this delta, and a test is made to determine whether *Finish.s* is greater than or equal to *Start.s.* In either case, delta_s is calculated by subtracting *Start.y* from *Finish.y.*

```
sub     LP.gen.delta_p,Finish.x,Start.x
sra     LP.gen.error,LP.gen.delta_p,31
jmpf    LP.gen.delta_p,L_01
sub     LP.gen.delta_s,Finish.y,Start.y
subr    LP.gen.delta_p,LP.gen.delta_p,0
constn  LP.gen.move_p,-PIXEL_SIZE
subr    LP.gen.p,Start.x,0
subr    LP.gen.min_p,GP.wnd.max_x,0
jmp     L_02
subr    LP.gen.max_p,GP.wnd.min_x,0
```

If *Finish.x* is not equal to or greater than *Start.s*, delta_p is negated. The pixel offset (the move value) in the primary direction is set to the negative of the pixel size (the distance between pixels in bytes in the horizontal direction). The minimum and maximum clipping boundaries are set to the complement of the maximum and minimum x values of the clipping window, respectively.

If *Finish.x* is greater than or equal to *Start.x*, the logic at L_01 sets the primary movement value to the distance between pixels (in bytes) in the x direction. The minimum and maximum clipping boundaries are set to the minimum and maximum x values of the clipping window, respectively.

```
L_01:
  const   LP.gen.move_p,PIXEL_SIZE
  add     LP.gen.p,Start.x,0
  add     LP.gen.min_p,GP.wnd.min_x,0
  add     LP.gen.max_p,GP.wnd.max_x,0
```

At label L_02 the Clipping Skip Vector is set to the label Loop (partially). This is completed during a delay instruction of a jump just after label L_04. The secondary delta is tested to determine whether *Finish.y* was equal to or greater than *Start.y.* If not, the secondary delta is negated, and the secondary movement value is set to the memory width (that is, the number of bytes in the linear address space between vertically adjacent pixels). The secondary initial clipping location is set to the complement of *Start.y.* The minimum and maximum clipping boundaries are set to the negative of the maximum and minimum y values of the clipping window, respectively.

```
L_02:
  jmpf    LP.gen.delta_s,L_03
  const   LP.clp.skip_vec,Loop
  subr    LP.gen.delta_s,LP.gen.delta_s,0
  add     LP.gen.move_s,GP.mem.width,0
  subr    LP.gen.s,Start.y,0
  subr    LP.gen.min_s,GP.wnd.max_y,0
  jmp     L_04
  subr    LP.gen.max_s,GP.wnd.min_y,0
```

If *Finish.y* was greater than or equal to *Start.y*, then the code at label L_03 sets the secondary movement value to the negative of the memory width, and the secondary clipping initial value is set to *Start.y.* The minimum and maximum secondary clipping boundaries are set to the minimum and maximum y values of the window, respectively.

```
L_03:
  subr    LP.gen.move_s,GP.mem.width,0
  add     LP.gen.s,Start.y,0
  add     LP.gen.min_s,GP.wnd.min_y,0
  add     LP.gen.max_s,GP.wnd.max_y,0
```

At label L_04, the primary delta is compared to the secondary delta. The Clipping Skip Vector is completely set to the label Loop. If the primary delta was greater than or equal to the secondary delta, then the x direction is the primary direction and the code continues at label L_05. If not, then the primary direction is, in fact, the y direction, and some swapping is necessary.

The following five sets of values are exchanged:

```
LP.gen.delta_p          LP.gen.delta_s
LP.gen.move_p           LP.gen.move_s
LP.gen.p                LP.gen.s
LP.gen.min_p            LP.gen.min_s
LP.gen.max_p            LP.gen.max_s
```

That is, the primary and secondary deltas, movement values, initial clipping positions, and clipping boundaries are exchanged.

```
L_04:
   cpge     Temp0,LP.gen.delta_p,
            LP.gen.delta_s
   jmpt     Temp0,L_05
   consth   LP.clp.skip_vec,Loop
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_s,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_s,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.p,LP.gen.p,LP.gen.s
   xor      LP.gen.s,LP.gen.p,LP.gen.s
   xor      LP.gen.p,LP.gen.p,LP.gen.s
   xor      LP.gen.min_p,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.min_s,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.min_p,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
   xor      LP.gen.max_s,LP.gen.max_p,
            LP.gen.max_s
   xor      LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
```

At label L_05, the primary and secondary error increments are calculated. These error increments are exactly those described in Bresenham's algorithm, namely "2*delta_s" and "2*delta_s – 2*delta_p." The initial error term is calculated, and the Clipping Stop Vector is set. Now the vector can be drawn.

```
L_05:
   sll      LP.gen.slope,LP.gen.delta_s,1
   sll      LP.gen.x_slope,LP.gen.delta_p,1
   add      LP.gen.error,LP.gen.error,
            LP.gen.slope
   sub      LP.gen.error,LP.gen.error,
            LP.gen.delta_p
   const    LP.clp.stop_vec,Stop
   consth   LP.clp.stop_vec,Stop
   jmp      L_08
   sub      LP.gen.count,LP.gen.delta_p,1

Loop:
   jmpfdec  LP.gen.count,L_06
   nop
   jmp      Stop
   nop
```

If the vector is a single pixel, the routine jumps to L_08, where it is checked against the clipping boundaries. If the single pixel is inside the clipping window, it will be drawn.

The pixel count is calculated from the primary delta, and the routine falls through to label Loop.

Depending on whether the movement will be in the primary direction or the combined direction, the code will take one of two paths through the generation loop. If the movement is in the primary direction only, the path is as follows:

```
L_06:
   jump to L_07
   add prim inc to error

L_07:
   add prim move to addrs
   inc primary clipping adrs
L_08:
   assert pixel inside clipping
   dec loop count, jump L_06
   store value into addr
Stop:
   RELEASE
   nop
   LEAVE
```

This movement in the primary direction requires ten instructions, including the four ASSERT instructions that mechanize the clipping, plus one store per pixel.

If the movement is in the combined direction, the path is as follows:

```
L_06:
   do not jump to L_07
   add prim inc to error
   sub x error incr from error
   add secondary move to addrs
   inc secondary clipping adrs
```

```
L_07:
  add prim move to addrs
  inc primary clipping adrs

L_08:
  assert pixel inside clipping
  dec loop count, jump L_06
  store value into addr

Stop:
  RELEASE
  nop
  LEAVE
```

This movement in the combined direction requires 13 instructions, including the four ASSERT instructions that mechanize the clipping, plus one store per pixel.

Clipping is mechanized with a set of four ASSERT instructions, as shown in Figure 5.

The line is being drawn from left to right (that is, in the first quadrant). The current clipping locations are initially set to the *Start.x* and *Start.y* values. The clipping boundaries are set to the clipping window values.

When the generation loop begins, the first assert will fail because *LP.gen.p* will not be greater than or equal to *LP.gen.min.p.* Since the Clipping Skip Vector has been set to Loop, the routine will go back to the top and continue with the next point along the line. In this case, the loop generation count is decremented at the top of the loop. This point is not drawn.

When *LP.gen.p* has been incremented to a value greater than or equal to *LP.gen.min_p,* the assertions will succeed. Now the routine will execute the STORE instructions, and each pixel will be written. The generation loop count is decremented at the bottom of the loop.

When *LP.gen.p* has been incremented to a value greater than or equal to *LP.gen.max_p*, the third assertion will fail. Since the Clipping Stop Vector points to the label Stop, the generation loop will exit as though the count were negative, and the iterations will cease.

```
L_06:
  jmpt    LP.gen.error,L_07
  add     LP.gen.error,LP.gen.error,
          LP.gen.slope
  sub     LP.gen.error,LP.gen.error,
          LP.gen.x_slope
  add     LP.loc.addr,LP.loc.addr,
          LP.gen.move_s
  add     LP.gen.s,LP.gen.s,1

L_07:
  add     LP.loc.addr,LP.loc.addr,
          LP.gen.move_p
  add     LP.gen.p,LP.gen.p,1

L_08:
  asge    V_CLIP_SKIP,LP.gen.p,
          LP.gen.min_p
  asge    V_CLIP_SKIP,LP.gen.s,
          LP.gen.min_s
  asle    V_CLIP_STOP,LP.gen.p,
          LP.gen.max_p
  asle    V_CLIP_STOP,LP.gen.s,
          LP.gen.max_s
  jmpfdec LP.gen.count,L_06
  store   0,0,GP.pxl.value,LP.loc.addr
```

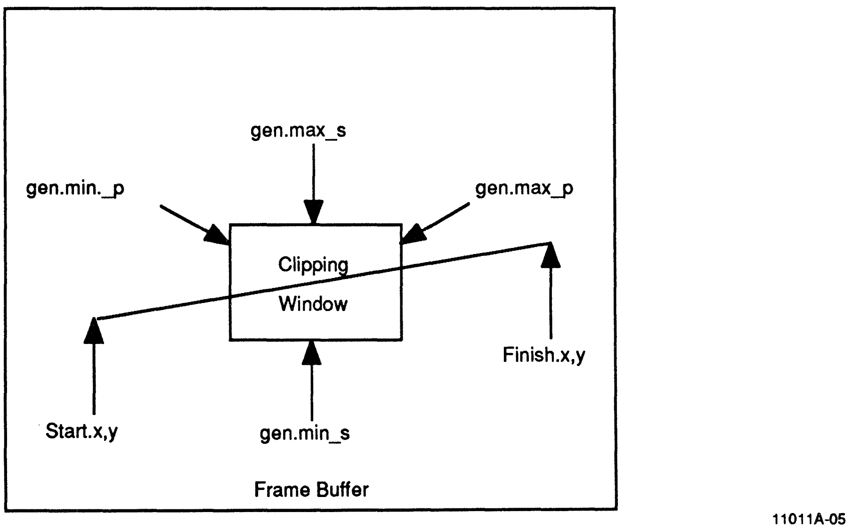


Figure 5. Clipping

```
Stop:
   RELEASE
   nop
   LEAVE
```

P_L2_01.S

This C-language callable routine is used to draw a single-width, unclipped line. The writing operation can be any the user wishes.

The operation (e.g., XOR, AND, ADD, etc.) is specified by the user by providing the address of the routine that performs the operation. The example user-supplied routine, **O1_02.S** (included on the distribution diskette) XORs the current contents of the addressed pixel with the pixel value.

The routine **P_L2_01.S** begins with the normal global functions. Ten parameters are loaded from the structure G29K_Params.

```
GP.pxl.value            lr6
GP.mem.width            lr7
GP.mem.depth            lr8
GP.wnd.base             lr9
GP.wnd.align            lr10
GP.pxl.op_vec           lr11
GP.pxl.in_mask          lr12
GP.pxl.do_mask          lr13
GP.pxl.do_value         lr14
GP.pxl.out_mask         lr15
const   Temp0,_G29K_Params + 4 * 4
consth  Temp0,_G29K_Params + 4 * 4
mtsrim  cr,(10 - 1)
loadm   0,0,GP.pxl.value,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x, Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr.*

```
add     LP.loc.x,Start.x,0
call    ret,S_M1_01
add     LP.loc.y,Start.y,0
```

The routine assumes that the x direction will be the primary direction, and that the y direction will be secondary. It calculates the delta for the primary direction by subtracting *Start.x* from *Finish.x.* The error adjustment for reversibly retraceable lines is set from the sign bit. The primary movement parameter is set to PIXEL_SIZE, which is the distance in bytes between horizontally adjacent pixels. If *Finish.x* is greater than or equal to *Start.x,* the code jumps to L_01. If *Finish.x* is less than *Start.x,* the value for the primary delta is negated, and the primary movement parameter is set to negative PIXEL_SIZE. (The line will be drawn from right to left.)

```
sub     LP.gen.delta_p,Finish.x,Start.x
sra     LP.gen.error,LP.gen.delta_p,31
jmpf    LP.gen.delta_p,L_01
const   LP.gen.move_p,PIXEL_SIZE
subr    LP.gen.delta_p,LP.gen.delta_p,0
constn  LP.gen.move_p,-PIXEL_SIZE
```

At label L_01, the secondary delta is calculated by subtracting *Start.y* from *Finish.y.* The secondary movement parameter is set to the negative of the memory width (that is, the distance between vertically adjacent pixels). If *Finish.y* is less than *Start.y,* the secondary delta is negated and the secondary movement parameter is set to the value of memory width.

```
L_01:
   sub     LP.gen.delta_s,Finish.y,Start.y
   jmpf    LP.gen.delta_s,L_02
   subr    LP.gen.move_s,GP.mem.width,0
   subr    LP.gen.delta_s,LP.gen.delta_s,0
   add     LP.gen.move_s,GP.mem.width,0
```

At label L_02, the primary and secondary deltas are compared. If the primary delta is greater than or equal to the secondary delta, then the x axis is the primary axis. The combined movement value is computed by adding the primary movement to the original secondary movement.

```
L_02:
   cpge    Temp0,LP.gen.delta_p,
           LP.gen.delta_s
   jmpt    Temp0,L_03
   add     LP.gen.move_s,LP.gen.move_p,
           LP.gen.move_s
   xor     LP.gen.delta_p,LP.gen.delta_p,
           LP.gen.delta_s
   xor     LP.gen.delta_s,LP.gen.delta_p,
           LP.gen.delta_s
   xor     LP.gen.delta_p,LP.gen.delta_p,
           LP.gen.delta_s
   sub     LP.gen.move_p,LP.gen.move_s,
           LP.gen.move_p
```

If the primary delta is less than the secondary delta, the deltas are swapped using three XOR instructions. The combined movement parameter is corrected.

At label L_03, the primary and secondary error increments are calculated. These error increments are derived from those described in Bresenham's algorithm. The initial error term is calculated. The vector can now be drawn.

```
L_03:
   sll     LP.gen.slope,LP.gen.delta_s,1
   sll     LP.gen.x_slope,LP.gen.delta_p,1
   add     LP.gen.error,LP.gen.error,
           LP.gen.slope
```

```
  sub     LP.gen.error,LP.gen.error,
          LP.gen.delta_p
  calli   ret,GP.pxl.op_vec
  sub     LP.gen.count,LP.gen.delta_p,1
  jmpt    LP.gen.count,L_06
  sub     LP.gen.count,LP.gen.count,1
```

The first pixel is always drawn, even if it is the only pixel. This routine draws pixels by executing a CALLI instruction through *GP.pxl.op.vec*. This pointer must have been set up by the caller in the structure G29K_Params. The routine performs whatever operation (XOR, Add, etc.) is specified by the caller.

Routine **O1_02.S** will be used as an example. It has access to the register names (at assemble time) because it includes **G29K_REG.H**. When it is called, *LP.loc.addr* contains the linear address of the pixel location, and *GP.pxl.value* contains the current pixel value (drawing color). The routine simply reads the address pixel into a temporary register and XORs it with current drawing color. The result is stored back into the current pixel location and the routine exits.

In routine **P_L2_01.S**, the code computes the number of pixels left to right and determines whether there are more. If so, the count is decremented, and the generation loop is entered.

There are two paths through the generation loop. In the case where the move is in the primary direction only, the path through the loop is:

```
L_04:
  jump to L_05
  add primary adjust to error

L_05:
  call the operation routine
  add the prim move to addr
  dec loop count, jmp L_04
  (nop for delay slot)
```

This (movement in the primary direction only) requires six instructions plus the operator routine.

In the case where the move is in the combined direction, the path through the loop is:

```
L_04:
  (do not) jump
  add primary adjust to error
  call the operator routine
  add combined adust to adr
  dec loop count, jmp L_04
  subtract out primary error
```

This movement in the combined direction requires six instructions plus the operator routine.

```
L_04:
  jmpt    LP.gen.error,L_05
  add     LP.gen.error,LP.gen.error,
          LP.gen.slope
  calli   ret,GP.pxl.op_vec
  add     LP.loc.addr,LP.loc.addr,
          LP.gen.move_s
  jmpfdec LP.gen.count,L_04
  sub     LP.gen.error,LP.gen.error,
          LP.gen.x_slope
  jmp     L_06
  nop
L_05:
  calli   ret,GP.pxl.op_vec
  add     LP.loc.addr,LP.loc.addr,
          LP.gen.move_p
  jmpfdec LP.gen.count,L_04
  nop
```

P_L2_02.S

This C-language callable routine is used to draw single-width clipped lines with any pixel operation.

The operation (e.g., XOR, AND, ADD, etc.) is specified by the user by providing the address of the routine that performs the operation. The routine **O1_02.S** XORs the current contents of the addressed pixel with the pixel value.

P_L2_02.S begins with the normal global functions. Fourteen parameters are loaded from the structure G29K_Params.

```
  GP.wnd.min_x          lr2
  GP.wnd.max_x          lr3
  GP.wnd.min_y          lr4
  GP.wnd.max_y          lr5
  GP.pxl.value          lr6
  GP.mem.width          lr7
  GP.mem.depth          lr8
  GP.wnd.base           lr9
  GP.wnd.align          lr10
  GP.pxl.op_vec         lr11
  GP.pxl.in_mask        lr12
  GP.pxl.do_mask        lr13
  GP.pxl.do_value       lr14
  GP.pxl.out_mask       lr15
  const   Temp0,_G29K_Params
  consth  Temp0,_G29K_Params
  mtsrim  cr,(14 - 1)
  loadm   0,0,GP.wnd.min_x,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x, Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr*.

```
add     LP.loc.x,Start.x,0
call    ret,S_M1_01
add     LP.loc.y,Start.y,0
```

This routine assumes that the primary direction will be x and the secondary direction will be y. This is tested at label L_04.

```
sub     LP.gen.delta_p,Finish.x,Start.x
sra     LP.gen.error,LP.gen.delta_p,31
jmpf    LP.gen.delta_p,L_01
sub     LP.gen.delta_s,Finish.y,Start.y
subr    LP.gen.delta_p,LP.gen.delta_p,0
constn  LP.gen.move_p,-PIXEL_SIZE
subr    LP.gen.p,Start.x,0
subr    LP.gen.min_p,GP.wnd.max_x,0
jmp     L_02
subr    LP.gen.max_p,GP.wnd.min_x,0
```

The routine computes the primary delta by subtracting *Start.x* from *Finish.y*. The sign bit is placed in the error variable to allow for reversibly retraceable lines. If *Finish.x* is less than *Start.x*, the line is drawn from right to left. In this case, the primary delta is negated, and the primary movement is set to the negative of PIXEL_SIZE, which is the distance between horizontally adjacent pixels, moving from right to left. The current primary clipping location is set to the negative of *Start.x*, and the primary minimum and maximum clipping boundaries are set to the maximum and minimum x window boundaries respectively. The secondary delta is computed by subtracting *Start.y* from *Finish.y*.

If *Finish.x* is equal to or greater than *Start.x*, the code jumps to label L_01. The primary movement value is set to PIXEL_SIZE, which is the distance between horizontally adjacent pixels, moving from left to right. The current primary clipping location is set to *Start.x*, and the primary minimum and maximum clipping boundaries are set to the minimum and maximum x window boundaries, respectively. The second delta is computed by subtracting *Start.y* from *Finish.y*.

```
L_01:
  const   LP.gen.move_p,PIXEL_SIZE
  add     LP.gen.p,Start.x,0
  add     LP.gen.min_p,GP.wnd.min_x,0
  add     LP.gen.max_p,GP.wnd.max_x,0
```

At label L_02, the secondary variables are computed similarly. If *Finish.y* is less than *Start.y*, the line is drawn from lower addresses to higher addresses. The secondary delta is negated, and the secondary movement value is set to the distance between vertically adjacent pixels. The current secondary clipping location is set to the negative of *Start.y*, and the secondary minimum and maximum clipping boundaries are set to the negative of the maximum and minimum y window boundaries, respectively.

```
L_02:
  jmpf    LP.gen.delta_s,L_03
  const   LP.clp.skip_vec,Loop
  subr    LP.gen.delta_s,LP.gen.delta_s,0
  add     LP.gen.move_s,GP.mem.width,0
  subr    LP.gen.s,Start.y,0
  subr    LP.gen.min_s,GP.wnd.max_y,0
  jmp     L_04
  subr    LP.gen.max_s,GP.wnd.min_y,0
```

If *Finish.y* is equal to or greater than *Start.y*, the code jumps to label L_03. The secondary movement value is set to the negative of the memory width, which is the distance between the vertically adjacent pixels, moving from bottom to top. The current secondary clipping location is set to *Start.y*, and the secondary minimum and maximum clipping boundaries are set to the minimum and maximum window boundaries, respectively.

```
L_03:
  subr    LP.gen.move_s,GP.mem.width,0
  add     LP.gen.s,Start.y,0
  add     LP.gen.min_s,GP.wnd.min_y,0
  add     LP.gen.max_s,GP.wnd.max_y,0
```

At label L_04, the routine determines whether the x direction is actually going to be the primary. If not, the primary and secondary values are swapped.

```
L_04:
  cpge    Temp0,LP.gen.delta_p,
          LP.gen.delta_s
  jmpt    Temp0,L_05
  consth  LP.clp.skip_vec,Loop
  xor     LP.gen.delta_p,LP.gen.delta_p,
          LP.gen.delta_s
  xor     LP.gen.delta_s,LP.gen.delta_p,
          LP.gen.delta_s
  xor     LP.gen.delta_p,LP.gen.delta_p,
          LP.gen.delta_s
  xor     LP.gen.move_p,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.move_s,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.move_p,LP.gen.move_p,
          LP.gen.move_s
  xor     LP.gen.p,LP.gen.p,LP.gen.s
  xor     LP.gen.s,LP.gen.p,LP.gen.s
  xor     LP.gen.p,LP.gen.p,LP.gen.s
  xor     LP.gen.min_p,LP.gen.min_p,
          LP.gen.min_s
  xor     LP.gen.min_s,LP.gen.min_p,
          LP.gen.min_s
```

```
    xor     LP.gen.min_p,LP.gen.min_p,
            LP.gen.min_s
    xor     LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
    xor     LP.gen.max_s,LP.gen.max_p,
            LP.gen.max_s
    xor     LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
```

There are a total of five sets of values to be swapped, requiring three XOR instructions.

```
LP.gen.delta_p          LP.gen.delta_s
LP.gen.move_p           LP.gen.move_s
LP.gen.p                LP.gen.s
LP.min_p                LP.min_s
LP.max_p                LP.max_s
```

That is, the primary and secondary values for the delta, the movement value, and the clipping parameters are swapped.

At label L_05, the primary and secondary error increments are calculated. These error increments are derived from those described in Bresenham's algorithm. The initial error term is calculated.

```
L_05:
    sll     LP.gen.slope,LP.gen.delta_s,1
    sll     LP.gen.x_slope,LP.gen.delta_p,1
    add     LP.gen.error,LP.gen.error,
            LP.gen.slope
    sub     LP.gen.error,LP.gen.error,
            LP.gen.delta_p
    const   LP.clp.stop_vec,Stop
    consth  LP.clp.stop_vec,Stop
    jmp     L_08
    sub     LP.gen.count,LP.gen.delta_p,1
```

The Clipping Stop Vector is set to the label Stop. The Clipping Skip Vector has already been set to the label Loop. The program jumps to the middle of the generation loop, where the first pixel will be drawn if it is inside the clipping window. The generation loop count is calculated by subtracting one from the primary delta.

There are two possible paths through the generation loop, depending on whether the movement is in the primary direction or in the combined direction. If the movement is in the primary direction only, the path through the loop is as follows:

```
L_06:
    jump to L_07
    add x error to error

L_07:
    add primary move to address
    inc primary clipping location

L_08:
    assert loc inside window
    call pixel op routine
    (nop)

Loop:
    decr count, jump L_06
    (nop)
```

This movement in the primary direction only requires 12 instructions plus any instructions in the pixel operation routine. This includes the four ASSERT instructions.

If the movement is in the combined direction, the path through the loop is as follows:

```
L_06:
    (do not) jump to L_07
    add x error to error
    subtract....
    add secondary move to address
    inc secondary clipping location

L_07:
    add primary move to address
    inc primary clipping location

L_08:
    assert loc inside window
    call pixel op routine
    (nop)

Loop:
    decr count, jump L_06
    (nop)
```

This movement in the combined direction requires 15 instructions plus any instructions in the pixel operation routine. This includes the four ASSERT instructions. Clipping is mechanized as described in section "P_L2_01.S" above.

```
L_06:
    jmpt    LP.gen.error,L_07
    add     LP.gen.error,LP.gen.error,
            LP.gen.slope
    sub     LP.gen.error,LP.gen.error,
            LP.gen.x_slope
    add     LP.loc.addr,LP.loc.addr,
            LP.gen.move_s
    add     LP.gen.s,LP.gen.s,1

L_07:
    add     LP.loc.addr,LP.loc.addr,
            LP.gen.move_p
    add     LP.gen.p,LP.gen.p,1

L_08:
```

```
    asge    V_CLIP_SKIP,LP.gen.p,
            LP.gen.min_p
    asge    V_CLIP_SKIP,LP.gen.s,
            LP.gen.min_s
    asle    V_CLIP_STOP,LP.gen.p,
            LP.gen.max_p
    asle    V_CLIP_STOP,LP.gen.s,
            LP.gen.max_s
    calli   ret,GP.pxl.op_vec
    nop

Loop:
    jmpfdecLP.gen.count,L_06
    nop
```

P_L3_01.S

The routine **P_L3_01.S** is used to draw anti-aliased wide lines with user-supplied writing and anti-aliasing functions. The routine can also be used to draw wide lines without anti-aliasing. Clipping is not performed.

The actual bit map operations are done by routines supplied by the calling function. One of the parameters supplied by the calling routine to **P_L3_01.S** is the address of a user-written routine. The input parameters to the routine are the address of the pixel location and the required "coverage" of that pixel.

The coverage parameter is an integer in the range 1 through 256 inclusively, indicating the portion of the pixel (in 256ths) that is actually covered by the line. Alternatively, it can be thought of as a real number in the range 1/256 to 1, with the radix point between bits 7 and 8.

The routine begins with the normal global functions. Eleven parameters are loaded from the structure G29K_Params.

```
    GP.pxl.value          lr6
    GP.mem.width          lr7
    GP.mem.depth          lr8
    GP.wnd.base           lr9
    GP.wnd.align          lr10
    GP.pxl.op_vec         lr11
    GP.pxl.in_mask        lr12
    GP.pxl.do_mask        lr13
    GP.pxl.do_value       lr14
    GP.pxl.out_mask       lr15
    GP.wid.actual         lr16
    const   Temp0,_G29K_Params + 4 * 4
    consth  Temp0,_G29K_Params + 4 * 4
    mtsrim  cr,(11 - 1)
    loadm   0,0,GP.pxl.value,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x*, *Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr*.

```
    add     LP.loc.x,Start.x,0
    call    ret,S_M1_01
    add     LP.loc.y,Start.y,0
    sub     LP.gen.delta_p,Finish.x,
            Start.x
    jmpf    LP.gen.delta_p,L_01
    sub     LP.gen.delta_s,Finish.y,
            Start.y
    subr    LP.gen.delta_p,LP.gen.delta_p,0
    jmp     L_02
    constn  LP.gen.move_p,-PIXEL_SIZE
```

This routine assumes that the primary direction will be x and the secondary direction will be y. If this is not the case, the parameters will be swapped at L_04. The primary delta is calculated by subtracting *Start.x* from *Finish.x*. If *Finish.x* is greater than or equal to *Start.x*, the code at L_01 sets the primary movement value to PIXEL_SIZE, which is the distance in bytes between horizontally adjacent pixels. The secondary delta is calculated by subtracting *Start.y* from *Finish.y*.

```
L_01:
    const   LP.gen.move_p,PIXEL_SIZE
```

If *Finish.x* is less than *Start.x*, the primary delta is negated and the primary movement value is set to the negative of PIXEL_SIZE. The secondary delta is calculated by subtracting *Start.y* from *Finish.y*.

At label L_02, the secondary parameters are calculated in a similar manner. If *Finish.y* is less than *Start.y*, the secondary delta is negated and the secondary movement value is set to *GP.mem.width*. If *Finish.y* is greater than or equal to *Start.y*, the code jumps to L_03, where the secondary movement value is set to the negative of *GP.mem.width*. In either case, the beginning address is set to the beginning pixel address calculated from *Start.x*, *Start.y*.

```
L_02:
    jmpf    LP.gen.delta_s,L_03
    add     LP.gen.addr,LP.loc.addr,0
    subr    LP.gen.delta_s,LP.gen.delta_s,0
    jmp     L_04
    add     LP.gen.move_s,GP.mem.width,0

L_03:
    subr    LP.gen.move_s,GP.mem.width,0
```

At label L_04, the code determines which axis will be primary. If the primary delta is greater than or equal to the secondary delta, the initial assumption was correct and nothing extra need be done. If the primary delta is less than the secondary delta, the primary direction will be y, and two pairs of parameters must be swapped.

```
LP.gen.delta_p          LP.gen.delta_s
LP.gen.move_p           LP.gen.move_s
```

```
L_04:
   cpge     Temp0,LP.gen.delta_p,
            LP.gen.delta_s
   jmpt     Temp0,L_05
   const    LP.gen.error,0
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_s,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_s,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
```

In either case, the error term is forced to zero. At label L_05, the axial half-width is calculated. Figure 6 illustrates this derivation.

The line (or a representative piece of the line) is illustrated by the heavy line from S to F. The wide line to be drawn covers the area between the two lines parallel to the heavy line. The specific line segment in the figure has a slope of $\Delta s/\Delta p = 2/4 = 0.5$. The distance to be determined is the distance from the center of the line to the exact edge of the line in the secondary direction. This is distance h in Figure 6.

```
L_05:
   mtsr     q,LP.gen.delta_p
   mulu     Temp0,LP.gen.delta_p,0
   mulu     Temp0,LP.gen.delta_p,Temp0
   .
   .
   .
   mulu     Temp0,LP.gen.delta_p,Temp0
   mfsr     Temp1,q
   mtsrim   fc,16
   extract  Temp0,Temp0,Temp1
   mtsr     q,LP.gen.delta_s
   mulu     LP.wid.axial,LP.gen.delta_s,0
   mulu     LP.wid.axial,LP.gen.delta_s,
            LP.wid.axial
   .
   .
   .
   mulu     LP.wid.axial,LP.gen.delta_s,
            LP.wid.axial
   mfsr     Temp1,q
   mtsrim   fc,16
   extract  LP.wid.axial,LP.wid.axial,Temp1
   add      LP.wid.axial,Temp0,LP.wid.axial
   srl      Temp2,LP.wid.axial,16
   sll      LP.wid.axial,LP.wid.axial,16
   mtsr     q,LP.wid.axial
   div0     Temp1,Temp2
   div      Temp1,Temp1,Temp0
```

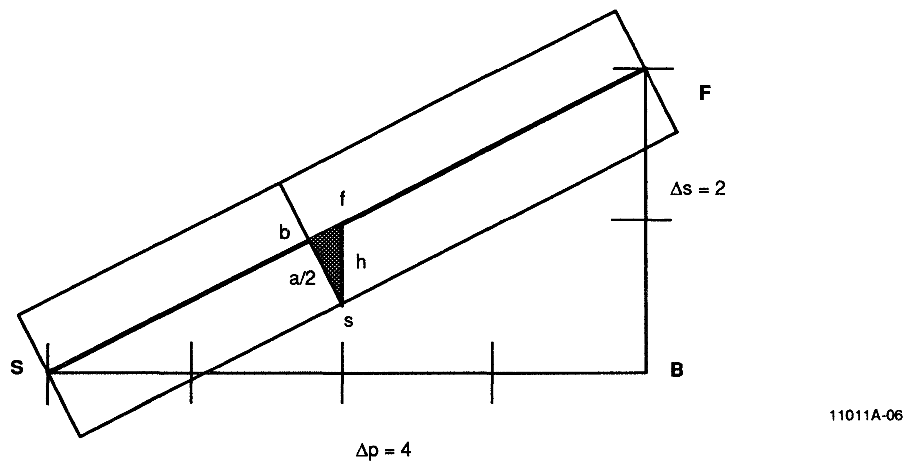


Figure 6. Axial Half-Width

```
        .
        .
        .
    div     Temp1,Temp1,Temp0
    divl    Temp1,Temp1,Temp0
    mfsr    LP.wid.axial,q
    clz     Temp0,LP.wid.axial
    subr    Temp0,Temp0,32
    srl     Temp0,Temp0,1
    srl     Temp1,LP.wid.axial,Temp0
    add     Temp0,Temp1,0
```

There are two triangles, SBF and the smaller sbf, each identified by its three vertices. Angle B and angle b are right angles. Sides FB and fs are vertical and are therefore parallel. Angle F and angle f are equal because they are formed by parallel lines intersected by a common line. The triangles are similar because they have two (and therefore three) congruent angles. Because the triangles are similar, corresponding sides must be in proportion. That is, the hypotenuses are proportional to the sides sb and SB.

$$\frac{h}{\Delta p^2 + \Delta s^2} = \frac{a/2}{\Delta p}$$

Rearranging, and moving the bottom Δp inside the radical for convenience, yields the equation:

$$h = \frac{a \bullet \sqrt{\dfrac{\Delta p^2 + \Delta s^2}{\Delta p^2}}}{2}$$

The equation is evaluated in the following manner. The square of delta_p is calculated, and the result is left in *Temp0*. The square of delta_s is calculated, and the result is left in *LP.wid.axial*. The sum of the squares is calculated, shifted left 16 bit positions, and divided by delta_p squared. The result will be a real number between 1 and 2, with the radix point between bit positions 15 and 16.

The square root of the quotient is calculated iteratively at L_07. The square root is multiplied by the actual line width and the result is divided by two to get the half-width. This is left in *LP.wid.axial*.

```
L_07:
    mtsr    q,LP.wid.axial
    div0    Temp2,0
    div     Temp2,Temp2,Temp1
        .
        .
        .
    div     Temp2,Temp2,Temp1
    divl    Temp2,Temp2,Temp1
    mfsr    Temp2,q
    add     Temp2,Temp2,Temp1
    srl     Temp2,Temp2,1
    cpeq    Temp0,Temp2,Temp0
    jmpt    Temp0,L_08
    add     Temp0,Temp1,0
    cpeq    Temp1,Temp2,Temp1
    jmpf    Temp1,L_07
    add     Temp1,Temp2,0
```

The slope of the line is calculated by dividing the secondary delta times 256 by the primary delta. The quotient is left in *LP.gen.slope*, and the remainder is left in *LP.gen.x_slope*. The primary error term is set to the negative of the primary delta.

The loop count is calculated by subtracting one from the primary delta, and the generation loop is entered at the middle. The first point is always drawn and is centered in the line.

