PIXELFLOW™ RASTERIZER FUNCTIONAL DESCRIPTION

John Eyles
Steven Molnar

Department of Computer Science
University of North Carolina at Chapel Hill

Rev. 7.0
November 20, 1997

— CONFIDENTIAL —
Do not distribute outside UNC-CH CS Dept.

TABLE OF CONTENTS

I INTRODUCTION

PixelFlow™ is a special-purpose computer architecture designed for high-speed, high-quality image generation. The central part of each PixelFlow circuit board is the *Rasterizer*. It consists of an two Image Generation Controller chips (IGCs), an array of 32 Enhanced Memory Chips (EMCs), an array of 8 Texture ASICs (TASICs), and 32 synchronous DRAM (SDRAM) memories, as shown in Figure 1.

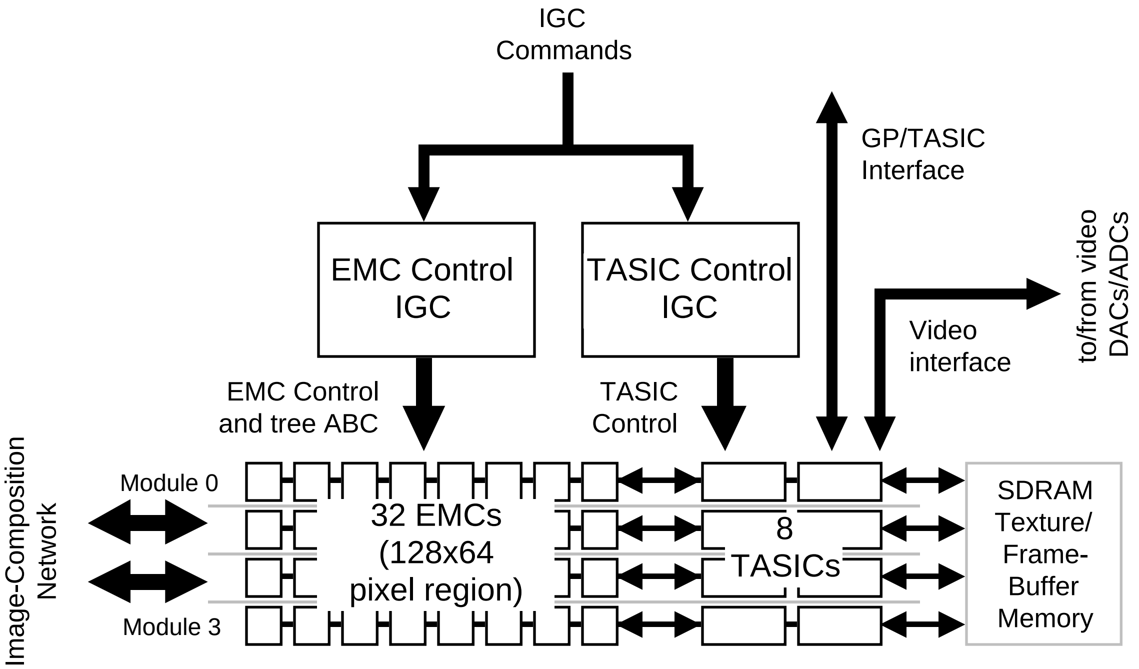


Figure 1: Block diagram of the PixelFlow rasterizer.

The EMCs contain an array of 8K (8192) processing elements (PEs). Each PE consists of a byte-wide ALU and 384 bytes of memory, operating at 100 MHz. The PEs operate in Single-Instruction/Multiple Data (SIMD) fashion. Each PE has an x,y address and is connected to a distributed Linear Expression Evaluator (LEE) which supplies the local value of the bi-linear expression $F(x, y) = Ax + By + C$ byte-serially at 100 MHz.

