Texas Instruments

TMS320C8x (DSP)

Fundamental Graphic Algorithms

Application Book

1996

Digital Signal Processing Solutions

Texas Instruments

Printed in U.S.A., June 1996

SPRA069

Application Book

TMS320C8x (DSP)
Fundamental Graphic Algorithms

1996

TMS320C8x (DSP)
Fundamental Graphic Algorithms

Application Book

Syd Poland
Digital Signal Processing Solutions

SPRA069
June 1996

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

PREFACE

The TMS320C8x is a single-chip parallel processor that can be used for applications such as:

- Image processing
- Two-dimensional graphics
- Three-dimensional graphics
- Virtual-reality graphics
- Audio/video digital compression

The ’C8x also supports applications within the security, image-recognition, and digital-telecom markets.

This book contains application notes that describe specific ’C8x capabilities. These notes are as follows:

- **Transform3 Command** examines the ’C8x master processor’s *transform3* command. The author demonstrates how two-dimensional graphic coordinate transformations can use the ’C8x floating-point unit efficiently. Performance estimation also is provided.
- **Transform4 Command** examines the ’C8x master processor’s *transform4* command. The author demonstrates how three-dimensional graphic coordinate transformations can use the ’C8x floating-point unit efficiently. Performance estimation also is provided.
- **Draw Colored Lines Command** examines the ’C8x parallel processor’s _PPCMD_COLOR_LINES command. The author demonstrates how this command can be used to generate colored lines efficiently. Performance estimation also is provided.
- **Draw Colored Trapezoids Command** examines the ’C8x parallel processor’s _PPCMD_COLOR_TRAPEZOIDS command. The author demonstrates how this command can be used to generate colored trapezoids efficiently. Performance estimation also is provided.
- **Parallel Processor Integer and Floating-Point Math** examines the floating-point and integer math subroutines that reside on the ’C8x parallel processor. The author describes these subroutines and provides performance estimation data.

The following related ’C8x documents are available through Texas Instruments. To obtain a copy of any of these documents, call the Texas Instruments Customer Response Center (CRC) at (800) 336–5236. When ordering, please identify the book by its title and literature number.

- *TMS320C8x Code Generation Tools User’s Guide* (literature number SPRU108)
- *TMS320C8x Master Processor User’s Guide* (literature number SPRU109)
- *TMS320C8x Multitasking Executive User’s Guide* (literature number SPRU112)
- *TMS320C8x Parallel Processor User’s Guide* (literature number SPRU110)
- *TMS320C8x Transfer Controller User’s Guide* (literature number SPRU105)

Contents

List of Illustrations

List of Tables

Transform3 Command

Syd Poland
Digital Signal Processing Solutions

Printed on Recycled Paper

INTRODUCTION

This application note demonstrates the use of the 'C8x MP *transform3* and *concat3x3* operations. These single-precision (SP) FP operations are entries in the MP's *xform3.asm* source subroutine. (The reader can examine the *xform3.asm* subroutine by obtaining the 'C8x tools CD-ROM available through Texas Instruments. The *xform3.asm* code is located in the *xform3* subdirectory.) The following topics are discussed:

- Problem Definitions
- Conventions and Structures
- Calling Sequences
- Application Notes
- Performance Estimates

PROBLEM DEFINITIONS

The following problem can be solved using the *transform3* operation:

{InVector} * **[Matrix]** = **{OutVect}**

for a number of sequential vectors **numvec** > 0.

- **InVector** is a 1x3 SP FP input vector (4-byte boundary).
- **Matrix** is a 3x3 SP FP input matrix (8-byte boundary).
- **OutVect** is a 1x3 SP FP output vector (4-byte boundary).

Another form of this problem can be solved using the *concat3x3* matrix-multiply operation:

{InVector} * **[Matrix]** = **{OutVect}**

for the number of sequential vectors **numvec** = 3.

- **InVector** is three 1x3 SP FP input vectors (4-byte boundary).
- **Matrix** is a 3x3 SP FP input matrix (8-byte boundary).
- **OutVect** is three 1x3 SP FP output vectors (4-byte boundary).

CONVENTIONS AND STRUCTURES

The following C structures are used to demonstrate data storage:

```
float InVector [numvec][3];
float Matrix [3][3];          /* 8-byte double-word boundary */
float OutVect [numvec][3];
int numvec;                   /* numvec is greater than zero.*/
```

where **numvec** is the number of sequential vectors (> 0) to be transformed.

The elements of the input/output 1x3 vectors are accessed using *vector load single-word* or *vector store single-word*. The elements of the 3x3 matrix are accessed using *vector load double-word*; therefore, **Matrix** must be aligned on an 8-byte boundary.

The following structure can be used to ensure that each matrix starts on a 64-bit (that is, 8-byte) boundary:

```
typedef struct s {float Matrix [3][3]; } S;
S A;
```

A.Matrix[0][0] is the first element of the 3x3 matrix and is located on a double-word boundary. For additional information, refer to the *TMS320C8x Code Generation Tools User's Guide*.(1)

C stores arrays by row (that is, in row-major order). Therefore,

Input	Matrix	=	Output
{x y 1}	$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$		{x' y' 1}

is equivalent to the following excerpt from *Computer Graphics, Principle and Practice:*(2)

Matrix	Input	=	Output
$\begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$	$\begin{Bmatrix} x \\ y \\ 1 \end{Bmatrix}$		$\begin{Bmatrix} x' \\ y' \\ 1' \end{Bmatrix}$

Thus, sequential vectors can be stored as rectangular arrays, thereby improving computational efficiency.

CALLING SEQUENCES

The standard C calling conventions used within a program are as follows:

- Global declarations of functions

```
void transform3();
void concat3x3();
```

- Local declaration of *transform3* function

```
transform3(InVector, Matrix, OutVect, numvec);
float InVector [numvec][3];
float Matrix [3][3];       /* 8-byte double-word boundary */
float OutVect [numvec][3];
int numvec;                /* numvec is greater than zero.*/
```

- Local declaration of *concat3* function

```
concat3x3(InVector, Matrix, OutVect);
float InVector [3][3];
float Matrix [3][3];       /* 8-byte double-word boundary */
float OutVect [3][3];
int numvec = 3;
```

The standard ’C8x MP assembly-calling conventions used by C include the following:

- Interrupt-enable *bit 0* must be set to one for FP operations.
- r2 = **InVector** address (4-byte boundary)
- r4 = **Matrix** input address (8-byte boundary)
- r6 = **OutVec** address (4-byte boundary)
- r8 = integer **numvec** > 0 (*concat3x3* sets **numvec** = 3)
- jsr.a *_transform3*(r0),r31

  or

- jsr.a *_concat3x3*(r0),r31
- Any register used in the range r20–r31 is saved on the stack and restored (zero stack words are used).

APPLICATION NOTES

The *concat3x3* entry is aligned on an instruction-cache boundary (multiple of 16 32-bit instruction words) and continues sequentially to *transform3* after setting **numvec** = 3.

Input/output 1x3 vectors are read and stored using *vector load/store single-word*. However, the 3x3 matrix is read using *vector load double-word* (five memory accesses instead of nine for vector load single-word) and, therefore, must be on an 8-byte boundary. *Big endian* is the expected addressing method.

As the ’C8x MP has a data-cache size of 4096 bytes (341 1x3 vectors), the data-cache hit rate can be improved by setting the **InVector** address to the **OutVect** address (336 1x3 vectors). Otherwise, 168 1x3 input and output vectors will fill the data cache. **Matrix** requires nine words.

If there is a large number of vectors to be processed, most of the data-cache times can be hidden by executing a *look-ahead read* of one vector (16 words = 5.333 vectors). This means that every 5.333 vectors can cause a read data-cache miss (or three cache misses for every 16 vectors).

Write data-cache will be required if the data cache is full and the least-recently-used (LRU) sub-block has been modified. Only after the data-cache has been written can a new sub-block be read from external memory.

Three instruction-cache reads will be required to load this program if it is not resident in the instruction cache.

PERFORMANCE ESTIMATES

Performance data presented in this section is based upon the following assumptions:

- The clock rate is 50 MHz.
- There is a 16-word cache sub-block (data or instructions).
- External DRAM is two cycles, and the read/write cache sub-block is a minimum of 22 cycles.
- The transfer controller (TC) is immediately available for all cache services.

- When performing *look-ahead read data*, the wait time for the data load is ~15 cycles net per cache miss (the rest of the read cycles are overlapped with computations).
- All data and programs begin on a cache sub-block boundary (multiple of 16 32-bit words).
- The **numvec** values shown provide the best performance; however, available memory determines the practical **numvec** limit.
- There are no FP exceptions and no trap or interrupt-service calls.
- Three instruction-cache reads for the program are not required.

Performance can be affected by a variety of factors. The *transform3* performance data in Table 1 is based upon the following assumptions:

- Instructions are already resident in the instruction cache (three sub-block reads are not needed).
- All **InVectors** and **Matrix** are also resident in the data cache.
- All **OutVects** overwrite **InVectors**, that is, in-address = out-address.
- **numvec** is within the range 100 to 330 for the same **Matrix**.
- Peak performance is nine cycles per vector for a computational rate of 5.5 million 1x3 vectors per second.
- Adding two loop-control instructions (11 cycles per vector) produces a computational rate of 4.5 million 1x3 vectors per second.
- Performance is 12 cycles per vector if a *load look-ahead read* is also included and there are no cache misses for a computational rate of 4.1 million 1x3 vectors per second.
- One read-cache miss (22 cycles less seven cycles are overlapped with the computation every 5.333 vectors for an average of three cycles per 1x3 vector) results in a 15-cycle-per-vector net time for a computational rate of 3.3 million 1x3 vectors per second.
- Two cache misses (read-read or write-read) add 22 cycles for 5.333 1x3 vectors for an average of 19.125 cycles per 1x3 vector. This results in a computational rate of 2.6 million 1x3 vectors per second.
- Three cache misses (read-write-read or write-read-read) are the performance asymptote. If the input data sets are not resident in the data cache, or the **InVector** address is not equal to the **OutVect** address, or **numvec** is greater than 330, the performance becomes 23.25 cycles per vector, and the computational rate becomes 2.1 million 1x3 vectors per second.

Table 1. MP Single-Precision FP Performance for {1x3} x [3x3] = {1x3}

CYCLES PER 1x3 VECTOR	50-MHz 'C8x, 2-Cycle DRAM		
	Million 1x3 VECTORS PER SECOND	MFLOPS	COMMENTS
9	5.5	100	Absolute Peak
11	4.5	81	Loop Control
12	4.1	75	Load Look-Ahead
15	3.3	60	1 Cache Miss
19.125	2.6	47	2 Cache Misses
23.25	2.1	39	3 Cache Misses

NOTES: 1. This assumes **numvec** > 100 with no FP exceptions, traps, interrupts, instruction-cache misses, contention, or waiting for the TC.
2. These preliminary estimates are subject to change. Silicon performance will determine the final values.

SUMMARY

The data presented demonstrates the efficiency of the 'C8x MP.[3] This device can exceed five million 1x3 SP FP vectors per second in two-dimensional coordinate transformations.

REFERENCES

1. *TMS320C8x Code Generation Tools User's Guide,* Texas Instruments, 1995, literature number SPRU108.
2. J.D. Foley et al., *Computer Graphics, Principle and Practice,* Addison-Wesley Publishing Co., 1990, pp. 201–210.
3. *TMS320C8x Master Processor User's Guide,* Texas Instruments, 1995, literature number SPRU109.

Transform4 Command

Syd Poland
Digital Signal Processing Solutions

Printed on Recycled Paper

INTRODUCTION

This application note demonstrates the use of the 'C8x MP *transform4* and *concat4x4* operations. These single-precision (SP) FP operations are entries in the MP's *xform4.asm* source subroutine. (The reader can examine the *xform4.asm* subroutine by obtaining the 'C8x tools CD-ROM available through Texas Instruments. The *xform4.asm* code is located in the *xform4* subdirectory.) The following topics are discussed:

- Problem Definitions
- Conventions and Structures
- Calling Sequences
- Application Notes
- Performance Estimates

PROBLEM DEFINITIONS

The following problem can be solved using the *transform4* operation:

{InVector} * **[Matrix]** = **{OutVect}**

for a number of sequential vectors **numvec** > 0.

- **InVector** is a 1x4 SP FP input vector (8-byte boundary).
- **Matrix** is a 4x4 SP FP input matrix (8-byte boundary).
- **OutVect** is a 1x4 SP FP output vector (8-byte boundary).

Another form of this problem can be solved using the *concat4x4* matrix-multiply operation:

{InVector} * **[Matrix]** = **{OutVect}**

for the number of sequential vectors **numvec** = 4.

- **InVector** is four 1x4 SP FP input vectors (8-byte boundary).
- **Matrix** is a 4x4 SP FP input matrix (8-byte boundary).
- **OutVect** is four 1x4 SP FP output vectors (8-byte boundary).