```
L_08:
    mtsr    q,GP.wid.actual
    mulu    Temp0,Temp2,0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mulu    Temp0,Temp2,Temp0
    mfsr    Temp1,q
    mtsrim  fc,7
    extract LP.wid.axial,Temp0,Temp1
    sll     Temp0,LP.gen.delta_s,8
    mtsr    q,Temp0
    div0    Temp1,0
    div     Temp1,Temp1,LP.gen.delta_p
        .
        .
        .
    div     Temp1,Temp1,LP.gen.delta_p
    divl    Temp1,Temp1,LP.gen.delta_p
    divrem  LP.gen.x_slope,Temp1,
            LP.gen.delta_p
    mfsr    LP.gen.slope,q
    subr    LP.gen.x_error,LP.gen.delta_p,0
    jmp     L_11
    sub     LP.gen.count,LP.gen.delta_p,1
```

The generation loop begins at L_09. The slope is added to the error term. If the *LP.gen.x.error* is negative, the primary delta is subtracted from it, and the error term is incremented by 1.

```
L_09:
    jmpt    LP.gen.x_error,L_10
    add     LP.gen.error,LP.gen.error,
```

```
        LP.gen.slope
sub     LP.gen.x_error,LP.gen.x_error,
        LP.gen.delta_p
add     LP.gen.error,LP.gen.error,1
```

At label L_10, the position of the next optimum pixel is calculated. This calculation is carried out with a resolution of 1/256 of a real pixel. The primary movement is added into the address. If a combined movement is required, the error is reduced by 256 (one full pixel), and the secondary movement is added in.

```
L_10:
  add     LP.gen.addr,LP.gen.addr,
          LP.gen.move_p
  cpge    Temp2,LP.gen.error,128
  jmpf    Temp2,L_11
  nop
  sub     LP.gen.error,LP.gen.error,128
  sub     LP.gen.error,LP.gen.error,128
  add     LP.gen.addr,LP.gen.addr,
          LP.gen.move_s
```

At label L_11, the distances the line will extend to either side of the optimum pixel are computed. Figure 7 illustrates this and shows why it is important to calculate the axial half-width so precisely. When the generation loop is calculating pixel addresses, it always chooses exact pixels in the primary direction. That is, each position it chooses in the primary direction maps to an actual pixel location in the primary direction. It almost never maps to an actual pixel location in the secondary direction, except at the end points. By calculating the width of the line in the secondary direction very precisely, the routine is able to determine with equal precision the portion of the "outside" pixels that are covered.

```
L_11:
  add     LP.wid.side_1,LP.wid.axial,0
  add     LP.wid.side_2,LP.wid.axial,0
  add     LP.gen.cover,LP.gen.error,128
  sub     LP.wid.side_1,LP.wid.side_1,
          LP.gen.cover
  jmpf    LP.wid.side_1,L_12
  subr    Temp2,LP.gen.error,128
  add     LP.gen.cover,LP.gen.cover,
          LP.wid.side_1
```

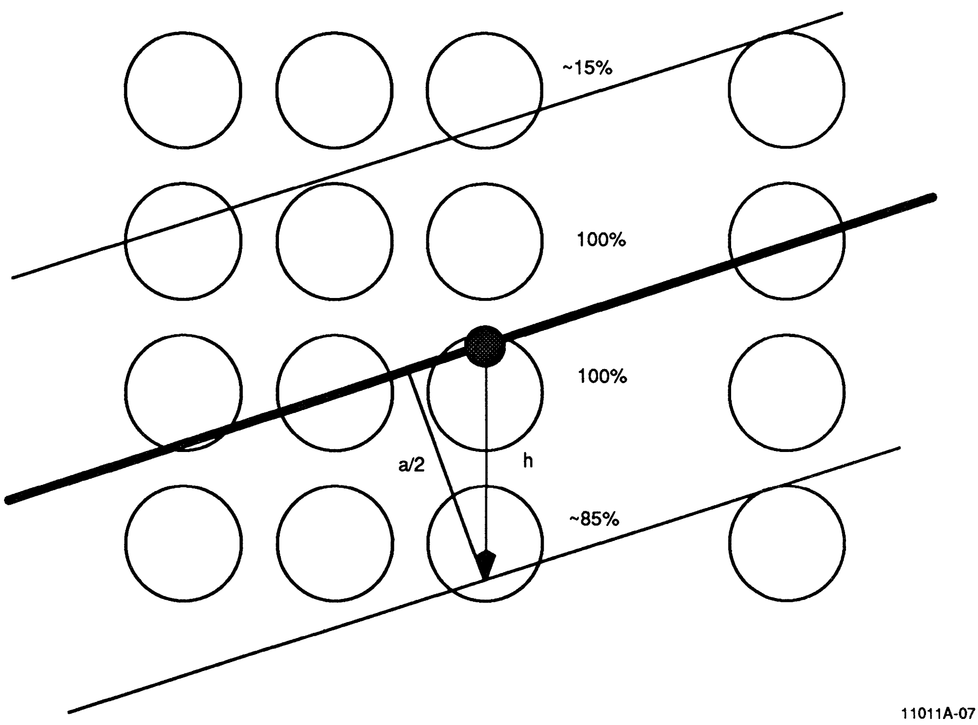


Figure 7. Pixel Coverage

In Figure 7, each actual pixel is shown as a circle. On a real monitor, the area illuminated by each pixel may be larger or smaller, depending on the intensity and focus.

The generation loop has chosen the shaded area as the point in the secondary direction that corresponds to the column chosen in the primary direction. (The primary direction is horizontal.) The column of pixels to the right of the column of interest has been deleted to make room for the coverage values. The actual width has been set to 3. The axial half-width with the indicated slope is something more than 1.5 but less than 2.0. The two pixels closest to the ideal point will be completely covered by the line. The next pixel down will be well over 50 percent covered by the line, perhaps 85 percent. The pixel at the top will be only somewhat covered, perhaps 15 percent.

```
L_12:
   add     LP.gen.cover,LP.gen.cover,Temp2
   sub     LP.wid.side_2,LP.wid.side_2,
           Temp2
   jmpf    LP.wid.side_2,L_13
   add     LP.loc.addr,LP.gen.addr,0
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side_2
```

At label L_13, the optimum pixel is drawn by calling the user-supplied routine with the address and coverage of the optimum pixel. This pixel may be fully covered or may be only partially covered. There is nothing special about the optimum pixel except that it will always be drawn, regardless of the line width, and will be drawn outside the loops for each of the two sides. The routine determines whether any pixels or partial pixels are required on side 1. If not, it jumps to L_18 to test for pixels on side 2.

```
L_13:
   calli   ret,GP.pxl.op_vec
   cpgt    Temp3,LP.wid.side_1,0
   jmpf    Temp3,L_18
   const   LP.gen.cover,256
```

If pixels are required on side 1, they are drawn in the loop beginning at L_14. For a line in the first octant, these pixels will be below the optimum pixel. The address is decremented by the secondary movement value to find the next pixel in the secondary direction. The width remaining to side 1 is decremented by the value in *LP.gen.cover*. This parameter was initialized to 256 (for full coverage). If reducing the width causes it to be less than zero, it is added into *LP.gen.cover* (this actually reduces *LP.gen.cover*).

```
L_14:
   sub     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   sub     LP.wid.side_1,LP.wid.side_1,
           LP.gen.cover
   jmpf    LP.wid.side_1,L_15
   nop
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side_1
```

At label L_15, the user-supplied routine is called to draw this pixel. The remaining width of side 1 is compared to zero. If it is greater than zero, the routine loops back to L_14 to draw the next pixel. In the example shown in Figure 7, the first pixel on side 1 is 85 percent covered and the routine does not loop back. The coverage is set back to 256 for full coverage.

```
L_15:
   calli   ret,GP.pxl.op_vec
   cpgt    Temp3,LP.wid.side_1,0
   jmpt    Temp3,L_14
   const   LP.gen.cover,256
```

At label L_18, the routine tests the remaining width of side 2 and restores the pixel address back to the optimum pixel. In the example in Figure 7, two pixels will be drawn on side 2 (above the optimum pixel).

```
L_18:
   cpgt    Temp3,LP.wid.side_2,0
   jmpf    Temp3,L_21
   add     LP.loc.addr,LP.gen.addr,0
```

In the loop starting at label L_19, the pixels above the optimum pixel are drawn. The pixel address is adjusted by adding the secondary movement, and the width remaining to side 2 is reduced by the amount the pixel is to be covered. This was previously set to 256 for complete coverage. If the remaining width is reduced to less than zero, the width is added back into the coverage (actually reducing it to below 256 for the outside pixel). In the example in Figure 7, the width will not be reduced to below zero.

```
L_19:
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   sub     LP.wid.side_2,LP.wid.side_2,
           LP.gen.cover
   jmpf    LP.wid.side_2,L_20
   nop
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side_2
```

At label L_20, the user-supplied routine is called with the address and coverage to actually draw the pixel. If the remaining width is greater than zero, the routine loops back to L_19 to draw the next pixel. The coverage is set to 256. In the example, the routine will loop once after drawing the pixel just above the optimum pixel. Every pixel except the outside one will have a coverage value of 256. That is, only the outside pixels are not completely covered.

```
L_20:
  calli   ret,GP.pxl.op_vec
  cpgt    Temp3,LP.wid.side_2,0
  jmpt    Temp3,L_19
  const   LP.gen.cover,256
```

At label L_21, the routine decrements and tests the loop count and adds the slope into the primary error. If more pixels are required in the primary direction—that is, if the line is not complete—the routine continues at label L_09.

```
L_21:
  jmpfdec LP.gen.count,L_09
  add     LP.gen.x_error,LP.gen.x_error,
          LP.gen.x_slope
```

An example of a user-supplied drawing routine is in the file **O2_02.S**, which is included on the distribution diskette. This is an almost trivial example that scales the input value to the top of the word and stores it. That is, the illumination of a pixel is linearly related to its coverage. Lines drawn with such a function will exhibit the "barberpole" effect.

The pixel operation routine can also use the coverage value as an independent variable in a more complicated anti-aliasing function. It can also correct for specific hardware differences, such as the actual shape or size of the pixel, or be used to scale color model component intensities for color anti-aliasing. Of course, these more complex operations require more computation, with a corresponding reduction in overall drawing speed.

Wide lines can be drawn without anti-aliasing by supplying a routine that ignores the coverage input and just draws with the current pixel color. Logical or arithmetic operations could be used as well.

P_L3_02.S

The routine **P_L3_02.S** is used to draw anti-aliased wide lines with user-supplied writing and anti-aliasing functions. The routine can also be used to draw wide lines without anti-aliasing. Clipping is also performed.

The actual bit map operations are done by routines supplied by the calling function. One of the parameters supplied by the calling routine to **P_L3_02.S** is the address of a user-written routine. The input parameters are the address of the pixel location and the required "coverage" of that pixel.

The form of the coverage parameter is an integer in the range 1 through 256 inclusively, indicating the portion of the pixel (in 256ths) that is actually covered by the line. Alternatively, it can be thought of as a real number in the range 1/256 to 1, with the radix point between bit 7 and bit 8.

The routine begins with the normal global functions.

Fifteen parameters are loaded from the structure G29K_Params.

```
GP.wnd.min_x            lr2
GP.wnd.max_x            lr3
GP.wnd.min_y            lr4
GP.wnd.max_y            lr5
GP.pxl.value            lr6
GP.mem.width            lr7
GP.mem.depth            lr8
GP.wnd.base             lr9
GP.wnd.align            lr10
GP.pxl.op_vec           lr11
GP.pxl.in_mask          lr12
GP.pxl.do_mask          lr13
GP.pxl.do_value         lr14
GP.pxl.out_mask         lr15
GP.wid.actual           lr16
const   Temp0,_G29K_Params
consth  Temp0,_G29K_Params
mtsrim  cr,(15 - 1)
loadm   0,0,GP.wnd.min_x,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x*, *Start.y* pair to a linear address. The linear address is returned in *LP.loc.addr*.

```
add     LP.loc.x,Start.x,0
call    ret,S_M1_01
add     LP.loc.y,Start.y,0
sub     LP.gen.delta_p,Finish.x,
        Start.x
jmpf    LP.gen.delta_p,L_01
sub     LP.gen.delta_s,Finish.y,
        Start.y
subr    LP.gen.delta_p,LP.gen.delta_p,0
constn  LP.gen.move_p,-PIXEL_SIZE
subr    LP.gen.p,Start.x,0
subr    LP.gen.min_p,GP.wnd.max_x,0
jmp     L_02
subr    LP.gen.max_p,GP.wnd.min_x,0
```

The routine assumes that the primary direction will be x and that the secondary direction will be y. The primary delta is calculated by subtracting *Start.x* from *Finish.x*. If the result is greater than or equal to zero, the code at L_01 sets the primary movement value to the distance between pixels (moving from left to right), sets the primary clipping point to *Start.x*, and sets the primary minimum and maximum clipping boundaries to the minimum and maximum window values, respectively. The secondary delta is calculated by subtracting *Start.y* from *Finish.y*.

```
L_01:
   const    LP.gen.move_p,PIXEL_SIZE
   add      LP.gen.p,Start.x,0
   add      LP.gen.min_p,GP.wnd.min_x,0
   add      LP.gen.max_p,GP.wnd.max_x,0
```

If *Finish.x* is less than *Start.x*, the line will be drawn from right to left. The primary delta is negated, and the primary movement value is set to the distance between pixels (from right to left). The primary clipping point is set to the negative of *Start.x*, and the minimum and maximum clipping boundaries are set to the maximum and minimum x window values, respectively. The secondary delta is calculated by subtracting *Start.y* from *Finish.y*.

At label L_02, the secondary values are calculated from the y-axis inputs. If *Finish.y* is greater than or equal to *Start.y*, the routine continues at label L_03. It sets the secondary movement value to the negative of the memory width (moving from bottom to top) and sets the current secondary clipping position to *Start.y*. The secondary minimum and maximum clipping boundaries are set to the minimum and maximum y window values, respectively. The general address is initialized to the beginning address.

```
L_02:
   jmpf     LP.gen.delta_s,L_03
   add      LP.gen.addr,LP.loc.addr,0
   subr     LP.gen.delta_s,LP.gen.delta_s,0
   add      LP.gen.move_s,GP.mem.width,0
   subr     LP.gen.s,Start.y,0
   subr     LP.gen.min_s,GP.wnd.max_y,0
   jmp      L_04
   subr     LP.gen.max_s,GP.wnd.min_y,0

L_03:
   subr     LP.gen.move_s,GP.mem.width,0
   add      LP.gen.s,Start.y,0
   add      LP.gen.min_s,GP.wnd.min_y,0
   add      LP.gen.max_s,GP.wnd.max_y,0
```

If *Finish.y* is less than *Start.y*, the line will be drawn from top to bottom. The secondary delta is negated, and the secondary movement value is set to the memory width. The current secondary clipping position is set to the negative of *Start.y*. The secondary minimum and maximum clipping boundaries are set to the maximum and minimum y window values, respectively. The general address is initialized to the beginning address.

At label L_04, the routine determines whether x should be the primary direction. If so, the initial assumption is correct and nothing special need be done. *LP.gen.error* is set to zero.

If the primary direction is to be y, the primary and secondary parameters must be swapped. In particular, the following five pairs of variables are swapped.

```
LP.gen.delta_p    LP.gen.delta_s
LP.gen.move_p     LP.gen.move_s
LP.gen.p          LP.gen.s
LP.gen.min_p      LP.gen.min_s
LP.gen.max_p      LP.gen.max_s
```

That is, the deltas and movement values, as well as all the clipping values, are swapped.

```
L_04:
   cpge     Temp0,LP.gen.delta_p,
            LP.gen.delta_s
   jmpt     Temp0,L_05
   const    LP.gen.error,0
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_s,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.delta_p,LP.gen.delta_p,
            LP.gen.delta_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_s,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.move_p,LP.gen.move_p,
            LP.gen.move_s
   xor      LP.gen.p,LP.gen.p,LP.gen.s
   xor      LP.gen.s,LP.gen.p,LP.gen.s
   xor      LP.gen.p,LP.gen.p,LP.gen.s
   xor      LP.gen.min_p,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.min_s,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.min_p,LP.gen.min_p,
            LP.gen.min_s
   xor      LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
   xor      LP.gen.max_s,LP.gen.max_p,
            LP.gen.max_s
   xor      LP.gen.max_p,LP.gen.max_p,
            LP.gen.max_s
```

At label L_05, the axial half-width is calculated. This is done in the same way as in routine **P_L3_01.S**.

```
L_05:
   mtsr     q,LP.gen.delta_p
   mulu     Temp0,LP.gen.delta_p,0
   mulu     Temp0,LP.gen.delta_p,Temp0
   mulu     Temp0,LP.gen.delta_p,Temp0
   mfsr     Temp1,q
   mtsrim   fc,16
   extract  Temp0,Temp0,Temp1
   mtsr     q,LP.gen.delta_s
```

```
    mulu     LP.wid.axial,LP.gen.delta_s,0
    mulu     LP.wid.axial,LP.gen.delta_s,
             LP.wid.axial
    mulu     LP.wid.axial,LP.gen.delta_s,
             LP.wid.axial
     .
     .
     .
    mfsr     Temp1,q
    mtsrim   fc,16
    extract  LP.wid.axial,LP.wid.axial,
             Temp1
    add      LP.wid.axial,Temp0,
             LP.wid.axial
    srl      Temp2,LP.wid.axial,16
    sll      LP.wid.axial,LP.wid.axial,16
    mtsr     q,LP.wid.axial
    div0     Temp1,Temp2
    div      Temp1,Temp1,Temp0
     .
     .
     .
    div      Temp1,Temp1,Temp0
    divl     Temp1,Temp1,Temp0
    mfsr     LP.wid.axial,q
    clz      Temp0,LP.wid.axial
    subr     Temp0,Temp0,32
    srl      Temp0,Temp0,1
    srl      Temp1,LP.wid.axial,Temp0
    add      Temp0,Temp1,0
```

At label L_07, the square root is calculated iteratively.

```
L_07:
    mtsr     q,LP.wid.axial
    div0     Temp2,0
    div      Temp2,Temp2,Temp1
     .
     .
     .
    div      Temp2,Temp2,Temp1
    divl     Temp2,Temp2,Temp1
    mfsr     Temp2,q
    add      Temp2,Temp2,Temp1
    srl      Temp2,Temp2,1
    cpeq     Temp0,Temp2,Temp0
    jmpt     Temp0,L_08
    add      Temp0,Temp1,0
    cpeq     Temp1,Temp2,Temp1
    jmpf     Temp1,L_07
    add      Temp1,Temp2,0
```

At label L_08, the slope is calculated, and then the clipping destinations are initialized. There are a total of six destinations: two in the primary direction and four in the secondary direction. The destinations are loaded into the vector pointers depending on whether the current clipping tests are primary, secondary side 1, or secondary side 2. The actions performed for each case are summarized in Table 6.

```
L_08:
    mtsr     q,GP.wid.actual
    mulu     Temp0,Temp2,0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mulu     Temp0,Temp2,Temp0
    mfsr     Temp1,q
    mtsrim   fc,7
    extract  LP.wid.axial,Temp0,Temp1
    sll      Temp0,LP.gen.delta_s,8
    mtsr     q,Temp0
    div0     Temp1,0
    div      Temp1,Temp1,LP.gen.delta_p
     .
     .
     .
    div      Temp1,Temp1,LP.gen.delta_p
    divl     Temp1,Temp1,LP.gen.delta_p
    divrem   LP.gen.x_slope,Temp1,
             LP.gen.delta_p
    mfsr     LP.gen.slope,q
    subr     LP.gen.x_error,LP.gen.delta_p,0
    const    LP.clp.skip_p,Skip_p
    consth   LP.clp.skip_p,Skip_p
    const    LP.clp.stop_p,Stop_p
    consth   LP.clp.stop_p,Stop_p
    const    LP.clp.skip_s,Skip_s
    consth   LP.clp.skip_s,Skip_s
    const    LP.clp.skip_s_1,Skip_s_1
    consth   LP.clp.skip_s_1,Skip_s_1
    const    LP.clp.stop_s_1,Stop_s_1
    consth   LP.clp.stop_s_1,Stop_s_1
    const    LP.clp.skip_s_2,Skip_s_2
    consth   LP.clp.skip_s_2,Skip_s_2
    const    LP.clp.stop_s_2,Stop_s_2
    consth   LP.clp.stop_s_2,Stop_s_2
    jmp      L_11
    sub      LP.gen.count,LP.gen.delta_p,1
```

Table 6. Clipping Tests

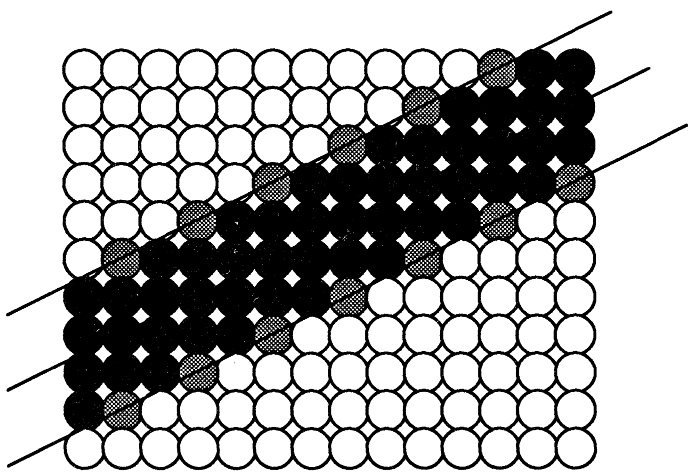
Destination	Vector	Near Label	Action If Outside Window
Skip_p	V_CLIP_SKIP	L_11	Decrement primary count
Stop_p	V_CLIP_STOP	L_11	Exit routine
Skip_s_1	V_CLIP_SKIP	L_15	Test side 1 width
Stop_s_1	V_CLIP_STOP	L_15	Exit side 1 loop
Skip_s_2	V_CLIP_SKIP	L_20	Test side 2 width
Stop_s_2	V_CLIP_STOP	L_20	Exit side 2 loop

The clipping method is an extension of the method described for single-width lines in section "P_L1_02.S." The algorithm generates pixel locations and asserts that the locations are inside the clipping window. If they are inside the clipping window, they are drawn. If they are outside the clipping window, the algorithm goes on to the next pixel (if the current pixel is "before" the window), or exits (if the pixel is "after" the window). In the two-dimensional extension implemented in **P_L3_02.S**, this is done in three phases. The algorithm finds each optimum pixel in the primary direction and does the primary clipping. Then it finds each pixel on side 1 of the optimum pixel (in the secondary direction) and does the first secondary clipping. Finally, it finds each pixel on side 2 of the optimum pixel (in the secondary direction) and does the other secondary clipping. This two-dimensional clipping will result in abrupt ends of lines. Further, the shape of the line end will be different if the line is clipped at a corner of the window. This is shown in Figure 8.

The generation loop begins at label L_09. The slope is added into the error term. If the primary error is not negative, it is reduced by the primary delta, and the error term is incremented by 1.

```
L_09:
   jmpt    LP.gen.x_error,L_10
   add     LP.gen.error,LP.gen.error,
           LP.gen.slope
   sub     LP.gen.x_error,LP.gen.x_error,
           LP.gen.delta_p
   add     LP.gen.error,LP.gen.error,1
```

At label L_10, the primary movement value is added into the address. The primary clipping position is incremented. If the error term is greater than or equal to 128 (0.5), it is decremented by 1, the secondary movement value is added into the address, and the secondary clipping position is incremented.



11011A-08

Figure 8. Wide-Line Clipping

```
L_10:
   add     LP.gen.addr,LP.gen.addr,
           LP.gen.move_p
   cpge    Temp2,LP.gen.error,128
   jmpf    Temp2,L_11
   add     LP.gen.p,LP.gen.p,1
   sub     LP.gen.error,LP.gen.error,128
   sub     LP.gen.error,LP.gen.error,128
   add     LP.gen.addr,LP.gen.addr,
           LP.gen.move_s
   add     LP.gen.s,LP.gen.s,1
```

At label L_11, the optimum pixel location is asserted to be within the primary clipping boundaries. If it is less than the minimum primary clipping boundary, control is transferred to Skip_p because the clipping window has not yet been reached. If it is greater than the maximum primary clipping boundary, control is transferred to Stop_p because the clipping window has just been exited in the primary axis. If the optimum pixel location is inside the primary clipping window, the remaining widths for sides 1 and 2 are set up.

```
L_11:
   add     LP.clp.skip_vec,LP.clp.skip_p,0
   asge    V_CLIP_SKIP,LP.gen.p,
           LP.gen.min_p
   add     LP.clp.stop_vec,LP.clp.stop_p,0
   asle    V_CLIP_STOP,LP.gen.p,
           LP.gen.max_p
   add     LP.wid.side_1,LP.wid.axial,0
   add     LP.wid.side_2,LP.wid.axial,0
   add     LP.gen.try_s,LP.gen.s,0
   add     LP.gen.cover,LP.gen.error,128
   sub     LP.wid.side_1,LP.wid.side_1,
           LP.gen.cover
   jmpf    LP.wid.side_1,L_12
   subr    Temp2,LP.gen.error,128
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side_1

L_12:
   add     LP.gen.cover,LP.gen.cover,
           Temp2
   sub     LP.wid.side_2,LP.wid.side_2,
           Temp2
   jmpf    LP.wid.side_2,L_13
   add     LP.loc.addr,LP.gen.addr,0
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side_2
```

At label L_13, the user-supplied routine is called to draw the optimum pixel. The clipping vectors are set up for side 1.

```
L_13:
   add     LP.clp.skip_vec,LP.clp.skip_s,0
   asle    V_CLIP_SKIP,LP.gen.try_s,
           LP.gen.max_s
   asge    V_CLIP_SKIP,LP.gen.try_s,
           LP.gen.min_s
   calli   ret,GP.pxl.op_vec

Skip_s:
   cpgt    Temp3,LP.wid.side_1,0
   jmpf    Temp3,L_18
   const   LP.gen.cover,256
   add     LP.clp.skip_vec,
           LP.clp.skip_s_1,0
   add     LP.clp.stop_vec,
           LP.clp.stop_s_1,0
```

Label L_14 is the top of the loop that draws pixels on one side (side 1) of the optimum pixel. The algorithm moves away from the optimum pixel in the secondary direction until the remaining width becomes zero or negative. Each pixel is asserted to be within the secondary clipping boundaries at label L_15. If it is greater than the maximum secondary clipping boundary, it is skipped. If it is less than the minimum secondary clipping boundary, the loop exits. If it is within the secondary clipping boundaries, the drawing routine is called with the address and coverage.

```
L_14:
   sub     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   sub     LP.wid.side_1,LP.wid.side_1,
           LP.gen.cover
   jmpf    LP.wid.side_1,L_15
   sub     LP.gen.try_s,LP.gen.try_s,1
   add     LP.gen.cover,LP.gen.cover,
           LP.wid.side

_1L_15:
   asle    V_CLIP_SKIP,LP.gen.try_s,
           LP.gen.max_s
   asge    V_CLIP_STOP,LP.gen.try_s,
           LP.gen.min_s
   calli   ret,GP.pxl.op_vec

Skip_s_1:
   cpgt    Temp3,LP.wid.side_1,0
   jmpt    Temp3,L_14

Stop_s_1:
   const   LP.gen.cover,256
   add     LP.gen.try_s,LP.gen.s,0

L_18:
   cpgt    Temp3,LP.wid.side_2,0
   jmpf    Temp3,Skip_paddLP.loc.addr,
           LP.gen.addr,0
   add     LP.clp.skip_vec,
           LP.clp.skip_s_2,0
```

```
   add      LP.clp.stop_vec,
            LP.clp.stop_s_2,0
```

Label L_19 is the top of the loop that draws pixels on the second side (side 2) of the optimum pixel. This is exactly the same as side 1, except that pixels are drawn on the other side of the optimum pixel.

```
L_19:
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_s
   sub      LP.wid.side_2,LP.wid.side_2,
            LP.gen.cover
   jmpf     LP.wid.side_2,L_20
   add      LP.gen.try_s,LP.gen.try_s,1
   add      LP.gen.cover,LP.gen.cover,
            LP.wid.side

_2L_20:
   asge     V_CLIP_SKIP,LP.gen.try_s,
            LP.gen.min_s
   asle     V_CLIP_STOP,LP.gen.try_s,
            LP.gen.max_s
   calli    ret,GP.pxl.op_vec

Skip_s_2:
   cpgt     Temp3,LP.wid.side_2,0
   jmpt     Temp3,L_19

Stop_s_2:
   const    LP.gen.cover,256

Skip_p:
   jmpfdecLP.gen.count,L_09
   add      LP.gen.x_error,LP.gen.x_error,
            LP.gen.x_slope
Stop_p:
```

P_L4_01.S

The routine **P_L4_01.S** draws single-width lines in a monochrome bit map. The pixels in the bit map are assumed to be packed 32 to a word. Bit 31 of a word is assumed to be displayed to the left of bit 30. Lower-addressed words are displayed to the left of, and above, higher-addressed words.

The caller of **P_L4_01.S** must provide a subroutine to perform the actual writes to the bit map. The user routine is called with the linear address of the 32-bit word in the bit map and a mask with 1s for the pixels within the word that must be written. This is illustrated for a 16-bit word in Figure 9.

If the line being drawn is close to horizontal, multiple pixels will occupy any given word. By accumulating pixels until the line enters the "next" word, memory accesses can be amortized over several pixels. This does not help with lines that are closer to vertical than 45°, and is not used for lines where the primary axis is y.

The routine begins with the normal global functions.

Ten parameters are loaded from the structure G29K_Params.

```
GP.pxl.value        lr6
GP.mem.width        lr7
GP.mem.depth        lr8
GP.wnd.base          lr9
GP.wnd.align        lr10
GP.pxl.op_vec       lr11
GP.pxl.in_mask      lr12
GP.pxl.do_mask      lr13
GP.pxl.do_value     lr14
GP.pxl.out_mask      lr15
const    Temp0,_G29K_Params + 4 * 4
consth   Temp0,_G29K_Params + 4 * 4
mtsrim   cr,(10 - 1)
loadm    0,0,GP.pxl.value,Temp0
```

Routine **S_M1_01** is called to convert the *Start.x*, *Start.y* pair to a linear address, which is returned in *LP.loc.addr*. The bit position within the word is indicated by *LP.loc.align*, which is the number of bits to the left of the addressed bit within the addressed word.

```
add      LP.loc.x,Start.x,0
call     ret,S_M1_01
add      LP.loc.y,Start.y,0
cpeq     LP.gen.cover,LP.gen.cover,
         LP.gen.cover
srl      LP.gen.cover,LP.gen.cover,
         LP.loc.align
```

This routine assumes that the primary direction will be x, and the secondary direction will be y. The primary delta is calculated by subtracting *Start.x* from *Finish.x*. The sign bit is left in the error term for reversibly retraceable lines. The primary movement value is set to 4. If *Finish.x* is less than *Start.x*, the primary delta is negated, and the primary movement value is set to negative 4.

```
sub      LP.gen.delta_p,Finish.x,Start.x
sra      LP.gen.error,LP.gen.delta_p,31
jmpf     LP.gen.delta_p,L_01
const    LP.gen.move_p,4
subr     LP.gen.delta_p,LP.gen.delta_p,0
constn   LP.gen.move_p,-4
```

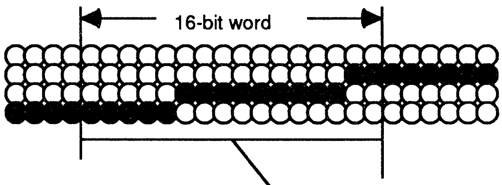


Figure 9. Monochrome Lines

At label L_01, the secondary delta is computed by subtracting *Start.y* from *Finish.y*. The secondary movement value is set to the negative of memory width. If *Finish.y* is less than *Start.y*, the secondary delta is negated, and the secondary movement value is set to the negative of memory width.

```
L_01:
   sub     LP.gen.delta_s,Finish.y,
           Start.y
   jmpf    LP.gen.delta_s,L_02
   subr    LP.gen.move_s,GP.mem.width,0
   subr    LP.gen.delta_s,LP.gen.delta_s,0
   add     LP.gen.move_s,GP.mem.width,0
L_02:
   cpge    Temp0,LP.gen.delta_p,
           LP.gen.delta_s
   jmpt    Temp0,L_03
   cpeq    LP.gen.try_s,LP.gen.try_s,
           LP.gen.try_s
   xor     LP.gen.delta_p,LP.gen.delta_p,
           LP.gen.delta_s
   xor     LP.gen.delta_s,LP.gen.delta_p,
           LP.gen.delta_s
   xor     LP.gen.delta_p,LP.gen.delta_p,
           LP.gen.delta_s
   xor     LP.gen.move_p,LP.gen.move_p,
           LP.gen.move_s
   xor     LP.gen.move_s,LP.gen.move_p,
           LP.gen.move_s
   xor     LP.gen.move_p,LP.gen.move_p,
           LP.gen.move_s
   const   LP.gen.try_s,0
L_03:
   sll     LP.gen.slope,LP.gen.delta_s,1
   sll     LP.gen.x_slope,LP.gen.delta_p,1
   add     LP.gen.error,LP.gen.error,
           LP.gen.slope
   jmpt    LP.gen.try_s,L_11
   sub     LP.gen.error,LP.gen.error,
           LP.gen.delta_p
   jmpt    LP.gen.move_s,L_07
   add     LP.gen.move_s,LP.gen.move_p,
           LP.gen.move_s
   mtsrim  fc,31
   sub     LP.gen.count,LP.gen.delta_p,1
   sub     LP.gen.slope,LP.gen.slope,
           LP.gen.x_slope
```

The deltas are compared to test the initial assumption that x is the primary direction. If the primary direction is y, the primary and secondary deltas and movement values must be swapped. A flag is set in *LP.gen.try_s* to indicate the primary axis. This will be used to choose a loop. The primary and secondary error increments are calculated.

These error increments are derived from those described in Bresenham's algorithm. The initial error term is calculated. The vector can now be drawn.

There are actually four loops (see Table 7). One will be chosen depending on the primary axis, and depending on the direction in which the line is to be drawn.

Table 7. Loop Top and Line Direction

Primary	Direction	Loop Top
y-axis	Positive x	L_04
y-axis	Negative x	L_07
x-axis	Positive y	L_11
x-axis	Negative y	L_16

The loop for the case where the primary axis is the y axis and movement is in the positive x direction begins at label L_04. The funnel-count register is set to 31 so that the extract between label L_04 and label L_05 actually cycles *LP.gen.cover* (the pixel mask) 1 bit to the right.