The 8K PEs in the Rasterizer can be mapped onto a region of the display screen (the *rasterizer region*) in several different ways:

1) 128 x 64 pixel region, with 1 PE per pixel
2) 64 x 32 pixel region, with 4 PEs per pixel

3) 32 x 32 pixel region, with 8 PEs per pixel

For option 1, each PE's *x*,*y* address is the screen position of the pixel with which that PE is associated.[1] For option 2, the PEs are divided into groups of 4, and each group is assigned to a pixel; the 4 PEs in each group are assigned to four samples in an anti-aliasing kernel. Since the *x*,*y* addresses are integers, pixel centers are defined to lie on an 8x8 *pixlet* grid, where F(*x*, *y*) is evaluated for pixlet *x*,*y* values. Thus, the region is 512 x 256 pixlets in size, and pixels are referenced to multiples of 8 on this pixlet grid. The 4 samples can be placed in a 16 x 16 pixlet box with origin at the pixel reference point. Option 3 is similar to option 2, except the region is 256 x 256 pixlets in size, and there are 8 samples in the anti-aliasing kernel.

In all 3 cases, the PEs operate on the F(*x*, *y*) data from the LEE and pixel-memory, storing their results back into pixel-memory.

The TASICs are CMOS datapath chips, which connect the EMCs to the GP bus (via the GNI chip—described elsewhere), to external SDRAM memory (for texture and image storage), and optionally, to video circuitry (on frame-buffer and frame-grabber boards).

The IGCs interpret instructions from the GP that are queued in their input FIFOs. Each is a microcoded sequencer which executes high-level instructions for the EMCs and TASICs, controlling their cycle-by-cycle operation.

Section II describes the rasterizer's hardware components. Sections III through VI describe its programming interface.

II RASTERIZER HARDWARE OVERVIEW

II.1 Rasterizer Input Interface

Before rasterization of a frame can begin, the Geometry Processor (GP) on each system board generates rendering commands and stores them in its local memory. The GP writes these commands to the rasterizer's 32-bit input interface, and each command is loaded into one of the IGCs.

Each IGC can process commands at peak rates of 100 Mword/second. Each IGC has programmable almost-full flags for its RFIFO and TFIFO, which are asserted when the FIFO can accept only a limited number of additional commands; these flags are or'ed together and go back to the GP.

[1] We sometimes refer to the PEs as pixel processors, but there is a one-to-one correspondence between PEs and pixels only for the 128x64 case; sometimes, in fact, the PEs may contain data totally unrelated to pixel values.

II.2 Image Generation Controllers

Each *Image Generation Controller* (IGC) is a custom chip with a sequencer and microcode store which controls the EMCs (in the case of the EMC Control IGC, or *EIGC*) or the TASICs (in the case of the TASIC Control IGC, or *TIGC*); the EIGC includes a serializer to convert integer or floating-point input coefficients into the fixed-point, byte-serial form required by the EMCs. The two IGCs can be thought of as a single logical entity which processes commands from the rasterizer input interface, controls and sequences the EMCs and TASICs, and synchronizes the rasterizer with the image-composition network. A set of semaphores interlocks operation of the Sequencers of the two IGCs, and processing of RFIFO and TFIFO commands. Figure 2 shows a block diagram of the IGCs.