CONVENTIONS AND STRUCTURES

The following C structures are used to demonstrate data storage:

```
float InVector [numvec][4];   /* 8-byte double-word boundary */
float Matrix [4][4];          /* 8-byte double-word boundary */
float OutVect [numvec][4];    /* 8-byte double-word boundary */
int numvec;                   /* numvec is greater than zero.*/
```

where **numvec** is the number of sequential vectors (> 0) to be transformed.

The elements of the input/output 1x4 vectors are accessed using *vector load double-word* or *vector store double-word.* The elements of the 4x4 matrix are accessed using *vector load double-word*; therefore, **Matrix** and both input/output vectors must be aligned on an 8-byte boundary.

The following structure can be used to ensure that each matrix starts on a 64-bit (that is, 8-byte) boundary:

```
typedef struct s {float Matrix [4][4]; } S;
S A;
```

A. Matrix[0][0] is the first element of the 4x4 matrix and is located on a double-word boundary. For additional information, refer to *the TMS320C8x Code Generation Tools User's Guide*.(1)

C stores arrays by row (that is, in row-major order). Therefore,

Input	Matrix	=	Output
{x y z 1}	$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$		{x' y' z' 1}

is equivalent to the following excerpt from *Computer Graphics, Principle and Practice:*(2)

Matrix	Input	=	Output
$\begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$	$\begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$		$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix}$

Thus, sequential vectors are stored as rectangular arrays, thereby improving computational efficiency.

CALLING SEQUENCES

The standard C calling conventions used within a program are as follows:

- Global declarations of functions

```
void transform4();
void concat4x4();
```

- Local declaration of *transform4* function

```
transform4(InVector, Matrix, OutVect, numvec);
float InVector [numvec][4];  /* 8-byte double-word boundary */
float Matrix [4][4];         /* 8-byte double-word boundary */
float OutVect [numvec][4];   /* 8-byte double-word boundary */
int numvec;                  /* numvec is greater than zero.*/
```

- Local declaration of *concat4x4* function

```
concat4x4(InVector, Matrix, OutVect);
float InVector [4][4];       /* 8-byte double-word boundary */
float Matrix [4][4];         /* 8-byte double-word boundary */
float OutVect [4][4];        /* 8-byte double-word boundary */
int numvec = 4;
```

The standard ’C8x assembly-calling conventions used by C include the following:

- Interrupt-enable *bit 0* must be set to one for FP operations.
- r2 = **InVector** address (8-byte boundary)
- r4 = **Matrix** input address (8-byte boundary)
- r6 = **OutVec** address (8-byte boundary)
- r8 = integer **numvec** > 0 (*concat4x4* sets **numvec** = 4)
- jsr.a *_transform4*(r0),r31

  or
- jsr.a *_concat4x4*(r0),r31
- Any register used in the range r20–r31 is saved on the stack and restored (eight stack words are used).

APPLICATION NOTES

The *concat4x4* entry is aligned on an instruction-cache boundary (multiple of 16 32-bit instruction words) and continues sequentially to *transform4* after setting **numvec** = 4.

Input/output 1x4 vectors are read and stored using *vector load/store double-word* (two memory accesses instead of four for vector load/store single-word). Also, the 4x4 matrix is read using *vector load double-word* (eight memory accesses instead of sixteen for vector load single-word); therefore, both vectors and **Matrix** must be on an 8-byte boundary. *Big endian* is the expected addressing method.

As the ’C8x MP has a data-cache size of 4096 bytes (256 1x4 vectors), the data-cache hit rate can be improved by setting the **InVector** address to the **OutVect** address (252 1x4 vectors). Otherwise, 126 1x4 input and output vectors will fill the data cache. **Matrix** requires 16 words.

If there is a large number of vectors to be processed, most of the data-cache times can be hidden by executing a *look-ahead read* of one vector (16 words = 4 vectors). This means that every 4 vectors can cause a read data-cache miss.

Write data-cache will be required if the data cache is full and the least-recently-used (LRU) sub-block has been modified. Only after the data-cache has been written can a new sub-block be read from external memory.

Four instruction-cache reads will be required to load this program if it is not resident in the instruction cache.

PERFORMANCE ESTIMATES

Performance data presented in this section is based upon the following assumptions:

- The clock rate is 50 MHz.
- There is a 16-word cache sub-block (data or instructions).
- External DRAM is two cycles, and the read/write cache sub-block is a minimum of 22 cycles.
- The transfer controller (TC) is immediately available for all cache services.

- When performing *look-ahead read data*, the wait time for the data load is ~6 cycles net per cache miss (the rest of the read cycles are overlapped with computations).
- All data and programs begin on a cache sub-block boundary (multiple of 16 32-bit words).
- The **numvec** values shown provide the best performance; however, available memory determines the practical **numvec** limit.
- Performance assumes no FP exceptions and no trap or interrupt-service calls.
- Four instruction-cache reads for the program are not required.

Performance can be affected by a variety of factors. The *transform4* performance data in Table 2 is based upon the following assumptions:

- Instructions are already resident in the instruction cache (four sub-block reads are not needed).
- All **InVectors** and **Matrix** are also resident in data cache.
- All **OutVects** overwrite **InVectors**, that is, in-address = out-address.
- **numvec** is within the range 75 to 248 for the same **Matrix.**
- Peak performance is 16 cycles per vector for a computational rate of 3.1 million 1x4-vectors per second.
- Adding two loop-control instructions (18 cycles per vector) produces a computational rate of 2.8 million 1x4-vectors per second.
- Performance is 19 cycles per vector if a *load look-ahead read* is also included and there are no cache misses for a computational rate of 2.6 million 1x4-vectors per second.
- One read-cache miss (22 cycles less 16 cycles are overlapped with the computation every 4 vectors for an average of 1.5 cycles per 1x4 vector) results in a 20.5-cycle-per-vector net time for a computational rate of 2.4 million 1x4-vectors per second.
- Two cache misses (read-read or write-read) add 22 cycles for four 1x4 vectors for an average of 26 cycles per 1x4 vector. This results in a computational rate of 1.9 million 1x4-vectors per second.
- Three cache misses (read-write-read or write-read-read) are the performance asymptote. If the input data sets are not resident in the data cache, or the **InVector** address is not equal to the **OutVect** address, or **numvec** is greater than 248, the performance becomes 31.5 cycles per vector, and the computational rate becomes 1.6 million 1x4-vectors per second.

Table 2. MP Single-Precision FP Performance for {1x4} x [4x4] = {1x4}

CYCLES PER 1x4 VECTOR	50-MHz 'C8x, 2-Cycle DRAM		
	Million 1x4 VECTORS PER SECOND	MFLOPS	COMMENTS
16	3.1	100	Absolute Peak
18	2.8	89	Loop Control
19	2.6	84	Load Look-Ahead
20.5	2.4	78	1 Cache Miss
26	1.9	62	2 Cache Misses
31.5	1.6	51	3 Cache Misses

NOTES: 3. This assumes **numvec** > 75 with no FP exceptions, traps, interrupts, instruction-cache misses, contention, or waiting for the TC.
4. These preliminary estimates are subject to change. Silicon performance determines the final values.

SUMMARY

The data presented demonstrates the efficiency of the 'C8x MP.(3) This device can exceed three million 1x4 SP FP vectors per second in three-dimensional coordinate transformations.

REFERENCES

1. *TMS320C8x Code Generation Tools User's Guide,* Texas Instruments, 1995, literature number SPRU108.
2. J. D. Foley et al., *Computer Graphics, Principle and Practice,* Addison-Wesley Publishing Co., 1990, pp. 201–227.
3. *TMS320C8x Master Processor User's Guide,* Texas Instruments, 1995, literature number SPRU109.

Draw Colored Lines Command

Syd Poland
Digital Signal Processing Solutions

Printed on Recycled Paper

INTRODUCTION

This document describes the operation of the _PPCMD_COLOR_LINES command as it is used on the PP core of a single-PP TMS320C8x. The PP draw colored lines source code is the *cololine.s* subroutine. (The reader can examine the *cololine.s* subroutine by obtaining the 'C8x tools CD-ROM available through Texas Instruments. The *cololine.s* code is located in the graphics subdirectory.) The following topics are discussed:

- Function Definitions
- Conventions and Structures
- Calling Sequence
- Application Notes
- Performance Estimates
- Notes for Single Long Lines
- Sample MP C Program

FUNCTION DEFINITIONS

The _PPCMD_COLOR_LINES command draws a suite of colored (Bresenham) lines. Each line can have a color that is different from the color of any other line. For more information on the Bresenham line draw algorithm, refer to the *TMS34010 Application Guide*.(1)

Double buffering is used to overlap line calculation in the PP's core with I/O in the transfer controller (TC). Each line is one 16-bit-pixel thick and is output as an unconditional write – that is, no transparency, plane masking, blending, or anti-aliasing is applied. This process occurs primarily on PP0 of the 'C8x, and results in the use of the two extra PP0 data RAM banks by the _PPCMD_COLOR_LINES command. The master processor (MP) performs any clipping needed to fit lines into a display window before _PPCMD_COLOR_LINES is called. The MP also ensures that the maximum line length of 495 pixels and the maximum number of lines per call (63) are not exceeded. These processes provide the user with the capability to draw any number of lines in any length. A sample line within a window is illustrated in Figure 1.

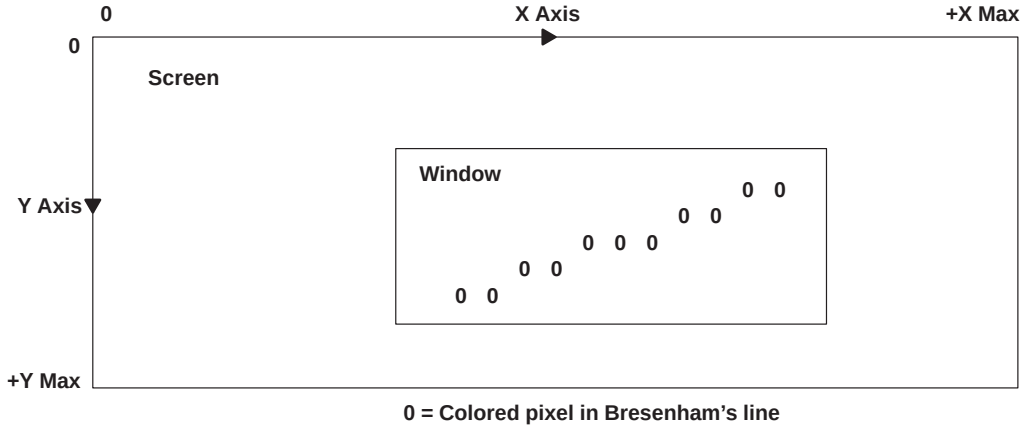

Figure 1. Line Draw in Window on Screen

Note the following statements/assumptions:

- The MP must pre-clip data to fit within the display window.
- The MP must divide individual lines into shorter lengths to ensure a maximum of 495 pixels per line.
- The MP must not exceed 63 lines for each call to _PPCMD_COLOR_LINES.
- External memory is *big endian*.
- Lines are one pixel thick.
- Pixels are 16-bits and are aligned on a two-byte address.
- The PP may require the two extra data RAMs found on PP0 – the MP must assign the LINE argument lists in all cases.
- The PP does not preserve registers upon returning from a call to _PPCMD_COLOR_LINES (with the exception of the stack pointer).
- The PP is dedicated to _PPCMD_COLOR_LINES until this function terminates.
- The screen display originates in the upper lefthand corner: (X,Y) = (0,0).
- X increases horizontally – that is, from left to right, across the screen.
- Y increases vertically – that is, from top to bottom, across the screen.
- The origin of the visible screen, which is an even byte address in external memory, is in the *graphics context*.
- *Pitch*, an even byte-address increment, is the number of bytes in a horizontal screen row and is in the *graphics context*.
- Integers are assumed to be signed unless otherwise noted.
- Integers are assumed to be 32-bit unless 16-bit halfwords are designated.
- PP0 is the expected processor on the single-PP 'C8x; however, if the 'C8x with four PPs is used, then PP1, PP2, or PP3 can be used. *Argument lists for PP3 must be assigned by the MP as valid on-chip addresses!*
- Because two data RAM banks from another PP could be required on a four-PP 'C8x system, precautions must be taken to ensure that this *other* PP does not use these two data RAM banks. The safest rule is for the *other* PP to remain idle.
- *Reserved* means a value can be zero or non-zero. Application programs should make no assumptions concerning the value in a reserved field.
- Lines are drawn in the order supplied.
- _PPCMD_COLOR_LINES exits without waiting for the TC to complete the last line draw request. If called immediately after _PPCMD_COLOR_LINES exits, the PP should wait for TC completion before using its data RAMs.

CONVENTIONS AND STRUCTURES

The MP generates a list of lines to be drawn (see Figure 2). The MP ensures that the number of lines is less than 64, and the number of pixels in each line is less than 496. The assumption is that the MP program

pre-clips any line to the desired window before the colored line draw is called. If any line is too long, the MP subdivides the line into shorter line segments. LINE is a pointer to the colored line argument list.