At label L_04, the error is tested and incremented by the slope. If movement in the secondary direction is not necessary, the jump to L_05 is taken. The pixel is written using the current mask (*LP.gen.cover*), and *LP.gen. x_slope* is added to the error term. At label L_06, the loop count is decremented and tested, and the primary movement is added to the location.

```
L_04:
   jmpt    LP.gen.error,L_05
   add     LP.gen.error,LP.gen.error,
           LP.gen.slope
   calli   ret,GP.pxl.op_vec
   cpeq    LP.loc.align,LP.gen.cover,1
   jmpf    LP.loc.align,L_06
   extract LP.gen.cover,LP.gen.cover,
           LP.gen.cover
   jmpfdec LP.gen.count,L_04
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   jmp     L_22
   nop
L_05:
   calli   ret,GP.pxl.op_vec
   add     LP.gen.error,LP.gen.error,
           LP.gen.x_slope
L_06:
   jmpfdec LP.gen.count,L_04
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_p
   jmp     L_22
   nop
```

If movement in the secondary direction is needed, the current pixel is written, and the mask is compared to the value 1. If the mask is not equal to 1, it can be right-shifted and remain in the same word. If this is the case, the jump to L_06 is taken. The loop count is decremented and tested, and the address is modified by the primary movement value.

If the mask is 1, it is at the right edge of the word, and it is necessary to change the address into the next secondary word. The jump to L_06 is not taken, and the address is modified by the primary movement value.

It is not possible to complete this loop without modifying the address with one movement value or the other. This is reasonable, since no word ever has more than a single pixel written.

The case where the primary direction is y and there is negative movement in x is essentially the same. The *Funnel Count Register* is set to 1 so that the extract is a left cycle of 1 bit.

```
L_07:
   mtsrim  fc,1
   sub     LP.gen.count,LP.gen.delta_p,1
L_08:
   jmpt    LP.gen.error,L_09
   add     LP.gen.error,LP.gen.error,
           LP.gen.slope
   calli   ret, GP.pxl.op_vec
   sub     LP.gen.error,LP.gen.error,
           LP.gen.x_slope
   jmpf    LP.gen.cover,L_10
   extract LP.gen.cover,LP.gen.cover,
           LP.gen.cover
   jmpfdec LP.gen.count,L_08
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   jmp     L_22
   nop
L_09:
   calli   ret, GP.pxl.op_vec
   nop
L_10:
   jmpfdec LP.gen.count,L_08
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_p
   jmp     L_22
   nop
```

Label L_11 begins the case where primary movement is along the x axis. In this case, multiple pixels can be written into a word. If positive y-axis movement is required, the loop at L_12 is used.

```
L_11:
   jmpt    LP.gen.move_p,L_16
   add     LP.loc.align,LP.gen.cover,0
   mtsrim  fc,31
   sub     LP.gen.x_slope,LP.gen.slope,
           LP.gen.x_slope
   sub     LP.gen.count,LP.gen.delta_p,1
   jmpt    LP.gen.count,L_21
   sub     LP.gen.count,LP.gen.count,1
L_12:
   jmpt    LP.gen.error,L_13
   extract LP.loc.align,LP.loc.align,
           LP.loc.align
   calli   ret,GP.pxl.op_vec
   add     LP.gen.error,LP.gen.error,
           LP.gen.x_slope
   jmpt    LP.loc.align,L_14
   add     LP.loc.addr,LP.loc.addr,
           LP.gen.move_s
   jmpfdec LP.gen.count,L_12
   add     LP.gen.cover,LP.loc.align,0
   jmp     L_21
   nop
```

The Funnel Count register is set to 31 so that the extract just past L_12 will be a right cycle of 1 bit. The loop count is set to the primary delta minus one; if this is negative, only a single pixel is written at L_21.

At label L_12, the error term is tested, and *LP.loc.align* is right-cycled 1 bit. This register always has a single 1, and is right-cycled 1 bit for every pass through the loop.

If *LP.gen.error* is positive, control passes to L_13. Here, *LP.loc.align* is tested to determine whether the single 1 is in bit 31. The error term is adjusted for a primary move. If the bit in *LP.loc.align* is not in position 31, the pixel is still in the same word. Control passes to label L_15, where the loop count is decremented and tested, and the new bit is ORed into the mask, *LP.gen.cover*. This continues until either a movement in the secondary direction is needed, in which case the write function is called just past label L_12, or a movement in the primary direction is needed, in which case the write function is called just past label L_13.

```
L_13:
   jmpf     LP.loc.align,L_15
   add      LP.gen.error,LP.gen.error,
            LP.gen.slope
   calli    ret,GP.pxl.op_vec
   nop
L_14:
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_p
   const    LP.gen.cover,0
L_15:
   jmpfdec LP.gen.count,L_12
   or       LP.gen.cover,LP.gen.cover,
            LP.loc.align
   jmp      L_21
   nop
```

Label L_16 begins the case where primary movement is along the x axis, and negative movement is needed in the y axis. It is essentially similar to the code at L_12, except that the single 1 cycles to the left rather than the right.

```
L_16:
   mtsrim   fc,1
   sub      LP.gen.count,LP.gen.delta_p,1
   jmpt     LP.gen.count,L_21
   sub      LP.gen.count,LP.gen.count,1
L_17:
   jmpt     LP.gen.error,L_19
   add      LP.gen.error,LP.gen.error,
            LP.gen.slope
   jmpf     LP.loc.align,L_18
   sub      LP.gen.error,LP.gen.error,
            LP.gen.x_slope
   calli    ret,GP.pxl.op_vec
   extract LP.loc.align,LP.loc.align,
            LP.loc.align
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_p
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_s
   jmpfdec LP.gen.count,L_17
   add      LP.gen.cover,LP.loc.align,0
   jmp      L_21
   nop
L_18:
   calli    ret,GP.pxl.op_vec
   extract LP.loc.align,LP.loc.align,
            LP.loc.align
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_s
   jmpfdec LP.gen.count,L_17
   add      LP.gen.cover,LP.loc.align,0
   jmp      L_21
   nop
L_19:
   jmpf     LP.loc.align,L_20
   extract LP.loc.align,LP.loc.align,
            LP.loc.align
   calli    ret,GP.pxl.op_vec
   nop
   add      LP.loc.addr,LP.loc.addr,
            LP.gen.move_p
   const    LP.gen.cover,0
L_20:
   jmpfdec LP.gen.count,L_17
   or       LP.gen.cover,LP.gen.cover,
            LP.loc.align
L_21:
   calli    ret,GP.pxl.op_vec
   nop
```

S_M1_01.S

The routine **S_M1_01.S** is an internal subroutine that calculates the linear address of a pixel from the x and y coordinates. It is not a C-language callable routine.

Figure 10 is a diagram of the memory system. Location zero is shown at the top, and machine addresses increase from top to bottom. This is also the way the bit map is expected to be displayed.

The routines are written so that higher y coordinates are written into lower machine addresses and therefore displayed higher on the screen. Higher x coordinates are shown further to the right on the screen.

The memory width is multiplied times the y address, and the result is left in *LP.loc.addr*. It is negated since it must be subtracted from the window base in order to appear higher on the screen. *GP.mem.depth* is then tested to determine whether the bit map is monochrome.

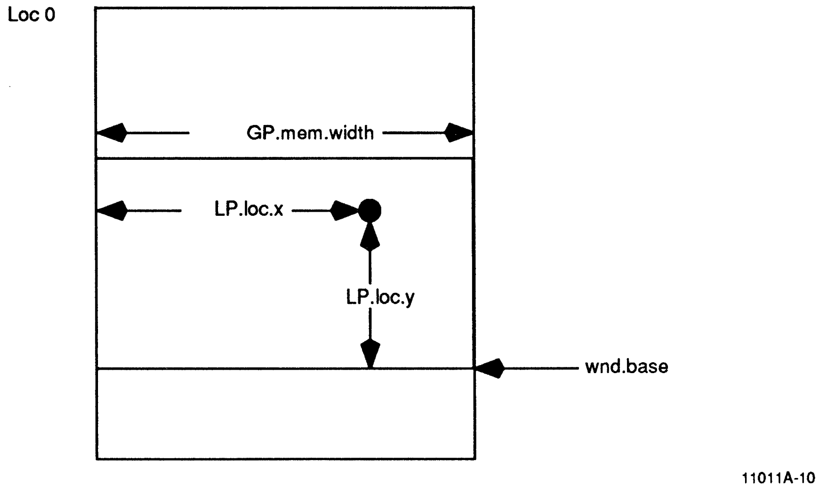


Figure 10. X–Y Translation

If the bit map is not monochrome, the jump is not taken. The x pixel location is multiplied by the number of bytes per pixel (1, 2, 4), and the result is added into *LP.loc.addr*. Then the window base is added into *LP.loc.addr*, and the routine exits. The variable *LP.loc.align* is left with the low-order 2 bits of the address times eight. This indicates which byte or half-word is being addressed in bit maps with 8 or 16 bits per pixel, in a manner similar to the bit position indicator for a monochrome display.

If the bit map is monochrome, the jump to label $01 is taken. The window alignment (offset into a word) is added to the x location, and the low-order 5 bits are left in *LP.loc.align*. This indicates the bit within the word that corresponds to the pixel coordinates: the number of bits to the left of the addressed bit. The byte address corresponding to the x location is added to *LP.loc.addr*, and the window base is added into *LP.loc.addr*. The routine exits.

S_C1_01.S

There are three routines in file **S_C1_01.S**; they are **_S_C1_01**, **S_C2_01**, and **S_C2_02**.

Routine **_S_C1_01** (which is contained in Appendix A) loads the clipping trap vectors to point to their handlers. The code assumes that the Vector Fetch (VF) bit in the *Configuration Register* (*CFG*) is 1, which means that the Vector Area is a table of vectors.

The address of **S_C2_01** is stored into absolute location decimal 400, and the address of **S_C2_02** is stored into absolute location decimal 404. Since the R bit(s) are not set, the routines are expected to be in instruction/data memory.

When a Clipping Skip Trap occurs, control is transferred to routine **S_C2_01**. It executes in freeze mode and supervisor mode. The routine sets *pc1* to the value contained in *LP.clp.skip_vec*, and sets *pc0* to the value contained in *LP.clp.skip_vec* + 4. It exits with an IRET instruction. This transfers control to the skipping destination in the rendering routine.

When a Clipping Stop Trap occurs, control is transferred to routine **S_C2_02**. It executes in freeze mode and supervisor mode. One expects that an operating system would normally handle this. The routine sets *pc1* to the value contained in *LP.clp.stop_vec* and sets *pc0* to the value contained in *LP.clp.stop_vec* + 4. It exits with an IRET instruction. This transfers control to the stopping destination in the rendering routine.

This method of clipping is most efficient if not much clipping is done. In the non-clipped case, it requires only four cycles for the four ASSERT instructions for each pixel.

A skip-if-clipped exception is fairly expensive in terms of cycles, requiring the following operations for each skip:

1. ASSERT instruction.
2. Fetch vector.
3. Fetch and execute trap.
4. Restart at skip destination.

If many skips are expected, it may be better to replace the skip asserts with explicit compare/jump combinations. In this case, each skip test will require only two cycles per pixel, regardless whether clipping occurs.

Since the Stop Trap can occur only once per object, it is probably most efficient to implement it with an ASSERT instruction, as is done here.

Copy Block Routines

There are a total of eight functions for Copy Block, which are listed in Table 8.

Table 8. Copy Block Routines

Routine	Function
P_B1_01.S	Color, copy only, no clipping
P_B1_02.S	Color, copy only, clipping
P_B2_01.S	Color, general operation, no clipping
P_B2_02.S	Color, general operation, clipping
P_B3_01.S	Monochrome, copy only, no clipping
P_B3_02.S	Monochrome, copy only, clipping
P_B4_01.S	Monochrome, general operation, no clipping
P_B4_02.S	Monochrome, general operation, clipping

All Copy Block routines use the Am29000 load- and store-multiple instructions, proceeding one scan line at a time. As many pixels as will fit into a reserved block of registers are fetched with a Load Multiple, and are placed in the destination area with a Store Multiple. The use of Load and Store Multiple instructions allows the memory system to run at maximum speed. If the memory system supports burst-mode reads and writes, it is possible to load or store a 32-bit word every cycle.

The size of the block of reserved registers is set by a pair of .equ statements in **G29K_REG.h**. The constant MAX_SHIFT must be set to correspond to the constant MAX_WORDs, as indicated in Table 9. Values other than those in the Table will not work.

Table 9. Block Size Values

MAX_WORDS	MAX_SHIFT
32	5
16	4
8	3

If MAX_WORDS is set to any value other than 32, the Byte Pointer register arrays should be compressed, and the BLOCK_PRIMITIVE .equ statement in g29K_reg.h should be adjusted to reflect the change. If this is not done, there is no reason to change the .equ statements.

The tradeoff here is between the potential for a spill/fill in the case where MAX_WORDS is large, and the potential for fewer Load-Multiple/Store-Multiple instructions where MAX_WORDS is small. If there are only a few scan lines to be moved, then perhaps the extra Load-Multiple/Store-Multiple overhead is not worth the spill/fill. If there are many scan lines to be moved, then the spill/fill will be better amortized.

These routines do not check for overlapping blocks and always proceed from lower addresses to higher addresses.

All the copy routines begin with the same declarations. The function name is declared to be global, the ENTER macro is used to specify that 111 general registers are required, and the routine name appears as a label.

```
  .global_P_B1_01
  ENTER  BLOCK_PRIMITIVE
_P_B1_01:
```

The eight parameter register names are declared with PARAM macros. These assign local register numbers above (higher) than the registers previously defined. These parameters are passed in the local registers shown below:

Macro	Register Name	Register Number
PARAM	Dest.x	lr111
PARAM	Dest.y	lr112
PARAM	Size.x	lr113
PARAM	Size.y	lr114
PARAM	Source.x	lr115
PARAM	Source.y	lr116
PARAM	Source.b	lr117
PARAM	Source.w	lr118

The CLAIM macro is the function prologue. If a spill operation is not necessary, this requires five instructions. If a spill is necessary, this is the standard **SPILL** routine, and may involve a Load/Store Multiple instruction.

P_B1_01.S

Routine **P_B1_01.S** is a C-language callable program that performs a copy block operation in a color (32-plane) bit map. No clipping is performed. This routine is optimized for moving data in deep bit maps.

The routine begins with the normal global functions.

Four parameters are loaded from the structure G29K_Params.

```
GP.mem.width lr7
GP.mem.depth lr8
GP.wnd.base  lr9
GP.wnd.align lr10
const   Templ,_G29K_Params+(5*4)
consth  Templ,_G29K_Params+(5*4)
mtsrim  cr,(4 - 1)
loadm   0,0,GP.mem.width,Templ
```

The routine checks to make sure the size of the block is non-negative in either dimension. If it is negative, it exits immediately.

```
sub     Size.x,Size.x,1
jmpt    Size.x,L_04
```

```
sub      Size.y,Size.y,1
jmpt     Size.y,L_04
sub      Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr.*

```
add      LP.loc.x,Dest.x,0
call     ret,S_M1_01
add      LP.loc.y,Dest.y,0
add      BP.dst.lft_addr,LP.loc.addr,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The source bit-map width and base can be different from the global bit-map width and base. The linear address of the source is left in *BP.src.lft_addr.* This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps respectively. They will be modified at the end of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
add      Temp1,GP.mem.width,0
add      GP.mem.width,Source.w,0
add      GP.wnd.base,Source.b,0
add      LP.loc.x,Source.x,0
call     ret,S_M1_01
add      LP.loc.y,Source.y,0
add      BP.src.lft_addr,LP.loc.addr,0
add      GP.mem.width,Temp1,0
```

The number of groups per scan line is calculated by right shifting the x dimension of the block size. The number of pixels in the first (or only) group is calculated.

```
and      BP.fst.count,Size.x,
         (MAX_WORDS - 1)
srl      BP.grp.repeat,Size.x,
         MAX_SHIFT
sub      BP.grp.repeat,BP.grp.repeat,1
sll      BP.fst.incr,BP.fst.count,2
add      BP.fst.incr,BP.fst.incr,4
```

The movement of each scan line begins at label L_01. The current values of the group count and source and destination addresses are copied into working registers.

```
L_01:
  add      BP.grp.count,BP.grp.repeat,0
  add      BP.dst.addr,BP.dst.lft_addr,0
  add      BP.src.addr,BP.src.lft_addr,0
  mtsr     cr,BP.fst.count
  loadm    0,0,BP.dst.array,BP.src.addr
  mtsr     cr,BP.fst.count
  jmpt     BP.grp.count,L_03
  storem   0,0,BP.dst.array,BP.dst.addr
  add      BP.dst.addr,BP.dst.addr,
           BP.fst.incr
  add      BP.src.addr,BP.src.addr,
           BP.fst.incr
  sub      BP.grp.count,BP.grp.count,1
```

The first (or only) group of the scan line is loaded and then stored. If the first group is the only group, the code jumps to L_03. Otherwise the source and destination addresses are incremented by the number of bytes in the first group. The group count is decremented.

The loop that moves the second and subsequent group for each scan line begins at label L_02. Each group moved in the loop will be the maximum size. The odd group, if any, was moved first. The group is loaded into the register block and the source address is incremented. The group is stored from the register block and the destination address is incremented. The loop count is decremented and tested.

```
L_02:
  mtsrim cr,(MAX_WORDS - 1)
  loadm    0,0,BP.dst.array,BP.src.addr
  add      BP.src.addr,BP.src.addr,
           (MAX_WORDS * 4)
  mtsrim cr,(MAX_WORDS - 1)
  storem   0,0,BP.dst.array,BP.dst.addr
  jmpfdec BP.grp.count,L_02
  add      BP.dst.addr,BP.dst.addr,
           (MAX_WORDS * 4)
```

At label L_03, the pixels for a scan line have all been moved. The left edge addresses, *BP.dst.lft_addr* and *BP.src.lpt_addr* are incremented by the width of the respective bit maps and the scan line count is decremented and tested.

```
L_03:
  add      BP.dst.lft_addr,
           BP.dst.lft_addr,GP.mem.width
  jmpfdec Size.y,L_01
  add      BP.src.lft_addr,
           BP.src.lft_addr,Source.w
```

P_B1_02.S

Routine **P_B1_02.S** is a C-language callable program that performs a copy-block operation in a color (32-plane) bit map. Clipping is performed. This routine is optimized for moving data in deep bit maps.

The routine begins with the normal global functions.

Nine parameters are loaded from the structure G29K_Params.

```
GP.wnd.min_x          lr2
GP.wnd.max_x          lr3
GP.wnd.min_y          lr4
GP.wnd.max_y          lr5
GP.pxl.value          lr6
```

```
GP.mem.width                lr7
GP.mem.depth                lr8
GP.wnd.base                 lr9
GP.wnd.align                lr10
const   Templ,_G29K_Params
consth  Templ,_G29K_Params
mtsrim  cr,(9 - 1)
loadm   0,0,GP.wnd.min_x,Templ
```

The routine determines the size and location of the destination region that overlaps the clipping window.

```
sub     Templ,Dest.x,GP.wnd.min_x
jmpf    Templ,L_01
add     Temp2,GP.mem.width,0
add     Size.x,Size.x,Templ
sub     Source.x,Source.x,Templ
add     Dest.x,GP.wnd.min_x,0
```

Figure 11 shows how the source is cropped so that it becomes the same size as the destination and clipping window overlap.

If necessary, the left edge of the source and destination blocks are cropped. *Size.x* is decremented, *Source.x* is incremented, and *Dest.x* is set to the left edge of the window.

```
L_01:
  add     Templ,Dest.x,Size.x
  sub     Templ,Templ,1
  sub     Templ,GP.wnd.max_x,Templ
  jmpf    Templ,L_02
  sub     Size.x,Size.x,1
  add     Size.x,Size.x,Templ
L_02:
  jmpt    Size.x,L_08
  sub     Templ,GP.wnd.max_y,Dest.y
```

The right edge of the destination block is cropped to the right edge of the clipping window, and the *Size.x* is reduced if necessary. If the result is less than or equal to zero, the routine exits.

The top edge of the destination block is cropped to the top of the clipping window. If necessary, the *Source.y* and *Size.y* are adjusted.

```
  jmpf    Templ,L_03
  sub     Temp3,GP.wnd.min_y,1
  add     Size.y,Size.y,Templ
  add     Source.y,Source.y,Templ
  add     Dest.y,GP.wnd.max_y,0
L_03:
  sub     Templ,Dest.y,Size.y
  sub     Templ,Templ,Temp3
  jmpf    Templ,L_04
  sub     Size.y,Size.y,1
  add     Size.y,Size.y,Templ
```

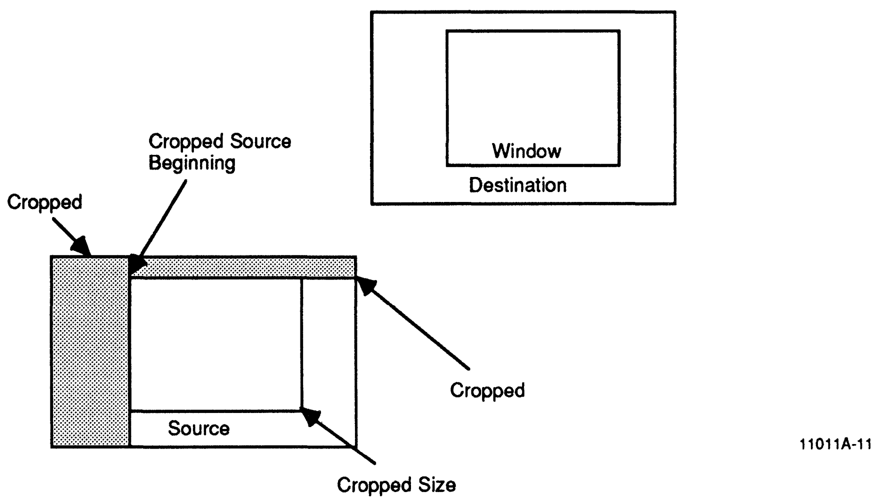


Figure 11. Cropping

The bottom edge of the destination block is cropped to the bottom of the clipping window. If necessary, the *Size.y* is adjusted. If the result is less than or equal to zero, the routine exits.

```
L_04:
   jmpt     Size.y,L_08
   sub      Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr.*

```
   add      LP.loc.x,Dest.x,0
   call     ret,S_M1_01
   add      LP.loc.y,Dest.y,0
   add      BP.dst.lft_addr,LP.loc.addr,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The linear address of the source is left in variable *BP.src.lft_addr.* This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps, respectively. They will be modified at the end of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
   add      GP.mem.width,Source.w,0
   add      GP.wnd.base,Source.b,0
   add      LP.loc.x,Source.x,0
   call     ret,S_M1_01
   add      LP.loc.y,Source.y,0
   add      BP.src.lft_addr,LP.loc.addr,0
   add      GP.mem.width,Temp2,0
```

The number of groups per scan line is calculated by right shifting the x dimension of the block size. The number of pixels in the first (or only) group is calculated.

```
   and      BP.fst.count,Size.x,
             (MAX_WORDS - 1)
   srl      BP.grp.repeat,Size.x,
            MAX_SHIFT
   sub      BP.grp.repeat,BP.grp.repeat,1
   sll      BP.fst.incr,BP.fst.count,2
   add      BP.fst.incr,BP.fst.incr,4

L_05:
   add      BP.grp.count,BP.grp.repeat,0
   add      BP.dst.addr,BP.dst.lft_addr,0
   add      BP.src.addr,BP.src.lft_addr,0
   mtsr     cr,BP.fst.count
   loadm    0,0,BP.dst.array,BP.src.addr
   mtsr     cr,BP.fst.count
   jmpt     BP.grp.count,L_07
   storem   0,0,BP.dst.array,BP.dst.addr
   add      BP.dst.addr,BP.dst.addr,
            BP.fst.incr
   add      BP.src.addr,BP.src.addr,
            BP.fst.incr
   sub      BP.grp.count,BP.grp.count,1
```

The movement of each scan line begins at label L_06. The current values of the group count and source and destination addresses are copied into working registers.

```
L_06:
   mtsrim cr,(MAX_WORDS - 1)
   loadm    0,0,BP.dst.array,BP.src.addr
   add      BP.src.addr,BP.src.addr,
            (MAX_WORDS * 4)
   mtsrim cr,(MAX_WORDS - 1)
   storem 0,0,BP.dst.array,BP.dst.addr
   jmpfdecBP.grp.count,L_06
   add      BP.dst.addr,BP.dst.addr,
            (MAX_WORDS * 4)
```

The first (or only) group of the scan line is loaded and then stored. If the first group is the only group, the code jumps to L_08; otherwise, the source and destination addresses are incremented by the number of bytes in the first group. The group count is decremented.

The loop that moves the second and subsequent group for each scan line begins at label L_07. Each group moved in the loop will be the maximum size. The odd group is moved first by loading it into the register block and incrementing the source address. The group is stored from the register block, and the destination address is incremented. The loop count is decremented and tested.

```
L_07:
   add      BP.dst.lft_addr,
            BP.dst.lft_addr,GP.mem.width
   jmpfdecSize.y,L_05
   add      BP.src.lft_addr,
            BP.src.lft_addr,Source.w
L_08:
```

At label L_08, the pixels for a scan line have all been moved. The scan line count is decremented and tested. If there are more scan lines, the left-edge addresses, *BP.dst.lft_addr* and *BP.src.lpt_addr*, are incremented by the width of the respective bit maps at label L_05.

P_B2_01.S

Routine **P_B2_01.S** is a C-language callable program that performs a general BITBLT operation in a color (32-plane) bit map. No clipping is performed. The calling program is expected to supply the address of a routine that actually combines the source and destination arrays, according to the desired operation.

The routine begins with the normal global functions.

Ten parameters are loaded from the structure G29K_Params.

```
GP.pxl.value          lr6
GP.mem.width          lr7
GP.mem.depth          lr8
GP.wnd.base           lr9
GP.wnd.align          lr10
GP.pxl.op_vec         lr11
GP.pxl.in_mask        lr12
GP.pxl.do_mask        lr13
GP.pxl.do_value       lr14
GP.pxl.out_mask       lr15
const   Temp1,_G29K_Params+(4*4)
consth  Temp1,_G29K_Params+(4*4)
mtsrim  cr,(10 - 1)
loadm   0,0,GP.pxl.value,Temp1
```

The size of the block is tested to make sure it is greater than zero. If it is less than or equal to zero, the routine exits immediately.

```
sub     Size.x,Size.x,1
jmpt    Size.x,L_04
sub     Size.y,Size.y,1
jmpt    Size.y,L_04
sub     Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr*.

```
add     LP.loc.x,Dest.x,0
call    ret,S_M1_01
add     LP.loc.y,Dest.y,0
add     BP.dst.lft_addr,LP.loc.addr,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The linear address of the source is left in variable *BP.src.lft_addr*. This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps, respectively. They will be modified at the top of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
add     Temp1,GP.mem.width,0
add     GP.mem.width,Source.w,0
add     GP.wnd.base,Source.b,0
add     LP.loc.x,Source.x,0
call    ret,S_M1_01
add     LP.loc.y,Source.y,0
add     BP.src.lft_addr,LP.loc.addr,0
add     GP.mem.width,Temp1,0
```

The number of groups in each scan line and the number of pixels in the first group are calculated. If each scan line is not an integer number of groups, the odd pixels will be done in the first group.

```
and     BP.fst.count,Size.x,
        (MAX_WORDS - 1)
srl     BP.grp.repeat,Size.x,
        MAX_SHIFT
sub     BP.grp.repeat,BP.grp.repeat,1
sll     BP.fst.incr,BP.fst.count,2
subr    BP.fst.skip,BP.fst.incr,
        ((MAX_WORDS - 1) * 4)
add     BP.fst.incr,BP.fst.incr,4

L_01:
add     BP.grp.count,BP.grp.repeat,0
add     BP.dst.addr,BP.dst.lft_addr,0
add     BP.src.addr,BP.src.lft_addr,0
mtsr    cr,BP.fst.count
loadm   0,0,BP.dst.array,BP.dst.addr
mtsr    cr,BP.fst.count
loadm   0,0,BP.src.array,BP.src.addr
calli   ret,GP.pxl.op_vec
add     BP.grp.op_skip,BP.fst.skip,0
mtsr    cr,BP.fst.count
jmpt    BP.grp.count,L_03
storem  0,0,BP.dst.array,BP.dst.addr
add     BP.dst.addr,BP.dst.addr,
        BP.fst.incr
add     BP.src.addr,BP.src.addr,
        BP.fst.incr
sub     BP.grp.count,BP.grp.count,1
```

The top of the loop for each scan line is L_02. The left addresses of the source and destination are copied into working registers, and the group count is copied into a working register.

```
L_02:
mtsrim  cr,(MAX_WORDS - 1)
loadm   0,0,BP.dst.array,BP.dst.addr
mtsrim  cr,(MAX_WORDS - 1)
loadm   0,0,BP.src.array,BP.src.addr
add     BP.src.addr,BP.src.addr,
        (MAX_WORDS * 4)
calli   ret,GP.pxl.op_vec
const   BP.grp.op_skip,0
mtsrim  cr,(MAX_WORDS - 1)
storem  0,0,BP.dst.array,BP.dst.addr
jmpfdecBP.grp.count,L_02
add     BP.dst.addr,BP.dst.addr,
        (MAX_WORDS * 4)
```

The source pixels for the first group are moved into the source register block. The destination pixels for the first group are moved into the destination-register block. The user routine is called to perform the operation on the two operands. An example of such a routine is **O4_02.S**, which is included on the distribution diskette. This particular routine adds the two operands as 32-bit unsigned numbers. If the result overflows the 32-bit register, it is forced to all 1s. Thus, an add with saturate is done. The conditional assembly in **04_02.S** avoids register destruction when MAX_WORDS is set to less than 32.

When the user-supplied routine returns, the data to be stored are in the destination-register block, and the store takes place.

The routine determines whether the first group is the only group. If so, the scan line is complete and control transfers to L_04. If more groups are needed for the scan line, the source and destination addresses are adjusted by the amount transferred in the first group and the group count is decremented.

At label L_03, the remaining groups are transferred. The destination register block is loaded, the source register block is loaded, and the source address is modified. Then, the user routine is called, the destination-register block is written, and the destination address is modified. The group count is decremented and tested, and more groups are transferred, if necessary.

```
L_03:
   add      BP.dst.lft_addr,
            BP.dst.lft_addr,GP.mem.width
   jmpfdec  Size.y,L_01
   add      BP.src.lft_addr,
            BP.src.lft_addr,Source.w

L_04:
```

At label L_04, the scan-line count is decremented and tested, and more scan lines are done, if necessary. In this case, the source and destination left addresses are adjusted at L_01.

P_B2_02.S

Routine **P_B2_02.S** is a C-language callable program that performs a copy block operation in a color (32-plane) bit map. Clipping is performed.

The routine begins with the normal global functions.

Fourteen parameters are loaded from the structure G29K_Params.

```
   GP.wnd.min_x            lr2
   GP.wnd.max_x            lr3
   GP.wnd.min_y            lr4
   GP.wnd.max_y            lr5
   GP.pxl.value            lr6
   GP.mem.width            lr7
   GP.mem.depth            lr8
   GP.wnd.base             lr9
   GP.wnd.align            lr10
   GP.pxl.op_vec           lr11
   GP.pxl.in_mask          lr12
   GP.pxl.do_mask          lr13
   GP.pxl.do_value         lr14
   GP.pxl.out_mask         lr15
   const    Temp1,_G29K_Params
   consth   Temp1,_G29K_Params
   mtsrim   cr,(14 - 1)
   loadm    0,0,GP.wnd.min_x,Temp1
```

The routine determines the size and location of the destination region that overlaps the clipping window. This is similar to the code in **P_B1_02.S**.

The left edge of the destination block is cropped to the left edge of the window. The *Size.x* is decremented, the *Source.x* is incremented, and *Dest.x* is set to the left edge of the window.

```
   sub      Temp1,Dest.x,GP.wnd.min_x
   jmpf     Temp1,L_01
   add      Temp2,GP.mem.width,0
   add      Size.x,Size.x,Temp1
   sub      Source.x,Source.x,Temp1
   add      Dest.x,GP.wnd.min_x,0
```

The right edge of the destination block is cropped to the right edge of the clipping window, and the *Size.x* is reduced if necessary. If the result is less than or equal to zero, the routine exits.

```
L_01:
   add      Temp1,Dest.x,Size.x
   sub      Temp1,Temp1,1
   sub      Temp1,GP.wnd.max_x,Temp1
   jmpf     Temp1,L_02
   sub      Size.x,Size.x,1
   add      Size.x,Size.x,Temp1
```

The top edge of the destination block is cropped to the top of the clipping window. If necessary, the *Source.y* and *Size.y* are adjusted so that the beginning of the destination and source correspond to the beginning of the window.

```
L_02:
   jmpt     Size.x,L_08
   sub      Temp1,GP.wnd.max_y,Dest.y
   jmpf     Temp1,L_03
   sub      Temp3,GP.wnd.min_y,1
   add      Size.y,Size.y,Temp1
   add      Source.y,Source.y,Temp1
   add      Dest.y,GP.wnd.max_y,0

L_03:
   sub      Temp1,Dest.y,Size.y
   sub      Temp1,Temp1,Temp3
   jmpf     Temp1,L_04
   sub      Size.y,Size.y,1
   add      Size.y,Size.y,Temp1
```

The bottom edge of the destination block is cropped to the bottom of the clipping window. If necessary, *Size.y* is adjusted. If the result is less than or equal to zero, the routine exits.

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr*.

```
L_04:
    jmpt     Size.y,L_08
    sub      Size.y,Size.y,1
    add      LP.loc.x,Dest.x,0
    call     ret,S_M1_01
    add      LP.loc.y,Dest.y,0
    add      BP.dst.lft_addr,LP.loc.addr,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The linear address of the source is left in variable *BP.src.lft_addr*. This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps, respectively. They will be modified at the top of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
    add      GP.mem.width,Source.w,0
    add      GP.wnd.base,Source.b,0
    add      LP.loc.x,Source.x,0
    call     ret,S_M1_01
    add      LP.loc.y,Source.y,0
    add      BP.src.lft_addr,LP.loc.addr,0
    add      GP.mem.width,Temp2,0
```

The number of groups per scan line is calculated by right shifting the x dimension of the block size. The number of pixels in the first (or only) group is calculated.