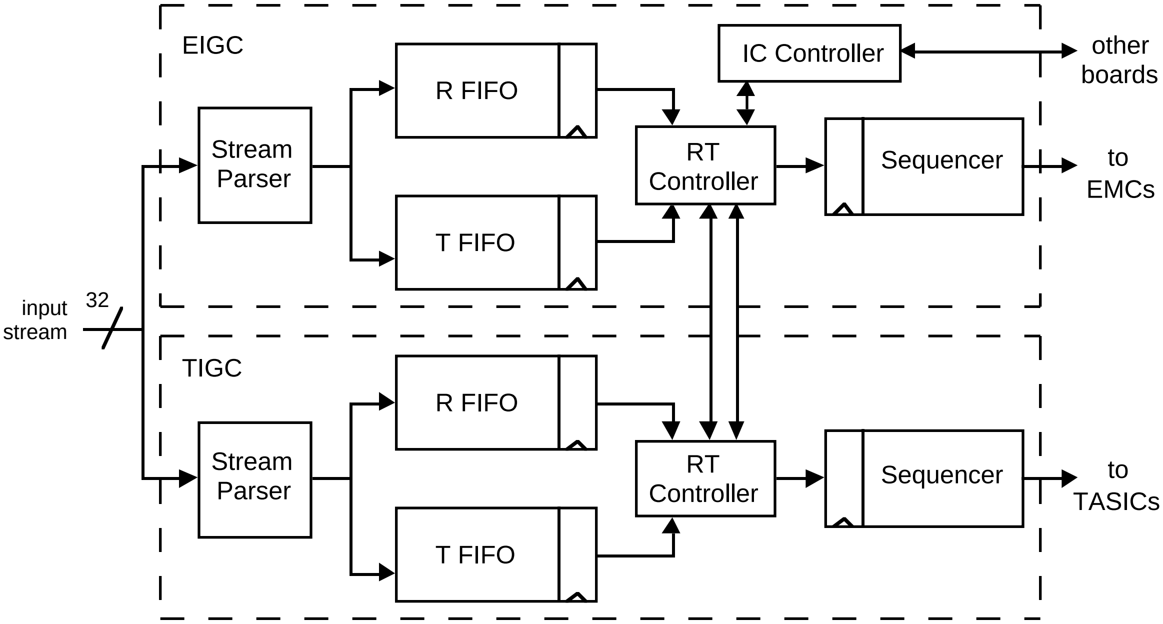


Figure 2: Block diagram of the IGCs.

Stream Parser. IGC Commands consist of an I-word opcode, and optional additional arguments, depending on the purpose of the instruction:

- **I-word.** Mandatory. Contains the instruction opcode and parameters for the microcode routine. The I-word is present in every command and is the first word in the command. The I-word also specifies which FIFO and which Sequencer the command is for.
- **P-word.** Optional. Contains additional parameters for the microcode routine.
- ***A*, *B*, and *C* coefficients.** Optional. Operands for the linear expression evaluator. C coefficient also used in some initialization commands.

The I-word and P-word are each 32-bit quantities. The A, B, and C coefficients may be either 32- or 64-bit integers or floating-point numbers. The RFIFOs contain 256-bit wide slots for each command; the TFIFOs contain 64-bit wide slots (TFIFO commands cannot use the A, B, and C operands). The physical input interface to the IGCs is 32-bits wide; a *stream parser* parses the input stream (rejecting commands for the other IGC) and building complete commands, which are loaded into a slot in the specified FIFO. The FIFOs, RT controllers, and Sequencers, handle each comand as a single unit, irrespective of its original format. The formats for IGC commands are described in more detail in Sections IV through VI below.

RFIFO and TFIFO. The rasterizer's task consists of two parts: (1) rasterization—the calculation (or shading) of pixel values, and (2) compositor setup—copying pixel data to the transfer buffer section of EMC pixel memory and controlling the compositor logic. Unfortunately, the two parts must be performed as asynchronous processes, since region transfers can occur at unpredictable times (whenever all system boards are ready to transfer the next region).

To be able to execute these tasks asynchronously, incoming IGC commands must be buffered into two streams: the *RFIFO* buffers rendering commands, and the *TFIFO* buffers transfer commands. Semaphores, described below, synchronize the operation of the two FIFOs.

The FIFOs are wide enough to hold an entire command in each entry. A bit in the I-word of each command determines whether the command is to be loaded into the RFIFO or TFIFO. The RFIFO can hold up to 128 commands and the TFIFO can hold up to 512 commands. Hardware keeps track of the number of commands stored in the FIFOs and asserts the status register bit *SRFullH* or *STFullH* if either FIFO reaches a programmable high-water mark. If the high-water mark is close to the FIFO size, then the FIFO can inadvertently be overflowed because there is a several instruction pipeline between the IGC inputs and the *Full* flags.

RT Controller. Each IGC contains an *RT Controller* which reads commands from the R- and T-FIFOs and writes them to the Sequencer. The RT Controller includes four semaphore counters; two can block the RFIFO, and two can block the TFIFO. Each semaphore blocks its FIFO if its value is zero and a P command is in the FIFO's read latch. One of the RFIFO-blocking semaphores can be V'ed by the appropriate TFIFO command on the same IGC, and one of the RFIFO-blocking semaphores can be V'ed by the appropriate RFIFO command on the other IGC; similarly, one of the TFIFO-blocking semaphores can V'ed by the appropriate RFIFO command on the same IGC, and one of the TFIFO-blocking semaphores can be V'ed by the appropriate TFIFO command on the other IGC. There is also a "preference" register which determines which of the two FIFOs on each IGC is the preferred one for executing commands; the preferred FIFO takes precedence unless it is blocked, even if it is empty.

IC Controller. Each IGC also contains an IC Controller, which controls Image Composition port operation. However, only the IC Controller on the EIGC is used. Each FIFO can also be blocked by a P-like command which waits until the IC Controller no longer has a pending composition cycle.