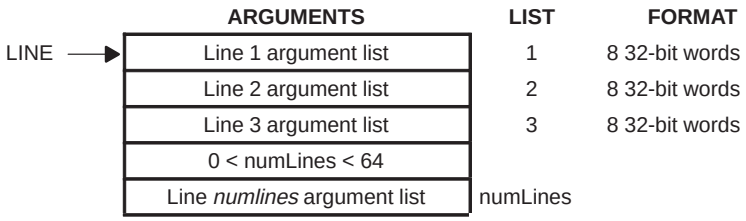

Figure 2. Argument List for Colored Lines

Colored Line Argument List

The structure of each colored line request as generated by the MP is shown in Figure 3.

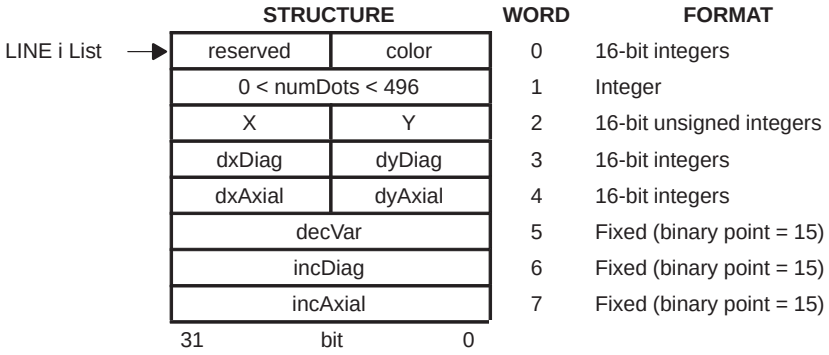

Figure 3. Argument List for an Individual Colored Line

Elements of the argument list structure are shown in Table 3.

Table 3. Structure of an Argument List for an Individual Colored Line

color	16-bit color value for this colored line
numDots	Number of pixels in this line (0 < numDots < 496)
X	First X coordinate for this line (unsigned 16-bit integer)
Y	First Y coordinate for this line (unsigned 16-bit integer)
dxDiag	Δx step in the diagonal direction (–1, 0, +1 pixel)
dyDiag	Δy step in the diagonal direction (–1, 0, +1 pixel)
dxAxial	Δx step in the axial direction (–1, 0, +1 pixel)
dyAxial	Δy step in the axial direction (–1, 0, +1 pixel)
decVar	Initial Bresenham decision variable at (X,Y)
incDiag	Decision variable increment for a diagonal step
incAxial	Decision variable increment for an axial step

Command List

Once the line arguments are generated, the MP passes a command structure to the PP (see Figure 4). The standard pointer to this command list is *arg1*.

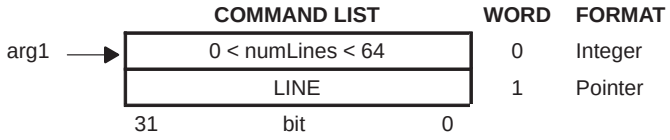

Figure 4. Argument List for Colored Lines Call

Elements of the command list are shown in Table 4.

Table 4. Structure of an Argument List for a Colored Lines Call

numLines	Number of lines in this list (0 < numLines < 64)
LINE	Pointer to the first line's argument list (see Figure 2)

Delta Guide Table

The Bresenham line algorithm is employed to generate the pixel locations of a requested line. The TC packet request form used in this document is the fixed-patch delta guided tour, which requires a delta guide table as shown in Figure 5.

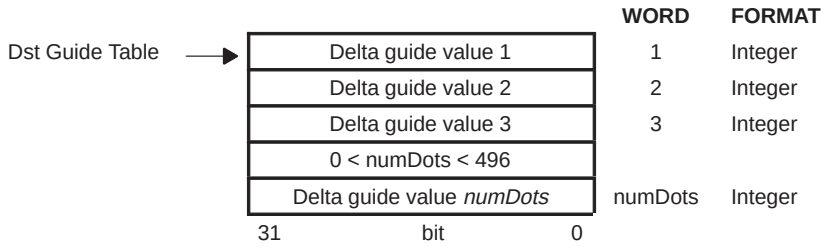

Figure 5. Fixed-Patch Delta Guide

The even-byte addresses of pixels in the line are formed as follows:

pixel address 1 = Dst start address + Delta guide value 1

pixel address 2 = pixel address 1 + Delta guide value 2

pixel address 3 = pixel address 2 + Delta guide value 3

•
•
•

pixel address *numDots* = pixel address (*numDots* – 1) + Delta guide value *numDots*

Transfer Packet Request

Once the guide table has been generated, a transfer packet request is issued to the TC. The request format is illustrated in Figure 6. *tPT* is the pointer to the packet request parameter list.

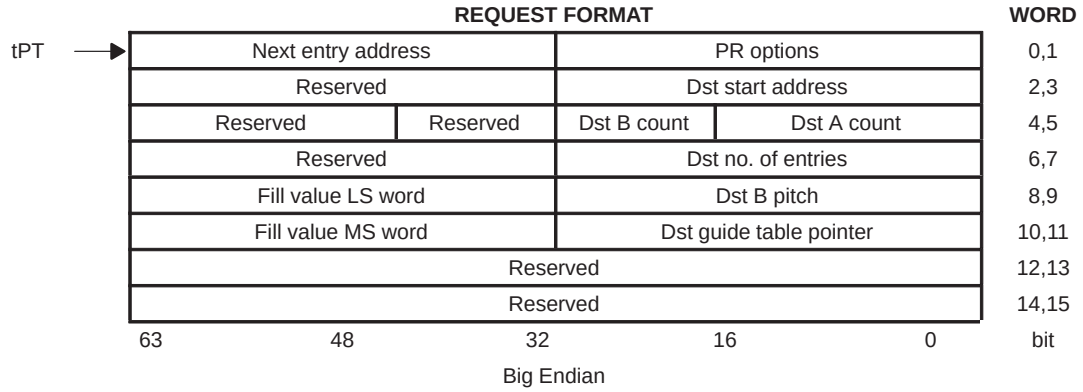

Figure 6. Fill-With-Value to Fixed-Patch Delta Guided Tour Packet Request

The transfer packet request contents are shown in Table 5.

Table 5. Structure of a Fill-With-Value to Fixed-Patch Delta Guided Tour Packet

Next entry address	Address of the next packet request
PR options	Options for this packet request (fill-with-value to fixed-patch delta guided tour, stop when this packet request is done)
Dst start address	Even byte address of screen's origin in external memory
Dst B count	Destination B count is supplied as zero
Dst A count	Destination A count is supplied as two bytes per 16-bit pixel
Dst no. of entries	Destination number of entries (0 < numDots < 496) in guide table
Fill value LS word	Two copies of this line's 16-bit color value
Dst B pitch	Destination B pitch is not required (as Dst B count = 0)
Fill value MS word	Two copies of this line's 16-bit color value
Dst guide table pointer	Pointer to the delta guide table for this line (see Figure 5)

Memory Allocation

The target system of this implementation is the single-PP 'C8x. Line argument lists are double-buffered using PP0 data RAM 3 and PP0 data RAM 4 (see top portion of Figure 7). Packet requests and delta guide tables are also double-buffered using PP0 data RAM 0 and PP0 data RAM 1 (see middle portion of Figure 7). PP parameter RAM is used to issue the packet request to the TC and also to provide transfer addresses for PP interrupts (see bottom-right portion of Figure 7). PP0 data RAM 2 contains the *graphics context* (see bottom-left of Figure 7) as pointed to by the *GC* pointer.

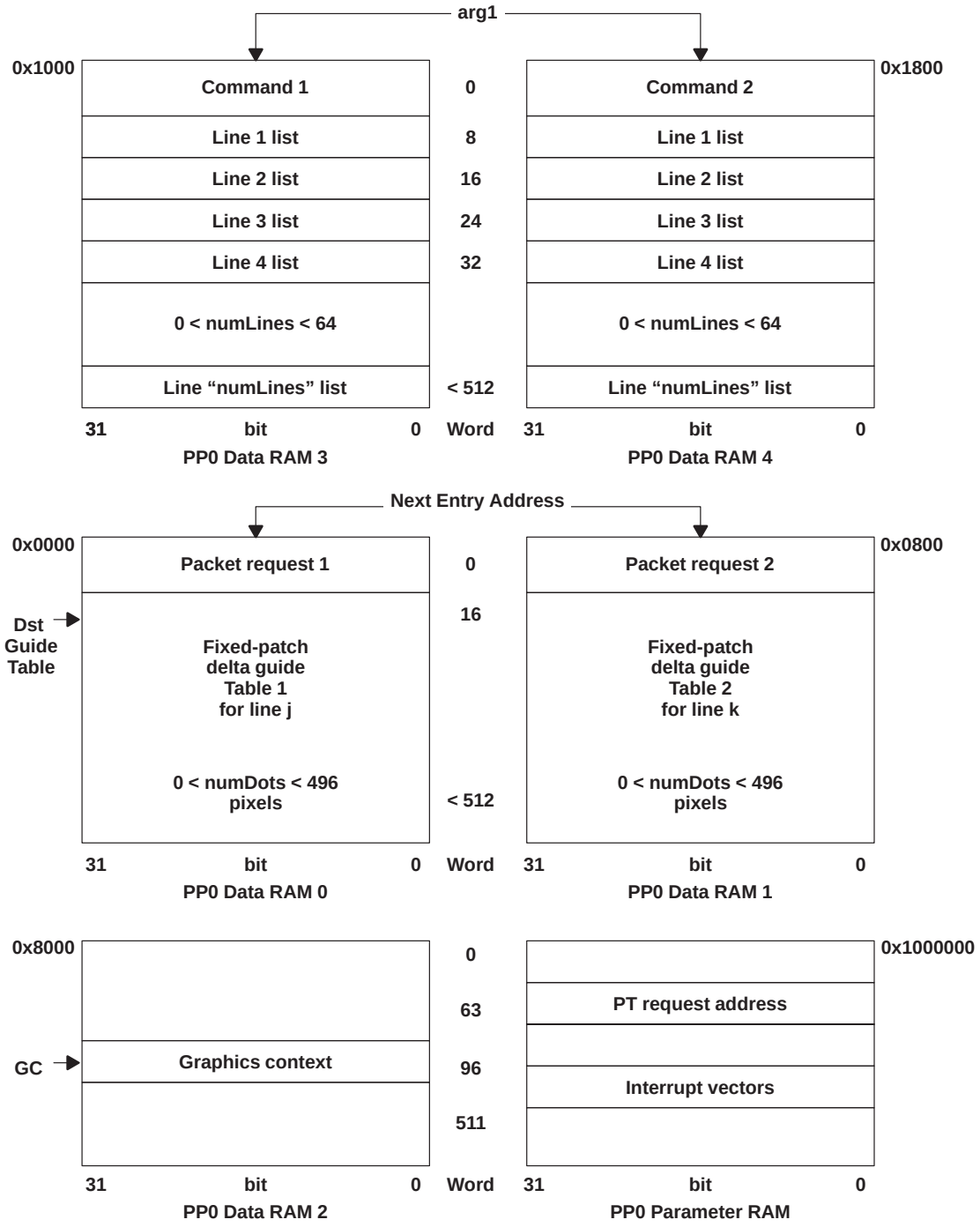

Figure 7. Memory Allocation

The *graphics context* variables accessed (see *graphics.i*) are shown in Table 6.

Table 6. Graphics Context Variables

GC.SCRNADRS	32-bit integer, even-byte base address of the visible screen's origin in external memory. This becomes Dst start address in the packet request (see Figure 6).
GC.SCRNPITCH	32-bit integer, even number of external memory bytes in each horizontal screen row (*pitch*)

CALLING SEQUENCE

_PPCMD_COLOR_LINES is called in accordance with the PP command structure as initiated by the MP. The sequence of events could include the following:

- MP events
  - Obtains message and command buffers
  - Builds message and command buffers
  - Performs pre-clipping and line partitioning as required for argument lists
  - Initiates a message to the target PP
- PP events
  - Responds to the message interrupt
  - Decodes a message and calls the application program (sets pointer to *arg1* and begins execution of _PPCMD_COLOR_LINES)
  - Upon entry, target PP spins until the TC is idle (for the local PP)
  - Generates each line and issues packet requests to the TC
  - Uses double buffering to overlap PP computations with TC output
  - Returns, signals the MP when the application is done (if required)

Refer to the *TMS320C8x (MVP) Multitasking Executive User's Guide* for a complete treatise on this subject.(2) The PP command buffer is shown in Figure 8.