```
    and      BP.fst.count,Size.x,
             (MAX_WORDS - 1)
    srl      BP.grp.repeat,Size.x,
             MAX_SHIFT
    sub      BP.grp.repeat,BP.grp.repeat,1
    sll      BP.fst.incr,BP.fst.count,2
    subr     BP.fst.skip,BP.fst.incr,
             ((MAX_WORDS - 1) * 4)
    add      BP.fst.incr,BP.fst.incr,4

L_05:
    add      BP.grp.count,BP.grp.repeat,0
    add      BP.dst.addr,BP.dst.lft_addr,0
    add      BP.src.addr,BP.src.lft_addr,0
    mtsr     cr,BP.fst.count
    loadm    0,0,BP.dst.array,BP.dst.addr
    mtsr     cr,BP.fst.count
    loadm    0,0,BP.src.array,BP.src.addr
    calli    ret,GP.pxl.op_vec
    add      BP.grp.op_skip,BP.fst.skip,0
    mtsr     cr,BP.fst.count
    jmpt     BP.grp.count,L_07
    storem   0,0,BP.dst.array,BP.dst.addr
    add      BP.dst.addr,BP.dst.addr,
             BP.fst.incr
    add      BP.src.addr,BP.src.addr,
             BP.fst.incr
    sub      BP.grp.count,BP.grp.count,1
```

Label L_06 is the beginning of each scan line. A set of working registers is set to the number of groups in the scan line, and to the beginning source and destination addresses for the scan line.

```
L_06:
    mtsrim   cr,(MAX_WORDS - 1)
    loadm    0,0,BP.dst.array,BP.dst.addr
    mtsrim   cr,(MAX_WORDS - 1)
    loadm    0,0,BP.src.array,BP.src.addr
    add      BP.src.addr,BP.src.addr,
             (MAX_WORDS * 4)
    calli    ret,GP.pxl.op_vec
    const    BP.grp.op_skip,0
    mtsrim   cr,(MAX_WORDS - 1)
    storem   0,0,BP.dst.array,BP.dst.addr
    jmpfdec  BP.grp.count,L_06
    add      BP.dst.addr,BP.dst.addr,
             (MAX_WORDS * 4)
```

One group of the destination operands are moved into the destination-register block. One group of the source operands are moved into the source-register block. The user routine is called to perform the operation on the blocks.

When the user routine returns with the finished destination data, the destination field in memory is written. The group count is decremented and tested. If this is the only group in the scan line, the code continues at label L_08. If more groups are required, the left-edge addresses are adjusted for the first group, and the group count is reduced.

At label L_07, the remaining groups are moved. For each group, the destination and source operands are moved into the register block, and the user routine is called. When the routine returns, the results are stored, and the group count is decremented and tested.

```
L_07:
    add      BP.dst.lft_addr,
             BP.dst.lft_addr,GP.mem.width
    jmpfdec  Size.y,L_05
    add      BP.src.lft_addr,
             BP.src.lft_addr,Source.w

L_08:
```

At label L_08, the scan line count is reduced and tested. If more scan lines are required, the left-edge addresses are adjusted at label L_05.

P_B3_01.S

Routine **P_B3_01.S** is a C-language callable program that performs a copy-block operation in a monochrome (1-plane) bit map. No clipping is performed. This routine is optimized for moving data.

The routine begins with the normal global functions.

Four parameters are loaded from the structure G29K_Params.

```
GP.mem.width            lr7
GP.mem.depth            lr8
GP.wnd.base             lr9
GP.wnd.align            lr10
const   Temp1,_G29K_Params+(5*4)
consth  Temp1,_G29K_Params+(5*4)
mtsrim  cr,(4 - 1)
loadm   0,0,GP.mem.width,Temp1
```

The routine checks to be sure that the size of the block is non-negative in either dimension. If so, it exits immediately.

```
cple    BP.grp.repeat,Size.x,0
jmpt    BP.grp.repeat,L_16
sub     Size.y,Size.y,1
jmpt    Size.y,L_16
sub     Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr*, and the alignment is left in *LP.loc.align*.

```
add     LP.loc.x,Dest.x,0
call    ret,S_M1_01
add     LP.loc.y,Dest.y,0
add     BP.dst.lft_addr,LP.loc.addr,0
add     BP.dst.align,LP.loc.align,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The linear address of the source is left in variable *BP.src.lft_addr*. This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps, respectively. They will be modified at the top of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
add     BP.grp.count,GP.mem.width,0
add     GP.mem.width,Source.w,0
add     GP.wnd.base,Source.b,0
add     GP.wnd.align,Source.a,0
add     LP.loc.x,Source.x,0
call    ret,S_M1_01
add     LP.loc.y,Source.y,0
add     GP.mem.width,BP.grp.count,0
add     BP.src.lft_addr,LP.loc.addr,0
```

The amount that the source must be shifted to align with the destination is calculated and left in *BP.src.shift*.

```
sub     BP.src.shift,LP.loc.align,
        BP.dst.align
```

Masks are generated for the left and right ends of the destination field. These will be used at the beginning and end of each scan line to avoid affecting partial words not actually inside the destination.

```
constn  BP.dst.rgt_mask,-1
srl     BP.dst.lft_mask,
        BP.dst.rgt_mask,BP.dst.align
add     BP.grp.align,BP.dst.align,
        Size.x
add     BP.grp.repeat,BP.grp.align,31
subr    BP.grp.align,BP.grp.align,32
and     BP.grp.align,BP.grp.align,31
sll     BP.dst.rgt_mask,
        BP.dst.rgt_mask,BP.grp.align
```

The number of groups in each scan line is calculated. Each group except the first will contain exactly 32 words; any "extra" words will be in the first group.

```
srl     BP.grp.repeat,BP.grp.repeat,5
sub     BP.grp.repeat,BP.grp.repeat,1
and     BP.fst.count,BP.grp.repeat,
        (MAX_WORDS - 1)
srl     BP.grp.repeat,BP.grp.repeat,
        MAX_SHIFT
sub     BP.grp.repeat,BP.grp.repeat,1

sll     BP.fst.incr,BP.fst.count,2
subr    BP.fst.skip,BP.fst.incr,
        (4 * (MAX_WORDS - 1))
const   BP.fst.shift,L_09
consth  BP.fst.shift,L_09
add     BP.fst.shift,BP.fst.shift,
        BP.fst.skip
setip   BP.dst.array,BP.src.array,
        BP.src.array
mfsr    BP.src.rgt_ptr, ipa
add     BP.src.rgt_ptr,BP.src.rgt_ptr,
        BP.fst.incr
add     BP.fst.incr,BP.fst.incr,4
cpgt    BP.grp.align,BP.src.shift,0
add     BP.src.shift,BP.src.shift,32
and     BP.src.shift,BP.src.shift,31
cpeq    BP.src.save,BP.src.shift,0
or      BP.src.shift,BP.src.shift,
        BP.src.save
```

For the first group, there is a sequence of shift instructions beginning at L_09. Since the first group may not contain all 32 words, an indirect jump address into the sequence is generated and left in *BP.fst.shift*. The indirect address pointers are set to the source array and destination array. The address of the right-most word of the first scan line of the source array is computed.

```
L_05:
   add     BP.grp.count,BP.grp.repeat,0
   add     BP.dst.addr,BP.dst.lft_addr,0
   add     BP.src.addr,BP.src.lft_addr,0
   load    0,0,BP.dst.lft_end,BP.dst.addr
   jmpf    BP.grp.count,L_06
   andn    BP.dst.lft_end,BP.dst.lft_end,
           BP.dst.lft_mask
   add     Temp1,BP.dst.addr,BP.fst.incr
   sub     Temp1,Temp1,4
   load    0,0,BP.dst.rgt_end,Temp1
   andn    BP.dst.rgt_end,BP.dst.rgt_end,
           BP.dst.rgt_mask
```

The sign of the relative alignment is placed into *BP.grp.align*; it will be used to determine whether to do a left or right shift. *BP.src.shift* is set to the value to be loaded into the funnel count register for use with the extract instructions when shifting the source registers. The value is appropriate for either left or right shifting. If the value of the shift amount is zero (no shifting is needed), then the sign bit is set for use with a conditional jump prior to the sequence of extract instructions.

Each scan line begins at L_06. Copies of the source and destination addresses of the edge of the scan lines and the group count are moved into working registers.

```
L_06:
   jmpf    BP.grp.align,L_07
   mtsr    ipa,BP.src.rgt_ptr
   add     BP.src.extra,BP.fst.count,1
   mtsr    cr,BP.src.extra
   loadm   0,0,BP.src.extra,BP.src.addr
   jmp     L_08
   add     BP.src.addr,BP.src.addr,4
```

The word at the left end of the first group of the destination (possibly an incomplete word) is fetched, and the portion outside the destination is retained for storing to memory. If the first group is the only group, the word at the right end is fetched and masked as well.

The alignment direction flag in *BP.grp.align* is tested. A right shift results in a jump to L_07. If a left shift is necessary, an extra word is loaded into a register that is prefixed to the source register block. The remaining words of the source array are loaded into the register block.

```
L_07:
   mtsr    cr,BP.fst.count
   loadm   0,0,BP.src.array,BP.src.addr
```

At label L_08, the right-most source word of the first group is saved to be prefixed to the next group.

```
L_08:
   jmpt    BP.src.shift,L_10
   add     BP.src.save,gr0,0
   jmpi    BP.fst.shift
   mtsr    fc,BP.src.shift
```

If the source and destination are identically aligned, no shifts are necessary and the code jumps over the shift array to label L_10. This is the case regardless of the absolute alignment of the operands, which is handled by the special treatment of words at the left and right ends of the scan line.

```
L_09:
   .if MAX_WORDS == 32
   extract BP.src.array_31,
           BP.src.array_30,BP.src.array_31
     .
     .
     .
   extract BP.src.array_16,
           BP.src.array_15,BP.src.array_16
   .endif

   .if MAX_WORDS >= 16
   extract BP.src.array_15,
           BP.src.array_14,BP.src.array_15
     .
     .
     .
   extract BP.src.array_08,
           BP.src.array_07,BP.src.array_08
   .endif

   .if MAX_WORDS >= 8
   extract BP.src.array_07,
           BP.src.array_06,BP.src.array_07
   extract BP.src.array_06,
           BP.src.array_05,BP.src.array_06
   extract BP.src.array_05,
           BP.src.array_04,BP.src.array_05
   extract BP.src.array_04,
           BP.src.array_03,BP.src.array_04
   .endif

   extract BP.src.array_03,
           BP.src.array_02,BP.src.array_03
   extract BP.src.array_02,
           BP.src.array_01,BP.src.array_02
   extract BP.src.array_01,
           BP.src.array_00,BP.src.array_01
```

```
extract BP.src.array_00,
        BP.src.extra,BP.src.array_00
```

If the shift is necessary, the code jumps somewhere into the sequence of shift instructions. Each instruction either left- or right-shifts two adjacent registers in the source block and leaves the result in the register that is further to the right.

At label L_10, the bits outside the destination (in the left-most word) are placed into the left-most word of the source array. If there is only one group, the bits outside the destination (in the right-most word) are placed into the right-most word of the source array.

```
L_10:
  and     BP.src.array,BP.src.array,
          BP.dst.lft_mask
  jmpf    BP.grp.count,L_11
  or      BP.src.array,BP.src.array,
          BP.dst.lft_end
  mtsr    ipa,BP.src.rgt_ptr
  mtsr    ipc,BP.src.rgt_ptr
  nop
  and     gr0,gr0,BP.dst.rgt_mask
  or      gr0,gr0,BP.dst.rgt_end
```

At label L_11, the resulting register block is written into the destination bit map. If there is only a single group per scan line, code jumps to L_15.

```
L_11:
  mtsr    cr,BP.fst.count
  jmpt    BP.grp.count,L_15
  storem  0,0,BP.src.array,BP.dst.addr
  add     BP.dst.addr,BP.dst.addr,
          BP.fst.incr
  add     BP.src.addr,BP.src.addr,
          BP.fst.incr
  sub     BP.grp.count,BP.grp.count,1
```

If there is more than one group per scan line, the addresses are adjusted by the amount moved in the first group, and the code enters a loop at L_12 to move the rest of the scan line 32 words at a time.

At label L_12, the code determines whether this is the last group. If so, the right-most word is fetched from the destination. The bits outside the destination block are preserved.

```
L_12:
  jmpf    BP.grp.count,L_12a
  add     BP.src.extra,BP.src.save,0
  add     Temp1,BP.dst.addr,
          (4 * (MAX_WORDS - 1))
  load    0,0,BP.dst.rgt_end,Temp1
  andn    BP.dst.rgt_end,BP.dst.rgt_end,
          BP.dst.rgt_mask
```

At label L_12a, the right-most word from the previous group is prefixed, and then the 32 words are fetched from the source bit map. The source address is incremented, and the right-most word is saved for the next group.

```
L_12a:
  mtsrim  cr,(MAX_WORDS - 1)
  loadm   0,0,BP.src.array,BP.src.addr
  add     BP.src.addr,BP.src.addr,
          (4 * MAX_WORDS)
  jmpt    BP.src.shift,L_13
  add     BP.src.save,BP.src.array_end,0
  mtsr    fc,BP.src.shift

  .if MAX_WORDS == 32
  extract BP.src.array_31,
          BP.src.array_30,BP.src.array_31
          .
          .
          .
  extract BP.src.array_16,
          BP.src.array_15,BP.src.array_16
  .endif

  .if MAX_WORDS >= 16
  extract BP.src.array_15,
          BP.src.array_14,BP.src.array_15
          .
          .
          .
  extract BP.src.array_08,
          BP.src.array_07,BP.src.array_08
  .endif

  .if MAX_WORDS >= 8
  extract BP.src.array_07,
          BP.src.array_06,BP.src.array_07
  extract BP.src.array_06,
          BP.src.array_05,BP.src.array_06
  extract BP.src.array_05,
          BP.src.array_04,BP.src.array_05
  extract BP.src.array_04,
          BP.src.array_03,BP.src.array_04
  .endif

  extract BP.src.array_03,
          BP.src.array_02,BP.src.array_03
  extract BP.src.array_02,
          BP.src.array_01,BP.src.array_02
  extract BP.src.array_01,
          BP.src.array_00,BP.src.array_01
  extract BP.src.array_00,
          BP.src.extra,BP.src.array_00
```

If no shift is necessary (the source and destination are identically aligned), the shift array is skipped. If a shift is necessary, it occurs.

At label L_13, the right-most destination word is fetched, masked, and merged, if this is the last group.

```
L_13:
   jmpf     BP.grp.count,L_14
   mtsrim   cr,(MAX_WORDS - 1)
   and      BP.src.array_end,
            BP.src.array_end,
            BP.dst.rgt_mask
   or       BP.src.array_end,
            BP.src.array_end,
            BP.dst.rgt_end
```

At label L_14, the group is written into the destination bit map and the destination address is modified. The group count is decremented and tested. If further groups are necessary for the scan line, they are moved beginning at label L_12.

```
L_14:
   storem   0,0,BP.src.array,BP.dst.addr
   jmpfdec  BP.grp.count,L_12
   add      BP.dst.addr,BP.dst.addr,
            (4 * MAX_WORDS)
L_15:
   add      BP.dst.lft_addr,
            BP.dst.lft_addr,GP.mem.width
   jmpfdec  Size.y,L_05
   add      BP.src.lft_addr,
            BP.src.lft_addr,Source.w
L_16:
```

The scan-line count is decremented and tested. If further scan lines are necessary, the address of the left edge of each is calculated at label L_05.

P_B3_02.S

Routine **P_B3_02.S** is a C-language callable program that performs a copy-block operation in a monochrome (1-plane) bit map. Clipping is performed. This routine is optimized for moving data. The routine begins with the normal global functions.

Nine parameters are loaded from the structure G29K_Params.

```
GP.wnd.min_x     lr2
GP.wnd.max_x     lr3
GP.wnd.min_y     lr4
GP.wnd.max_y     lr5
GP.pxl.value     lr6
GP.mem.width     lr7
GP.mem.depth     lr8
GP.wnd.base      lr9
GP.wnd.align     lr10
```

```
   const    Temp1,_G29K_Params
   consth   Temp1,_G29K_Params
   mtsrim   cr,(9 - 1)
   loadm    0,0,GP.wnd.min_x,Temp1
```

The destination array is cropped so that it consists only of the array originally within the destination array *and* within the clipping window.

The left edge of the destination array is cropped, if necessary, to the left edge of the clipping window. If the left edge must be cropped, the left edge of the source array and the block size are adjusted as well.

```
   sub      Temp1,Dest.x,GP.wnd.min_x
   jmpf     Temp1,L_01
   add      Temp3,GP.wnd.max_x,1
   add      Size.x,Size.x,Temp1
   sub      Source.x,Source.x,Temp1
   add      Dest.x,GP.wnd.min_x,0
```

The right edge of the destination array is cropped, if necessary, to the right edge of the clipping window. If the right edge must be adjusted, the block size is adjusted as well.

```
L_01:
   add      Temp1,Dest.x,Size.x
   sub      Temp1,Temp3,Temp1
   jmpf     Temp1,L_02
   add      Temp2,GP.mem.width,0
   add      Size.x,Size.x,Temp1
```

If the width of the resulting block is less than or equal to zero, the routine exits immediately.

If necessary, the top edge of the destination array is cropped to the top edge of the clipping window. If the top edge must be adjusted, the source top and the block size are adjusted as well.

```
L_02:
   cple     Temp1,Size.x,0
   jmpt     Temp1,L_16
   sub      Temp1,GP.wnd.max_y,Dest.y
   jmpf     Temp1,L_03
   sub      Temp3,GP.wnd.min_y,1
   add      Size.y,Size.y,Temp1
   add      Source.y,Source.y,Temp1
   add      Dest.y,GP.wnd.max_y,0
```

The bottom edge of the destination array is cropped, if necessary, to the bottom edge of the clipping window. If the bottom edge must be adjusted, the block size is adjusted as well.

```
L_03:
   sub      Temp1,Dest.y,Size.y
   sub      Temp1,Temp1,Temp3
```

```
    jmpf    Temp1,L_04
    sub     Size.y,Size.y,1
    add     Size.y,Size.y,Temp1
```

If the height of the resulting block is less than or equal to zero, the routine exits immediately.

```
L_04:
    jmpt    Size.y,L_16
    sub     Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination coordinates to a linear address and alignment. The results are left in *BP.dst.lft_addr* and *BP.dst.align*.

```
    add     LP.loc.x,Dest.x,0
    call    ret,S_M1_01
    add     LP.loc.y,Dest.y,0
    add     BP.dst.lft_addr,LP.loc.addr,0
    add     BP.dst.align,LP.loc.align,0
```

Routine **S_M1_01** is called a second time to convert the source coordinates to a linear address and alignment.

```
    add     GP.mem.width,Source.w,0
    add     GP.wnd.base,Source.b,0
    add     GP.wnd.align,Source.a,0
    add     LP.loc.x,Source.x,0
    call    ret,S_M1_01
    add     LP.loc.y,Source.y,0
    add     GP.mem.width,Temp2,0
    add     BP.src.lft_addr,LP.loc.addr,0
```

The difference in alignment between the source and destination is calculated to determine how far the source must be shifted. This is left in *BP.src.shift*.

```
    sub     BP.src.shift,LP.loc.align,
            BP.dst.align
```

The masks for the left and right ends of a destination line are formed. These will be used to mask bits in the words at the ends of each scan line that are not in the destination block.

```
    constn  BP.dst.rgt_mask,-1
    srl     BP.dst.lft_mask,
            BP.dst.rgt_mask,BP.dst.align
    add     BP.grp.align,BP.dst.align,
            Size.x
    add     BP.grp.repeat,BP.grp.align,31
    subr    BP.grp.align,BP.grp.align,32
    and     BP.grp.align,BP.grp.align,31
    sll     BP.dst.rgt_mask,
            BP.dst.rgt_mask,BP.grp.align
```

The number of groups in each scan line is computed. The number of words in the first or only group of each scan line is computed. All other groups of each scan line will be exactly MAX_WORDS (32) words.

```
    srl     BP.grp.repeat,BP.grp.repeat,5
    sub     BP.grp.repeat,BP.grp.repeat,1
    and     BP.fst.count,BP.grp.repeat,
            (MAX_WORDS - 1)
    srl     BP.grp.repeat,BP.grp.repeat,
            MAX_SHIFT
    sub     BP.grp.repeat,BP.grp.repeat,1
    sll     BP.fst.incr,BP.fst.count,2
    subr    BP.fst.skip,BP.fst.incr,
            (4 * (MAX_WORDS - 1))
    const   BP.fst.shift,L_09
    consth  BP.fst.shift,L_09
    add     BP.fst.shift,BP.fst.shift,
            BP.fst.skip
    setip   BP.dst.array,BP.src.array,
            BP.src.array
    mfsr    BP.src.rgt_ptr, ipa
    add     BP.src.rgt_ptr,BP.src.rgt_ptr,
            BP.fst.incr
    add     BP.fst.incr,BP.fst.incr,4
    cpgt    BP.grp.align,BP.src.shift,0
    add     BP.src.shift,BP.src.shift,32
    and     BP.src.shift,BP.src.shift,31
    cpeq    BP.src.save,BP.src.shift,0
    or      BP.src.shift,BP.src.shift,
            BP.src.save

L_05:
    add     BP.grp.count,BP.grp.repeat,0
    add     BP.dst.addr,BP.dst.lft_addr,0
    add     BP.src.addr,BP.src.lft_addr,0
    load    0,0,BP.dst.lft_end,BP.dst.addr
    jmpf    BP.grp.count,L_06
    andn    BP.dst.lft_end,BP.dst.lft_end,
            BP.dst.lft_mask
    add     Temp1,BP.dst.addr,BP.fst.incr
    sub     Temp1,Temp1,4
    load    0,0,BP.dst.rgt_end,Temp1
    andn    BP.dst.rgt_end,BP.dst.rgt_end,
            BP.dst.rgt_mask
```

For the first group, there is a sequence of shift instructions beginning at L_09. Since the first group may not contain all 32 words, an indirect jump address into the sequence is generated and left in *BP.fst.shift*. The indirect address pointers are set to the source array and destination array. The address of the right-most word of the first scan line of the source array is computed.

The sign of the relative alignment is placed into *BP.grp.align*; it will be used to determine whether to do a left or right shift. *BP.src.shift* is set to the value to be loaded into the funnel count register for use with the extract instructions when shifting the source registers. The value is appropriate for either left or right shifting. If the actual value of the shift amount is zero—that is, no shifting is needed—then the sign bit is set for use with a conditional jump prior to the extract instructions sequence.

Each scan line begins at L_06. Copies of the source and destination addresses of the edge of the scan lines and the group count are moved into working registers.

```
L_06:
  jmpf    BP.grp.align,L_07
  mtsr    ipa,BP.src.rgt_ptr
  add     BP.src.extra,BP.fst.count,1
  mtsr    cr,BP.src.extra
  loadm   0,0,BP.src.extra,BP.src.addr
  jmp     L_08
  add     BP.src.addr,BP.src.addr,4
```

The word at the left end of the first group (which may be an incomplete word) of the destination is fetched, and the portion outside the destination is retained and eventually written back into memory. If the first group is the only group, the word at the right end is fetched and masked as well.

The alignment direction flag in *BP.grp.align* is tested. A right shift results in a jump to L_07. If a left shift is necessary, an extra word is loaded into a register that is prefixed to the source register block. The remaining words of the source array are loaded into the register block.

```
L_07:
  mtsr    cr,BP.fst.count
  loadm   0,0,BP.src.array,BP.src.addr
```

At label L_08, the right-most source word of the first group is saved to be prefixed to the next group.

```
L_08:
  jmpt    BP.src.shift,L_10
  add     BP.src.save,gr0,0
  jmpi    BP.fst.shift
  mtsr    fc,BP.src.shift
```

If the source and destination are identically aligned, no shifts are necessary and the code jumps over the shift array to label L_10. This is the case regardless of the absolute alignment of the operands, which is handled by the special treatment of words at the left and right ends of the scan line.

```
L_09:
  .if MAX_WORDS == 32
  extract BP.src.array_31,
          BP.src.array_30,BP.src.array_31
   .
   .
   .
  extract BP.src.array_16,
          BP.src.array_15,BP.src.array_16
  .endif

  .if MAX_WORDS >= 16
  extract BP.src.array_15,
          BP.src.array_14,BP.src.array_15
   .
   .
   .
  extract BP.src.array_08,
          BP.src.array_07,BP.src.array_08
  .endif

  .if MAX_WORDS >= 8
  extract BP.src.array_07,
          BP.src.array_06,BP.src.array_07
  extract BP.src.array_06,
          BP.src.array_05,BP.src.array_06
  extract BP.src.array_05,
          BP.src.array_04,BP.src.array_05
  extract BP.src.array_04,
          BP.src.array_03,BP.src.array_04
  .endif

  extract BP.src.array_03,
          BP.src.array_02,BP.src.array_03
  extract BP.src.array_02,
          BP.src.array_01,BP.src.array_02
  extract BP.src.array_01,
          BP.src.array_00,BP.src.array_01
  extract BP.src.array_00,
          BP.src.extra,BP.src.array_00
```

If the shift is necessary, the code jumps somewhere into the sequence of shift instructions. Each instruction either left- or right-shifts two adjacent registers in the source block and leaves the result in the register that is further to the right.

At label L_10, the bits outside the destination (in the left-most word) are placed into the left-most word of the source array. If there is only one group, the bits outside the destination (in the right-most word) are placed into the right-most word of the source array.

```
L_10:
  and     BP.src.array,BP.src.array,
          BP.dst.lft_mask
  jmpf    BP.grp.count,L_11
  or      BP.src.array,BP.src.array,
          BP.dst.lft_end
  mtsr    ipa,BP.src.rgt_ptr
  mtsr    ipc,BP.src.rgt_ptr
```

```
  nop
  and     gr0,gr0,BP.dst.rgt_mask
  or      gr0,gr0,BP.dst.rgt_end
```

At label L_11, the resulting register block is written into the destination bit map. If there is only a single group per scan line, the code jumps to L_15.

```
L_11:
  mtsr    cr,BP.fst.count
  jmpt    BP.grp.count,L_15
  storem  0,0,BP.src.array,BP.dst.addr
  add     BP.dst.addr,BP.dst.addr,
          BP.fst.incr
  add     BP.src.addr,BP.src.addr,
          BP.fst.incr
  sub     BP.grp.count,BP.grp.count,1
```

If there is more than one group per scan line, the addresses are adjusted by the amount moved in the first group, and the code enters a loop at L_12 to move the rest of the scan line 32 words at a time.

At label L_12, the code tests to find out if this is the last group. If so, the right-most word is fetched from the destination. The bits outside the destination block are preserved.

```
L_12:
  jmpf    BP.grp.count,L_12a
  add     BP.src.extra,BP.src.save,0
  add     Templ,BP.dst.addr,
          (4 * (MAX_WORDS - 1))
  load    0,0,BP.dst.rgt_end,Templ
  andn    BP.dst.rgt_end,BP.dst.rgt_end,
          BP.dst.rgt_mask
```

At label L_12a, the right-most word from the previous group is prefixed, and then the 32 words are fetched from the source bit map. The source address is incremented and the right-most word is saved for the next group.

```
L_12a:
  mtsrim  cr,(MAX_WORDS - 1)
  loadm   0,0,BP.src.array,BP.src.addr
  add     BP.src.addr,BP.src.addr,
          (4 * MAX_WORDS)
  jmpt    BP.src.shift,L_13
  add     BP.src.save,BP.src.array_end,0
  mtsr    fc,BP.src.shift
  .if MAX_WORDS == 32
  extract BP.src.array_31,
          BP.src.array_30,BP.src.array_31
          .
          .
          .
  extract BP.src.array_16,
          BP.src.array_15,BP.src.array_16
  .endif

  .if MAX_WORDS >= 16
  extract BP.src.array_15,
          BP.src.array_14,BP.src.array_15
          .
          .
          .
  extract BP.src.array_08,
          BP.src.array_07,BP.src.array_08
  .endif
  .if MAX_WORDS >= 8
  extract BP.src.array_07,
          BP.src.array_06,BP.src.array_07
  extract BP.src.array_06,
          BP.src.array_05,BP.src.array_06
  extract BP.src.array_05,
          BP.src.array_04,BP.src.array_05
  extract BP.src.array_04,
          BP.src.array_03,BP.src.array_04
  .endif

  extract BP.src.array_03,
          BP.src.array_02,BP.src.array_03
  extract BP.src.array_02,
          BP.src.array_01,BP.src.array_02
  extract BP.src.array_01,
          BP.src.array_00,BP.src.array_01
  extract BP.src.array_00,
          BP.src.extra,BP.src.array_00
```

If no shift is necessary (the source and destination are identically aligned), the shift array is skipped. A shift occurs, if necessary.

At label L_13, the right-most destination word is fetched, masked, and merged, if this is the last group.

```
L_13:
  jmpf    BP.grp.count,L_14
  mtsrim  cr,(MAX_WORDS - 1)
  and     BP.src.array_end,
          BP.src.array_end,
          BP.dst.rgt_mask
  or      BP.src.array_end,
          BP.src.array_end,
          BP.dst.rgt_end
```

At label L_14, the group is written into the destination bit map, and the destination address is modified. The group count is decremented and tested. If further groups are necessary for the scan line, they are moved beginning at label L_12.

```
L_14:
   storem 0,0,BP.src.array,BP.dst.addr
   jmpfdec BP.grp.count,L_12
   add     BP.dst.addr,BP.dst.addr,
           (4 * MAX_WORDS)


L_15:
   add     BP.dst.lft_addr,
           BP.dst.lft_addr,GP.mem.width
   jmpfdec Size.y,L_05
   add     BP.src.lft_addr,
           BP.src.lft_addr,Source.w

L_16:
```

The scan-line count is decremented and tested. If further scan lines are necessary, the address of the left edge of each is calculated at label L_05.

P_B4_01.S

Routine **P_B4_01.S** is a C-language callable program that performs a general BITBLT operation in a monochrome (1-plane) bit map. No clipping is performed.

The caller is responsible for supplying the address of a routine that combines the two operands after they have been moved into the source and destination register blocks, and after the source operand has been shifted to align with the destination. An example of such a routine is **O5_01.S**, which is included on the distribution diskette. This routine XORs the source array into the destination array.

The routine begins with the normal global functions.

Nine parameters are loaded from the structure G29K_Params.

```
   GP.mem.width         lr7
   GP.mem.depth         lr8
   GP.wnd.base          lr9
   GP.wnd.align         lr10
   GP.pxl.op_vec        lr11
   GP.pxl.in_mask       lr12
   GP.pxl.do_mask       lr13
   GP.pxl.do_value      lr14
   GP.pxl.out_mask      lr15
   const   Temp1,_G29K_Params+(5*4)
   consth  Temp1,_G29K_Params+(5*4)
   mtsrim  cr,(9 - 1)
   loadm   0,0,GP.mem.width,Temp1
```

The routine checks to be sure that the size of the block is not negative or zero in either dimension. If so, it exits immediately.

```
   cple    Temp1,Size.x,0
   jmpt    Temp1,L_12
   sub     Size.y,Size.y,1
   jmpt    Size.y,L_12
   sub     Size.y,Size.y,1
```

From here on, the routine is exactly like **P_B3_01.S**, except that the user routine is called to perform the operation on the two register blocks, and the labels have been changed.

Routine **S_M1_01** is called to convert the destination address to a linear address. The linear destination address is left in *BP.dst.lft_addr* and the alignment is left in *LP.loc.align*.

```
   add     LP.loc.x,Dest.x,0
   call    ret,S_M1_01
   add     LP.loc.y,Dest.y,0
   add     BP.dst.lft_addr,LP.loc.addr,0
   add     BP.dst.align,LP.loc.align,0
```

Routine **S_M1_01** is called a second time to convert the source address to a linear address. The linear address of the source is left in variable *BP.src.lft_addr*. This address and *BP.dst.lft_addr* point to the left edge of the source and destination bit maps, respectively. They will be modified at the top of each scan line by *GP.mem.width* and *Source.w*, respectively.

```
   add     BP.grp.count,GP.mem.width,0
   add     GP.mem.width,Source.w,0
   add     GP.wnd.base,Source.b,0
   add     GP.wnd.align,Source.a,0
   add     LP.loc.x,Source.x,0
   call    ret,S_M1_01
   add     LP.loc.y,Source.y,0
   add     GP.mem.width,BP.grp.count,0
   add     BP.src.lft_addr,LP.loc.addr,0
```

The amount that the source must be shifted in order to align with the destination is calculated and left in *BP.src.shift*.

```
   sub     BP.src.shift,LP.loc.align,
           BP.dst.align
```

Masks are generated for the left and right end of the destination field. These will be used at the beginning and end of each scan line to avoid affecting partial words not actually inside the destination.

```
   constn  BP.dst.rgt_mask,-1
   srl     BP.dst.lft_mask,
           BP.dst.rgt_mask,BP.dst.align
   add     BP.grp.align,BP.dst.align,
           Size.x
   add     BP.grp.repeat,BP.grp.align,31
```

```
    subr    BP.grp.align,BP.grp.align,32
    and     BP.grp.align,BP.grp.align,31
    sll     BP.dst.rgt_mask,
            BP.dst.rgt_mask,BP.grp.align
```

The number of groups in each scan line is calculated. Each group except the first will contain exactly 32 words; any "extra" words will be in the first group.

For the first group, there is a sequence of shift instructions beginning at L_05. Since the first group may not contain all 32 words, an indirect jump address into the sequence is generated and left in *BP.fst.shift*. The indirect address pointers are set to the source array and destination array. The address of the right-most word of the first scan line of the source array is computed.

```
    sll     BP.fst.incr,BP.fst.count,2
    subr    BP.fst.skip,BP.fst.incr,
            (4 * (MAX_WORDS - 1))
    const   BP.fst.shift,L_05
    consth  BP.fst.shift,L_05
    add     BP.fst.shift,BP.fst.shift,
            BP.fst.skip
    setip   BP.dst.array,BP.src.array,
            BP.src.array
    mfsr    BP.dst.rgt_ptr,ipc
    add     BP.dst.rgt_ptr,BP.dst.rgt_ptr,
            BP.fst.incr
    mfsr    BP.src.rgt_ptr, ipa
    add     BP.src.rgt_ptr,BP.src.rgt_ptr,
            BP.fst.incr
    add     BP.fst.incr,BP.fst.incr,4
```

The sign of the relative alignment is placed into *BP.grp.align*; it will be used to determine whether to do a left or right shift. *BP.src.shift* is set to the value to be loaded into the funnel count register for use with the extract instructions, when shifting the source registers. The value is appropriate for either left or right shifting. If the actual value shift amount is zero—that is, no shifting is needed—then the sign bit is set for use with a conditional jump prior to the sequence of extract CT instructions.

```
    cpgt    BP.grp.align,BP.src.shift,0
    add     BP.src.shift,BP.src.shift,32
    and     BP.src.shift,BP.src.shift,31
    cpeq    BP.src.save,BP.src.shift,0
    or      BP.src.shift,BP.src.shift,
            BP.src.save
```

Each scan line begins at L_01. Copies of the source and destination addresses of the edge of the scan lines and the group count are moved into working registers.