These semaphores and their control commands are described in Section VI.

IGC Sequencers. Each IGC contains a sequencer. The EIGC Sequencer generates micro-instructions for the pixel-ALUs, addresses into pixel-memory, and *ABC* coefficients for the linear expression evaluator; commands to the EIGC are used to rasterize polygons (on Renderer boards) and to perform shading calculations (on Shader boards). The TIGC Sequencer controls the TASICs and attached memory; commands to the TIGC are used for moving data between the pixel-memory on the EMCs and external texture/frame-buffer memory, or between pixel-memory on the EMCs and the geometry processor bus.

The Sequencers each contain local microcode memory. A bit in the I-word of each command indicates which sequencer is to execute the command. The I-word also specifies the starting microcode address for the command. Each sequencer can conditionally branch, both on internal conditions (such as the value of loop counters) and external conditions (three condition code inputs for each sequencer). The sequencers have a one-level stack; they can store a single microcode return address, allowing one level of procedure calls. They have several external control outputs which allow them to perform miscellaneous control functions on the rasterizer board.

There is a delay of approximately 20 cycles between the time a Sequencer issues a micro-instruction and when the micro-instruction is executed within the EMCs or TASICs. The external control outputs are subject to a somewhat smaller latency. The programmer need not normally be concerned with these latencies, except in some situations described in the command descriptions and examples in the remaining sections.

Sequencer commands and their formats are described in detail in Sections IV and V.

II.3 EMC Array

The array of 32 PixelFlow EMCs implements a 2-dimensional SIMD processor array that covers a 128x64-, 64x32-, or 32x32 - pixel region of the screen, as described above. This logical array can be "moved" to process any region of the display screen; the region is normally aligned to region boundaries, but it *can* be positioned arbitrarily.

Each PE is provided with its own 8-bit ALU, an output of the linear-expression evaluator tree (the LEE), 256 bytes of local memory, two 32-byte *transfer buffers*, and two 32-byte *local-port buffers*. Figure 3 shows a logical diagram of an EMC.

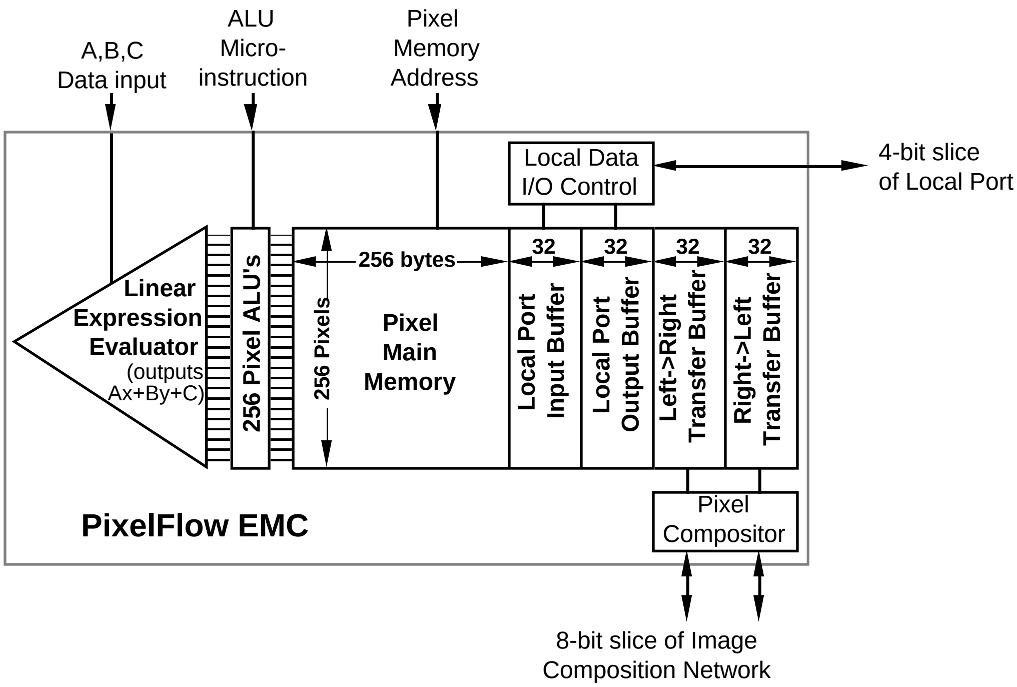


Figure 3: Logical diagram of a PixelFlow EMC.