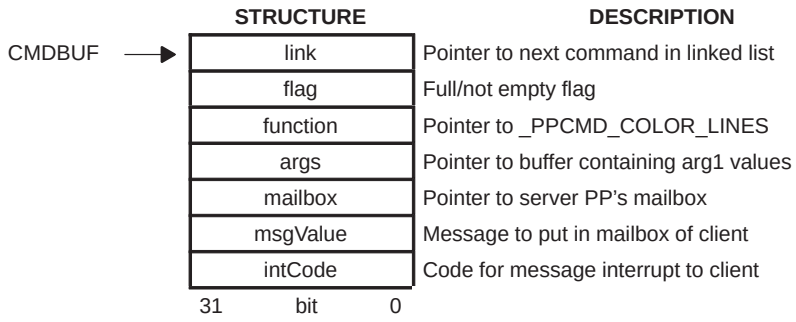

Figure 8. PP Command Buffer

Call in PP Assembly Language Versus Call in C

The colored line draw can be called from PP assembly code after all structures are established, as shown in Figure 9. The standard C subroutine call (Figure 9, lower-case label) with the pointer in *d1* continues to the upper-case label (assembly language alternate subroutine call) by moving *d1* to *a9*.

STANDARD ASSEMBLY EQUIVALENT OF A PP C CALL	PP ASSEMBLY LANGUAGE ALTERNATE
.global _ppcmd_color_lines call = _ppcmd_color_lines d1 = arg1 ; see Figure 4 nop ; delay slot 2 Return: – – – ; call returns here	.global _PPCMD_COLOR_LINES call = _PPCMD_COLOR_LINES a9 = arg1 ; see Figure 4 nop ; delay slot 2 Return: – – – ; call returns here

Figure 9. PP Colored Line Call

APPLICATION NOTES

This section discusses the following topics:

- Double Buffering
- Convert (X,Y) Coordinates to Byte Offsets
- Overlapped Processing
- Future Options

Double Buffering

The double buffering of argument lists and delta guide tables provides the opportunity to overlap MP and PP computations with TC I/O. Upon entry, the PP spins until there is no TC activity for the PP. Thereafter, the PP calculates a Bresenham line and issues a TC request with *stop on complete* enabled. If the PP completes its next line computation before the TC is done, the next TC request forces a packet-request-busy interrupt, thereby causing the PP to spin until the TC has completed the earlier packet request. The current packet request then begins, and the PP continues to the next line.

Convert (X,Y) Coordinates to Byte Offsets

The (X,Y) coordinates are converted to even byte offsets relative to the screen origin as:

`offset = Y * pitch + X * 2 bytes per pixel`

where *pitch* is the even number of bytes in each horizontal screen row.

Overlapped Processing

The preferred embodiment of colored line draw includes (see Figure 10):

- MP window pre-clipping and segmentation of long lines
- PP development of the line's pixel addresses
- TC writing of colored pixel values to external RAM

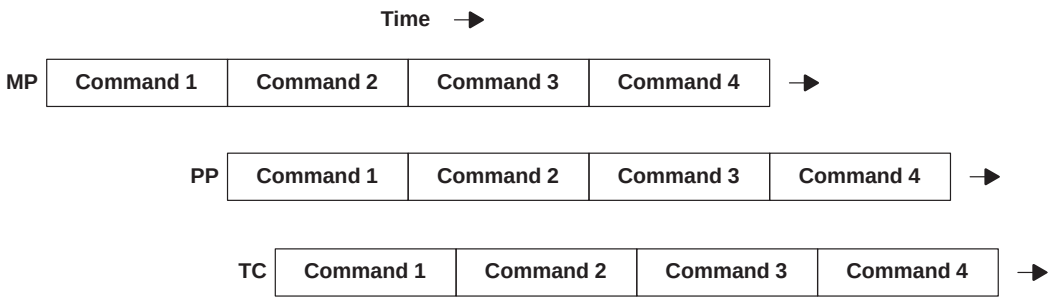

Figure 10. MP, PP, TC Overlapped Processing

These commands can be subdivided further to illustrate individual lines (see Figure 11). The MP's Command **K** issues all numLines as a single command. The PP's response (Command **J**) is to generate a line and issue a TC request until all lines have been rendered. In general:

MP $Line_K$ (M) time ≠ PP $Line_J$ (L) time ≠ TC $Line_J$ (L–1) time

Performance is dominated by the largest line time value of the three processors – that is, the MP, PP, and TC.

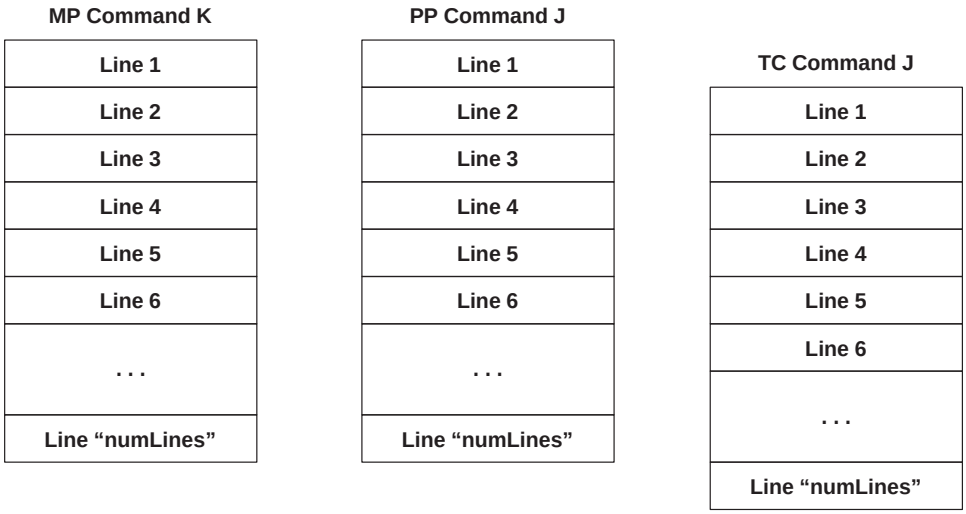

Figure 11. Three Processor Overlapped Times

Future Options

One way to improve performance would be to detect special line orientations such as horizontal, vertical, or diagonal lines. These cases (if detected by additional code) would provide improved TC performance using the fill-with-value to dimensioned packet request form. The following changes would be required in the packet request parameters for horizontal lines and for vertical or diagonal lines (see Figure 6).

Horizontal Lines

Table 7 shows the packet request parameter changes required to detect horizontal line orientations.

Table 7. Parameter Changes Required to Detect Horizontal Line Orientations

ELEMENT	DESCRIPTION
PR options	Packet request options (*fill-with-value* to *dimensioned*; stop when packet request is processed)
Dst start address	Even byte address in external memory of the initial (X,Y) coordinates
Dst A count	Destination A count = numDots * 2 bytes per pixel

Vertical or Diagonal Lines

Table 8 shows the packet request parameter changes required to detect vertical or diagonal lines.

Table 8. Parameter Changes Required to Detect Vertical/Diagonal Lines

ELEMENT	DESCRIPTION
PR options	Packet request options (*fill-with-value* to *dimensioned*; stop when packet request is processed)
Dst start address	Even byte address in external memory of the initial (X,Y) coordinates
Dst B count	Destination B count = numDots – 1
Dst B pitch	Destination B pitch = pitch + τ * 2 bytes/pixel

Pitch = the even number of bytes per horizontal screen row and τ = (–1, 0, +1) depending on the line orientation (0 = vertical line, –1 = –45° diagonal line, +1 = +45° diagonal line).

NOTES: 5. Packet request values must be initialized for the random line orientation if these options are employed.
6. The fixed-patch delta guide tables (see Figure 5) are not required for this form of horizontal, vertical, or diagonal lines.

PERFORMANCE ESTIMATES

Performance estimates assume no contention for on-chip or off-chip RAM access and that there is no TC round-robin contention. The following must be considered:

- MP computational time
- PP computational time
- TC I/O time including external memory organization
- Throughput as determined by max(MP time, PP time, TC time)
- Estimated time for 50 10-pixel vectors

MP Line Clipping and Segmentation

The MP must clip a line to the desired window and also must segment lines so that the maximum number of pixels per line is not exceeded. Line values then are placed in the colored line argument list (see Figure 3). Once the set of lines has been entered, the MP issues a message to the PP for the colored line draw command. MP computational time is determined by the program(s) on the MP and by another specific, application-dependent value, which is not discussed within this document.

PP Line Draw Estimates

PP line draw estimates are as follows:

- Overhead (excluding instruction cache) = 24 cycles if the TC is idle (more if the TC is busy)
- Time for each line draw with numDots pixels = 17 + numDots cycles
- Time for *constant line length* numLines = 17 + numDots * numLines cycles
- Time for *variable line length* numLines = Σ (17 + $numDots_k$) cycles for k = 1, 2, ..., numLines
- Total time for *constant line length* = 24 + (17 + numDots) * numLines cycles
- Total time for *variable line length* = 24 + Σ (17 + $numDots_k$) cycles for k = 1, 2, ..., numLines

TC Line Draw Estimates

Assume that external memory organization uses two-cycle DRAMs with n rows per page (n = 1, 2, 4) and that the page fault select time is six additional cycles. The orientation of the line determines how many *write* page faults occur. The minimum number of cycles required for each line is as follows:

- For zero page faults: 15 + 2 * numDots cycles (near horizontal line within same page)
- For numDots in 4 rows per page: 15 + λ * 3.5 * numDots cycles
- For numDots in 2 rows per page: 15 + λ * 5 * numDots cycles
- For numDots in 1 row per page: 15 + λ * 8 * numDots cycles

The value λ is the statistical average of the number of page faults that occurs assuming the line orientation is equally likely to be in any direction. If numDots equals ten, the lines from 0° to (but not including) 45° average 4.5 rows per vector. Likewise, the lines from 45° through 90° are always ten rows per vector. Combining these two values yields an average of 7.25 rows per vector. Normalizing this by ten pixels per vector yields λ = 0.725 average rows per pixel.

Line Draw Throughput of Combined (PP, TC)

The total throughput limit is max(PP time, TC time); MP time is not included within this document.

Time = max[24 + (17 + numDots) * numLines, (15 + λ * 5 * numDots) * numLines] cycles

assuming constant line length, two-cycle DRAMs with two rows per page, and no instruction cache misses.

Estimated Time for 50 10-Pixel Vectors

The definition of a vector (as used here) is ten 16-bit pixels per vector. Estimated times for drawing 50 vectors are as follows:

- PP time = 24 + (17 + 10) * 50 = 1374 cycles (excludes instruction cache)
- TC time = (15 + λ * 3.5 * 10) * 50 = 2019 cycles (four rows per page)
- TC time = (15 + λ * 5 * 10) * 50 = 2563 cycles (two rows per page)
- TC time = (15 + λ * 8 * 10) * 50 = 3650 cycles (one row per page)
- (PP, TC) time = max (1374, 2019) = 2019 cycles (four rows per page)
- (PP, TC) time = max (1374, 2563) = 2563 cycles (two rows per page)
- (PP, TC) time = max (1374, 3650) = 3650 cycles (one row per page)

See Figure 12 for performance estimates that assume the use of a 50-MHz part.

			TC, Two Cycle DRAM n Rows/Page		
Item	**Units**	**PP**	**n = 4**	**n = 2**	**n = 1**
PP, no i-cache	Cycles/50 vec	1374			
TC	Cycles/50 vec		2019	2563	3650
MAX (PP, TC)	Cycles/50 vec		2019	2563	3650
PP at 50 MHz	Million vec/sec	1.8			
TC at 50 MHz	Million vec/sec		1.2	1.0	0.7

Vec/sec = Vectors per second (vec = vector = 10 16-bit pixels)

Figure 12. PP and TC Performance for 50 Vectors

NOTES FOR SINGLE LONG LINES

Long lines can be written by having each call generate only ONE output line. This is accomplished by chaining more of the contiguous PP RAMs together for the longer delta guide tables. *Extreme care must be used when employing this approach!* The packet transfer request (PTR) parameter list in Figure 6 requires 16 words in PP0 data RAM 0; remaining memory is available for use by the first 495 delta offset values in Figure 5 (the TC writes color value to output pixels). Each additional PP data RAM provides 512 extra delta offset values, as shown in Figure 13.