```
L_01:
    add     BP.grp.count,BP.grp.repeat,0
    add     BP.dst.addr,BP.dst.lft_addr,0
    add     BP.src.addr,BP.src.lft_addr,0
    mtsr    cr,BP.fst.count
    loadm   0,0,BP.dst.array,BP.dst.addr
    mtsr    ipa,BP.dst.rgt_ptr
    andn    BP.dst.lft_end,BP.dst.array,
            BP.dst.lft_mask
    jmpf    BP.grp.align,L_03
    andn    BP.dst.rgt_end,gr0,
            BP.dst.rgt_mask
    add     BP.src.extra,BP.fst.count,1
    mtsr    cr,BP.src.extra
    loadm   0,0,BP.src.extra,BP.src.addr
    jmp     L_04
    add     BP.src.addr,BP.src.addr,4
```

The word at the left end of the first group (which may be an incomplete word) of the destination is fetched, and the portion outside the destination is retained and eventually written back into memory. If the first group is the only group, the word at the right end is fetched and masked as well.

The alignment direction flag in *BP.grp.align* is tested. A right shift results in a jump to L_03. If a left shift is necessary, an extra word is loaded into a register that is prefixed to the source register block. The remaining words of the source array are loaded into the register block.

```
L_03:
    mtsr    cr,BP.fst.count
    loadm   0,0,BP.src.array,BP.src.addr
```

At label L_04, the right-most source word of the first group is saved to be prefixed to the next group.

```
L_04:
    mtsr    ipa,BP.src.rgt_ptr
    jmpt    BP.src.shift,L_06
    add     BP.src.save,gr0,0
    jmpi    BP.fst.shift
    mtsr    fc,BP.src.shift
```

If the source and destination are identically aligned, no shifts are necessary, and the code jumps over the shift array to label L_06. This is the case regardless of the absolute alignment of the operands, which is handled by the special treatment of words at the left and right ends of the scan line.

```
L_05:
    .if MAX_WORDS == 32
    extract BP.src.array_31,
            BP.src.array_30,BP.src.array_31
    .
    .
    .
    extract BP.src.array_16,
            BP.src.array_15,BP.src.array_16
    .endif
```

```
.if MAX_WORDS >= 16
extract BP.src.array_15,
        BP.src.array_14,BP.src.array_15

.
.
.

extract BP.src.array_08,
        BP.src.array_07,BP.src.array_08
.endif

.if MAX_WORDS >= 8
extract BP.src.array_07,
        BP.src.array_06,BP.src.array_07
extract BP.src.array_06,
        BP.src.array_05,BP.src.array_06
extract BP.src.array_05,
        BP.src.array_04,BP.src.array_05
extract BP.src.array_04,
        BP.src.array_03,BP.src.array_04
.endif

extract BP.src.array_03,
        BP.src.array_02,BP.src.array_03
extract BP.src.array_02,
        BP.src.array_01,BP.src.array_02
extract BP.src.array_01,
        BP.src.array_00,BP.src.array_01
extract BP.src.array_00,
        BP.src.extra,BP.src.array_00
```

If the shift is necessary, the code jumps somewhere into the sequence of shift instructions. Each instruction either left- or right-shifts two adjacent registers in the source block and leaves the result in the register that is further to the right.

At label L_06, the user-supplied routine is called to perform the operation. The word at the left end of the destination array is restored. If there is only one group, the bits outside the destination (in the right-most word) are placed into the right-most word of the source array.

```
L_06:
   calli   ret,GP.pxl.op_vec
   add     BP.grp.op_skip,BP.fst.skip,0
   and     BP.dst.array,BP.dst.array,
           BP.dst.lft_mask
   jmpf    BP.grp.count,L_07
   or      BP.dst.array,BP.dst.array,
           BP.dst.lft_end
   mtsr    ipa,BP.dst.rgt_ptr
   mtsr    ipc,BP.dst.rgt_ptr
   nop
   and     gr0,gr0,BP.dst.rgt_mask
   or      gr0,gr0,BP.dst.rgt_end
```

At label L_07, the resulting register block is written into the destination bit map. If there is only a single group per scan line, the code jumps to L_11.

```
L_07:
   mtsr    cr,BP.fst.count
   jmpt    BP.grp.count,L_11
   storem  0,0,BP.dst.array,BP.dst.addr
   add     BP.dst.addr,BP.dst.addr,
           BP.fst.incr
   add     BP.src.addr,BP.src.addr,
           BP.fst.incr
   sub     BP.grp.count,BP.grp.count,1
```

If there is more than one group per scan line, the addresses are adjusted by the amount moved in the first group, and the code enters a loop at L_08 to process the rest of the scan line, 32 words at a time.

At label L_08, the 32 words of the destination array are loaded into the destination-register block, and the right-end word is saved.

```
L_08:
   mtsrim  cr,(MAX_WORDS - 1)
   loadm   0,0,BP.dst.array,
           BP.dst.addr
   andn    BP.dst.rgt_end,
           BP.dst.array_end,
           BP.dst.rgt_mask
   add     BP.src.extra,BP.src.save,0
   mtsrim  cr,(MAX_WORDS - 1)
   loadm   0,0,BP.src.array,
           BP.src.addr
   add     BP.src.addr,BP.src.addr,
           (4 * MAX_WORDS)
   jmpt    BP.src.shift,L_09
   add     BP.src.save,BP.src.array_end,0
   mtsr    fc,BP.src.shift

   .if MAX_WORDS == 32
   extract BP.src.array_31,
           BP.src.array_30,BP.src.array_31

   .
   .
   .

   extract BP.src.array_16,
           BP.src.array_15,BP.src.array_16
   .endif

   .if MAX_WORDS >= 16
   extract BP.src.array_15,
           BP.src.array_14,BP.src.array_15

   .
   .
   .
```

```
  extract BP.src.array_08,
          BP.src.array_07,BP.src.array_08
  .endif

  .if MAX_WORDS >= 8
  extract BP.src.array_07,
          BP.src.array_06,BP.src.array_07
  extract BP.src.array_06,
          BP.src.array_05,BP.src.array_06
  extract BP.src.array_05,
          BP.src.array_04,BP.src.array_05
  extract BP.src.array_04,
          BP.src.array_03,BP.src.array_04
  .endif

  extract BP.src.array_03,
          BP.src.array_02,BP.src.array_03
  extract BP.src.array_02,
          BP.src.array_01,BP.src.array_02
  extract BP.src.array_01,
          BP.src.array_00,BP.src.array_01
  extract BP.src.array_00,
          BP.src.extra,BP.src.array_00
```

The right-most word from the previous group is prefixed, and then the 32 words are fetched from the source bit map. The source address is incremented, and the right-most word is saved for the next group.

If no shift is necessary (the source and destination are identically aligned), the shift array is skipped. If a shift is necessary, it takes place.

At label L_09, the user routine is called to perform the operation. The right-most destination word is masked and merged if this is the last group.

```
L_09:
  calli   ret,GP.pxl.op_vec
  const   BP.grp.op_skip,0
  jmpf    BP.grp.count,L_10
  mtsrim  cr,(MAX_WORDS - 1)
  and     BP.dst.array_end,
          BP.dst.array_end,
          BP.dst.rgt_mask
  or      BP.dst.array_end,
          BP.dst.array_end,
          BP.dst.rgt_end
```

At label L_10, the group is written into the destination bit map, and the destination address is modified. The group count is decremented and tested. If further groups are necessary for the scan line, they are moved beginning at label L_08.

```
L_10:
  storem  0,0,BP.dst.array,BP.dst.addr
  jmpfdec BP.grp.count,L_08
  add     BP.dst.addr,BP.dst.addr,
          (4 * MAX_WORDS)
L_11:
  add     BP.dst.lft_addr,
          BP.dst.lft_addr,GP.mem.width
  jmpfdec Size.y,L_01
  add     BP.src.lft_addr,
          BP.src.lft_addr,Source.w
L_12:
```

The scan line count is decremented and tested. If further scan lines are necessary, the address of the left edge of each is calculated at label L_01.

P_B4_02.S

Routine **P_B4_02.S** is a C-language callable program that performs a copy-block operation in a monochrome (1-plane) bit map. Clipping is performed.

The caller is responsible for supplying the address of a routine that combines the two operands after they have been moved into the source and destination register blocks, and the source operand has been shifted to align with the destination. An example of such a routine is **O5_01.S**, which is included on the distribution diskette. This routine XORs the source array into the destination array.

The routine begins with the normal global functions. Fourteen parameters are loaded from the structure G29K_Params.

```
GP.wnd.min _x      lr2
GP.wnd.max_x       lr3
GP.wnd.min_y       lr4
GP.wnd.max_y       lr5
GP.pxl.value       lr6
GP.mem.width       lr7
GP.mem.depth       lr8
GP.wnd.base        lr9
GP.wnd.align       lr10
GP.pxl.op_vec      lr11
GP.pxl.in_mask     lr12
GP.pxl.do_mask     lr13
GP.pxl.do_value    lr14
GP.pxl.out_mask    lr15

const   Temp1,_G29K_Params
consth  Temp1,_G29K_Params
mtsrim  cr,(14 - 1)
loadm   0,0,GP.wnd.min_x,Temp1
```

From this point on, the routine is just like **P_B3_02.S**, except that the user routine is called to perform the operation on the operands.

The destination array is cropped so that it consists only of the array originally within the destination array *and* within the clipping window. The left edge of the destination array is cropped, if necessary, to the left edge of the clipping window. If the left edge must be cropped, the left edge of the source array and the block size are adjusted as well.

```
sub     Temp1,Dest.x,GP.wnd.min_x
jmpf    Temp1,L_01
add     Temp3,GP.wnd.max_x,1
add     Size.x,Size.x,Temp1

sub     Source.x,Source.x,Temp1
add     Dest.x,GP.wnd.min_x,0
```

If necessary, the right edge of the destination array is cropped to the right edge of the clipping window. If the right edge must be adjusted, the block size is adjusted as well.

```
L_01:
   add     Temp1,Dest.x,Size.x
   sub     Temp1,Temp3,Temp1
   jmpf    Temp1,L_02
   add     Temp2,GP.mem.width,0
   add     Size.x,Size.x,Temp1
```

If the width of the resulting block is less than or equal to zero, the routine exits immediately.

```
L_02:
   cple    Temp1,Size.x,0
   jmpt    Temp1,L_16
   sub     Temp1,GP.wnd.max_y,Dest.y
```

The top edge of the destination array is cropped, if necessary, to the top edge of the clipping window. If the top edge must be adjusted, the source top and the block size are adjusted as well.

```
   jmpf    Temp1,L_03
   sub     Temp3,GP.wnd.min_y,1
   add     Size.y,Size.y,Temp1
   add     Source.y,Source.y,Temp1
   add     Dest.y,GP.wnd.max_y,0
```

The bottom edge of the destination array is cropped, if necessary, to the bottom edge of the clipping window. If the bottom edge must be adjusted, the block size is adjusted as well.

```
L_03:
   sub     Temp1,Dest.y,Size.y
   sub     Temp1,Temp1,Temp3
   jmpf    Temp1,L_04
   sub     Size.y,Size.y,1
   add     Size.y,Size.y,Temp1
```

If the height of the resulting block is less than or equal to zero, the routine exits immediately.

```
L_04:
   jmpt    Size.y,L_16
   sub     Size.y,Size.y,1
```

Routine **S_M1_01** is called to convert the destination coordinates to a linear address and alignment. The results are left in *BP.dst.lft_addr* and *BP.dst.align*.

```
add     LP.loc.x,Dest.x,0
call    ret,S_M1_01
add     LP.loc.y,Dest.y,0
add     BP.dst.lft_addr,LP.loc.addr,0
add     BP.dst.align,LP.loc.align,0
```

Routine **S_M1_01** is called a second time to convert the source coordinates to a linear address and alignment. The base and width of the source bit map may be different from those of the destination.

```
add     BP.grp.count,GP.mem.width,0
add     GP.mem.width,Source.w,0
add     GP.wnd.base,Source.b,0
add     GP.wnd.align,Source.a,0
add     LP.loc.x,Source.x,0
call    ret,S_M1_01
add     LP.loc.y,Source.y,0
add     GP.mem.width,BP.grp.count,0
add     BP.src.lft_addr,LP.loc.addr,0
```

The difference in alignment between the source and destination is calculated to determine how far the source must be shifted. This is left in *BP.src.shift*.

```
sub     BP.src.shift,LP.loc.align,
        BP.dst.align
```

The masks for the left and right ends of a destination line are formed. These will be used to mask bits in the words at the ends of each scan line that are not in the destination block.

```
constn  BP.dst.rgt_mask,-1
srl     BP.dst.lft_mask,
        BP.dst.rgt_mask,BP.dst.align
add     BP.grp.align,BP.dst.align,
        Size.x
add     BP.grp.repeat,BP.grp.align,31
subr    BP.grp.align,BP.grp.align,32
and     BP.grp.align,BP.grp.align,31
sll     BP.dst.rgt_mask,
        BP.dst.rgt_mask,BP.grp.align
```

The number of groups in each scan line and the number of words in the first or only group of each scan line are computed. All other groups of each scan line will be exactly MAX_WORDS (32) words.

```
srl     BP.grp.repeat,BP.grp.repeat,5
sub     BP.grp.repeat,BP.grp.repeat,1
and     BP.fst.count,BP.grp.repeat,
```

```
        (MAX_WORDS - 1)
srl     BP.grp.repeat,BP.grp.repeat,
        MAX_SHIFT
sub     BP.grp.repeat,BP.grp.repeat,1
```

For the first group, there is a sequence of shift instructions beginning at L_09. Since the first group may not contain all 32 words, an indirect jump address into the sequence is generated and left in *BP.fst.shift*. The indirect address pointers are set to the source array and destination array. The address of the right-most word of the first scan line of the source array is computed.

```
sll     BP.fst.incr,BP.fst.count,2
subr    BP.fst.skip,BP.fst.incr,
        (4 * (MAX_WORDS - 1))
const   BP.fst.shift,L_09
consth  BP.fst.shift,L_09
add     BP.fst.shift,BP.fst.shift,
        BP.fst.skip
setip   BP.dst.array,BP.src.array,
        BP.src.array
mfsr    BP.dst.rgt_ptr, ipc
add     BP.dst.rgt_ptr,BP.dst.rgt_ptr,
        BP.fst.incr
mfsr    BP.src.rgt_ptr, ipa
add     BP.src.rgt_ptr,BP.src.rgt_ptr,
        BP.fst.incr
add     BP.fst.incr,BP.fst.incr,4
```

The sign of the relative alignment is placed into *BP.grp.align*; it will be used to determine whether to do a left or right shift.

```
cpgt    BP.grp.align,BP.src.shift,0
add     BP.src.shift,BP.src.shift,32
and     BP.src.shift,BP.src.shift,31
cpeq    BP.src.save,BP.src.shift,0
or      BP.src.shift,BP.src.shift,
        BP.src.save
```

Each scan line begins at L_05. Copies of the source and destination addresses of the edge of the scan lines and the group count are moved into working registers.

```
L_05:
  add     BP.grp.count,BP.grp.repeat,0
  add     BP.dst.addr,BP.dst.lft_addr,0
  add     BP.src.addr,BP.src.lft_addr,0
  mtsr    cr,BP.fst.count
  loadm   0,0,BP.dst.array,BP.dst.addr
  mtsr    ipa,BP.dst.rgt_ptr
  andn    BP.dst.lft_end,BP.dst.array,
          BP.dst.lft_mask
  jmpf    BP.grp.align,L_07
  andn    BP.dst.rgt_end,gr0,
          BP.dst.rgt_mask
  add     BP.src.extra,BP.fst.count,1
  mtsr    cr,BP.src.extra
  loadm   0,0,BP.src.extra,BP.src.addr
  jmp     L_08
  add     BP.src.addr,BP.src.addr,4
```

The word at the left end of the first group (which may be an incomplete word) of the destination is fetched, and the part outside the destination is saved and eventually written back into memory. If the first group is the only group, the word at the right end is fetched and masked as well.

The alignment direction flag in *BP.grp.align* is tested. A right shift results in a jump to L_07. If a left shift is necessary, an extra word is loaded into a register that is prepended to the source register block. The remaining words of the source array are loaded into the register block.

```
L_07:
  mtsr    cr,BP.fst.count
  loadm   0,0,BP.src.array,BP.src.addr
```

At label L_08, the right-most source word of the first group is saved to be prefixed to the next group.

```
L_08:
  mtsr    ipa,BP.src.rgt_ptr
  jmpt    BP.src.shift,L_10
  add     BP.src.save,gr0,0
  jmpi    BP.fst.shift
  mtsr    fc,BP.src.shift
```

If the source and destination are identically aligned, no shifts are necessary, and the code jumps over the shift array to label L_10. This is the case regardless of the absolute alignment of the operands, which is handled by the special treatment of words at the left and right ends of the scan line. If the shift is necessary, the code jumps somewhere into the sequence of shift instructions. Each instruction either left- or right-shifts two adjacent registers in the source block and leaves the result in the register that is further to the right.

```
L_09:
  .if MAX_WORDS == 32
  extract BP.src.array_31,
          BP.src.array_30,BP.src.array_31
   .
   .
   .
  extract BP.src.array_16,
          BP.src.array_15,BP.src.array_16
  .endif

  .if MAX_WORDS >= 16
  extract BP.src.array_15,
          BP.src.array_14,BP.src.array_15
```

```
   .
   .
   .
   extract BP.src.array_08,
           BP.src.array_07,BP.src.array_08
   .endif
   .if MAX_WORDS >= 8
   extract BP.src.array_07,
           BP.src.array_06,BP.src.array_07
   extract BP.src.array_06,
           BP.src.array_05,BP.src.array_06
   extract BP.src.array_05,
           BP.src.array_04,BP.src.array_05
   extract BP.src.array_04,
           BP.src.array_03,BP.src.array_04
   .endif

   extract BP.src.array_03,
           BP.src.array_02,BP.src.array_03
   extract BP.src.array_02,
           BP.src.array_01,BP.src.array_02
   extract BP.src.array_01,
           BP.src.array_00,BP.src.array_01
   extract BP.src.array_00,
           BP.src.extra,BP.src.array_00
```

At label L_10, the operation routine is called. The bits outside the destination (in the left-most word) are placed into the left-most word of the source array. If there is only one group, the bits outside the destination (in the right-most word) are placed into the right-most word of the source array.

```
L_10:
   calli   ret,GP.pxl.op_vec
   add     BP.grp.op_skip,BP.fst.skip,0
   and     BP.dst.array,BP.dst.array,
           BP.dst.lft_mask
   jmpf    BP.grp.count,L_11
   or      BP.dst.array,BP.dst.array,
           BP.dst.lft_end
   mtsr    ipa,BP.dst.rgt_ptr
   mtsr    ipc,BP.dst.rgt_ptr
   nop
   and     gr0,gr0,BP.dst.rgt_mask
   or      gr0,gr0,BP.dst.rgt_end
```

At label L_11, the resulting register block is written into the destination bit map. If there is only a single group per scan line, the code jumps to L_15.

```
L_11:
   mtsr    cr,BP.fst.count
   jmpt    BP.grp.count,L_15
   storem  0,0,BP.dst.array,BP.dst.addr
   add     BP.dst.addr,BP.dst.addr,
           BP.fst.incr
   add     BP.src.addr,BP.src.addr,
           BP.fst.incr
   sub     BP.grp.count,BP.grp.count,1
```

If there is more than one group per scan line, the addresses are adjusted by the amount moved in the first group, and the code enters a loop at L_12 to move the rest of the scan line 32 words at a time.

At label L_12, the destination words are fetched. The bits to the right of the destination block are preserved in case this is the last group.

```
L_12:
   mtsrim  cr,(MAX_WORDS - 1)
   loadm   0,0,BP.dst.array,
           BP.dst.addr
   andn    BP.dst.rgt_end,
           BP.dst.array_end,
           BP.dst.rgt_mask
   add     BP.src.extra,BP.src.save,0
   mtsrim  cr,(MAX_WORDS - 1)
   loadm   0,0,BP.src.array,BP.src.addr
   add     BP.src.addr,BP.src.addr,
           (4 * MAX_WORDS)
   jmpt    BP.src.shift,L_13
   add     BP.src.save,BP.src.array_end,0
   mtsr    fc,BP.src.shift

   .if MAX_WORDS == 32
   extract BP.src.array_31,
           BP.src.array_30,BP.src.array_31
   .
   .
   .
   extract BP.src.array_16,
           BP.src.array_15,BP.src.array_16
   .endif

   .if MAX_WORDS >= 16
   extract BP.src.array_15,
           BP.src.array_14,BP.src.array_15
   .
   .
   .
   extract BP.src.array_08,
           BP.src.array_07,BP.src.array_08
   .endif
   .if MAX_WORDS >= 8
   extract BP.src.array_07,
           BP.src.array_06,BP.src.array_07
   extract BP.src.array_06,
           BP.src.array_05,BP.src.array_06
   extract BP.src.array_05,
           BP.src.array_04,BP.src.array_05
   extract BP.src.array_04,
           BP.src.array_03,BP.src.array_04
   .endif
   extract BP.src.array_03,
           BP.src.array_02,BP.src.array_03
```

```
extractBP.src.array_02,
       BP.src.array_01,BP.src.array_02
extractBP.src.array_01,
       BP.src.array_00,BP.src.array_01
extractBP.src.array_00,
       BP.src.extra,BP.src.array_00
```

The right-most word from the previous group is prefixed, and then 32 words are fetched from the source bit map. The source address is incremented, and the right-most word is saved for the next group.

If no shift is necessary (because the source and destination are identically aligned), the shift array is skipped. If a shift is necessary, it takes place.

At label L_13, the user routine is called to perform the operation. If this is the last group, the right-end word is restored.

```
L_13:
  calli   ret,GP.pxl.op_vec
  const   BP.grp.op_skip,0
  jmpf    BP.grp.count,L_14
  mtsrim  cr,(MAX_WORDS - 1)
  and     BP.dst.array_end,
          BP.dst.array_end,
          BP.dst.rgt_mask
  or      BP.dst.array_end,
          BP.dst.array_end,
          BP.dst.rgt_end
```

At label L_14, the group is written into the destination bit map and the destination address is modified. The group count is decremented and tested. If further groups are necessary for the scan line, they are moved beginning at label L_12.

```
L_14:
  storem  0,0,BP.dst.array,BP.dst.addr
  jmpfdecBP.grp.count,L_12
  add     BP.dst.addr,BP.dst.addr,
          (4 * MAX_WORDS)
L_15:
  add     BP.dst.lft_addr,
          BP.dst.lft_addr,GP.mem.width
  jmpfdecSize.y,L_05
  add     BP.src.lft_addr,
          BP.src.lft_addr,Source.w
L_16:
```

The scan line count is decremented and tested. If further scan lines are necessary, the address of the left edge of each is calculated at label L_05.

Text Routines

There are two C-language callable routines for text operations: **P_T1_01.S**, which does not perform clipping, and **P_T1_02.S**, which does.

The rasterized characters must be stored in memory before the text routines can be called. The first word of each character form specifies its size. The following words contain the bit patterns for the character. The bits begin with the top row, left to right, and continue with following rows, left to right. The bits are packed into just as many words as are necessary to contain them. The only unused bits are the least-significant bits of the last word. The first word contains five fields, as shown in Table 10.

Table 10. Bit Assignments for First Word of Rasterized Character Format

Bits	Function	Value Range	Value in Figure 12
31–21	Height in scan lines	0...63	9
25–20	Width in pixels	0...63	7
19–14	Inset to left side	–32...31	1
13–07	Ascent to top	–32...95	8
06–00	Pitch to next char	0...127	9

In Figure 12, the hex bit patterns used to form the character 'A' were 0x1020E1C6, 0xCDBFE3C6.

The two text routines begin with the same declarations. The function name is declared to be global, the ENTER macro is used to specify that 48 general registers are required, and the routine name appears as a label.

```
  .global_P_T1_01
  ENTER   TEXT_PRIMITIVE
_P_T1_01:
```

The macros used to form the words are in routine **TEST_T1.C**

The three parameter register names are declared with PARAM macros. These assign local register-numbers higher than (or above) the registers previously defined. These parameters are passed in the local registers shown below:

Macro	Register Name	Register Number
PARAM	Pos.x	lr50
PARAM	Pos.y	lr51
PARAM	Form	lr52

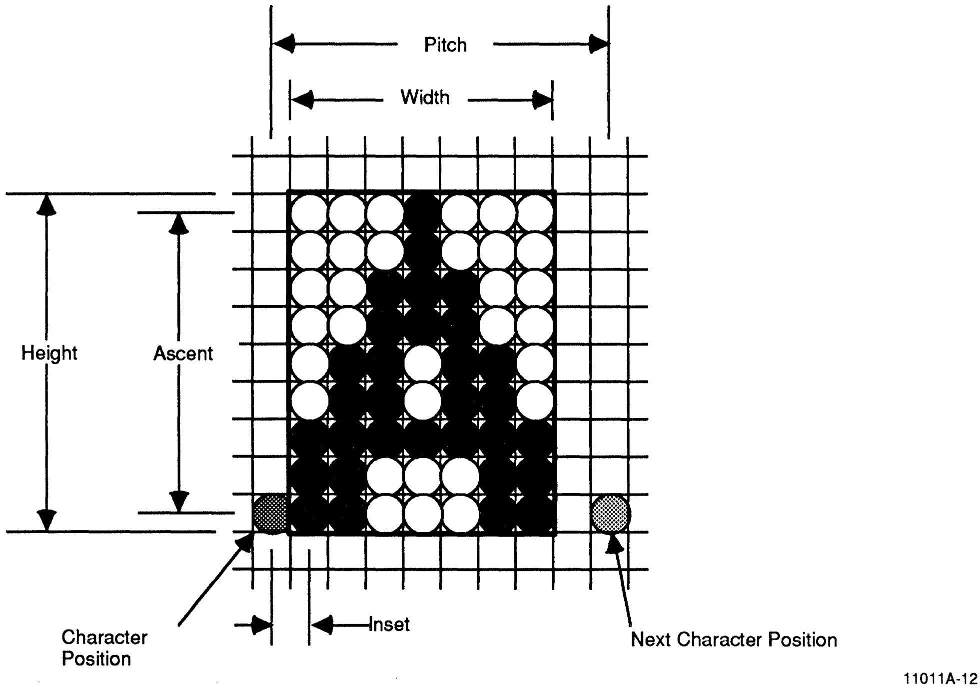


Figure 12. Character Parameters

Pos.x and *Pos.y* together indicate where the character is to be drawn. The routine updates *Pos.x* according to the pitch parameter of the character drawn. Form is the address of the character definition.

The CLAIM macro is the function prologue. If a spill operation is not necessary, this consists of five instructions. If a spill is necessary, the standard **SPILL** routine is used, which may involve a Load/Store Multiple instruction.

P_T1_01.S

Routine **P_T1_01.S** draws the indicated character into a color bit map at the specified location. No clipping is performed. The routine begins with the normal global functions.

Five parameters are loaded from the structure G29K_Params.

```
GP.pxl.value          lr6
GP.mem.width          lr7
GP.mem.depth          lr8
GP.wnd.base           lr9
GP.wnd.align         lr10
const   Temp0,_G29K_Params + (4 * 4)
consth Temp0,_G29K_Params + (4 * 4)
mtsrim cr,(5 - 1)
```

```
loadm   0,0,GP.pxl.value,Temp0
```

The first word of the character form is loaded into *Temp0* and divided into its five components. The variables *TP.char.inset* and *TP.char.ascent* are each offset by 32 and must be adjusted.

```
load    0,0,Temp0,Form
srl     TP.chr.high,Temp0,26
srl     TP.chr.wide,Temp0,20
and     TP.chr.wide,TP.chr.wide,63
srl     TP.chr.inset,Temp0,14
and     TP.chr.inset,TP.chr.inset,63
sub     TP.chr.inset,TP.chr.inset,32
srl     TP.chr.ascent,Temp0,7
and     TP.chr.ascent,TP.chr.ascent,127
sub     TP.chr.ascent,TP.chr.ascent,32
and     TP.chr.pitch,Temp0,127
```

Routine **S_M1_01** is called to convert the destination coordinates to a linear address and alignment. The inset is added to the x position and the ascent is added to the y position.

```
add     LP.loc.x,Pos.x,TP.chr.inset
call    ret,S_M1_01
add     LP.loc.y,Pos.y,TP.chr.ascent
```

```
sll     TP.ptn.next,TP.chr.wide,
        GP.mem.depth
sub     TP.ptn.next,GP.mem.width,
        TP.ptn.next
const   TP.ptn.shift_set,
        (0x80000000 + 30)
consth  TP.ptn.shift_set,
        (0x80000000 + 30)
sub     TP.ptn.high,TP.chr.high,2
sub     TP.ptn.wide,TP.chr.wide,2
```

Variable *TP.ptn.next* is set to the value necessary to increment from the last pixel of a scan line in the character cell to the first pixel of the next scan line. Variable *TP.ptn.shift_set* is initialized to count bits in the pattern words, and is loaded into *TP.ptn.shift_count* each time a new pattern word is fetched. The initial value is negative. *TP.ptn.high* and *TP.ptn.wide* are set to *TP.chr.high* – 2, and *TP.chr.wide* – 2, respectively.

The character-drawing loop begins at label L_01. The pointer in Form is moved to the first pattern word, and the word is loaded into *TP.ptn.mask*. *TP.ptn.shift_count* is renewed from *TP.ptn.shift_set*, and the code jumps to L_03.

```
L_01:
   add     Form, Form,4
   load    0,0,TP.ptn.mask,Form
   jmp     L_03
   add     TP.ptn.shift_count,
           TP.ptn.shift_set,0

L_02:
   jmpfdec TP.ptn.shift_count,L_01
   sll     TP.ptn.mask,TP.ptn.mask,1
```

At label L_03, the high-order bit of the current mask word is tested. If it is a zero, the code jumps to L_04. If it is necessary to write both a foreground and background color, one would add code to write a background color instead of just jumping to L_04. If the bit in the mask register is a 1, the current pixel color is written into the current pixel location.

```
L_03:
   jmpf    TP.ptn.mask,L_04
   nop
   store   0,0,GP.pxl.value,LP.loc.addr
```

At label L_04, variable *TP.ptn.wide* is decremented and tested. If it does not become negative, the destination scan line need not change. The pixel location is incremented to the next pixel, and the code returns to label L_02, where it tests for bits remaining in the current mask word.

```
L_04:
   jmpfdec TP.ptn.wide,L_02
   add     LP.loc.addr,LP.loc.addr,
           PIXEL_SIZE
   add     LP.loc.addr,LP.loc.addr,
           TP.ptn.next
   jmpfdec TP.ptn.high,L_02
   sub     TP.ptn.wide,TP.chr.wide,2
   add     v0,Pos.x,TP.chr.pitch
   add     v1,Pos.y,0
```

If *TP.ptn.wide* becomes negative, the current scan line is complete. The current pixel location is adjusted to the first pixel of the next scan line. *TP.ptn.high* is tested to determine if more scan lines are necessary. If not, the routine returns the address of the next character and exits. If it is necessary to process more scan lines, *TP.ptn.wide* is renewed, and the code jumps to L_02.

At label L_02, *TP.ptn.shift_count* is decremented and tested. If the current pattern word is not exhausted, it is left-shifted and tested at L_03, as described above. If the current pattern word is exhausted, the code continues at label L_01, where the next pattern word is fetched.

P_T1_02.S

Routine **P_T1_02.S** draws the indicated character into a color bit map at the specified location, with clipping. The routine begins with the normal global functions.

This routine is almost exactly the same as **P_T1_01.S** described above, except for clipping. The clipping is performed in the same way as described for **P_L1_02.S**. Just prior to writing a pixel, the routine asserts that the pixel is inside the clipping window. If the pixel is not inside the window, it is not drawn. If no subsequent pixels could be inside the window, the routine terminates.

Nine parameters are loaded from the structure G29K_Params.

```
GP.wnd.min_x          lr2
GP.wnd.max_x          lr3
GP.wnd.min_y          lr4
GP.wnd.max_y          lr5
GP.pxl.value          lr6
GP.mem.width          lr7
GP.mem.depth          lr8
GP.wnd.base           lr9
GP.wnd.align          lr10
const   Temp0,_G29K_Params
consth  Temp0,_G29K_Params
mtsrim  cr,(9 - 1)
loadm   0,0,GP.wnd.min_x,Temp0
```

The first word of the character form is loaded into *Temp0* and divided into its five components. The inset and ascent are offset by 32 and must be adjusted.

```
load    0,0,Temp0,Form
srl     TP.chr.high,Temp0,26
```

```
srl     TP.chr.wide,Temp0,20
and     TP.chr.wide,TP.chr.wide, 63
srl     TP.chr.inset,Temp0,14
and     TP.chr.inset,TP.chr.inset, 63
sub     TP.chr.inset,TP.chr.inset,32
srl     TP.chr.ascent,Temp0,7
and     TP.chr.ascent,TP.chr.ascent,127
sub     TP.chr.ascent,TP.chr.ascent,32
and     TP.chr.pitch,Temp0,127
```

Routine **S_M1_01** is called to convert the destination coordinates to a linear address and alignment. The inset is added to the x position and the ascent is added to the y position.

```
add     LP.loc.x,Pos.x,TP.chr.inset
call    ret,S_M1_01
add     LP.loc.y,Pos.y,TP.chr.ascent
sll     TP.ptn.next,TP.chr.wide,
        GP.mem.depth
sub     TP.ptn.next,GP.mem.width,
        TP.ptn.next
const   TP.ptn.shift_set,
        (0x80000000 + 30)
consth  TP.ptn.shift_set,
        (0x80000000 + 30)
sub     TP.ptn.high,TP.chr.high,2
sub     TP.ptn.wide,TP.chr.wide,2
const   LP.clp.skip_vec,L_04
consth  LP.clp.skip_vec,L_04
const   LP.clp.stop_vec,L_05
consth  LP.clp.stop_vec,L_05
```

Variable *TP.ptn.next* is set to the value necessary to increment from the last pixel of a scan line in the character cell, to the first pixel of the next scan line. Variable *TP.ptn.shift_set* is initialized to count bits in the pattern words. It is loaded into *TP.ptn.shift_count* each time a new pattern word is fetched. The initial value is negative. *TP.ptn.high* and *TP.ptn.wide* are set to *TP.chr.high*–2 and *TP.chr.wide*–2, respectively.

The character-drawing loop begins at label L_01. The pointer in Form is moved to the first pattern word and the word is loaded into *TP.ptn.mask*. The variable *TP.ptn.shift_count* is renewed from *TP.ptn.shift_set*, and the code jumps to L_03.

```
L_01:
  add     Form,Form,4
  load    0,0,TP.ptn.mask,Form
  jmp     L_03
  add     TP.ptn.shift_count,
          TP.ptn.shift_set,0

L_02:
  jmpfdec TP.ptn.shift_count,L_01
  sll     TP.ptn.mask,TP.ptn.mask,1
```

At label L_03, the high-order bit of the current mask word is tested. If it is a 0, the code jumps to L_04. If it is necessary to write both a foreground and background color, one would add code to write a background color instead of simply jumping to L_04.