PE ALUs. Each PE's ALU is a general-purpose 8-bit processor; it includes an enable register which allows operations to be performed on a subset of the PEs. The PE can use tree results or local memory as operands and can write results back to local memory. It can also transfer data between memory, the carry register, and the I/O buffers. Figure 4 shows a logical diagram of the ALU.

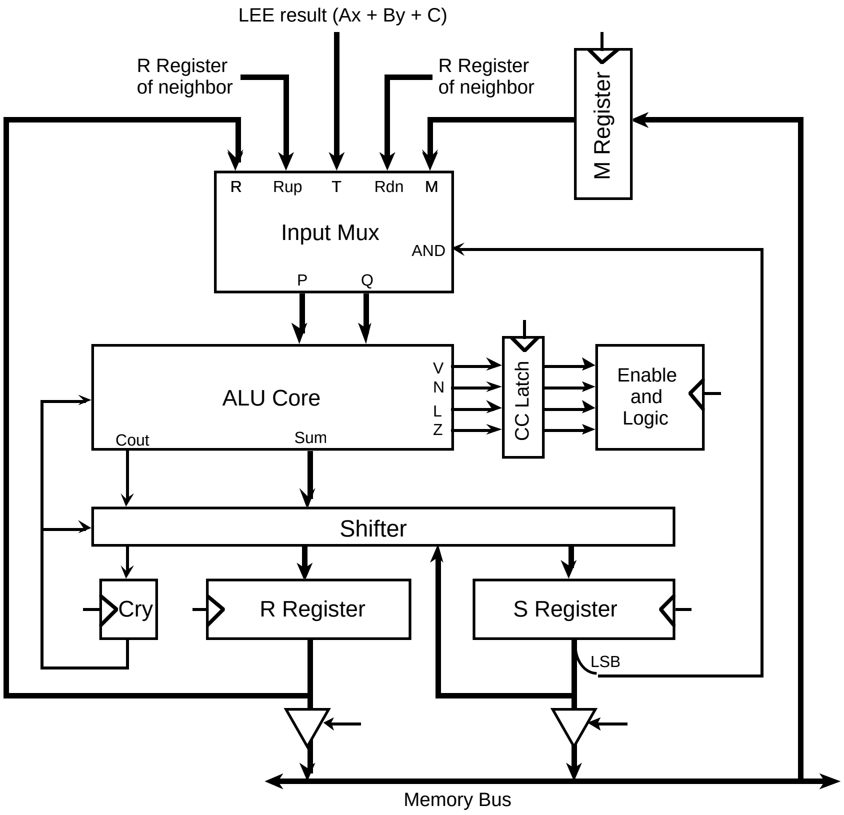


Figure 4: Logical diagram of EMC PE ALU.

The 256 PEs are divided into 8 *panels* of 32 PEs each. Some limited communication among the PEs in a panel is possible, via the ALU pathways.

Linear-expression evaluator (LEE). The linear-expression evaluator evaluates bilinear expressions $Ax + By + C$ for each PE of the array in parallel. A, B, and C are coefficients loaded from the IGC and (x, y) represent the PE's x,y address. Many graphics calculations can be cast into the form of bilinear expressions, such as the edge, depth, and color calculations required to render Gouraud-shaded triangles.

The IGC controls the operation of the EMC array. IGC instructions and coefficients are serialized and broadcast to all of the EMCs in parallel. The SIMD array of PEs execute these instructions in lock-step. The enable registers in each PE are used to control which subset of the PEs are active at any given time.

Pixel Memory. Each PE is provided with 256+4*32 bytes of local memory. The memory is divided into 5 partitions: a 256-byte main partition, which is used for most computation, and four 32-byte partitions used for external communication. Two of these, the local port buffers, are connected to the local port. The local port is connected to the TASICs, so that data can be exchanged between the local buffer and attached external memory. The others, the transfer buffers (or image-compostion buffers) are connected to the image-composition port. Data that is ready to be composited is placed into this area.