HEXADECIMAL ADDRESS		TOTAL NO. OF PIXELS	
0x0000	PTR + 495 offsets	495	PP0 Data RAM 0
0x8000	+ 512 offsets	1007	PP0 Data RAM 1
0x1000	+ 512 offsets	1519	PP0 Data RAM 3 = PP1 Data RAM 0
0x1800	+ 512 offsets	2031	PP0 Data RAM 4 = PP1 Data RAM 1

Figure 13. Single Long Line Memory

SAMPLE MP C PROGRAM

The following MP C program section (Figure 14) illustrates the line argument list portion of the colored line structure (see Figure 3) after clipping and line segmentation.

```
/* Rectangle defined by two diagonally opposite points  */
typedef struct {int xs, ys, xe, ye;} RECT;
/* A colored line (one-pixel thick, Bresenham-style line)   */
typedef struct {
   unsigned long color;              /*  Pixel value*/
   long numDots;                     /*  Number of pixels in line  */
   short x, y;                       /*  Integer coordinates of first pixel   */
   short dxDiag, dyDiag;             /*  X–Y increments for diagonal step  */
   short dxAxial, dyAxial;           /*  X–Y increments for axial step */
   long decVar;                      /*  Initial value of decision variable   */
   long incDiag, incAxial;           /*  Increments to decision variable  */
} COLOR_LINE;
/* Tell the PP to draw 63 colored lines (max) with 495 pixels (max).  */
for (j = 0; j < 63; ++j) {
       x = lines[j].xs;
       y = lines[j].ys;
       a = lines[j].xe – x;
       b = lines[j].ye – y;
       if (a < 0) {
              a = –a;
              dxDiag = –1;
       } else
              dxDiag = 1;
       if (b < 0) {
              b = –b;
              dyDiag = –1;
       } else
              dyDiag = 1;
       if (a < b) {
              t = a; a = b; b = t;
              dxAxial = 0;
              dyAxial = dyDiag;
       } else {
              dxAxial = dxDiag;
              dyAxial = 0;
```

```
        }
        decVar = 2 * b – a;
        incAxial = 2 * b;
        incDiag = 2 * b – 2 * a;
        p = nextLine( );                    /*  Get pointer to line buffer    */
        p–>color = –1;                      /*  Constant color used here      */
        p–>numDots = a + 1;                 /*  Number of pixels in one line  */
        p–>x = x;
        p–>y = y;
        p–>dxDiag = dxDiag;
        p–>dyDiag = dyDiag;
        p–>dxAxial = dxAxial;
        p–>dyAxial = dyAxial;
        p–>decVar = decVar;
        p–>incDiag = incDiag;
        p–>incAxial = incAxial;
}
flushLines( );                          /*  Send buffer to PP for rendering   */
/*  The task now terminates itself by returning.   */
```

Figure 14. MP C Program Line Structure Sample

SUMMARY

The data presented demonstrates the efficiency of the ’C8x PP.[5] This device can generate in excess of one million ten-pixel colored vectors per second.

REFERENCES

1. *TMS34010 Application Guide*, Texas Instruments, 1991, literature number SPV007A, pp. 239–254.
2. *TMS320C8x Multitasking Executive User’s Guide*, Texas Instruments, 1995, literature number SPRU112.
3. *Fundamentals of Interactive Computer Graphics*, J.D. Foley, A. van Dam, Addison-Wesley Publishing Co., 1983, pp. 432–439.
4. *Algorithm for Computer Control of a Digital Plotter*, IBM System Journal, Vol. 4, No. 1, 1965, pp. 25–30.
5. *TMS320C8x Parallel Processor User’s Guide*, Texas Instruments, 1995, literature number SPRU110.
6. *TMS320C8x Transfer Controller User’s Guide*, Texas Instruments, 1995, literature number SPRU105.
7. *TMS320C8x Code Generation Tools User’s Guide*, Texas Instruments, 1995, literature number SPRU108.

Draw Colored Trapezoids Command

Syd Poland
Digital Signal Processing Solutions

Printed on Recycled Paper

INTRODUCTION

This document describes the operation of the _PPCMD_COLOR_TRAPEZOIDS command as it is used on the PP core of a single-PP TMS320C8x. The source code for the PP Draw Colored Trapezoids command is the *colotrap.s* subroutine. (The reader can examine the *colotrap.s* subroutine by obtaining the 'C8x tools CD-ROM available through Texas Instruments. The *colotrap.s* code is located in the graphics subdirectory.) The following topics are discussed:

- Function Definitions
- Conventions and Structures
- Calling Sequence
- Application Notes
- Performance Estimates
- Notes for Single Tall Trapezoids

FUNCTION DEFINITIONS

The_PPCMD_COLOR_TRAPEZOIDS command draws a suite of colored trapezoids. Each trapezoid can have a color that is different from that of any other trapezoid.

Double buffering is used to overlap trapezoid calculation in the PP's core with input/output (I/O) in the transfer controller (TC). Each trapezoid uses 16-bit pixels and is output as an unconditional write – that is, no transparency, plane masking, blending, or anti-aliasing is applied. This process occurs primarily on PP0 of the 'C8x, and results in the use of the two extra PP0 data RAM banks by _PPCMD_COLOR_TRAPEZOIDS. The master processor (MP) performs any clipping needed to fit trapezoids into a display window before _PPCMD_COLOR_TRAPEZOIDS is called. The MP also ensures that the maximum trapezoid height (248 lines) and the maximum number of trapezoids per call (83) are not exceeded. These processes provide the user with the capability to draw any number of trapezoids at various heights. A sample trapezoid within a window is illustrated in Figure 15.

Note the following statements/assumptions:

- The MP must pre-clip the trapezoid data to fit into the display window.
- The MP must divide individual trapezoids into shorter heights to ensure that the maximum height of 248 lines per trapezoid is not exceeded.
- The MP must not exceed 83 trapezoids for each call to _PPCMD_COLOR_TRAPEZOIDS.
- External memory is big endian.
- Horizontal lines are one pixel thick.
- Pixels are 16-bits and are aligned on a two-byte address.
- The PP can require the two extra data RAMs found on PP0 – the MP must assign the COLOR_TRAPEZOID argument lists in all cases.
- The PP does not preserve registers upon returning from a call to _PPCMD_COLOR_TRAPEZOIDS (with the exception of the stack pointer).
- The PP is dedicated to _PPCMD_COLOR_TRAPEZOIDS until this function terminates.

- The screen display originates in the upper lefthand corner: (X,Y) = (0,0).
- X increases horizontally – that is – from left to right, across the screen.
- Y increases vertically – that is – from top to bottom, across the screen.
- The origin of the visible screen, which is an even byte address in external memory, is in the graphics context.

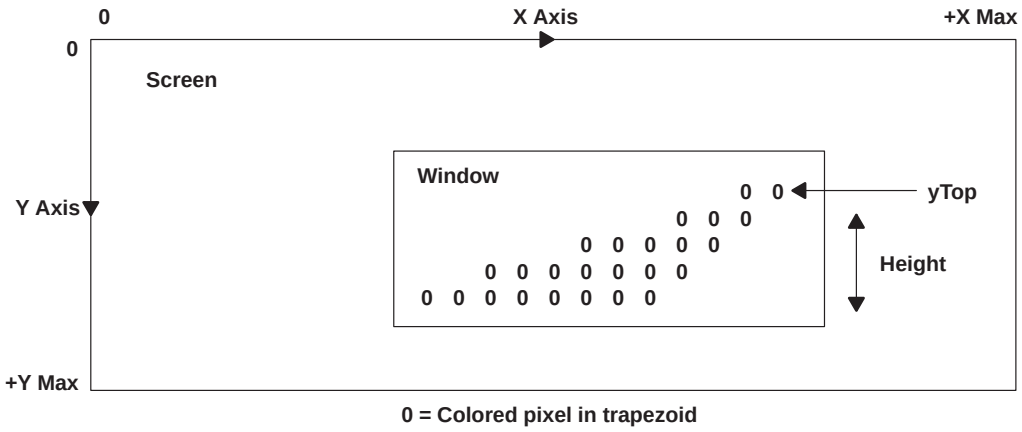

Figure 15. Trapezoid Fill in Window on Screen

- *Pitch*, an even byte-address increment, is the number of bytes in a horizontal screen row and is in the graphics context.
- Halfword 16-bit integers are assumed to be signed unless otherwise noted (for example, *yTop*, height).
- *yTop* ≥ 0 and height increases in the positive Y direction.
- *height* > 0
- *xL* ≤ *xR*
- Fixed values are assumed to be 32 bits long. The binary point is located between bits 15 and 16; the most significant (MS) 16 bits represent the integer, the least significant (LS) 16 bits contain the fraction for *xL, dxL, xR, dxR*.
- *Color* is the lower 16 bits in a 32-bit word.
- PP0 is the expected processor on the single-PP ’C8x; however, if the ’C8x DSP with four PPs is used, then PP1, PP2, or PP3 can be used. Argument lists for PP3 must be assigned by the MP as valid on-chip addresses!
- Because two data RAM banks from another PP could be required on a four-PP ’C8x system, precautions must be taken to ensure that this other PP does not use these two data RAM banks. The safest rule is for the other PP to remain idle.
- “Reserved” means a value can be zero or non-zero. The applications program should make no assumptions concerning the value in the reserved field.
- Trapezoids are drawn in the order supplied.

- _PPCMD_COLOR_TRAPEZOIDS exits without waiting for the TC to complete the last trapezoid fill request. If called immediately after _PPCMD_COLOR_TRAPEZOIDS exits, the PP should wait for TC completion before using its data RAMs.

CONVENTIONS AND STRUCTURES

The MP generates a list of trapezoids to be drawn (see Figure 16). The MP ensures that the number of trapezoids is ≤ 83, and the height of each trapezoid is ≤ 248 horizontal lines. The assumption is that the MP program pre-clips any trapezoid to the desired window before the colored trapezoid fill algorithm is called. If any trapezoid is too large, the MP subdivides that trapezoid into smaller trapezoid segments. pTzds is a pointer to the colored trapezoid argument list.

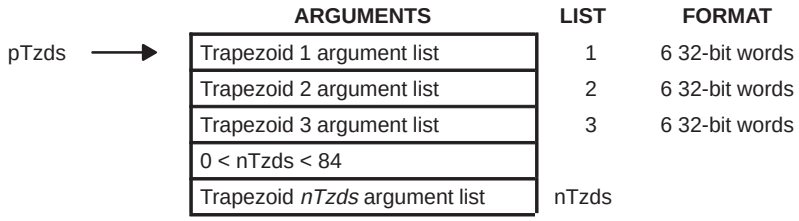

Figure 16. Argument List for Colored Trapezoids

Colored Trapezoid Argument List

The structure of each colored trapezoid request as generated by the MP is shown in Figure 17.

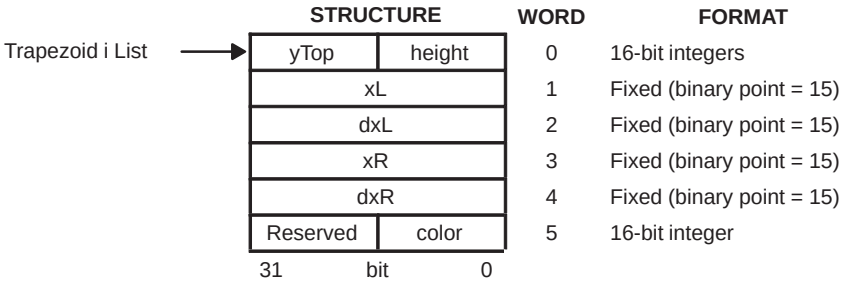

Figure 17. Argument List for an Individual Colored Trapezoid

Elements of the argument list structure are shown in Table 9.

Table 9. Structure of an Argument List for an Individual Colored Trapezoid

yTop	16-bit value for the Y-coordinate at the top of the trapezoid ≥ 0
height	Number of horizontal lines in this trapezoid (0 < height ≤ 248)
xL	Initial X left coordinate for this trapezoid (fixed 32-bit value: MS halfword is integer, LS halfword is fraction)
dxL	Δ X left coordinate for this trapezoid (fixed 32-bit value: MS halfword is integer, LS halfword is fraction)
xR	Initial X right coordinate for this trapezoid (fixed 32-bit value: MS halfword is integer, LS halfword is fraction)
dxR	Δ X right coordinate for this trapezoid (fixed 32-bit value: MS halfword is integer, LS halfword is fraction)
color	16-bit color value for this colored trapezoid
Reserved	16 bits, not used

Command List

Once the trapezoid arguments are generated, the MP passes a command structure to the PP (see Figure 18). The standard pointer to this command list is *arg1*.

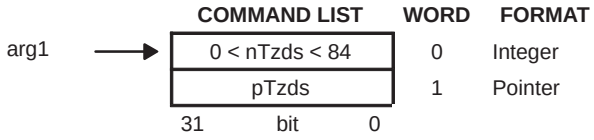

Figure 18. Argument List for Colored Trapezoid Call

Elements of the command list structure are shown in Table 10.

Table 10. Structure of an Argument List for Colored Trapezoid Call

nTzds	Number of trapezoids in this list (0 < nTzds < 84)
pTzds	Pointer to the first trapezoid's argument list (see Figure 16)

Offset Guide Table

The fill trapezoid algorithm is employed to generate the pixel locations of the requested horizontal line segments within the trapezoid. The TC packet request form used in this document is the variable-patch offset-guided tour, which requires an offset-guide table as shown in Figure 19.