```
L_03:
  jmpf    TP.ptn.mask,L_04
  nop
  asle    V_CLIP_SKIP,LP.loc.y,
          GP.wnd.max_y
  asge    V_CLIP_SKIP,LP.loc.x,
          GP.wnd.min_x
  asle    V_CLIP_SKIP,LP.loc.x,
          GP.wnd.max_x
  asge    V_CLIP_STOP,LP.loc.y,
          GP.wnd.min_y
  store   0,0,GP.pxl.value,LP.loc.addr
```

If the bit in the mask register is a 1, the current pixel color is written into the current pixel location, if it is within the clipping window. If any of the first three asserts fail, the pixel is outside the window, but there is a possibility that further pixels in the character cell may still be in the window. In this case, the store is skipped. If the fourth assert fails, the pixel is below the window, which means no more pixels can possibly be in the window. In this case, the routine terminates.

At label L_04, variable *TP.ptn.wide* is decremented and tested. If it is not negative, the destination scan line need not change. The pixel location is incremented to the next pixel, and the code returns to label L_02, where it tests for bits remaining in the current mask word.

```
L_04:
  add     LP.loc.x,LP.loc.x,1
  jmpfdec TP.ptn.wide,L_02
  add     LP.loc.addr,LP.loc.addr,
          PIXEL_SIZE
  add     LP.loc.x,Pos.x,TP.chr.inset
  sub     LP.loc.y,LP.loc.y,1
  add     LP.loc.addr,LP.loc.addr,
          TP.ptn.next
  jmpfdec TP.ptn.high,L_02
  sub     TP.ptn.wide,TP.chr.wide,2

L_05:
  add     v0,Pos.x,TP.chr.pitch
  add     v1,Pos.y,0
```

When *TP.ptn.wide* becomes negative, the current scan line is complete. The current pixel location is adjusted to the first pixel of the next scan line. *TP.ptn.high* is tested to determine if more scan lines are necessary. If not, the routine returns the address of the next character and then exits.

If it is necessary to process more scan lines, *TP.ptn.wide* is renewed, and the code jumps to L_02.

At label L_02, *TP.ptn.shift_count* is decremented and tested. If the current pattern word is not exhausted, it is left-shifted and tested at L_03, as described above. If the current pattern word is exhausted, the code continues at label L_01, where the next pattern word is fetched.

Filled-Triangle Routines

There are eight functions for filled triangles. They are shown in Table 11.

Table 11. Routines For Filled Triangles

Routine	Function
P_S1_01.S	Shaded triangle, no clipping
P_S1_02.S	Shaded triangle, with clipping
P_F1_01.S	Solid filled triangle, no clipping
P_F1_02.S	Solid filled triangle, with clipping
P_F2_01.S	General filled triangle, no clipping
P_F2_02.S	General filled triangle, with clipping
P_F3_01.S	Monochrome filled triangle, no clipping
P_F4_01.S	General monochrome triangle, no clipping

Shaded Triangles

All shaded-triangle routines begin with similar declarations. The function name is declared to be global, the ENTER macro is used to specify that the appropriate general registers are required, and the routine name appears as a label.

```
 .global_P_S1_01
 ENTER   SHADE_PRIMITIVE
_P_S1_01:
```

The nine parameter register names are declared with PARAM macros. These assign local-register numbers higher than (or above) the registers previously defined. These parameters are passed in registers.

```
 PARAM   P1.x
 PARAM   P1.y
 PARAM   I1
 PARAM   P2.x
 PARAM   P2.y
 PARAM   I2
 PARAM   P3.x
 PARAM   P3.y
 PARAM   I3
```

P(n).x, P(n).y, and I(n) together specify the x.y coordinates and the intensity of a point. The triangle must be specified with three points.

The CLAIM macro is the function prologue. If a spill operation is not necessary, this consists of five instructions. If a spill is necessary, the standard SPILL routine is used, which may involve a Load/Store Multiple.

Filled Triangles

The filled-triangle routines begin with similar declarations. The function name is declared to be global, the ENTER macro is used to specify that the appropriate general registers are required, and the routine name appears as a label.

```
 .global_P_F1_01
 ENTER   FILL_PRIMITIVE
_P_F1_01:
```

The six parameter register names are declared with PARAM macros. These assign local register-numbers higher than (or above) the registers previously defined. These parameters are passed in registers.

```
 PARAM   P1.x
 PARAM   P1.y
 PARAM   P2.x
 PARAM   P2.y
 PARAM   P3.x
 PARAM   P3.y
```

Pn.x and *Pn.y* together specify the x.y coordinates of a vertex. The triangle must be specified with three vertexes.

The CLAIM macro is the function prologue. If a spill operation is not necessary, this consists of five instructions. If a spill is necessary, the standard SPILL routine is used, which may involve a Load/Store Multiple.

Table 12 shows the input parameters that must be present before each routine can be called.

Table 12. Local-Register Usage in Filled-Triangle Routines

Variable	Reg	S1_01	S1_02	F1_01	F1_02	F2_01	F2_02	F3_01	F4_01
GP.wnd.min_x	lr2		X		X		X		
GP.wnd.max_x	lr3		X		X		X		
GP.wnd.min_y	lr4		X		X		X		
GP.wnd.max_y	lr5		X		X		X		
GP.pxl.value	lr6	X	X	X	X	X	X	X	X
GP.mem.width	lr7	X	X	X	X	X	X	X	X
GP.mem.depth	lr8	X	X	X	X	X	X	X	X
GP.wnd.base	lr9	X	X	X	X	X	X	X	
GP.wnd.align	lr10	X	X	X	X	X	X	X	
GP.pxl.op_vec	lr11	X	X			X	X		
GP.pxl.in_mask	lr12	X	X			X	X		
GP.pxl.do_mask	lr13	X	X			X	X		
GP.pxl.do_value	lr14	X	X			X	X		
GP.pxl.out_mask	lr15	X	X			X	X		

BENCHMARKS

The purpose of this section is to present some benchmarks for the Am29000 as a graphics processor. Rendering times are presented for a number of fundamental operations, including line drawing (vectors), BITBLT, strings, and triangle fill. The benchmarks were obtained by running the programs on the Architectural Simulator.

Each benchmarking program performs the following basic operations:

1. Initialize the bit map if necessary.
2. Read cycle counter.
3. Call a null function n times.
4. Read cycle counter. Calculate overhead.
5. Call the object function n times.
6. Read cycle counter. Calculate actual.
7. Print results (overhead, actual, actual minus overhead).

The bit map is initialized if its contents affect the execution time, as is the case for some arithmetic operations.

```
Mem = (unsigned long *)BitMap;
Count = 65536;
while ( Count--)
      *Mem++ = 0L;
```

In step 2, the Am29000 built-in cycle counter is read. This counter increments continuously, once every machine cycle, whenever the Am29000 is running.

```
Time = _cycles ();
```

A null function is called the same number of times that the actual function is called.

```
Items = 0;
for (ENDX = 0, EndY = 10;
       EndX < 10; ++EndX)
{
T_Empty (0,0,EndX,EndY);
++ Items;
 }
```

The cycle counter is read to determine the number of cycles that are spent calling the null function. Then, a new base time is obtained.

```
Over = _cycles () - Time;
Time = _cycles ();
```

The object function is then called n times.

```
Items = 0;
for (ENDX = 0, EndY = 10;
       EndX < 10; ++EndX)
{
/*draw a line*/
P_L1_01 (0,0,EndX,EndY);
++ Items;
 }
```

The cycle counter is read again, and the actual time is determined.

```
Time = _cycles () - Time;
```

Finally, a report is printed (see listing 1). An actual printout looks like Listing 2.

Listing 1. Benchmark Program Listing

```
Avrg = ((Time - Over) + Items / 2) / Items;    /*round up*/
printf ("Time_001:  %6u  cycles/vector     ", Avrg);
printf ("[ %5u vectors : %-20s ]\n", Items, "10 pixels each");
printf ("     %10u (actual)    - %10u (overhead)  = %10u cycles \n"
          Time, Over, Time - Over);
printf ("--(10-pixel vectors with P_L1_01, direct,  ");
printf ("unclipped )  --\n");
printf (" \n");
```

Listing 2. Benchmark Printout

```
 Time_001:     125 cycles/vectors  [80 vectors : 10 pixels each]
           338972 (actual)      328952  (overhead) = 10020 cycles
--(10-pixel vectors with P_L1_01,  direct,  unclipped)
```

Hardware Models

Three hardware models are benchmarked. In each case, the cycle time is 40 ns.

The first model is the Personal Computer Execution Board (PCEB29K), with limited burst-mode capability and Branch Target Cache disabled.

The second model is a typical mid-range system, with two-cycle first access and single-cycle burst. Such a system could be implemented using an instruction cache, or with an interleaved static memory, as described in the *Am29000 Memory Design Handbook*.

The third model is a very fast, single-cycle system.

The memory parameters for each model are given in Table 13.

Table 13. Memory Parameters (Wait States) for Hardware Models

Model	PCEB	Mid-Range	Fast
I–Fetch (First)	5	2	1
I–Fetch (Burst)	1	1	(n/a)
D–Fetch (First)	4	4	1
D–Fetch(Burst)	(n/a)	1	(n/a)

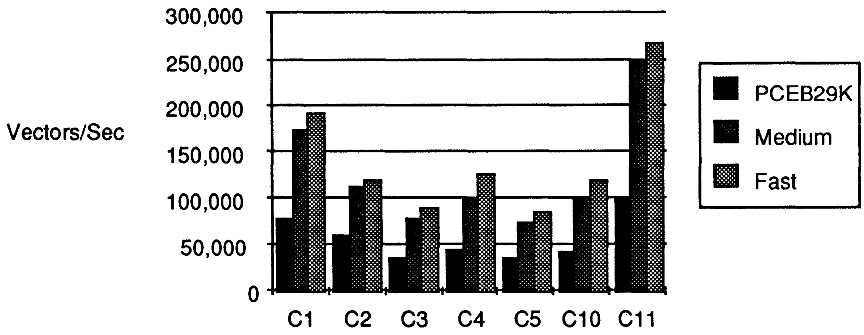
Benchmark Results

Benchmark results are presented in Figures 13 through 27 and are discussed in the subsections below.

Vectors

A total of 11 numbers are reported for each of the three models. The metric is made up of 10-pixel, randomly-oriented vectors per second. Both ends are specified for each vector. All numbers are for 32-bit pixels, except C10 which is monochrome. The numbers include setup and actual pixel-drawing time. A graphic representation is used.

Figure 13 shows the drawing performance in vectors per second for single-width vectors. Figure 14 shows the performance for wide and anti-aliased lines.

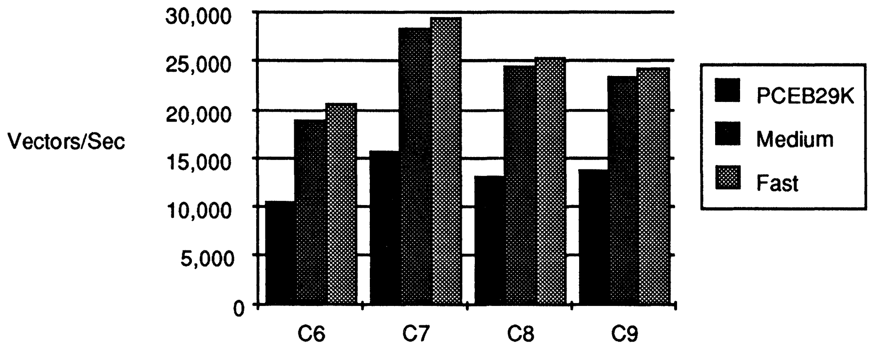


Case	PCEB29K	Medium	Fast	
C1	87,108	176,056	192,308	(Vectors/Sec)
C2	65,963	110,619	117,925	
C3	41,254	77,882	91,575	
C4	47,259	102,459	127,551	
C5	37,879	72,674	84,746	
C10	45,372	101,215	119,617	
C11	100,806	250,000	268,817	

Case	Routine	Function
C1	P_L1_01	Unclipped, set
C2	P_L1_02	Clipped, set
C3	P_L2_01	Unclipped, XOR (restricted)
C4	P_L2_01	Unclipped, XOR (unrestricted)
C5	P_L2_02	Clipped, XOR (unrestricted)
C10	P_L4_01	Unclipped, monochrome, set
C11	P_L5_01	Unclipped, fixed width, no window, set

11011A-13

Figure 13. Benchmark Results for Single-Width Line Functions (Vectors/Sec)



Case	PCEB29K	Medium	Fast	
C6	12,880	19,577	21,386	(Vectors/Sec)
C7	16,689	28,027	29,274	
C8	13,959	24,062	25,304	
C9	14,393	22,810	23,607	

Case	Routine	Function
C6	P_L3_01	Unclipped, anti-aliased, max, width = 1
C7	P_L3_01	Unclipped, anti-aliased, set, width = 1
C8	P_L3_01	Unclipped, anti-aliased, set, width = 2
C9	P_L3_02	Clipped, anti-aliased, set, width = 1

11011A-14

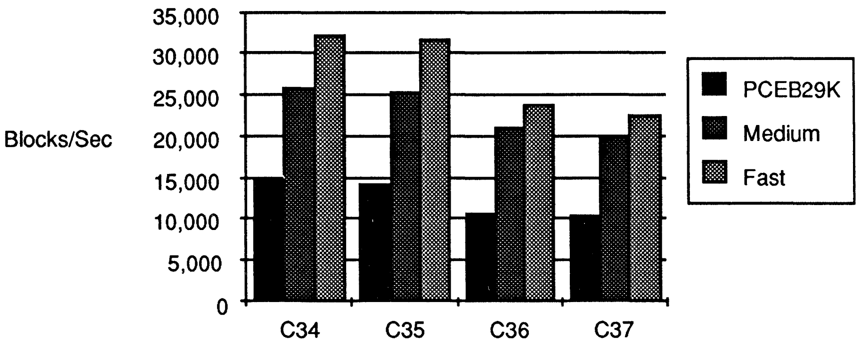
Figure 14. Benchmark Results for Wide/AA Line Functions (Vectors/Sec)

BITBLT

A total of 16 numbers are reported for each of the three models. There are four variables, each with two cases. The variables are:

Block Size	16 x 16	256 x 256
Bits/Pixel	1	32
Clipping	Off	On
Operation	Copy	XOR/Add with Saturation

The benchmark performance for BITBLT is summarized in Figures 15 through 22.

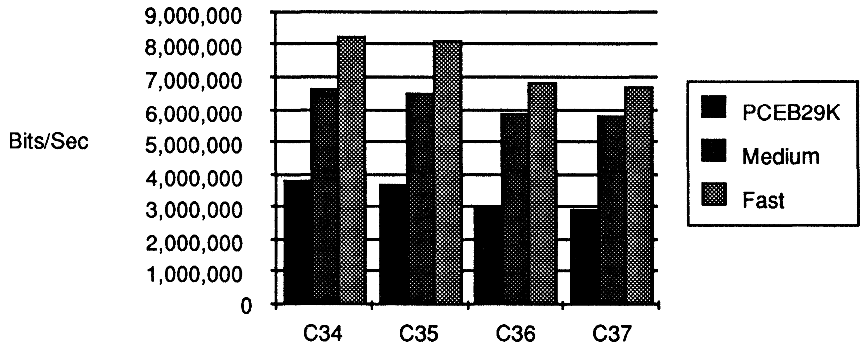


Case	PCEB	Medium	Fast	
C34	14,775	25,934	32,216	(Blocks/Sec)
C35	14,269	25,407	31,807	
C36	11,579	23,020	26,767	
C37	11,251	22,748	26,288	

Case	Routine	Function
C34	P_B3_01	Copy, unclipped
C35	P_B3_02	Copy, clipped
C36	P_B4_01	XOR, unclipped
C37	P_B4_02	XOR, clipped

11011A-15

Figure 15. Benchmark Results for BITBLT 16x16 Monochrome Functions (Blocks/Sec)

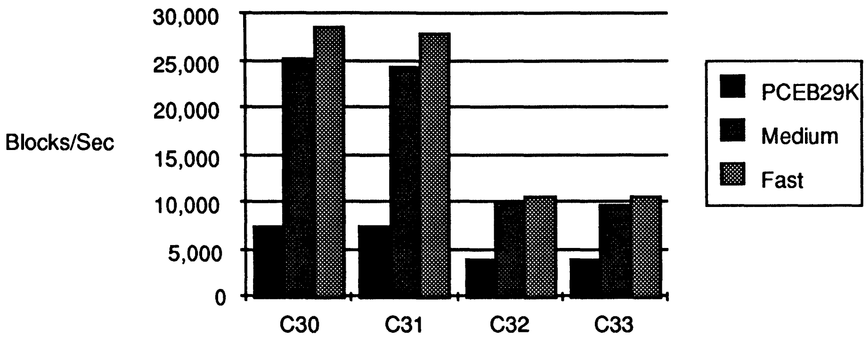


Case	PCEB	Medium	Fast	
C34	3,782,506	6,639,004	8,247,423	(Bits/Sec)
C35	3,652,968	6,504,065	8,142,494	
C36	2,964,335	5,893,186	6,852,248	
C37	2,880,288	5,823,476	6,729,758	

Case	Routine	Function
C34	P_B3_01	Copy, unclipped
C35	P_B3_02	Copy, clipped
C36	P_B4_01	XOR, unclipped
C37	P_B4_02	XOR, clipped

11011A-16

Figure 16. Benchmark Results for BITBLT 16x16 Monochrome Functions (Bits/Sec)

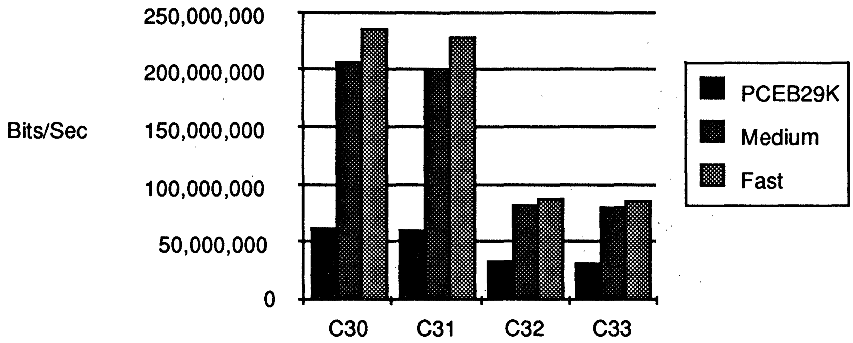


Case	PCEB29K	Medium	Fast	
C30	7,485	25,253	28,703	(Blocks/Sec)
C31	7,403	24,534	27,964	
C32	3,976	9,913	10,684	
C33	3,922	9,827	10,566	

Case	Routine	Function
C30	P_B1_01	Copy, unclipped
C31	P_B1_02	Copy, clipped
C32	P_B2_01	Add w/saturation, unclipped
C33	P_B2_02	Add w/saturation, clipped

11011A-17

Figure 17. Benchmark Results for BITBLT 16x16 Color Functions (Blocks/Sec)

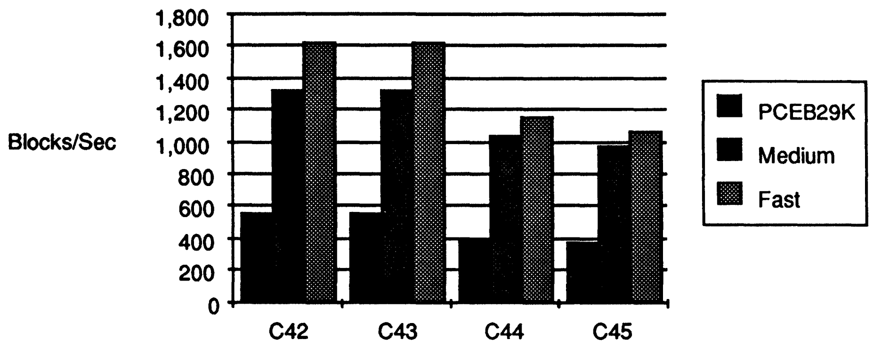


Case	PCEB	Medium	Fast	
C30	61,317,365	206,868,687	235,132,032	(Bits/Sec)
C31	60,645,543	200,981,354	229,082,774	
C32	32,575,155	81,205,393	87,521,368	
C33	32,125,490	80,503,145	86,559,594	

Case	Routine	Function
C30	P_B1_01	Copy, unclipped
C31	P_B1_02	Copy, clipped
C32	P_B2_01	Add w/saturation, unclipped
C33	P_B2_02	Add w/saturation, clipped

11011A-18

Figure 18. Benchmark Results for BITBLT 16x16 Color Functions (Bits/Sec)

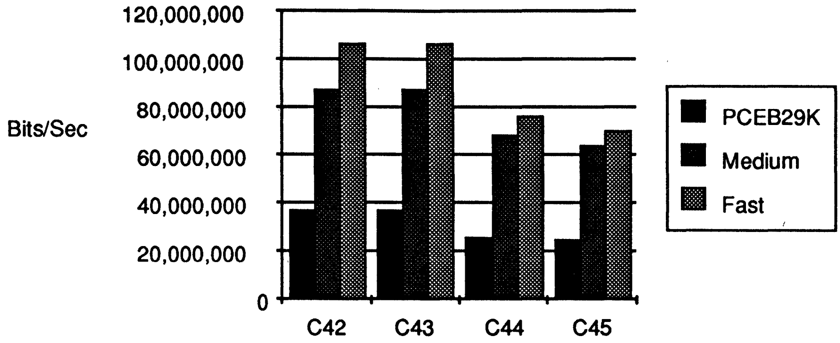


Case	PCEB29K	Medium	Fast	
C42	558	1,331	1,618	(Blocks/Sec)
C43	558	1,331	1,617	
C44	392	1,035	1,157	
C45	378	972	1,068	

Case	Routine	Function
C42	P_B3_01	Copy, unclipped
C43	P_B3_02	Copy, clipped
C44	P_B4_01	XOR, unclipped
C45	P_B4_02	XOR, clipped

11011A-19

Figure 19. Benchmark Results for BITBLT 256x256 Monochrome Functions (Blocks/Sec)

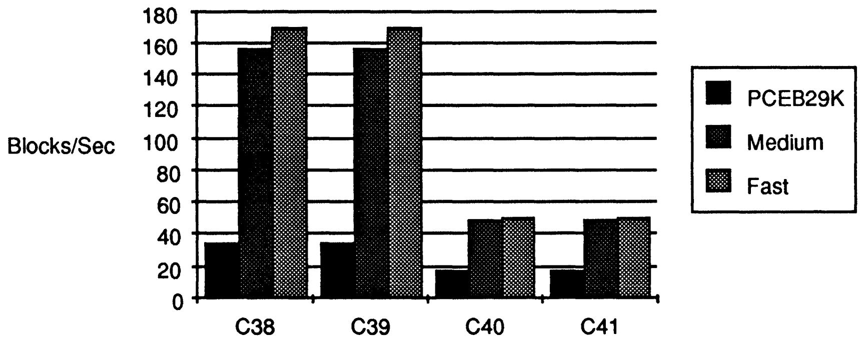


Case	PCEB29K	Medium	Fast	
C42	36,595,117	87,251,038	106,059,037	(Bits/Sec)
C43	36,553,478	87,237,101	105,976,714	
C44	25,666,975	67,839,841	75,802,720	
C45	24,775,442	63,701,400	70,002,136	

Case	Routine	Function
C42	P_B3_01	Copy, unclipped
C43	P_B3_02	Copy, clipped
C44	P_B4_01	XOR, unclipped
C45	P_B4_02	XOR, clipped

11011A-20

Figure 20. Benchmark Results for BITBLT 256x256 Monochrome Functions (Bits/Sec)

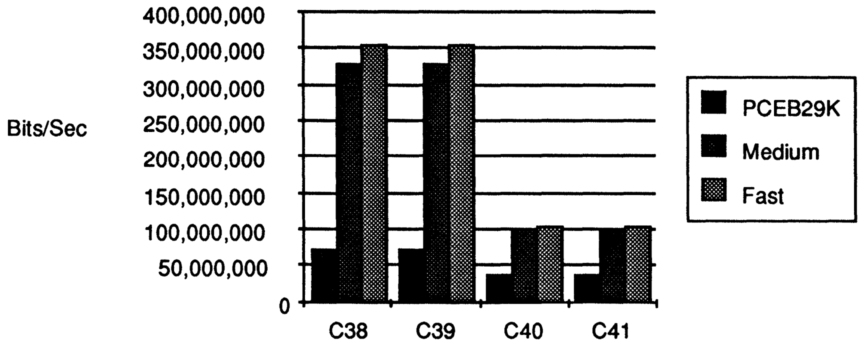


Case	PCEB29K	Medium	Fast	
C38	35	157	170	(Blocks/Sec)
C39	35	157	170	
C40	18	48	50	
C41	18	48	50	

Case	Routine	Function
C38	P_B1_01	Copy, unclipped
C39	P_B1_02	Copy, clipped
C40	P_B2_01	Add w/saturation, unclipped
C41	P_B2_02	Add w/saturation, clipped

11011A-21

Figure 21. Benchmark Results for BITBLT 256x256 Color Functions (Blocks/Sec)



Case	PCEB29K	Medium	Fast	
C38	73,920,216	328,312,001	355,741,320	(Bits/Sec)
C39	73,914,380	328,252,390	355,685,812	
C40	37,791,752	101,590,254	105,362,074	
C41	37,794,776	101,587,695	105,357,416	

Case	Routine	Function
C38	P_B1_01	Copy, unclipped
C39	P_B1_02	Copy, clipped
C40	P_B2_01	Add w/saturation, unclipped
C41	P_B2_02	Add w/saturation, clipped

11011A-22

Figure 22. Benchmark Results for BITBLT 256x256 Color Functions (Bits/Sec)

Text

The performance of text representation was benchmarked using a single case, in which each character is represented as a 7-by-9-pixel matrix. The results of the text benchmark are shown in Figure 23.

Filled Triangles

Two sets of benchmarks were run for filled triangles. The small triangles have 10-pixel sides, and the large triangles have 50-pixel sides. The shading is linear along scan lines (Gouraud shading).

The benchmark results for filled triangles, given in triangles per second, are summarized in Figures 24 through 27.

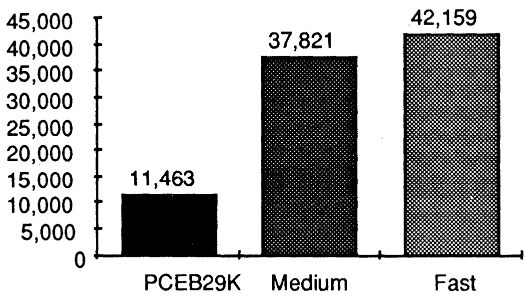
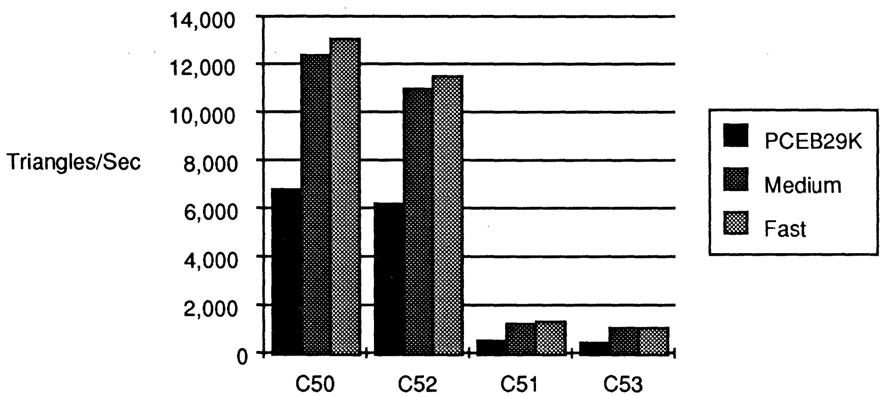


Figure 23. Benchmark Results for Text Functions (Characters/Sec)

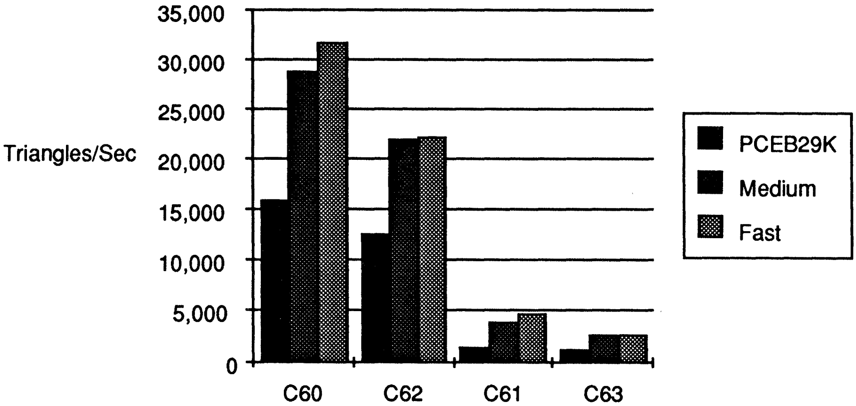


Case	PCEB29K	Medium	Fast	
C50	6,792	12,389	13,062	(Triangles/Sec)
C52	6,241	11,008	11,457	
C51	544	1,248	1,329	
C53	500	1,056	1,111	

Case	Routine	Function
C50	P_S1_01.S	10 Pixel sides, shaded, unclipped
C52	P_S1_02.S	10 Pixel sides, shaded, clipped
C51	P_S1_01.S	50 Pixel sides, shaded, unclipped
C53	P_S1_02.S	50 Pixel sides, shaded, clipped

11011A-24

Figure 24. Benchmark Results for Shaded Triangle Functions (Triangles/Sec)

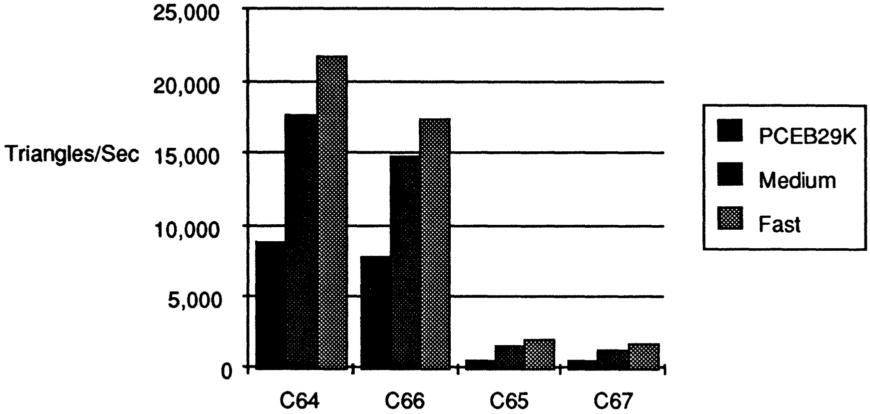


Case	PCEB29K	Medium	Fast	
C60	15,924	28,802	31,847	(Triangles/Sec)
C62	12,697	21,949	22,222	
C61	1,530	3,893	4,773	
C63	1,261	2,655	2,660	

Case	Routine	Function
C60	P_F1_01.S	10 Pixel sides, solid direct, unclipped
C62	P_F1_02.S	10 Pixel sides, solid direct, clipped
C61	P_F1_01.S	50 Pixel sides, solid direct, unclipped
C63	P_F1_02.S	50 Pixel sides, solid direct, clipped

11011A-25

Figure 25. Benchmark Results for Solid Direct Triangle Functions (Triangles/Sec)

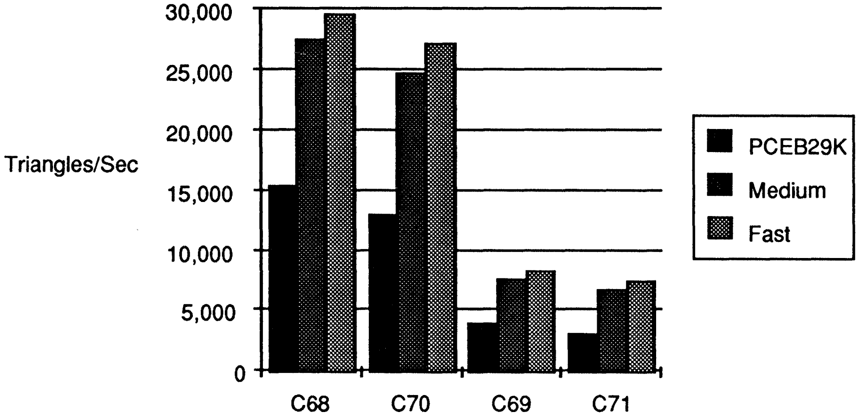


Case	PCEB29K	Medium	Fast	
C64	8,821	17,705	21,758	(Triangles/Sec)
C66	7,879	14,828	17,385	
C65	622	1,504	2,074	
C67	588	1,273	1,656	

Case	Routine	Function
C64	P_F2_01.S	10 Pixel sides, XOR, unclipped
C66	P_F2_02.S	10 Pixel sides, XOR, clipped
C65	P_F2_01.S	50 Pixel sides, XOR, unclipped
C67	P_F2_02.S	50 Pixel sides, XOR, clipped

11011A-26

Figure 26. Benchmark Results for Solid XOR Triangle Functions (Triangles/Sec)



Case	PCEB29K	Medium	Fast	
C68	15,366	27,533	29,656	(Triangles/Sec)
C70	12,927	24,851	27,203	
C69	3,912	7,583	8,300	
C71	3,106	6,631	7,372	

Case	Routine	Function
C68	P_F3_01.S	10 Pixel Sides, Monochrome, unclipped
C70	P_F4_01.S	10 Pixel Sides, Monochrome XOR, unclipped
C69	P_F3_01.S	50 Pixel sides, Monochrome, unclipped
C71	P_F4_01.S	50 Pixel sides, Monochrome XOR, unclipped

11011A-27

Figure 27. Benchmark Results for Monochrome Triangle Functions (Triangles/Sec)

Summary

Table 14 summarizes all preceding benchmark results.

Table 14. Am29000 Graphics Performance

Primitive	Case	Medium Performance	Maximum Performance	Units
10-Pixel Vectors:				
Monochrome SET	C10	101,215	119,617	Vectors/Sec
2–32 Bits/Pixel SET	C11	250,000	268,817	Vectors/Sec
2–32 Bits/Pixel XOR	C5	72,674	84,746	Vectors/Sec
2–32 Bits/Pixel (AA)	C7	28,027	29,274	Vectors/Sec
16x16 BITBLTs:				
Monochrome Copy	C34	25,934	32,216	Blocks/Sec
2–32 Bits/Pixel Copy	C30	25,253	28,703	Blocks/Sec
256x256 BITBLTs:				
Monochrome Copy	C42	1331	1618	Blocks/Sec
2–32 Bits/Pixel Copy	C38	157	170	Blocks/Sec
Text(7X9):				
2–32 Bits/Pixel		37,821	42,159	Characters/Sec
Filled Triangles (10-Pixel Sides):				
Shaded 2–32 Bits/Pixel	C50	12,389	13,062	Triangles/Sec
Solid 2–32 Bits/Pixel	C60	28,802	31,847	Triangles/Sec
Monochrome	C68	27,533	29,656	Triangles/Sec

ADDITIONAL PERFORMANCE CONSIDERATIONS

Pipelines

The performance of graphics-processing systems can be significantly increased through the use of pipelining. This is because graphics-processing operations can easily be partitioned into sequential tasks, such as transformations, end-point determination, and rendering. Since these tasks depend on the results of one another in only a single direction, each can be performed on a particular machine and the results passed on to the next.

Figure 28 shows three Am29000s pipelined together to execute fast line drawing.

Scaled Arithmetic

In a system without a floating-point processor, scaled arithmetic can be used to avoid floating-point emulation. The 3-D rendering examples in this handbook use scaled arithmetic. The scaled operations consist of several integer operations on real numbers, having an integer and a fractional part.

The software in this handbook assumes the presence of a radix point between bits 16 and 15 of words. Numbers therefore comprise 1 sign bit, 15 bits of integer part, and 16 bits of fractional part. This is shown in Figure 29.

With addition, subtraction, and compare, the values are simply ordered lists of bits. The assumed radix point introduces no special complication. Multiplication is slightly more complex. The standard multiply assumes two 32-bit signed numbers and produces a 64-bit signed number. Since there are a total of 32 bits of fraction, the radix point must lie between bits 31 and 32 of the product. After the multiplication has taken place, the middle 32 bits are extracted, leaving the radix point between bits 16 and 15. It is often unnecessary to execute a full 32-bit multiplication; the same result can be obtained by using a combination of shifts and adds when one operand is a constant.

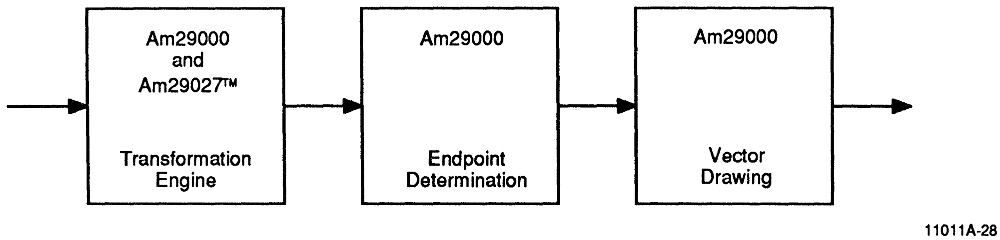


Figure 28. Pipelining

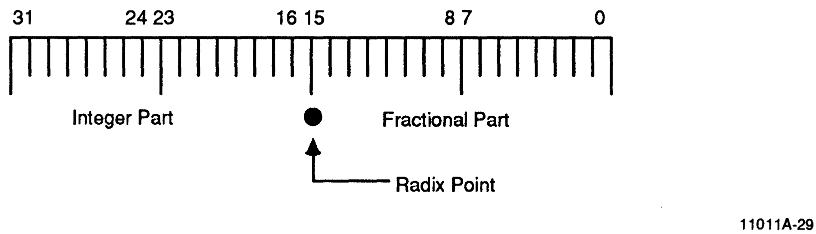


Figure 29. Scaled Arithmetic

Hardware Assist

There are always tradeoffs in determining the division of tasks between hardware and software. This handbook has assumed that the hardware is relatively simple. This section discusses moving some activities into hardware. Though this will not result in faster primitives, it may result in the speed-up of some ancillary operations.

For purposes of discussion, a typical bit map is defined as a 2K-wide by 1K-high buffer based on 256K × 4 VRAMs and an 81C458 Color Palette (see Figure 30). This bit map and serializer/palette is suitable for a 1280 × 1024 60 Hz, non-interlaced display.

A 2K × 1K × 8 bit map based on 256K × 4 VRAMs requires 16 devices. The example bit map is wired so that each pixel is contained in two adjacent devices. It is possible to wire the bit map differently, but this method has the advantage of not requiring the use of masked writes, which in turn has the substantial advantage of not requiring additional buffers on the data bus to inject the write mask, and also makes the timing generator slightly less complex.

Eight adjacent VRAM pairs contain eight horizontally adjacent pixels. Each VRAM pair contains every eighth pixel.

VRAMs Used for Bit Maps

The memory bandwidth required to refresh the screen has long been a problem with bit maps. For a screen of only modest resolution, the bandwidth can be hundreds of millions of bits per second. This can interfere significantly with the bit-map update process.

The VRAM can be thought of as a dual-port memory with one random port and one very specialized port. The random port is a standard dynamic RAM port with *RAS, *CAS, addresses, and *WE. In addition, *DT/OE is used to control transfers to and from an internal serializer. The second port can be accessed only serially. Once started, up to 256 or 512 nibbles per VRAM can be transferred independently of the standard port. Since VRAMs are typically arranged in parallel, only a single transfer cycle is required per scan line.

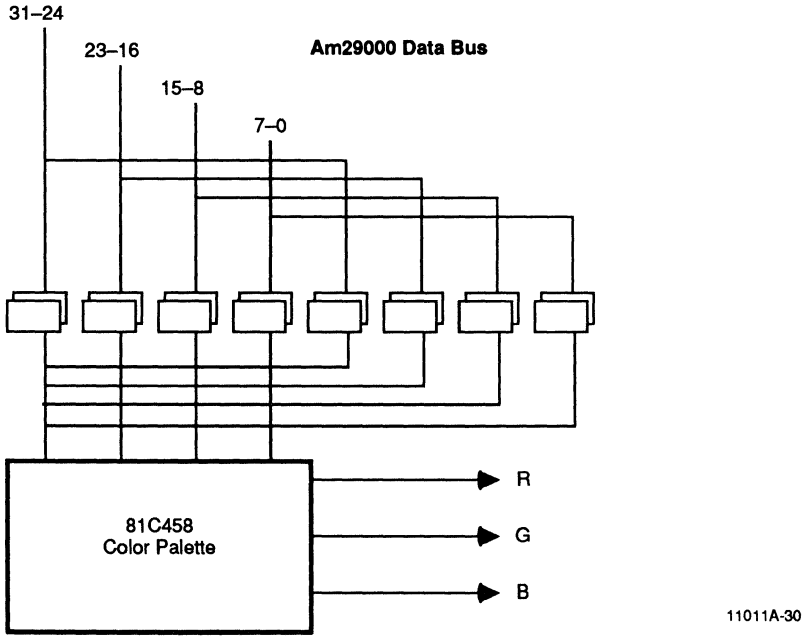


Figure 30. Frame Buffer

Because the raster traverses the screen in a very regular manner, accesses to the bit map required for screen refresh are very regular. In fact, for a typical system, all the bits necessary to refresh an entire scan line share a common row address.

In a bit-map application, the standard port is controlled by the graphics processor. It is used for bit-map update, dynamic memory refresh, and transfer cycles. The serial port is used for screen refresh.

A typical VRAM configuration is shown in Figure 31.

Bit-Map Clearing

In many applications, the bit map must be cleared occasionally. This could be as seldom as once every few minutes for a CAD system, to as often as 30 or 60 times a second for a real-time animation system. When the bit map is cleared often, this task can take a good percentage of the total available time and result in an appreciable reduction in performance. In some cases, it might be worth a modest investment in hardware to improve performance.

Note that in Figure 31 the path between the serializer and the dynamic memory is bidirectional. This means that an entire row of memory can be loaded with some value (typically zero) per memory cycle. In the discussion below, we will assume the value zero generates a blank screen.

Such an approach requires two things. First, it is necessary to get the pattern (0s) into the serializer. Second, it is necessary to build the logic to execute write transfer cycles. Since this operation requires the use of the serializer, it must be done when the serializer is otherwise unused for some significant period of time. Clearly this has to be during the vertical blanking period.

The serializer can be set to zero by transferring a row from the dynamic portion of the VRAM. This would require that one row always be either kept at zero or set to zero before executing the read transfer. In our example bit map, this is two (contiguous) scan lines. Keeping two scan lines at zero would imply either not displaying them at all, leaving 1022 scan lines on the screen, or displaying them as blanks. Using this method, the serializer can be set to zero in about 300 nsec.

The serializer can also be set to zero by shifting in a row of 0s. This would require a pseudo-write-transfer cycle, followed by the clocking of 512 nibbles into the serializer. Using this method, the serializer can be set to zero in about 300+512*50, or 26000 ns. The first method is certainly faster.

Once the serializer has been cleared to zero, 512 write transfer cycles are required to actually clear the bit map. At 300 ns per cycle, this would require about 154 µs or about 1/10 of one frame time.

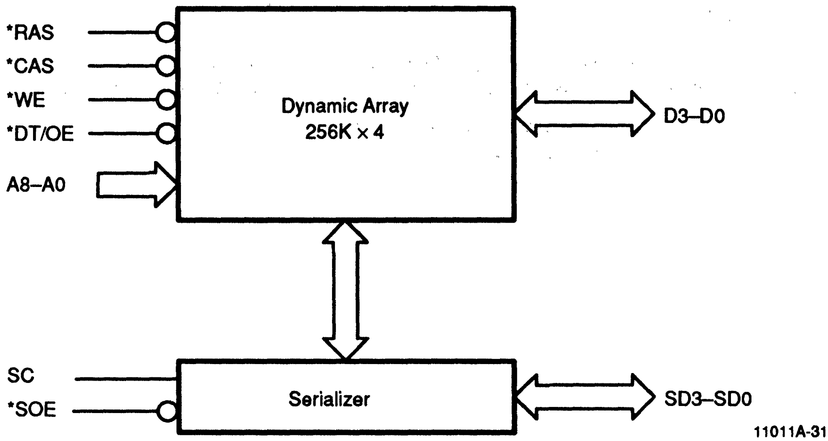


Figure 31. VRAM Block Diagram

Saturation in Hardware

Consider a 24-plane bit map. In that case, the three color values were calculated individually as fixed-point real numbers. The format of the numbers is shown in Figure 29, in the section entitled "Scaled Arithmetic."

If the frame store already has 8-bit latches to hold the values prior to being stored, one can mechanize the latches as 22V10 PAL devices for a small incremental cost. This is shown in Figure 32.

In the PAL equation shown in Figure 32, bit 31 is the sign bit. Whenever it is set, a 0 is latched in the register, regardless of the state of any other bits. If bit 31 is not set, then the bit in the register is set if the calculated bit is a 1, or if any of the first 4 overflow bits are set. Since each bit in the register has a similar set of equations, overflow will force a value of all 1s, and underflow will force a value of all 0s. If neither overflow nor underflow has occurred, the calculated value is placed in the register for storing into memory.

Hardware Cycles

Memory transfer and refresh cycles can be performed by the Am29000 in interrupt routines or in hardware. The advantage of software is that the memory responds only to the Am29000, and thus no arbiter is required. The disadvantage of software is that it requires some overhead, and transfer cycles must take place during the horizontal blank period; which means that the interrupt response must be guaranteed to be within less than 5 to 8 µs.

The disadvantage of executing the refresh and transfer cycles in hardware is that the memory controller must include additional arbitration logic, and that an additional cycle is required for arbitration.

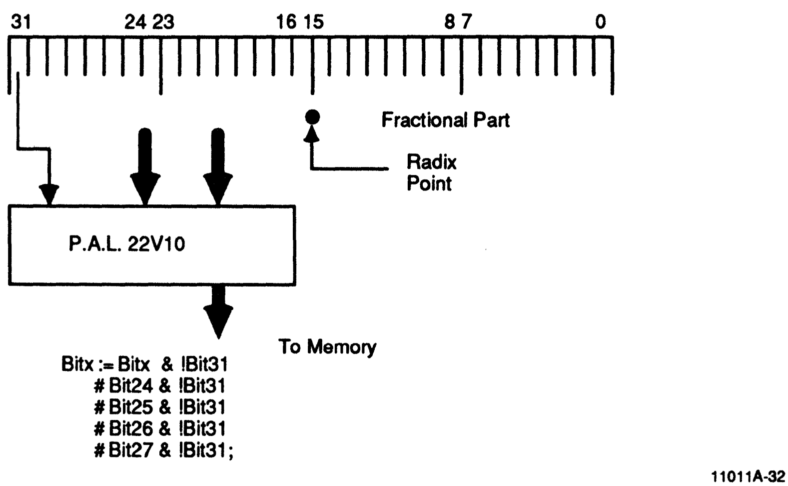


Figure 32. Hardware Saturation

BIBLIOGRAPHY

Newman, W.M. and Sproull, R.F., *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.

Foley, J.D. and van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison Wesley, Reading, 1983.

Raster Graphics Handbook, Conrac Division/Conrac Corporation, Van Nostrand Reinhard, New York, 1985.

APPENDIX A: PROGRAM LISTINGS

G29K_REG.H

```
        .eject
        .sbttl  "Register Names and Trap Definitions"

;*******************************************************************
;*                                                               **
;*      g29k_reg.h                  C 29000 Graphics Benchmarks **
;*                                                               **
;*  Copyright 1988 Advanced Micro Devices, Inc.                  **

;*  Written by Gibbons and Associates, Inc.                      **
;*                                                               **
;*                                                               **
;*  Register names, trap definitions, and C function calling    **
;*  convention macros.                                           **
;*                                                               **
;*******************************************************************


;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                               ++
;+  Calling Convention Registers                                 ++
;+                                                               ++
;+-----------------------------------------------------------------++

        .reg    rsp,    grl             ; register stack pointer
        .reg    tav,    gr121           ; trap handler argument
        .reg    tpc,    gr122           ; trap handler return
        .reg    lrp,    gr123           ; large return pointer
        .reg    slp,    gr124           ; static link pointer
        .reg    msp,    gr125           ; memory stack pointer
        .reg    rsb,    gr126           ; register spill bound
        .reg    rfb,    gr127           ; register fill bound

        .reg    ffb,    lrl             ; function frame bound
        .reg    ret,    lr0             ; return address


;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                               ++
;+  User Registers (General Names)                               ++
;+                                                               ++
;|                                                               ||
;|  Function return registers (16)                               ||
;|                                                               ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
        .reg    v0,     gr96
        .reg    vl,     gr97
        .reg    v2,     gr98
        .reg    v3,     gr99
```

```
        .reg    v4,     gr100
        .reg    v5,     gr101
        .reg    v6,     gr102
        .reg    v7,     gr103
        .reg    v8,     gr104
        .reg    v9,     gr105
        .reg    v10,    gr106
        .reg    v11,    gr107
        .reg    v12,    gr108
        .reg    v13,    gr109
        .reg    v14,    gr110
        .reg    v15,    gr111

;-------------------------------------------------------------
;|                                                          ||
;|  Volatile temporary registers (25)                       ||
;|                                                          ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
        .reg    t0,     gr116
        .reg    t1,     gr117
        .reg    t2,     gr118
        .reg    t3,     gr119
        .reg    t4,     gr120
        .reg    t5,     gr99                    ; (v3)
        .reg    t6,     gr98                    ; (v2)
        .reg    t7,     gr97                    ; (v1)
        .reg    t8,     gr96                    ; (v0)
        .reg    t9,     gr121                   ; (tav)
        .reg    t10,    gr122                   ; (tpc)
        .reg    t11,    gr123                   ; (lrp)
        .reg    t12,    gr124                   ; (slp)
        .reg    t13,    gr111                   ; (v15)
        .reg    t14,    gr110                   ; (v14)
        .reg    t15,    gr109                   ; (v13)
        .reg    t16,    gr108                   ; (v12)
        .reg    t17,    gr107                   ; (v11)
        .reg    t18,    gr106                   ; (v10)
        .reg    t19,    gr105                   ; (v9)
        .reg    t20,    gr104                   ; (v8)
        .reg    t21,    gr103                   ; (v7)
        .reg    t22,    gr102                   ; (v6)
        .reg    t23,    gr101                   ; (v5)
        .reg    t24,    gr100                   ; (v4)

;-------------------------------------------------------------
;|                                                          ||
;|  Reserved registers (4)                                  ||
;|                                                          ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
        .reg    r0,     gr112
        .reg    r1,     gr113
        .reg    r2,     gr114
        .reg    r3,     gr115
;-------------------------------------------------------------
```

```
;|                                                              ||
;|  Parameter registers (16)                                    ||
;|                                                              ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    p0,     lr2
        .reg    p1,     lr3
        .reg    p2,     lr4
        .reg    p3,     lr5
        .reg    p4,     lr6
        .reg    p5,     lr7
        .reg    p6,     lr8
        .reg    p7,     lr9
        .reg    p8,     lr10
        .reg    p9,     lr11
        .reg    p10,    lr12
        .reg    p11,    lr13
        .reg    p12,    lr14
        .reg    p13,    lr15
        .reg    p14,    lr16
        .reg    p15,    lr17


;------------------------------------------------------------------
;|                                                              ||
;|  Global control parameter registers                          ||
;|                                                              ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    GP.wnd.min_x,       lr2
        .reg    GP.wnd.max_x,       lr3
        .reg    GP.wnd.min_y,       lr4
        .reg    GP.wnd.max_y,       lr5
        .reg    GP.pxl.value,       lr6
        .reg    GP.mem.width,       lr7
        .reg    GP.mem.depth,       lr8
        .reg    GP.wnd.base,        lr9
        .reg    GP.wnd.align,       lr10
        .reg    GP.pxl.op_vec,      lr11
        .reg    GP.pxl.in_mask,     lr12
        .reg    GP.pxl.do_mask,     lr13
        .reg    GP.pxl.do_value,    lr14
        .reg    GP.pxl.out_mask,    lr15
        .reg    GP.wid.actual,      lr16
        .reg    GP.pxl.op_code,     lr17
        .reg    GP.mem.base,        lr18
        .reg    GP.wnd.origin_x,    lr19
        .reg    GP.wnd.origin_y,    lr20
```

```
;-------------------------------------------------------------
;|                                                        ||
;|  Line parameter registers                              ||
;|                                                        ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    LP.loc.x,           lr21
        .reg    LP.loc.y,           lr22
        .reg    LP.loc.addr,        lr23
        .reg    LP.loc.align,       lr24

        .reg    LP.wid.axial,       lr25
        .reg    LP.wid.side_1,      lr26
        .reg    LP.wid.side_2,      lr27

        .reg    LP.gen.cover,       lr28
        .reg    LP.gen.delta_p,     lr29
        .reg    LP.gen.delta_s,     lr30
        .reg    LP.gen.move_p,      lr31
        .reg    LP.gen.move_s,      lr32
        .reg    LP.gen.p,           lr33
        .reg    LP.gen.s,           lr34
        .reg    LP.gen.min_p,       lr35
        .reg    LP.gen.max_p,       lr36
        .reg    LP.gen.min_s,       lr37
        .reg    LP.gen.max_s,       lr38
        .reg    LP.gen.slope,       lr39
        .reg    LP.gen.x_slope,     lr40
        .reg    LP.gen.error,       lr41
        .reg    LP.gen.x_error,     lr42
        .reg    LP.gen.addr,        lr43
        .reg    LP.gen.try_s,       lr44
        .reg    LP.gen.count,       lr45

        .reg    LP.clp.skip_vec,    lr46
        .reg    LP.clp.stop_vec,    lr47
        .reg    LP.clp.skip_p,      lr48
        .reg    LP.clp.stop_p,      lr49
        .reg    LP.clp.skip_s,      lr50
        .reg    LP.clp.skip_s_1,    lr51
        .reg    LP.clp.stop_s_1,    lr52
        .reg    LP.clp.skip_s_2,    lr53
        .reg    LP.clp.stop_s_2,    lr54
```

```
;-------------------------------------------------------------
;|                                                          ||
;|  Block parameter registers                               ||
;|                                                          ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .equ    MAX_WORDS,  32
        .equ    MAX_SHIFT,  5

        .reg    BP.grp.op_skip,     lr25
        .reg    BP.grp.align,       lr26
        .reg    BP.grp.repeat,      lr27
        .reg    BP.grp.count,       lr28

        .reg    BP.dst.align,       lr29
        .reg    BP.dst.lft_addr,    lr30
        .reg    BP.dst.lft_mask,    lr31
        .reg    BP.dst.lft_end,     lr32
        .reg    BP.dst.addr,        lr33
        .reg    BP.dst.array,       lr34
        .reg    BP.dst.array_00,    lr34
        .reg    BP.dst.array_01,    lr35
        .reg    BP.dst.array_02,    lr36
        .reg    BP.dst.array_03,    lr37
        .reg    BP.dst.array_04,    lr38
        .reg    BP.dst.array_05,    lr39
        .reg    BP.dst.array_06,    lr40
        .reg    BP.dst.array_07,    lr41
        .reg    BP.dst.array_08,    lr42
        .reg    BP.dst.array_09,    lr43
        .reg    BP.dst.array_10,    lr44
        .reg    BP.dst.array_11,    lr45
        .reg    BP.dst.array_12,    lr46
        .reg    BP.dst.array_13,    lr47
        .reg    BP.dst.array_14,    lr48
        .reg    BP.dst.array_15,    lr49
        .reg    BP.dst.array_16,    lr50
        .reg    BP.dst.array_17,    lr51
        .reg    BP.dst.array_18,    lr52
        .reg    BP.dst.array_19,    lr53
        .reg    BP.dst.array_20,    lr54
        .reg    BP.dst.array_21,    lr55
        .reg    BP.dst.array_22,    lr56
        .reg    BP.dst.array_23,    lr57
        .reg    BP.dst.array_24,    lr58
        .reg    BP.dst.array_25,    lr59
        .reg    BP.dst.array_26,    lr60
        .reg    BP.dst.array_27,    lr61
        .reg    BP.dst.array_28,    lr62
        .reg    BP.dst.array_29,    lr63
        .reg    BP.dst.array_30,    lr64
        .reg    BP.dst.array_31,    lr65
```

```
.if MAX_WORDS == 32
.reg    BP.dst.array_end,   BP.dst.array_31
.endif

.if MAX_WORDS == 16
.reg    BP.dst.array_end,   BP.dst.array_15
.endif

.if MAX_WORDS == 8
.reg    BP.dst.array_end,   BP.dst.array_07
.endif

.if MAX_WORDS == 4
.reg    BP.dst.array_end,   BP.dst.array_03
.endif

.reg    BP.src.lft_addr,    lr66
.reg    BP.src.rgt_ptr,     lr67
.reg    BP.src.save,        lr68
.reg    BP.src.shift,       lr69
.reg    BP.src.addr,        lr70
.reg    BP.src.extra,       lr71
.reg    BP.src.array,       lr72
.reg    BP.src.array_00,    lr72
.reg    BP.src.array_01,    lr73
.reg    BP.src.array_02,    lr74
.reg    BP.src.array_03,    lr75
.reg    BP.src.array_04,    lr76
.reg    BP.src.array_05,    lr77
.reg    BP.src.array_06,    lr78
.reg    BP.src.array_07,    lr79
.reg    BP.src.array_08,    lr80
.reg    BP.src.array_09,    lr81
.reg    BP.src.array_10,    lr82
.reg    BP.src.array_11,    lr83
.reg    BP.src.array_12,    lr84
.reg    BP.src.array_13,    lr85
.reg    BP.src.array_14,    lr86
.reg    BP.src.array_15,    lr87
.reg    BP.src.array_16,    lr88
.reg    BP.src.array_17,    lr89
.reg    BP.src.array_18,    lr90
.reg    BP.src.array_19,    lr91
.reg    BP.src.array_20,    lr92
.reg    BP.src.array_21,    lr93
.reg    BP.src.array_22,    lr94
.reg    BP.src.array_23,    lr95
.reg    BP.src.array_24,    lr96
.reg    BP.src.array_25,    lr97
.reg    BP.src.array_26,    lr98
.reg    BP.src.array_27,    lr99
.reg    BP.src.array_28,    lr100
.reg    BP.src.array_29,    lr101
```

```
        .reg    BP.src.array_30,    lr102
        .reg    BP.src.array_31,    lr103

        .if MAX_WORDS == 32
        .reg    BP.src.array_end,   BP.src.array_31
        .endif

        .if MAX_WORDS == 16
        .reg    BP.src.array_end,   BP.src.array_15
        .endif

        .if MAX_WORDS == 8
        .reg    BP.src.array_end,   BP.src.array_07
        .endif

        .if MAX_WORDS == 4
        .reg    BP.src.array_end,   BP.src.array_03
        .endif

        .reg    BP.fst.shift,       lr104
        .reg    BP.fst.skip,        lr105
        .reg    BP.fst.incr,        lr106
        .reg    BP.fst.count,       lr107

        .reg    BP.dst.rgt_ptr,     lr108
        .reg    BP.dst.rgt_mask,    lr109
        .reg    BP.dst.rgt_end,     lr110


;-----------------------------------------------------------------
;|                                                              ||
;|  Text parameter registers                                    ||
;|                                                              ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    TP.chr.high,        lr25
        .reg    TP.chr.wide,        lr26
        .reg    TP.chr.inset,       lr27
        .reg    TP.chr.ascent,      lr28
        .reg    TP.chr.pitch,       lr29
        .reg    TP.ptn.mask,        lr30
        .reg    TP.ptn.shift_count, lr31
        .reg    TP.ptn.shift_set,   lr32
        .reg    TP.ptn.high,        lr33
        .reg    TP.ptn.wide,        lr34
        .reg    TP.ptn.next,        lr35
```

```
;-------------------------------------------------------------------
;|                                                                ||
;|  Shading parameter registers                                   ||
;|                                                                ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
        .reg    SP.gen.more,        lr25
        .reg    SP.gen.left,        lr26
        .reg    SP.gen.right,       lr27
;       .reg    LP.gen.cover,       lr28
        .reg    SP.gen.count,       lr29
        .reg    SP.gen.grade,       lr30
        .reg    SP.gen.extra,       lr31
        .reg    SP.gen.resid,       lr32
        .reg    SP.gen.inc_r,       lr33
        .reg    SP.gen.dec_r,       lr34

        .reg    SP.lft.count,       lr35
        .reg    SP.lft.move_p,      lr36
        .reg    SP.lft.move_s,      lr37
        .reg    SP.lft.fix_p,       lr38
        .reg    SP.lft.fix_s,       lr39
        .reg    SP.lft.error,       lr40
        .reg    SP.lft.point,       lr41
        .reg    SP.lft.grade,       lr42
        .reg    SP.lft.extra,       lr43
        .reg    SP.lft.resid,       lr44
        .reg    SP.lft.inc_r,       lr45

;       .reg    LP.clp.skip_vec,    lr46
;       .reg    LP.clp.stop_vec,    lr47

        .reg    SP.lft.dec_r,       lr48
        .reg    SP.lft.shade,       lr49
        .reg    SP.lft.flag,        lr50
        .reg    SP.lft.x,           lr51
        .reg    SP.lft.y,           lr52
        .reg    SP.lft.m_s_x,       lr53
        .reg    SP.lft.m_p_x,       lr54

        .reg    SP.clp.skip_x,      lr55
        .reg    SP.clp.skip_y,      lr56
        .reg    SP.clp.stop_x,      lr57
        .reg    SP.clp.stop_y,      lr58

        .reg    SP.rgt.count,       lr59
        .reg    SP.rgt.move_p,      lr60
        .reg    SP.rgt.move_s,      lr61
        .reg    SP.rgt.fix_p,       lr62
        .reg    SP.rgt.fix_s,       lr63
        .reg    SP.rgt.error,       lr64
        .reg    SP.rgt.point,       lr65
        .reg    SP.rgt.grade,       lr66
        .reg    SP.rgt.extra,       lr67
        .reg    SP.rgt.resid,       lr68
```

```
        .reg    SP.rgt.inc_r,       lr69
        .reg    SP.rgt.dec_r,       lr70
        .reg    SP.rgt.shade,       lr71
        .reg    SP.rgt.flag,        lr72
;-----------------------------------------------------------------
;|                                                             ||
;|  Shading parameter registers                                ||
;|                                                             ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    FP.lft.pixel,       lr33
        .reg    FP.rgt.pixel,       lr34


;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                             ++
;+  Programmed Traps                                           ++
;+                                                             ++
;+-------------------------------------------------------------++

        .equ    V_SPILL,            64  ; spill register stack
        .equ    V_FILL,             65  ; fill register stack
        .equ    V_CLIP_SKIP,        100 ; skip if clipped
        .equ    V_CLIP_STOP,        101 ; stop if clipped


;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                             ++
;+  Function Prolog/Epilog Macros and Constants                ++
;+                                                             ++
;+-------------------------------------------------------------++
;
;  The following are fixed memory configuration parameters.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

        .equ    PIXEL_SIZE, 4

;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
;
;  The following constants are used as the argument to the ENTER
;  macro.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

        .equ    LINE_PRIMITIVE,     53
        .equ    BLOCK_PRIMITIVE,    109
        .equ    TEXT_PRIMITIVE,     46
        .equ    SHADE_PRIMITIVE,    71
        .equ    FILL_PRIMITIVE,     71
```

```
;+-------------------------------------------------------------++
;
;  Used at the beginning of a function that is callable from a
;  C program, immediately before the label.  Only definitions of
;  management symbols are made; no code is generated.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

 .macro ENTER,  __FNTYPE

 .ifndef __FN_MANAGE
        .set    __FN_MANAGE, 0
 .endif

 .if __FN_MANAGE != 0
        .print  "ENTER without prior LEAVE"
        .err
 .else
        .set    __FN_MANAGE, 1
        .set    __LP_ALLOC, (__FNTYPE + 3) & 0xFFFFFFFE
        .set    __PX, 2
 .endif
 .endm

;+-------------------------------------------------------------++
;
;  Used in a function that is callable from a C program to name
;  an argument to the function, immediately after the function's
;  label.  Only a definition of a local register symbol is made;
;  no code is generated.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

 .macro PARAM,  __PNAME
 .ifndef __FN_MANAGE
        .set    __FN_MANAGE, 0
 .endif
 .if __FN_MANAGE != 1
        .print  "PARAM without prior ENTER, or after CLAIM"
        .err
 .else
        .reg    __PNAME, %%(__LP_ALLOC + __PX + 128)
        .set    __PX, __PX + 1
 .endif
 .endm

;+-------------------------------------------------------------++
;
;  Used in a function that is callable from a C program to claim
;  space in the local register stack cache, immediately after the
;  last PARAM.  The calling convention prolog code is generated.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
.macro CLAIM
.ifndef __FN_MANAGE
        .set    __FN_MANAGE, 0
.endif
.if __FN_MANAGE != 1
        .print  "CLAIM without prior ENTER"
        .err
.else
        .set    __FN_MANAGE, 2
.if (__LP_ALLOC * 4) >= 256
        const   tav, __LP_ALLOC * 4
        sub     rsp, rsp, tav
.else
        sub     rsp, rsp, __LP_ALLOC * 4
.endif
        asgeu   V_SPILL, rsp, rsb
.if ((__LP_ALLOC + __PX) * 4) >= 256
        const   tav, (__LP_ALLOC + __PX) * 4
        add     ffb, rsp, tav
.else
        add     ffb, rsp, (__LP_ALLOC + __PX) * 4
.endif
.endif
.endm

;+-------------------------------------------------------------++
;
;  Used in a function that is callable from a C program to release
;  space from the local register stack cache.  Part of the calling
;  convention epilog code is generated.  A normal instruction must
;  follow this and precede the LEAVE invocation.  This instruction
;  may be a NOP.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

.macro RELEASE
.ifndef __FN_MANAGE
        .set    __FN_MANAGE, 0
.endif
.if __FN_MANAGE != 2
        .print  "RELEASE without prior CLAIM"
        .err
.else
        .set    __FN_MANAGE, 3
.if (__LP_ALLOC * 4) >= 256
        const   tav, __LP_ALLOC * 4
        add     rsp, rsp, tav
.else
        add     rsp, rsp, __LP_ALLOC * 4
.endif
.endif
.endm
```

```
;+-----------------------------------------------------------++
;
;  Used to end a function that is callable from a C program.  The
;  remainder of the calling convention epilog code is generated.
;
;-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

 .macro LEAVE
 .ifndef __FN_MANAGE
        .set    __FN_MANAGE, 0
 .endif
 .if __FN_MANAGE != 3
        .print  "LEAVE without prior RELEASE"
        .err
 .else
        .set    __FN_MANAGE, 0
        jmpi    ret
        asleu   V_FILL, ffb, rfb
 .endif
 .endm


;*****************************************************************
; end of g29k_reg.h
;*****************************************************************
```

GRAPH29K.H

```
/****************************************************************\
**                                                             **
**      graph29k.h                C 29000 Graphics Benchmarks **
**                                                             **
**   Copyright 1988 Advanced Micro Devices, Inc.               **
**   Written by Gibbons and Associates, Inc.                   **
**                                                             **
**                                                             **
**   This file contains functions to provide queries to the    **
**   tester functions.                                         **
**                                                             **
\****************************************************************/

#if ! defined(GRAPH29K)
#define GRAPH29K

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*\
++                                                             ++
++   Type Definitions                                          ++
++                                                             ++
\*--------------------------------------------------------------*/

typedef struct parameters          /* for controlling graphics */
        {
        int             wnd_min_x;      /* min window x-coord   */
                                        /*   (origin relative)  */
        int             wnd_max_x;      /* max window x-coord   */
                                        /*   (origin relative)  */
        int             wnd_min_y;      /* min window y-coord   */
                                        /*   (origin relative)  */
        int             wnd_max_y;      /* max window y-coord   */
                                        /*   (origin relative)  */

        unsigned long   pxl_value;      /* current pixel color  */
                                        /*   or shading value   */

        unsigned int    mem_width;      /* no. bytes added to   */
                                        /*   move down one      */
        int             mem_depth;      /* pixels/word code:    */
                                        /*   -1=32 0=4 1=2 2=1  */

        unsigned char * wnd_base;       /* base address of      */
                                        /*   window origin      */
        unsigned int    wnd_align;      /* no. pixels added to  */
                                        /*   get actual origin  */

        void    ( *     pxl_op_vec ) ();/* pointer to routine   */
                                        /*   to do pixel-op     */
        unsigned long   pxl_in_mask;    /* pixel-op memory src  */
                                        /*   input mask         */
```

```
        unsigned long   pxl_do_mask;     /* pixel-op memory src */
                                         /*   acceptance mask   */
        unsigned long   pxl_do_value;    /* pixel-op memory src */
                                         /*   acceptance value  */
        unsigned long   pxl_out_mask;    /* pixel-op memory dst */
                                         /*   output mask       */

        int             wid_actual;      /* actual pixel width  */
                                         /*   of line segment   */
        int             pxl_op_code;     /* encoded value for   */
                                         /*   current pixel-op  */

        unsigned char * mem_base;        /* base address of     */
                                         /*   graphics raster   */
        unsigned int    wnd_origin_x;    /* x-coord of origin   */
                                         /*   (raster relative) */
        unsigned int    wnd_origin_y;    /* y-coord of origin   */
                                         /*   (raster relative) */
        }
        parameters;