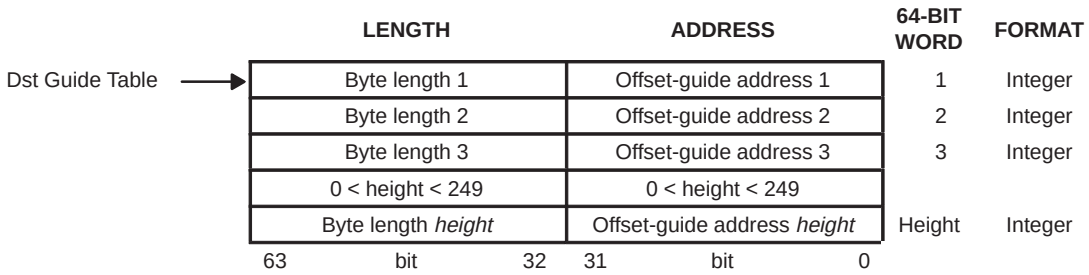

Figure 19. Variable-Patch Offset Guide Table (Big Endian)

The even-byte starting addresses of the pixels in each horizontal line segment in the colored trapezoid are formed as follows:

```
pixel address 1 = Dst start address + Offset-guide address 1
pixel address 2 = Dst start address + Offset-guide address 2
pixel address 3 = Dst start address + Offset-guide address 3
•
•
•
pixel address height = Dst start address + Offset-guide address height
```

Transfer Packet Request

Once the guide table has been generated, a transfer packet request is issued to the TC. The request format is illustrated in Figure 20. *tPT* is the pointer to the packet request parameter list.

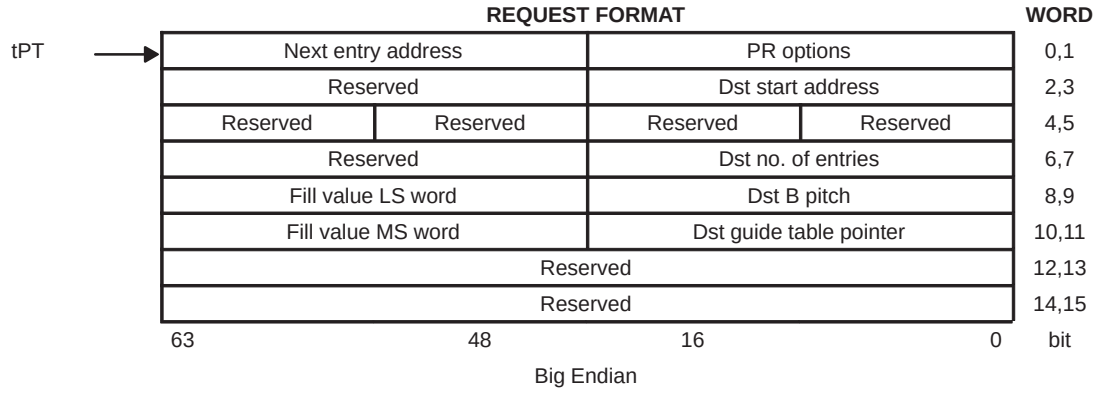

Figure 20. Fill-With-Value to Variable-Patch Offset-Guided Tour Packet Request

The transfer packet request contents are shown in Table 11.

Table 11. Structure of a Fill-With-Value to Variable-Patch Offset-Guided Tour Packet Request

Next entry address	Address of the next packet request
PR options	Options for this packet request (fill-with-value to variable-patch offset-guided tour, stop when this packet request is done)
Dst start address	Even-byte address of screen's origin in external memory
Dst no. of entries	Destination number of entries ($0 < \text{height} \leq 248$) in guide table
Fill value LS word	Two copies of this line's 16-bit color value
Dst B pitch	Destination B pitch is not required (as Dst B count = 0)
Fill value MS word	Two copies of this line's 16-bit color value
Dst guide table pointer	Pointer to the offset-guide table for the horizontal line segments in this trapezoid (see Figure 19)
Reserved	128 bits, not used

Memory Allocation

The target system of this implementation is the single-PP ’C8x. Line argument lists are double-buffered using PP0 data RAM 3 and PP0 data RAM 4 (see top portion of Figure 21). Packet requests and offset-guide tables are also double-buffered using PP0 data RAM0 and PP0 data RAM1 (see middle portion of Figure 21). The PP parameter RAM is used to issue the packet request to the TC and also to provide transfer vector addresses for PP interrupts (see bottom-right portion of Figure 21). PP0 data RAM 2 contains the graphics context (see bottom-left of Figure 21) as pointed to by the graphics context (GC) pointer.

The graphics context variables accessed (see *graphics.i*) are shown in Table 12.

Table 12. Graphics Context Variables

GC.SCRNADRS	32-bit integer, even-byte base address of the visible screen’s origin in external memory. This becomes Dst Start Address in the packet request (see Figure 6).
GC.SCRNPITCH	32-bit integer, even number of external memory bytes in each horizontal screen row (pitch)

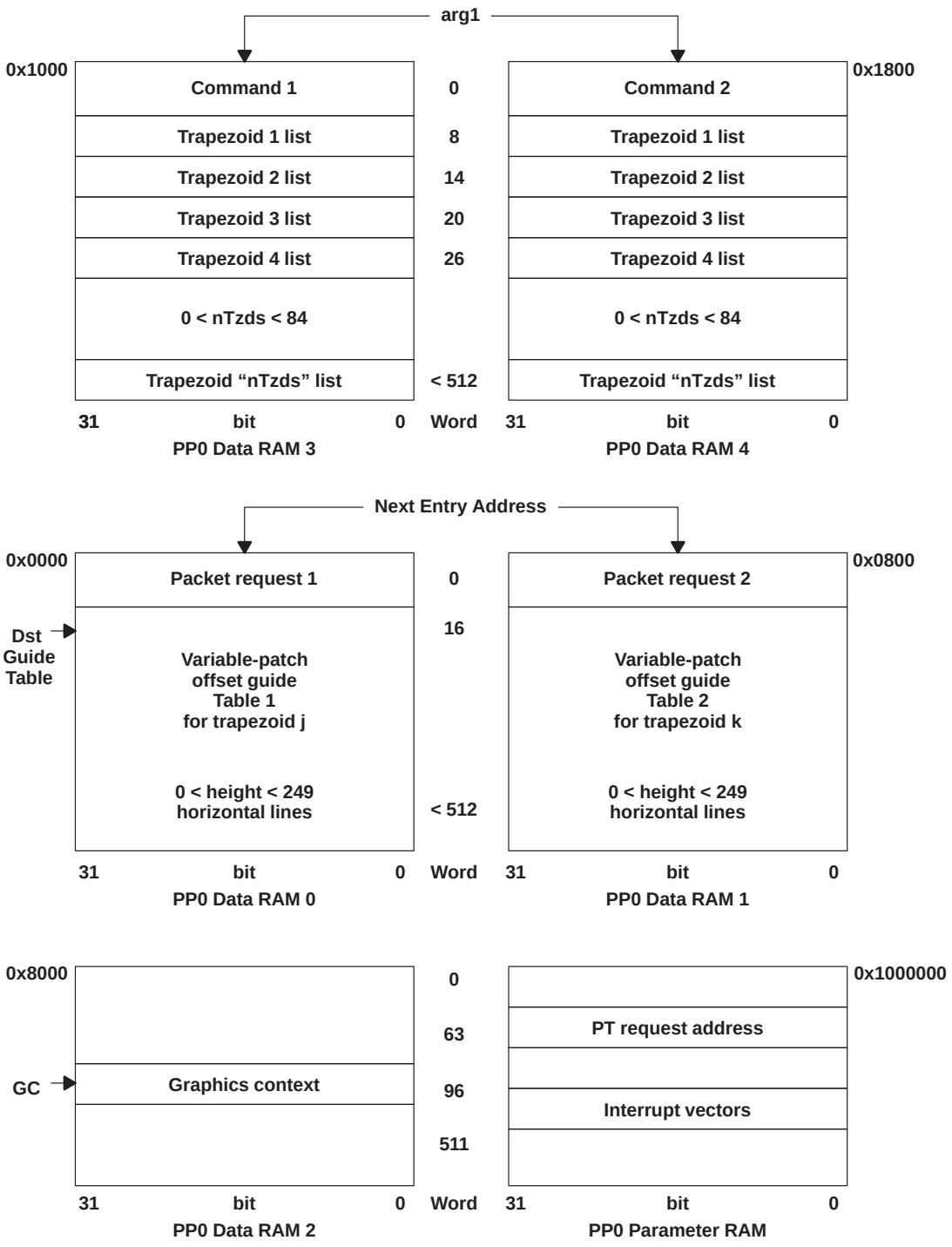

Figure 21. Memory Allocation

CALLING SEQUENCE

_PPCMD_COLOR_TRAPEZOIDS is called in accordance with the PP command structure as initiated by the MP. The sequence of events could include the following:

MP events

- Obtains message and command buffers
- Builds message and command buffers
- Performs pre-clipping and trapezoid partitioning as required for argument lists
- Initiates a message to the target PP

PP events

- Responds to the message interrupt
- Decodes a message and calls the application program (sets pointer to *arg1* and begins execution of _PPCMD_COLOR_TRAPEZOIDS)
- Upon entry, target PP spins until the TC is idle (for the local PP)
- Generates each trapezoid and issues packet requests to the TC
- Uses double buffering to overlap PP computations with TC output
- Returns, signals the MP when the application is done (if required)

Refer to the *TMS320C8x Multitasking Executive User's Guide* for a complete treatise on this subject.(1) The PP command buffer is shown in Figure 22.

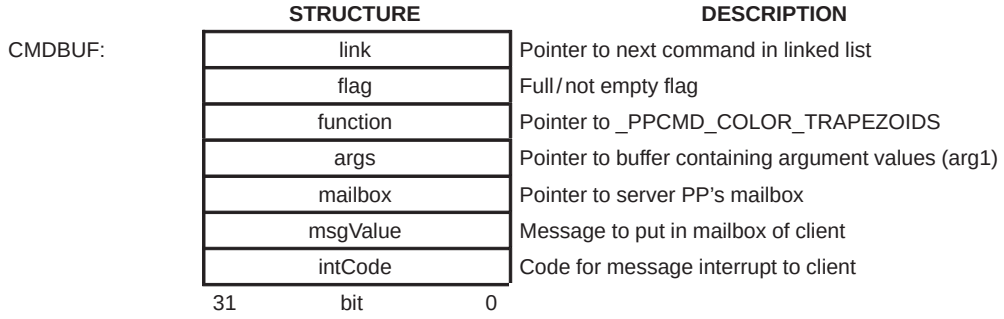

Figure 22. PP Command Buffer

Call in PP Assembly Language Versus Call in C

The colored trapezoid fill algorithm can be called from PP assembly code after all structures are established, as shown in Figure 23. The standard C subroutine call (Figure 23, lower-case label) with the pointer in *d1* continues to the upper-case label (assembly language alternate subroutine call) by moving *d1* to *a9*.

STANDARD ASSEMBLY EQUIVALENT OF A PP C CALL	PP ASSEMBLY LANGUAGE ALTERNATE
`.global _ppcmd_color_trapezoids`	`.global _PPCMD_COLOR_TRAPEZOIDS`
`call = _ppcmd_color_trapezoids`	`call = _PPCMD_COLOR_TRAPEZOIDS`
`d1 = arg1 ; see Figure 18`	`a9 = arg1 ; see Figure 18`
`nop ; delay slot 2`	`nop ; delay slot 2`
`Return:---; call returns here`	`Return:--- ; call returns here`

Figure 23. PP Colored Trapezoid Call

APPLICATION NOTES

This section discusses the following topics:

- Double Buffering
- Convert (X,Y) Coordinates to Byte Offsets
- *xL* and *xR* Computation
- Overlapped Processing
- Future Options

Double Buffering

The double buffering of argument lists and offset-guide tables provides the opportunity to overlap MP and PP computations with TC I/O. Upon entry, the PP spins until there is no TC activity for the PP. Thereafter, the PP calculates the trapezoid offset-guide table values and issues a TC request with *stop-on-complete* enabled. If the PP completes its next trapezoid computation before the TC is done, the next TC request forces a packet-request busy-interrupt, thereby causing the PP to spin until the TC has completed the earlier packet request. The current packet request then begins, and the PP continues to another trapezoid.

Convert (X,Y) Coordinates to Byte Offsets

The (X,Y) coordinates are converted to even-byte offsets relative to the screen origin as:

`offset = Y * pitch + X * 2 bytes per pixel`

where *pitch* is the even number of bytes in each horizontal screen row.

xL and xR Computation

The left and right X coordinates are updated as follows:

- $xL = xL + \Delta xL$
- $xR = xR + \Delta xR$
- *xLint = integer (xL)*
- *xRint = integer (xR)*
- *no. pixels = xRint – xLint*
- *no. bytes = 2 * (xRint – xLint)*
- *Offset = Y * Pitch + 2 * xLint*

Overlapped Processing

The preferred embodiment of colored trapezoid draw includes (see Figure 24):

- MP window pre-clipping and segmentation of tall trapezoids
- PP development of the trapezoid's pixel addresses for each horizontal row
- TC writing of the colored pixel values to external RAM

These commands can be subdivided further to illustrate individual trapezoids (see Figure 25). The MP's Command **K** issues all nTzds as a single command. The PP's response (Command **J**) is to generate a trapezoid and issue a TC request until all trapezoids have been rendered. In general:

$\texttt{MP Trapezoid}_K\ \texttt{(M) time} \neq \texttt{PP Trapezoid}_J\ \texttt{(L) time} \neq \texttt{TC Trapezoid}_J\ \texttt{(L-1) time}$

Performance is dominated by the largest trapezoid time value of the three processors – that is, the MP, PP, and TC.

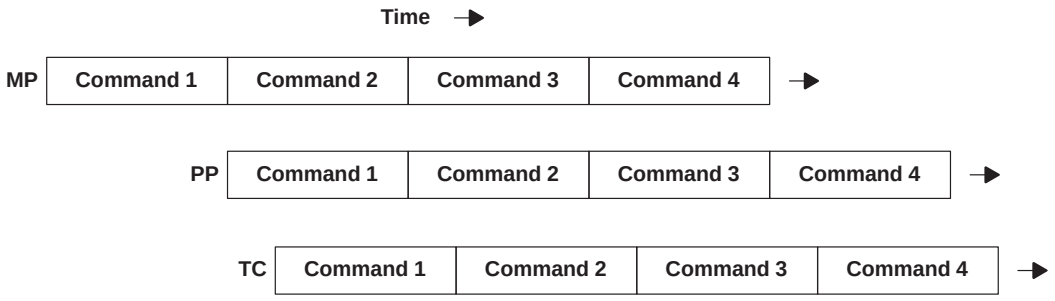

Figure 24. MP, PP, TC Overlapped Processing

The performance is dominated by the largest trapezoid time value of the three processors – the MP, PP, and TC.

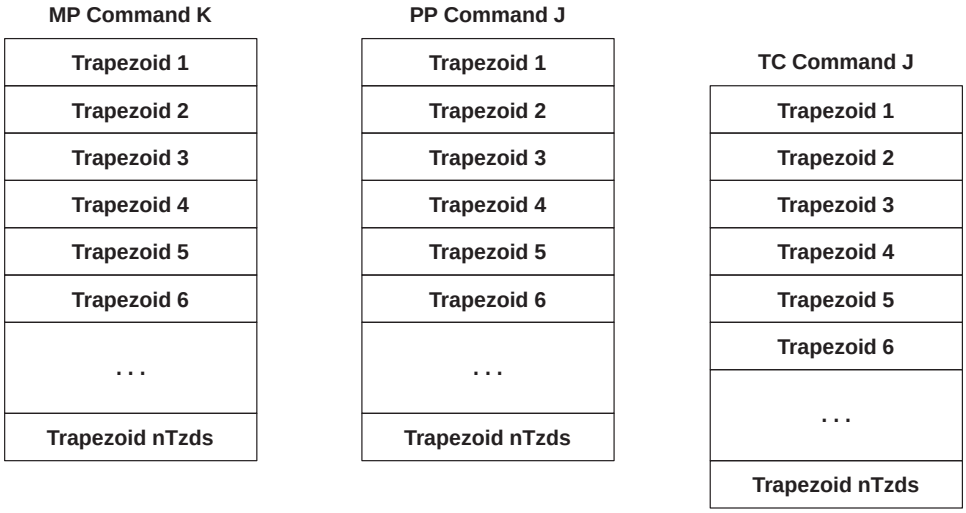

Figure 25. Three Processor Overlapped Times

Future Options

_PPCMD_DRAW_COLORED_TRAPEZOIDS can be modified to accept 8-bit and 32-bit pixels along with the 16-bit pixels already implemented. Register dOne and register dTwo values are shown in Table 13.

Table 13. Pixel Options (8-, 16-, and 32-Bit)

d REGISTER	8-BIT PIXEL	16-BIT PIXEL	32-BIT PIXEL
dOne	0	1	2
dTwo	1	2	4

PERFORMANCE ESTIMATES

Performance estimates assume no contention for on-chip or off-chip RAM access and that there is no TC round-robin contention. The following must be considered:

- MP computational time
- PP computational time
- TC I/O time including external memory organization
- Throughput as determined by the maximum of (MP time, PP time, TC time)
- Estimated time for 50 100-pixel trapezoids that are ten rows high

MP Trapezoid Clipping and Segmentation

The MP must clip a trapezoid to the desired window and also must segment trapezoids so that the maximum height of a trapezoid is not exceeded. Trapezoid values are then placed in the colored trapezoid argument list structure (see Figure 17). Once the set of trapezoids has been entered, the MP issues a message to the PP for the colored trapezoid fill command. MP computational time is determined by the program(s) on the MP and another specific, application-dependent value, which is not discussed in this document.

PP Color Trapezoid Fill Estimates

PP trapezoid fill estimates are as follows:

- Overhead (excluding instruction cache) = 23 cycles if the TC is idle (more if the TC is busy)
- Time for each trapezoid fill with **height** lines = 20 + 4 * **height** cycles
- Time for *constant trapezoid height* nTzds trapezoids = (20 + 4 * **height**) * nTzds cycles
- Time for *variable trapezoid heights* nTzds trapezoids = $\sum$(20 + 4 * $\mathbf{height_k}$) cycles for k = 1, 2, ..., nTzds
- Total time with *constant trapezoid height* = 23 + (20 + 4 * **height**) * nTzds cycles
- Total time with *variable trapezoid heights* = 23 + $\sum$(20 + 4 * $\mathbf{height_k}$) cycles for k = 1, 2, ..., nTzds

TC Trapezoid Fill Estimates

Assume that memory organization uses two-cycle DRAMs with n rows per page (n = 1, 2, 4) and that the page fault select time is six additional cycles. The orientation of the trapezoid determines how many *write* page faults occur. Assume **numDots** is the total number of pixels in this trapezoid. The estimated number of cycles required for each trapezoid is listed below. (NOTE: "/" means ceiling function; for example, 10/8 = 2.)

- For page faults: 6 * height / n where n = rows per page = 1, 2, 4
- To write data: 2 * height * **numDots** / height /4
- Total TC cycles: 15 + 6 * height / n + 2 * height * **numDots** / height /4

Assume that **numDots** = 100 and height = 10. TC time estimates are as follows:

- For zero page faults: 15 + 2 * 100 / 4 = 65 cycles (horizontal lines are all within the same page)
- For **numDots** in four rows per page: 15 + 6 * 10 / 4 + 2 * 10 * 10 / 4 = 93 cycles
- For **numDots** in two rows per page: 15 + 6 * 10 / 2 + 2 * 10 * 10 / 2 = 145 cycles
- For **numDots** in one row per page: 15 + 6 * 10 / 1 + 2 * 10 * 10 / 1 = 275 cycles

Trapezoid Fill Throughput of Combined (PP, TC)

The total throughput limit is max(PP time, TC time); MP time is not included in this document.

```
Time = max[ 23 + (20 + 4 * height) * nTzds, (Total TC) ] cycles
```

assuming constant trapezoid height and horizontal line length, two-cycle DRAMs with two rows per page, and no instruction cache misses.

Estimated Time for 50 100-Pixel, Ten-Row Trapezoids

The definition of a trapezoid (as used here) is 100 16-bit pixels per trapezoid – that is, height is ten rows. Estimated times for filling 50 trapezoids are as follows:

- PP time = 23 + (20 + 4 * 10) * 50 = 3023 cycles (excludes instruction cache)
- TC time = (93) * 50 = 4650 cycles (four rows per page)
- TC time = (145) * 50 = 7250 cycles (two rows per page)
- TC time = (275) * 50 = 13750 cycles (one row per page)
- (PP, TC) time = max (3023, 4650) = 4650 cycles (four rows per page)
- (PP, TC) time = max (3023, 7250) = 7250 cycles (two rows per page)
- (PP, TC) time = max (3023, 13750) = 13750 cycles (one row per page)

See Figure 26 for performance estimates that assume the use of a 50-MHz part.

			TC, Two-Cycle DRAM n Rows/Page		
Item	**Units**	**PP**	**n = 4**	**n = 2**	**n = 1**
PP, no i-cache	**50 tzd cycles**	**3023**			
TC	**50 tzd cycles**		**4650**	**7250**	**13750**
MAX (PP, TC)	**50 tzd cycles**		**4650**	**7250**	**13750**
PP at 50 MHz	**Million tzd/sec**	**0.83**			
TC at 50 MHz	**Million tzd/sec**		**0.54**	**0.34**	**0.18**

- **tzd/sec – Trapezoids per second**
- **tzd – Trapezoid is 10 rows high with 100 16-bit pixels total**

Figure 26. PP and TC Performance for 50 Trapezoids

NOTES FOR SINGLE TALL TRAPEZOIDS

Tall trapezoids can be written by having each call generate only *one* output trapezoid. This is accomplished by chaining more of the contiguous PP RAMs together for the longer offset-guide tables. *Extreme care must be used when employing this approach!* The packet transfer request (PTR) parameter list in Figure 20 requires 16 words in PP0, data RAM 0; remaining memory is available for use by the first 248 offset-guide values in Figure 19 (the TC writes color value to output trapezoids). Each additional PP data RAM provides 256 extra offset-guide values, as shown in Figure 27.

HEXADECIMAL ADDRESS		TOTAL NUMBER OF HORIZONTAL LINES (= HEIGHT)	
0x0000	PTR + 248 offsets	248	PP0 Data RAM 0
0x0800	+ 256 offsets	504	PP0 Data RAM 1
0x1000	+ 256 offsets	760	PP0 Data RAM 3 = PP1 Data RAM 0
0x1800	+ 256 offsets	1016	PP0 Data RAM 4 = PP1 Data RAM 1

Figure 27. Single Tall Trapezoid Memory

SUMMARY

The data presented demonstrates the efficiency of the 'C8x PP.[3] This device can generate in excess of one-half million colored trapezoids per second.

REFERENCES

1. *TMS320C8x Multitasking Executive User's Guide*, Texas Instruments, 1995, literature number SPRU112.
2. J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., 1983, pp. 91–99.
3. *TMS320C8x Parallel Processor User's Guide*, Texas Instruments, 1995, literature number SPRU110.
4. *TMS320C8x Transfer Controller User's Guide*, Texas Instruments, 1995, literature number SPRU105.
5. *TMS320C8x Code Generation Tools User's Guide*, Texas Instruments, 1995, literature number SPRU108.

Parallel Processor Integer and Floating-Point Math

Syd Poland
Texas Instruments Incorporated

Printed on Recycled Paper

INTRODUCTION

This application note describes PP math subroutines that were designed to be called by C programs. These subroutines use the short, 32-bit IEEE 754 standard FP format, which is in accordance with the IEEE 754 hardware implementation found on the TMS320C8x master processor (MP). The math routines are written in assembly language for use by the PP and are independent of which PP is used. (The reader can examine individual math functions by obtaining the 'C8x tools CD-ROM available through Texas Instruments. These functions are grouped within a single source file, which is located in the PP runtime support source subdirectory.)

Integer multiply, divide, and remainder are included for both signed and unsigned 32-bit arguments.

FORMAT

The format of the short IEEE 754 standard-compliant FP numbers is shown in Table 14. The 24-bit mantissa provides seven-plus decimal digits. The dynamic numeric range provided by the exponent includes:

- $2^{+127} = 10^{+38}$ when the exponent is 254
- $2^{-126} = 10^{-38}$ when the exponent is 1

The full value of a non-zero, short FP number is:

$$(-1)^{sign} \times 2^{(exponent-127)} \times 1.mantissa$$

Table 14. Short FP Data Format

31	30 23	22 0
Sign	Exponent	Fractional absolute mantissa

NOTES: 7. Sign = 0 for positive numbers, 1 for negative numbers.
8. Exponent has a bias of 127 (the exponent for 1.0).
9. Mantissa has an implied 1 (bit 23) so that the full mantissa is 1.mantissa when the exponent is non-zero.

SPECIAL CASES

The FP representation has unique cases for the following:

- Zero
- Underflow
- Overflow
- Denormal
- Signaling Not-A-Number
- Quiet Not-A-Number

These cases are described in subsequent sections.

Zero

Zero always has the entire word = 0.

Underflow

Underflow occurs when exponent <1; the answer returned is 0.

Overflow

Overflow is represented by one of the following:

- Positive infinity = 0x7fffffff (S = 0, exponent = 255, mantissa = 1s)
- Negative infinity = 0xffffffff (S = 1, exponent = 255, mantissa = 1s)

Denormal

Denormal FP numbers are never generated. If the exponent is zero, the mantissa and sign also are zero.

Signaling Not-A-Number

Signaling Not-A-Number (SNaN) is treated as infinity. SNaN has:

- Sign = 0 or 1
- Exponent = 255
- Mantissa bit 22 = 0, with remaining bits as *don't care* (same as MP).