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

typedef struct point                     /* for drawing position */
        {
        int             x;               /* x-coordinate         */
                                         /*   (origin relative)  */
        int             y;               /* x-coordinate         */
                                         /*   (origin relative)  */
        }
        point;


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

typedef struct          /* 3D floating-point coordinates        */
        {
        double          x;          /* x-coord of 3D vector     */
        double          y;          /* y-   "    "   "     "    */
        double          z;          /* z-   "    "   "     "    */
        }
        vector;

typedef struct                      /* vertex description       */
        {
        vector          n;          /* normal vector            */
        point           p;          /* graphics coord point     */
        int             i;          /* intensity value          */
        }
        vertex;

typedef struct                      /* triangle description     */
```

```
        {
        point          p1;          /* 1st pt coordinates        */
        int            i1;          /* 1st pt intensity value    */
        point          p2;          /* 2nd pt coordinates        */
        int            i2;          /* 2nd pt intensity value    */
        point          p3;          /* 3rd pt coordinates        */
        int            i3;          /* 3rd pt intensity value    */
        }
        triangle;


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*\
++                                                             ++
++  External Variables                                         ++
++                                                             ++
\*-------------------------------------------------------------*/

extern parameters       /* current graphics control parameters */

G29K_Params;


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

extern triangle         /* set of triangle arrays for spheres  */

Tri_00[512], Tri_01[512], Tri_02[512], Tri_03[512],
Tri_04[512], Tri_05[512], Tri_06[512], Tri_07[512],
Tri_08[512], Tri_09[512], Tri_10[512], Tri_11[512],
Tri_12[512], Tri_13[512], Tri_14[512], Tri_15[512],
Tri_16[512], Tri_17[512], Tri_18[512], Tri_19[512],
Tri_20[512], Tri_21[512], Tri_22[512], Tri_23[512],
Tri_24[512], Tri_25[512], Tri_26[512], Tri_27[512],
Tri_28[512], Tri_29[512], Tri_30[512], Tri_31[512];

extern vertex           /* set of vertex arrays for spheres    */

Vrt_00[512], Vrt_01[512];

extern triangle *       /* base of triangles list for spheres  */

Triangles;


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

extern unsigned char    /* bit-map memory                      */

BitMap[],
BM_MB_LFT[], BM_ML_MID[], BM_ML_LLC[],
BM_CB_LFT[], BM_CL_MID[], BM_CL_LLC[];
```

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*\
++                                                              ++
++  Function Prototypes                                         ++
++                                                              ++
\*--------------------------------------------------------------*/

unsigned int            /* get partial cycles counter          */

_cycles

(void);


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                              ||
||  Obtains the least 32 bits of the system cycle counter.      ||
||                                                              ||
||                                                              ||
||  Parameters:         none                                    ||
||                                                              ||
||                                                              ||
||  Return:     least significant 32 bits of cycle counter      ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */


/*--------------------------------------------------------------*/

void                    /* set clipping trap vectors           */

S_C1_01

(void);


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                              ||
||  Sets the vectors for the clipping traps.                    ||
||                                                              ||
||                                                              ||
||  Parameters:         none                                    ||
||                                                              ||
||                                                              ||
||  Return:     none                                            ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */


/*--------------------------------------------------------------*/

triangle *              /* model a sphere                      */

Sphere
```

```
(
int             L_Gamma,          /* <- light source gamma angle     */
int             L_Theta,          /* <- light source theta angle     */
int             L_Reflect,        /* <- reflection proportion */
int             L_Ambient,        /* <- ambient constant             */
int             M_Radius,         /* <- radius of modeled sphere     */
int             M_Rings,          /* <- no. of model rings    */
int             M_Sects           /* <- no. of 1st ring sections     */
);


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                              ||
||  Model a sphere as a set of triangles.                       ||
||                                                              ||
||                                                              ||
||  Parameters:                                                 ||
||                                                              ||
||                                                              ||
||  Return:     pointer to last triangle in list, or NULL       ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */


#endif

/*****************************************************************/
/* end of graph29k.h */
/*****************************************************************/
```

TEST_L1.C

```
/****************************************************************\
**                                                              **
**    test_l1.c                     C 29000 Graphics Benchmarks **
**                                                              **
**  Copyright 1988 Advanced Micro Devices, Inc.                 **
**  Written by Gibbons and Associates, Inc.                     **
**                                                              **
**                                                              **
**  These are the test functions for the L1 primitives.         **
**                                                              **
\****************************************************************/

#include        <stdio.h>
#include        <stdlib.h>
#include        "graph29k.h"
#include        "p_l1.h"
#include        "dumpmap.h"


/****************************************************************\
**                                                              **
**  Definitions                                                 **
**                                                              **
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*\
++                                                              ++
++  Global Functions                                            ++
++                                                              ++
\*--------------------------------------------------------------*/

void                     /* single line with P_L1_01          */

T_L1_01_01

(
unsigned int    Overhead,            /* <- timing overhead     */
FILE *          Report               /* <- report output file  */
)


/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                              ||
||  Runs a simple test of "P_L1_01" with a single, fixed line. ||
||                                                              ||
||                                                              ||
||  Parameters:                                                 ||
||                                                              ||
||      Overhead        overhead associated with the timing     ||
||                      measurement; should be subtracted       ||
||                      from the time for each repetition       ||
||                      of the function being timed.            ||
```

```
||                                                              ||
||      Report          stream pointer for the file to which    ||
||                      reports are to be written.              ||
||                                                              ||
||                                                              ||
||  Return:     none                                            ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --*/

{
unsigned char * Base;               /* base of graphics memory */
unsigned int    Time;               /* function time counter   */
unsigned int    Count;              /* memory clear counter    */

fprintf (Report, "L1_01.01 : ");

/*
**  The following parameters are initialized for use
**  by P_L1_01.
*/

G29K_Params.pxl_value = 0xFFFFFFFFL;
G29K_Params.mem_width = 256 * 4;
G29K_Params.mem_depth = 2;              /* 32 planes           */
G29K_Params.wnd_base = BM_CL_LLC;
G29K_Params.wnd_align = 0;

/*
**  The following parameters are initialized but are
**  -N-O-T- used by P_L1_01.
*/

G29K_Params.wnd_min_x = 0;
G29K_Params.wnd_max_x = 255;
G29K_Params.wnd_min_y = 0;
G29K_Params.wnd_max_y = 255;
G29K_Params.pxl_op_vec = NULL;
G29K_Params.pxl_in_mask = 0xFFFFFFFFL;
G29K_Params.pxl_do_mask = 0x00000000L;
G29K_Params.pxl_do_value = 0x00000000L;
G29K_Params.pxl_out_mask = 0xFFFFFFFFL;
G29K_Params.wid_actual = 1;
G29K_Params.pxl_op_code = 0;
G29K_Params.mem_base = BitMap;
G29K_Params.wnd_origin_x = 0;
G29K_Params.wnd_origin_y = 255;

/*
**  The bit-map memory is cleared.
*/

Base = BitMap;
Count = 256 * 256 * 4;
```

```
while ( Count-- )
        *Base++ = 0;
/*
**  The vector is drawn and the timing measurement is
**  taken.
*/

Time = _cycles ();
P_L1_01 (20, 20, 65, 54);
Time = (_cycles () - Time) - Overhead;

/*
**  The time measurement is reported and the bit-map
**  is compressed and dumped to a file.
*/

fprintf (Report, "%u cycles\n", Time);
DumpMap (BitMap, 256, 256, 2, 256 * 4, "DT_L1_01.01");
return;

}
/*----------------------------------------------------------------*/

void                    /* all 10-pixel lines with P_L1_01     */

T_L1_01_02

(
unsigned int    Overhead,          /* <- timing overhead       */
FILE *          Report             /* <- report output file    */
)

/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                              ||
||  Runs a test of "P_L1_01" using all possible segments that  ||
||  are ten pixels long.  These are drawn in six concentric    ||
||  rings around the center of the bit-map.                    ||
||                                                              ||
||                                                              ||
||  Parameters:                                                 ||
||                                                              ||
||      Overhead        overhead associated with the timing    ||
||                      measurement; should be subtracted      ||
||                      from the time for each repetition      ||
||                      of the function being timed.           ||
||                                                              ||
||      Report          stream pointer for the file to which   ||
||                      reports are to be written.             ||
||                                                              ||
||                                                              ||
||  Return:     none                                            ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */
```

```
{
unsigned char * Base;              /* base of graphics memory */
unsigned int    Time;              /* function time counter   */
unsigned int    Times;             /* sum of individual times */
unsigned int    Lines;             /* no. of lines drawn      */
unsigned int    BgnX;              /* x-coord of begin point  */
unsigned int    BgnY;              /* y-coord "    "     "    */
unsigned int    EndX;              /* x-coord of end point    */
unsigned int    EndY;              /* y-coord "   "    "      */
int             OffX;              /* x-coord of offset       */
int             OffY;              /* y-coord "    "          */
unsigned int    Count;             /* memory clear counter    */

fprintf (Report, "L1_01.02 : ");

/*
**  The following parameters are initialized for use by P_L1_01.
*/

G29K_Params.pxl_value = 0xFFFFFFFFL;
G29K_Params.mem_width = 256 * 4;
G29K_Params.mem_depth = 2;         /* 32 planes                      */
G29K_Params.wnd_base = BM_CL_MID;
G29K_Params.wnd_align = 0;

/*
**  The following parameters are initialized but are
**  -N-O-T- used by P_L1_01.
*/

G29K_Params.wnd_min_x = -128;
G29K_Params.wnd_max_x = 127;
G29K_Params.wnd_min_y = -128;
G29K_Params.wnd_max_y = 127;
G29K_Params.pxl_op_vec = NULL;
G29K_Params.pxl_in_mask = 0xFFFFFFFFL;
G29K_Params.pxl_do_mask = 0x00000000L;
G29K_Params.pxl_do_value = 0x00000000L;
G29K_Params.pxl_out_mask = 0xFFFFFFFFL;
G29K_Params.wid_actual = 1;
G29K_Params.pxl_op_code = 0;
G29K_Params.mem_base = BitMap;
G29K_Params.wnd_origin_x = 128;
G29K_Params.wnd_origin_y = 127;

/*
**  The bit-map memory is cleared.
*/

Base = BitMap;
Count = 256 * 256 * 4;
while ( Count-- )
        *Base++ = 0;
```

```
/*
**  Each possible 10-pixel long vector is drawn once, in a clockwise direction aroun
d
**  the center of the bit-map.  Cycles are counted for each vector and accumulated.
**  The number of vectors is also counted.
*/

Times = 0;  Lines = 0;
for ( OffX = 0 , OffY = 10;  OffX < 10;  ++OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffY > -10;  --OffY )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffX > -10;  --OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }
for ( ;  OffY < 10;  ++OffY )
```

```
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffX < 0;  ++OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_01 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

/*
**  The total time measurement and average time is
**  reported, then the bit-map is compressed and dumped
**  to a file.
*/

fprintf (Report, "%u cycles", Times);
Times = (Times + Lines / 2) / Lines;
fprintf (Report, "    (%u per segment)\n", Times);
DumpMap (BitMap, 256, 256, 2, 256 * 4, "DT_L1_01.02");
return;
}
```

```
/*-------------------------------------------------------------*/

void                    /* single line with P_L1_02          */
T_L1_02_01

(
unsigned int    Overhead,           /* <- timing overhead      */
FILE *          Report              /* <- report output file   */
)
/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --*\
||                                                              ||
||  Runs a simple test of "P_L1_02" with a single, fixed line. ||
||                                                              ||
||  Parameters:                                                 ||
||                                                              ||
||      Overhead        overhead associated with the timing    ||
||                      measurement; should be subtracted      ||
||                      from the time for each repetition      ||
||                      of the function being timed.           ||
||                                                              ||
||      Report          stream pointer for the file to which   ||
||                      reports are to be written.             ||
||                                                              ||
||  Return:     none                                            ||
||                                                              ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --*/

{
unsigned char * Base;               /* base of graphics memory */
unsigned int    Time;               /* function time counter   */
unsigned int    Count;              /* memory clear counter    */
fprintf (Report, "L1_02.01 : ");

/*
**  The following parameters are initialized for use
**  by P_L1_02.
*/

G29K_Params.pxl_value = 0xFFFFFFFFL;
G29K_Params.mem_width = 256 * 4;
G29K_Params.mem_depth = 2;              /* 32 planes           */
G29K_Params.wnd_base = BM_CL_LLC;
G29K_Params.wnd_align = 0;
G29K_Params.wnd_min_x = 0;
G29K_Params.wnd_max_x = 255;
G29K_Params.wnd_min_y = 0;
G29K_Params.wnd_max_y = 255;

/*
**  The following parameters are initialized but are
**  -N-O-T- used by P_L1_02.
*/
```

```
G29K_Params.pxl_op_vec = NULL;
G29K_Params.pxl_in_mask = 0xFFFFFFFFL;
G29K_Params.pxl_do_mask = 0x00000000L;
G29K_Params.pxl_do_value = 0x00000000L;
G29K_Params.pxl_out_mask = 0xFFFFFFFFL;
G29K_Params.wid_actual = 1;
G29K_Params.pxl_op_code = 0;
G29K_Params.mem_base = BitMap;
G29K_Params.wnd_origin_x = 0;
G29K_Params.wnd_origin_y = 255;

/*
**  The bit-map memory is cleared.
*/

Base = BitMap;
Count = 256 * 256 * 4;
while ( Count-- )
        *Base++ = 0;

/*
**  The vector is drawn and the timing measurement is
**  taken.
*/

Time = _cycles ();
P_L1_02 (20, 20, 65, 54);
Time = (_cycles () - Time) - Overhead;

/*
**  The time measurement is reported and the bit-map
**  is compressed and dumped to a file.
*/

fprintf (Report, "%u cycles\n", Time);
DumpMap (BitMap, 256, 256, 2, 256 * 4, "DT_L1_02.01");
return;
}

/*----------------------------------------------------------------*/

void                    /* all 10-pixel lines with P_L1_02     */

T_L1_02_02

(
unsigned int    Overhead,          /* <- timing overhead        */
FILE *          Report             /* <- report output file     */
)
```

```
/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --*\
||  Runs a test of "P_L1_02" using all possible segments that  ||
||  are ten pixels long.  These are drawn in six concentric    ||
||  rings around the center of the bit-map.                    ||
||                                                             ||
||  Parameters:                                                ||
||                                                             ||
||      Overhead        overhead associated with the timing    ||
||                      measurement; should be subtracted      ||
||                      from the time for each repetition      ||
||                      of the function being timed.           ||
||                                                             ||
||      Report          stream pointer for the file to which   ||
||                      reports are to be written.             ||
||                                                             ||
||  Return:     none                                           ||
||                                                             ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

{
unsigned char * Base;               /* base of graphics memory  */
unsigned int    Time;               /* function time counter    */
unsigned int    Times;              /* sum of individual times  */
unsigned int    Lines;              /* no. of lines drawn       */
unsigned int    BgnX;               /* x-coord of begin point   */
unsigned int    BgnY;               /* y-coord "     "     "    */
unsigned int    EndX;               /* x-coord of end point     */
unsigned int    EndY;               /* y-coord "    "    "      */
int             OffX;               /* x-coord of offset        */
int             OffY;               /* y-coord "     "          */
unsigned int    Count;              /* memory clear counter     */

fprintf (Report, "L1_02.02 : ");

/*
**  The following parameters are initialized for use
**  by P_L1_02.
*/

G29K_Params.pxl_value = 0xFFFFFFFFL;
G29K_Params.mem_width = 256 * 4;
G29K_Params.mem_depth = 2;              /* 32 planes            */
G29K_Params.wnd_base = BM_CL_MID;
G29K_Params.wnd_align = 0;
G29K_Params.wnd_min_x = -128;
G29K_Params.wnd_max_x = 127;
G29K_Params.wnd_min_y = -128;
G29K_Params.wnd_max_y = 127;

/*
**  The following parameters are initialized but are
**  -N-O-T- used by P_L1_02.
*/
```

```
G29K_Params.pxl_op_vec = NULL;
G29K_Params.pxl_in_mask = 0xFFFFFFFFL;
G29K_Params.pxl_do_mask = 0x00000000L;
G29K_Params.pxl_do_value = 0x00000000L;
G29K_Params.pxl_out_mask = 0xFFFFFFFFL;
G29K_Params.wid_actual = 1;
G29K_Params.pxl_op_code = 0;
G29K_Params.mem_base = BitMap;
G29K_Params.wnd_origin_x = 128;
G29K_Params.wnd_origin_y = 127;

/*
**  The bit-map memory is cleared.
*/

Base = BitMap;
Count = 256 * 256 * 4;
while ( Count-- )
        *Base++ = 0;

/*
**  Each possible 10-pixel long vector is drawn once, in a
**  clockwise direction around the center of the bit-map.
**  Cycles are counted for each vector and accumulated.
**  The number of vectors is also counted.
*/

Times = 0;  Lines = 0;
for ( OffX = 0 , OffY = 10;  OffX < 10;  ++OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffY > -10;  --OffY )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
```

```
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffX > -10;  --OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffY < 10;  ++OffY )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }

for ( ;  OffX < 0;  ++OffX )
        {
        BgnX = 3 * OffX;  BgnY = 3 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        BgnX = 10 * OffX;  BgnY = 10 * OffY;
        EndX = BgnX + OffX;  EndY = BgnY + OffY;
        Time = _cycles ();
        P_L1_02 (BgnX, BgnY, EndX, EndY);
        Times += (_cycles () - Time) - Overhead;
        ++Lines;
        }
```

```
/*
**  The total time measurement and average time is
**  reported, then the bit-map is compressed and dumped
**  to a file.
*/

fprintf (Report, "%u cycles", Times);
Times = (Times + Lines / 2) / Lines;
fprintf (Report, "    (%u per segment)\n", Times);
DumpMap (BitMap, 256, 256, 2, 256 * 4, "DT_L1_02.02");
return;
}

/*-----------------------------------------------------------------*/

void                    /* clipped pentagram                    */
T_L1_02_03

(
unsigned int    Overhead,          /* <- timing overhead       */
FILE *          Report             /* <- report output file    */
)

/*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- *\
||                                                               ||
||  Runs a test of "P_L1_02" drawing two pentagrams.  The        ||
||  larger one will be clipped while the smaller will be         ||
||  centered inside the larger.                                  ||
||                                                               ||
||                                                               ||
||  Parameters:                                                  ||
||                                                               ||
||      Overhead        overhead associated with the timing      ||
||                      measurement; should be subtracted        ||
||                      from the time for each repetition        ||
||                      of the function being timed.             ||
||                                                               ||
||      Report          stream pointer for the file to which     ||
||                      reports are to be written.               ||
||                                                               ||
||                                                               ||
||  Return:     none                                             ||
||                                                               ||
\*-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- */

{
unsigned char * Base;              /* base of graphics memory  */
unsigned int    Time;              /* function time counter    */
unsigned int    Times;             /* sum of individual times  */
unsigned int    Count;             /* memory clear counter     */
fprintf (Report, "L1_02.03 : ");
```

```
/*
**   The following parameters are initialized for use
**   by P_L1_02.
*/

G29K_Params.pxl_value = 0xFFFFFFFFL;
G29K_Params.mem_width = 256 * 4;
G29K_Params.mem_depth = 2;            /* 32 planes            */
G29K_Params.wnd_base = BM_CL_MID;
G29K_Params.wnd_align = 0;
G29K_Params.wnd_min_x = -128;
G29K_Params.wnd_max_x = 127;
G29K_Params.wnd_min_y = -128;
G29K_Params.wnd_max_y = 127;

/*
**   The following parameters are initialized but are
**   -N-O-T- used by P_L1_02.
*/

G29K_Params.pxl_op_vec = NULL;
G29K_Params.pxl_in_mask = 0xFFFFFFFFL;
G29K_Params.pxl_do_mask = 0x00000000L;
G29K_Params.pxl_do_value = 0x00000000L;
G29K_Params.pxl_out_mask = 0xFFFFFFFFL;
G29K_Params.wid_actual = 1;
G29K_Params.pxl_op_code = 0;
G29K_Params.mem_base = BitMap;
G29K_Params.wnd_origin_x = 128;
G29K_Params.wnd_origin_y = 127;

/*
**   The bit-map memory is cleared.
*/

Base = BitMap;
Count = 256 * 256 * 4;
while ( Count-- )
        *Base++ = 0;

/*
**   The pentagrams are drawn.  A timing measurment is
**   taken for each vector and accumulated.
*/

Times = 0;
Time = _cycles ();
P_L1_02 (-28, -39, 0, 48);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (0, 48, 28, -39);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
```

```
P_L1_02 (28, -39, -46, 15);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (-46, 15, 46, 15);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (46, 15, -28, -39);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (-113, -155, 0, 192);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (0, 192, 113, -155);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (113, -155, -183, 59);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (-183, 59, 183, 59);
Times += (_cycles () - Time) - Overhead;
Time = _cycles ();
P_L1_02 (183, 59, -113, -155);
Times += (_cycles () - Time) - Overhead;

/*
**  The total time measurement and average time is
**  reported, then the bit-map is compressed and dumped
**  to a file.
*/

fprintf (Report, "%u cycles", Times);
Times = (Times + 5) / 10;
fprintf (Report, "    (%u per segment)\n", Times);
DumpMap (BitMap, 256, 256, 2, 256 * 4, "DT_L1_02.03");
return;
}

/*****************************************************************/
/* end of test_l1.c */
/*****************************************************************/
```

S_M1_01.S

```
        .title  "C 29000 Graphics Benchmarks"
        .sbttl  "Translate Pixel Coords to Linear Address"
;******************************************************************
;*                                                              **
;*      s_m1_01.s                    C 29000 Graphics Benchmarks **
;*                                                              **
;*  Copyright 1988 Advanced Micro Devices, Inc.                 **
;*  Written by Gibbons and Associates, Inc.                     **
;*                                                              **
;*                                                              **
;*  An internal subroutine to translate signed integer, pixel   **
;*  coordinates to a linear address.                            **
;*                                                              **
;******************************************************************

        .include        "g29k_reg.h"

        .eject
        .sbttl  "Coordinates to Address Computation"
;******************************************************************
;*                                                              **
;*  Definitions                                                 **
;*                                                              **
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                              ++
;+  Global Functions                                            ++
;+                                                              ++
;+--------------------------------------------------------------++

        .sect           GRAPHX, text
        .use            GRAPHX

        .global         S_M1_01

S_M1_01:

;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
;|                                                              ||
;|  Translates signed integer, pixel coordinates to a linear    ||
;|  address, according to the current control parameters.      ||
;|                                                              ||
;|                                                              ||
;|  Parameters:                                                 ||
;|                                                              ||
;|      LP.loc.x        x-coordinate of pixel position          ||
;|                                                              ||
;|      LP.loc.y        y-coordinate of pixel position          ||
;|                                                              ||
;|      GP.mem.width    width, in bytes, of raster (assumed     ||
;|                      less than or equal to 65536)            ||
;|                                                              ||
;|      GP.mem.depth    depth code for raster                   ||
```

```
;|                        -1  - one bit per pixel (monochrome)  ||
;|                         0  - up to eight bits per pixel      ||
;|                         1  - up to sixteen bits per pixel    ||
;|                         2  - up to thirty-two bits per pixel ||
;|                                                              ||
;|      GP.wnd.base     linear address of the window origin    ||
;|                                                              ||
;|      GP.wnd.align    pixel alignment of window origin (only ||
;|                      used for monochrome raster)             ||
;|                                                              ||
;|                                                              ||
;|  Products:                                                   ||
;|                                                              ||
;|      LP.loc.addr     linear address of pixel position        ||
;|                                                              ||
;|      LP.loc.align    pixel alignment (used for monochrome)   ||
;|                                                              ||
;|                                                              ||
;|  Return:     none                                            ||
;|                                                              ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        .reg    Temp0,  t0

        const   LP.loc.addr, 0
        mtsr    q, GP.mem.width
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mul     LP.loc.addr, LP.loc.y, LP.loc.addr
        mfsr    Temp0, q
        mtsrim  fc, 18
        extract LP.loc.addr, LP.loc.addr, Temp0
        jmpt    GP.mem.depth, $01
        subr    LP.loc.addr, LP.loc.addr, 0
        sll     Temp0, LP.loc.x, GP.mem.depth
        add     LP.loc.addr, LP.loc.addr, Temp0
        add     LP.loc.addr, GP.wnd.base, LP.loc.addr
        and     LP.loc.align, LP.loc.addr, 3
        jmpi    ret
```

```
        sll     LP.loc.align, LP.loc.align, 3

$01:
        add     Temp0, LP.loc.x, GP.wnd.align
        and     LP.loc.align, Temp0, 31
        sra     Temp0, Temp0, 5
        sll     Temp0, Temp0, 2
        add     LP.loc.addr, LP.loc.addr, Temp0
        jmpi    ret
        add     LP.loc.addr, GP.wnd.base, LP.loc.addr

;*****************************************************************
        .end    ; of s_m1_01.s
;*****************************************************************
```

S_C1_01.S

```
        .title  "C 29000 Graphics Benchmarks"
        .sbttl  "Clipping Trap Vectors and Handlers"
;******************************************************************
;*                                                              **
;*      s_c1_01.s                  C 29000 Graphics Benchmarks **
;*                                                              **
;*  Copyright 1988 Advanced Micro Devices, Inc.                 **
;*  Written by Gibbons and Associates, Inc.                     **
;*                                                              **
;*                                                              **
;*  Function to set the clipping vectors to the clipping trap   **
;*  handlers.  The handlers are also here.                      **
;*                                                              **
;******************************************************************

        .include        "g29k_reg.h"
        .eject
        .sbttl  "Clipping Trap Handlers"

;******************************************************************
;*                                                              **
;*  Definitions                                                 **
;*                                                              **
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                              ++
;+  Local Functions                                             ++
;+                                                              ++
;+---------------------------------------------------------------++
        .sect           GRAPHX, text
        .use            GRAPHX

S_C2_01:

;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
;|                                                              ||
;|  Handles the skip if clipped trap.                           ||
;|                                                              ||
;|  Parameters:         none                                    ||
;|                                                              ||
;|  Products:           none                                    ||
;|                                                              ||
;|  Return:     none                                            ||
;|                                                              ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        add     tpc, LP.clp.skip_vec, 0
        mtsr    pc1, tpc
        add     tpc, LP.clp.skip_vec, 4
        mtsr    pc0, tpc
        iret
;-----------------------------------------------------------------
```

```
S_C2_02:

;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
;|                                                               ||
;|   Handles the stop if clipped trap.                           ||
;|                                                               ||
;|                                                               ||
;|   Parameters:        none                                     ||
;|                                                               ||
;|                                                               ||
;|   Products:          none                                     ||
;|                                                               ||
;|                                                               ||
;|   Return:     none                                            ||
;|                                                               ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||

        add     tpc, LP.clp.stop_vec, 0
        mtsr    pc1, tpc
        add     tpc, LP.clp.stop_vec, 4
        mtsr    pc0, tpc
        iret

;------------------------------------------------------------------

        .eject
        .sbttl  "Set Clipping Vectors"

;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;+                                                               ++
;+  Global Functions                                             ++
;+                                                               ++
;+---------------------------------------------------------------++
        .use            GRAPHX

        .global         _S_C1_01

_S_C1_01:

;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
;|                                                               ||
;|   Sets the clipping trap vectors to their handlers.           ||
;|                                                               ||
;|                                                               ||
;|   Parameters:        none                                     ||
;|                                                               ||
;|                                                               ||
;|   Products:          none                                     ||
;|                                                               ||
;|                                                               ||
;|   Return:     none                                            ||
;|                                                               ||
;|-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ||
```

```
        .reg     Temp0, t0
        .reg     Temp1, t1
        const    Temp0, V_CLIP_SKIP
        sll      Temp0, Temp0, 2
        const    Temp1, S_C2_01
        consth   Temp1, S_C2_01
        store    0, 0, Temp1, Temp0
        const    Temp0, V_CLIP_STOP
        sll      Temp0, Temp0, 2
        const    Temp1, S_C2_02
        consth   Temp1, S_C2_02
        jmpi     ret
        store    0, 0, Temp1, Temp0

;*****************************************************************
        .end     ; of s_c1_01.s
;*****************************************************************
```

A

B

C

F

G

H

I

L

M

P

R

S

T

V

W

X

Notes

Sales Offices

North American

ALABAMA	(205) 882-9122
ARIZONA	(602) 242-4400
CALIFORNIA,	
Culver City	(213) 645-1524
Newport Beach	(714) 752-6262
Roseville	(916) 786-6700
San Diego	(619) 560-7030
San Jose	(408) 452-0500
Woodland Hills	(818) 992-4155
CANADA, Ontario,	
Kanata	(613) 592-0060
Willowdale	(416) 224-5193
COLORADO	(303) 741-2900
CONNECTICUT	(203) 264-7800
FLORIDA,	
Clearwater	(813) 530-9971
Ft. Lauderdale	(305) 776-2001
Orlando (Casselberry)	(407) 830-8100
GEORGIA	(404) 449-7920
ILLINOIS,	
Chicago (Itasca)	(312) 773-4422
Naperville	(312) 505-9517
KANSAS	(913) 451-3115
MARYLAND	(301) 796-9310
MASSACHUSETTS	(617) 273-3970
MICHIGAN	(313) 347-1522
MINNESOTA	(612) 938-0001
NEW JERSEY,	
Cherry Hill	(609) 662-2900
Parsippany	(201) 299-0002
NEW YORK,	
Liverpool	(315) 457-5400
Poughkeepsie	(914) 471-8180
Rochester	(716) 272-9020
NORTH CAROLINA	(919) 878-8111
OHIO,	
Columbus (Westerville)	(614) 891-6455
OREGON	(503) 245-0080
PENNSYLVANIA	(215) 398-8006
SOUTH CAROLINA	(803) 772-6760
TEXAS,	
Austin	(512) 346-7830
Dallas	(214) 934-9099
Houston	(713) 785-9001
UTAH	(801) 264-2900

International

BELGIUM, Bruxelles	TEL	(02) 771-91-42
	FAX	(02) 762-37-12
	TLX	846-61028
FRANCE, Paris	TEL	(1) 49-75-10-10
	FAX	(1) 49-75-10-13
	TLX	263282F
WEST GERMANY,		
Hannover area	TEL	(0511) 736085
	FAX	(0511) 721254
	TLX	922850
München	TEL	(089) 4114-0
	FAX	(089) 406490
	TLX	523883
Stuttgart	TEL	(0711) 62 33 77
	FAX	(0711) 625187
	TLX	721882
HONG KONG,	TEL	852-5-8654525
Wanchai	FAX	852-5-8654335
	TLX	67955AMDAPHX
ITALY, Milan	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	843-315286
JAPAN,		
Kanagawa	TEL	462-47-2911
	FAX	462-47-1729
Tokyo	TEL	(03) 346-7550
	FAX	(03) 342-5196
	TLX	J24064AMDTKOJ
Osaka	TEL	06-243-3250
	FAX	06-243-3253

International (Continued)

KOREA, Seoul	TEL	822-784-0030
	FAX	822-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY, Hovik	TEL	(03) 010156
	FAX	(02) 591959
	TLX	79079HBCN
SINGAPORE	TEL	65-3481188
	FAX	65-3480161
	TLX	55650 AMDMMI
SWEDEN,		
Stockholm	TEL	(08) 733 03 50
(Sundbyberg)	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TEL	886-2-7213393
	FAX	886-2-7723422
	TLX	886-2-7122066
UNITED KINGDOM,		
Manchester area	TEL	(0925) 828008
(Warrington)	FAX	(0925) 827693
	TLX	851-628524
London area	TEL	(0483) 740440
(Woking)	FAX	(0483) 756196
	TLX	851-859103

North American Representatives

CANADA	
Burnaby, B.C.	
DAVETEK MARKETING	(604) 430-3680
Calgary, Alberta	
DAVETEK MARKETING	(403) 291-4984
Kanata, Ontario	
VITEL ELECTRONICS	(613) 592-0060
Mississauga, Ontario	
VITEL ELECTRONICS	(416) 676-9720
Lachine, Quebec	
VITEL ELECTRONICS	(514) 636-5951
IDAHO	
INTERMOUNTAIN TECH MKTG, INC	(208) 888-6071
ILLINOIS	
HEARTLAND TECH MKTG, INC	(312) 577-9222
INDIANA	
Huntington - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
Indianapolis - ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3450
IOWA	
LORENZ SALES	(319) 377-4666
KANSAS	
Merriam – LORENZ SALES	(913) 384-6556
Wichita – LORENZ SALES	(316) 721-0500
KENTUCKY	
ELECTRONIC MARKETING CONSULTANTS, INC	(317) 921-3452
MICHIGAN	
Birmingham - MIKE RAICK ASSOCIATES	(313) 644-5040
Holland – COM-TEK SALES, INC	(616) 392-7100
Novi – COM-TEK SALES, INC	(313) 344-1409
MISSOURI	
LORENZ SALES	(314) 997-4558
NEBRASKA	
LORENZ SALES	(402) 475-4660
NEW MEXICO	
THORSON DESERT STATES	(505) 293-8555
NEW YORK	
East Syracuse – NYCOM, INC	(315) 437-8343
Woodbury – COMPONENT CONSULTANTS, INC	(516) 364-8020
OHIO	
Centerville – DOLFUSS ROOT & CO	(513) 433-6776
Columbus – DOLFUSS ROOT & CO	(614) 885-4844
Strongsville – DOLFUSS ROOT & CO	(216) 238-0300
PENNSYLVANIA	
DOLFUSS ROOT & CO	(412) 221-4420
PUERTO RICO	
COMP REP ASSOC, INC	(809) 746-6550
UTAH, R^2 MARKETING	(801) 595-0631
WASHINGTON	
ELECTRA TECHNICAL SALES	(206) 821-7442
WISCONSIN	
HEARTLAND TECH MKTG, INC	(414) 792-0920

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE TOLL FREE: (800) 222-9323 • (408) 749-5703

IMP/9.2M/4-90/0
Printed in USA

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale,
California 94088-3453
_(408) 732-2400_
TWX: 910-339-9280
TELEX: 34-6306
TOLL-FREE
_(800) 538-8450_

APPLICATIONS HOTLINE
_(800) 222-9323_
_(408) 749-5703_

SUPPORT PRODUCTS ENGINEERING HOTLINE
USA (800) 2929-AMD
_UK 0-800-89-1131_
JAPAN 0-031-11-1129

Printed in USA
11011A