Quiet Not-A-Number

Quiet-Not-A-Number (QNaN) is treated as infinity. QNaN has:

- Sign = 0 or 1
- Exponent = 255
- Mantissa bit 22 = 1, with remaining bits as *don't care* (same as MP).

ROUNDING

The *rounding* mode adopted here is round-to-zero. No provisions are made for:

- Round-to-nearest (half-adjust)
- Round-to-positive-infinity
- Round-to-negative-infinity

CONVENTIONS FOR PASSING ARGUMENTS

Conventions for passing arguments are described in subsequent sections.

Input Integers

Input integer operands are passed using the data registers as follows:

- D1 is the input 32-bit integer operand 1.
- D2 is the input 32-bit integer operand 2 (if needed).

Input FP Numbers

Input FP operands are passed using the data registers as follows:

- D1 is the input short FP operand 1.
- D2 is the input short FP operand 2 (if needed).

Output Numbers

Output is always in data register D5, regardless of output format (32-bit integer or short FP).

TEST OR COMPARE

Test or compare operations return control to the calling routine with status bits set as follows:

- Z = 1 if the comparison yields equality (that is, Op1 = Op2).
- N = 1 if the comparison produces a negative quantity (that is, Op1 < Op2).
- C is the normal ALU carry-out value = Carry-Out$_{31}$.
- V is the normal ALU overflow value as V = Carry-Out$_{31}$ XOR Carry-In$_{31}$ (V is set for infinity).

TEMPORARY REGISTERS

All temporary registers used are restored upon exit, with the exception of *Loop Counter 2* which is used in FP divide, integer divide, or remainder operations (see the next section).

DIVIDE OR REMAINDER DESTROYS LOOP COUNTER 2

FP divide, integer divide, or remainder use *Loop Counter 2* and do *not* restore its original contents.

INTEGER MULTIPLY

Signed or unsigned integer (32 bits x 32 bits = lower 32 bits of 64 bits) product is returned. Integer multiply returns Ans = Op1 * Op2 . C generates this code in-line (that is, without calling a function).

INTEGER DIVIDE

Signed or unsigned integer divide (32 bits / 32 bits = 32-bit quotient) is returned. If the quotient does not fit, the maximum value that can be represented is returned. *Loop Counter 2* information is not restored. Note the following:

- I_DIV returns signed quotient as Ans = Op1 / Op2.
  - 0x7fffffff = signed positive maximum.
  - 0x80000000 = signed negative maximum.
- U_DIV returns unsigned quotient as Ans = Op1 / Op2.
  - 0xffffffff = unsigned maximum.

INTEGER REMAINDER OR MODULO

Signed or unsigned integer remainder from divide (32 bits / 32 bits = 32-bit remainder) is returned. The remainder, if non-zero, has the input sign of the numerator. The magnitude of the remainder is less than the magnitude of the divisor. If the quotient does not fit, the maximum value that can be represented is returned. *Loop Counter 2* information is not restored. Note the following:

- I_MOD returns signed remainder as Ans = Op1 modulo Op2.
  - 0x7fffffff = signed positive maximum.
  - 0x80000000 = signed negative maximum.
- U_MOD returns unsigned remainder as Ans = Op1 modulo Op2.
  - 0xffffffff = unsigned maximum.

INTEGER TO FP

Signed or unsigned 32-bit integers are converted to short FP numbers.

- F_ITOF returns short FP Ans = Op1 (32-bit signed integer).
- F_UTOF returns short FP Ans = Op1 (32-bit unsigned integer).

FP TO INTEGER

Short FP numbers are converted to signed or unsigned 32-bit words. If an answer does not fit, the maximum hexadecimal value that can be represented is returned (see Table 15).

FP numbers are returned as follows:

- F_FTOI returns 32-bit signed word as Ans = Op1 (short FP).
- F_FTOU returns 32-bit unsigned word as Ans = Op1 (short FP).

Table 15. Maximum Integer Returned in Float-to-Integer Conversions

MAXIMUM VALUES	WORD 32-BITS
Signed positive (S)	7 f f f f f f f
Signed negative (S)	8 0 0 0 0 0 0 0
Unsigned (U)	f f f f f f f f

FP ADD

The sum of the two input FP numbers is generated. Underflow returns zero, overflow returns + or – infinity. Status register (Z) = 1 if the result is zero, and status register (N) = 1 for negative numbers. These register settings facilitate the FP test/compare operation, if required.

F_ADD returns the short FP sum Ans = Op1 + Op2 (short FP).

FP SUBTRACT

The difference of the two input FP numbers is generated. Underflow returns zero, overflow returns + or – infinity. Status register (Z) = 1 if the result is zero, and status register (N) = 1 for negative numbers. These register settings facilitate the FP test/compare operation, if required.

F_SUB returns the short FP difference Ans = Op1 - Op2 (short FP).

FP MULTIPLY

The product of the two input FP numbers is generated. Underflow returns zero, overflow returns + or – infinity. Status register (Z) = 1 if the result is zero, and status register (N) = 1 for negative numbers.

F_MPY returns the short FP product Ans = Op1 * Op2 (short FP).

FP DIVIDE

The quotient of the two input FP numbers is generated. Underflow returns zero, overflow returns + or – infinity. If the divisor is 0, infinity is returned regardless of the value of the numerator. Status register (Z) = 1 if the result is zero, and status register (N) = 1 for negative numbers. *Loop Counter 2* information is *not* restored.

F_DIV returns the short FP quotient Ans = Op1 / Op2 (short FP).

FP INCREMENT OR DECREMENT BY 1.0

Overflow returns + or – infinity. Status register (Z) = 1 if the result is zero, and status register (N) = 1 for negative numbers. These register settings facilitate the FP test/compare operation, if required. Two different implementations include:

- The C compiler generates + or – 1.0 in operand 2 and calls FP add (method used here).
- A subroutine call overwrites operand 2 with + or – 1.0 and uses FP add.

F_INC returns the incremented value Ans = Op1 + 1.0 (short FP).

F_DEC returns the decremented value Ans = Op1 - 1.0 (short FP).

FP Test Versus 0.0

The following example shows the preferred implementation with the C compiler generating the in-line code.

```
OP1 = OP1 + 0              ; Determine if Op1 is +, 0, -
br = [cond] LABEL          ; Branch to LABEL if condition = True
```

Table 16 shows sample condition codes.

Table 16. Condition Codes for FP Test Op1 Versus 0.0

SYNTAX	CONDITION	TEST
[z]	Zero	Op1 = 0
[n]	Negative	Op1 < 0
[p]	Positive	Op1 > 0
[nz]	Non-zero	Op1 $\neq$ 0
[ge]	Non-negative	Op1 $\geq$ 0
[le]	Non-positive	Op1 $\leq$ 0

FP Comparison of Op1 Versus Op2

This operation compares two short FP numbers and returns a value in Ans, which sets the status register bits for conditional branching. This is similar to FP test where Op2 = 0.0. A code sample follows:

```
call   =  F_CMP           ; Call floating-point compare subroutine
OP1    =  argument1       ; In delay slot 1
Op2    =  argument2       ; In delay slot 2
br     =  [cond] LABEL    ; Branch to LABEL if condition = True
```

Table 17 shows sample condition codes.

Table 17. Condition Codes for FP Compare of Op1 Versus Op2

SYNTAX	CONDITION	TEST
[z]	Zero	Op1 = Op2
[n]	Negative	Op1 < Op2
[p]	Positive	Op1 > Op2
[nz]	Non-zero	Op1 $\neq$ Op2
[nn]	Non-negative	Op1 $\geq$ Op2
[le]	Non-positive	Op1 $\leq$ Op2

FP NEGATION

The following example shows the preferred implementation with the C compiler generating the in-line code.

```
OP1 = OP1 + 0                ; Determine if Op1 < 0
OP1 = [nz] Op1 ^ 1<<31       ; Reverse sign if Op1 is non-zero
```

PROGRAM SIZE AND TIMING ESTIMATES

The preferred loading of these PP subroutines begins on a cache boundary (that is, a multiple of 16 instructions). Clocks assume no cache misses. Table 18 shows some typical performance values.

Table 18. Performance of PP FP Math Routines

NAME	INSTRUCTIONS	CLOCKS†	MILLIONS/SEC. AT 50 MHZ
Integer Multiply	7	7	7.14
Unsigned Integer Divide	14	40	1.25
Unsigned Integer Remainder	14	40	1.25
Signed Integer Divide	15	42	1.19
Signed Integer Remainder	15	42	1.19
Unsigned Integer to Float	9	9	5.55
Signed Integer to Float	10	10	5.00
Float to Unsigned 32-Bit Integer	12	12	4.16
Float to Signed 32-Bit Integer	15	12	4.16
Floating Negation of Op1 (In-line)	2	2	25.00
Floating Test Versus 0.0 (In-line)	2	2	25.00
Floating Compare Op1 Versus Op2	6	6	8.33
Floating Inc/Dec by 1.0 (FADD)	39	14-41(38)	1.32
FP Add	39	14-41(38)	1.32
FP Subtract	41	16-43(40)	1.25
FP Multiply	38	10-32(31)	1.61
FP Divide	40	11-55(55)	0.91

† Parenthesized values denote typical execution.

IN-LINE FUNCTIONS

The following functions are in-line C code:

- FP test of Op1 versus 0 consists of two in-line instructions (see FP Test Versus 0.0 section).
- FP negation of Op1 consists of two in-line instructions (see FP Comparison of Op1 Versus Op2 section).
- FP increment/decrement by 1.0 uses F_ADD/F_SUB as generated by the C compiler.

COMPARISONS OF FP RATE ON PP, MP, AND SPARC-10

All comparisons are for short 32-bit FP values and assume no cache misses, a 50-MHz clock, and sequential execution of instructions. See the following ArcTAN example. Cramer's polynomial expansion depicts a register dependency example – that is, no pipelining is possible. The 'C8x pipeline assumes a suite of FP operations with no serial-register dependencies, that is, FP ADD output is not required for the next FP MPY. Table 19 shows clock comparison data.

Table 19. Clock Comparison of MP, PP, and SPARC™-10

NAME	'C8X - PP S/W	'C8X - MP SERIAL	'C8X - MP PIPELINE	SPARC-10 SERIAL
ADD	38	4	1	3
SUB	40	4	1	3
MPY	31	3	1	3
DIV	55	8-11	6-9	8-11~
INT-to-FP	10	3	1	3
FP-to-INT	12	3	1	3

Consider the following ArcTAN(X) where X is non-negative and Y = (X - 1) / (X + 1):

$$\text{ArcTAN}(X) = \text{Pi}/4 + C_1*Y + C_3*Y^3 + C_5*Y^5 + C_7*Y^7 + C_9*Y^9 + C_{11}*Y^{11} + C_{13}*Y^{13} + C_{15}*Y^{15}$$

Assuming 9 ADDs, 1 SUB, 9 MPYs, and 1 DIV, the clock totals for this ArcTAN evaluation are shown in Table 20.

Table 20. ArcTAN Comparison of MP, PP, and SPARC-10

NAME	'C8X - PP S/W	'C8X - MP SERIAL	'C8X - MP PIPELINE†	SPARC-10 SERIAL
ArcTAN instructions	716	78	28 (n/a)	68 clocks
Ratio/MP serial	9.18	1.0	0.36 (n/a)	0.87

† n/a - Not applicable for serial-register dependencies

OTHER FP ROUNDING

It may be possible to incorporate other types of rounding if there is a constant location that would contain the rounding value for all FP operations. This location can be FLT_ROUNDS and may contain the following values (as used on the MP):

- 0 for round to nearest
- 1 for round to zero
- 2 for round to positive infinity
- 3 for round to negative infinity

The cost for doing this is more instructions and, consequently, more execution time. This is not available in the current version.

FASTER VERSION OF FP ARITHMETIC (not implemented)

If one is willing to eliminate the tests for overflow and underflow, FP arithmetic can be implemented to test only for Divide_By_0 and to permit exponents to wrap for range violations. This improves the execution rate by approximately 30%. The following libraries can be used:

- Normal library with overflow/underflow tests as the default for safe operations
- Fast library that contains the faster arithmetic, which is referenced before the normal library functions are linked by the loader (fast library not included)

DOUBLE-PRECISION FP

These routines can be extended to include the 64-bit double-precision FP format but are not included in this library. Their performance would require more clocks for the extended precision. This could be a future addition to the PP integer and FP libraries.

SUMMARY

The data presented describes the functionality and usage of the 'C8x's PP integer and FP math subroutines. These routines efficiently generate the math values required by PP-resident C programs. (See Table 18 for detailed performance data.)

REFERENCES

1. IEEE 754-1985 Standard
2. *TMS320C8x Parallel Processor User's Guide*, Texas Instruments, 1995, literature number SPRU110.
3. *TMS320C8x Code Generation Tools User's Guide*, Texas Instruments, 1995, literature number SPRU108.
4. *TMS320C8x Master Processor User's Guide*, Texas Instruments, 1995, literature number SPRU109.

Index

G

I

M

O

P

R

S

T