Address Range	Length (bytes)	Partition
0 – 255	256	Main memory
256– 287	32	Left-to-Right Transfer Buffer
288 – 319	32	Right-to-Left Transfer Buffer
320-381	62	Unmapped
382	1	Local Port In Mark Register
383	1	Local Port Out Mark Register
384 – 415	32	Local Port Input Buffer
416 – 447	32	Local Port Output Buffer
448– 511	64	Base Address Offset Area

Table 1: EMC Pixel Memory Address Map.

Table 1 shows the memory map for pixel memory. Addresses 382 and 383 are 1-bit read/write registers used for the enable flags for local port operation; this is handled within the local port control instructions, and these addresses should not be accessed directly.

Addresses 448-511 are not physically implemented. This portion of the address space is used for base address registers, described below.

Normally, all 384 bytes of pixel memory can be accessed. However, when communication-port operations are performed, their buffer data temporarily is unavailable. For example, after pixel data to be composited is copied into the transfer buffers and the composition operation is initiated, memory in the transfer buffers cannot be accessed by the ALU until the composition operation is complete. Similarly, to perform a local-port operation, data is moved into the local output buffer, the local-port operation is initiated, and data may be unloaded from the local input buffer; during the local port operation, the local buffer being used must not be accessed by the ALU until the operation is complete (although the other local buffer can be accessed by the ALU if it is not involved in the local port operation). If any one of the four communications buffers is accessed while its port is in operation, unpredicatable results will occur; the write or read may or may not happen, but the port operation will not be disturbed. This occurs if the address being read or written is *anywhere* in the 32-byte address space of the active port, even if it is not one of the addresses actually being used by the port operation.

All pixel memory is dynamic and so must be periodically refreshed. This refresh is performed opportunistically by the EIGC Sequencer. Only under very unusual circumstances (mass quantities of memory-intensive EMC Sequencer instructions) may it be necessary to explicitly refresh EMC pixel memory. This can be done by interspersing refresh commands. The local and transfer buffer portions of pixel memory are not accessible under IGC control when port operations are in progress, so untouched data may be corrupted; therefore, unused portions of the local and transfer buffers must not be used for data storage. For example, if composited pixels are to be 192 bits in size, so only the low 24 bytes of a transfer buffer are needed, the remaining 8 bytes should not be used for general storage.

Communication Ports. The image-composition port and local port allow pixel data to be transferred serially to/from the EMCs to other EMCs (for compositing) or to/from the TASICs (to perform texture lookups or pixel-data writes to texture or video memory). Data from each PE is presented serially at each port. The number of bytes transferred to/from each PE and their location in the communication buffer are designated by configuration commands described in Section IV below. The image-composition port is an 8-bit port which runs at 200 MHz. The local port is a 4-bit port which runs at 200 MHz, with simultaneous bi-directional traffic.

Global enable. Each EMC has an output which represents the logical-OR of the enable registers of all PEs. These outputs are wire-anded together to form the *global-enable* signal (*EOrH*), the logical-OR of the enable registers for the entire SIMD array. *EOrH* is fed into an external condition-code input of the EMC sequencer. Commands to the EMC sequencer can test the status of *EOr,* and based on the result, can conditionally execute. The status of *EOr* can be communicated to the GP using a special command that waits until *EOr* is valid (must wait for last commands to be executed by the EMCs and *EOr* to become valid). Depending on the state of *EOr,* it asserts one of the two EMC sequencer external outputs to the GP status register. The GP status register contains a sticky bit for each of these signals. The GP can determine the status of *EOr* by waiting for one of these bits to be set, then clearing it. Note that the EMC sequencer may execute commands after the test-*EOr* instruction.

Panel organization of an EMC. The 256 PEs on an EMC are arranged in 8 *panels* of 32 PEs each. For some screen organizations, the panels are paired up, so that the PEs are effectively arranged as 4 panels of 64 PEs each. Note that no communication is possible between the lower and upper 32 PEs in these 64 PE panels, since they are separate panels hardware-wise. This organization is mostly transparent to the user, but the user must be aware of panels in several instances, so panels are discussed in the PE organization information given below.

PE Organization within a Region. The 256 PEs on each of the 32 EMCs in a rasterizer are mapped to the display screen in different patterns for different tasks. Figures 5 A-C show the three configurations (1, 4 and 8 samples per pixel) for rendering displayable images and Figures 5 D-F show the three configurations for rendering texture maps. Note that the EMCs are interleaved in both the *x* and *y* dimensions.

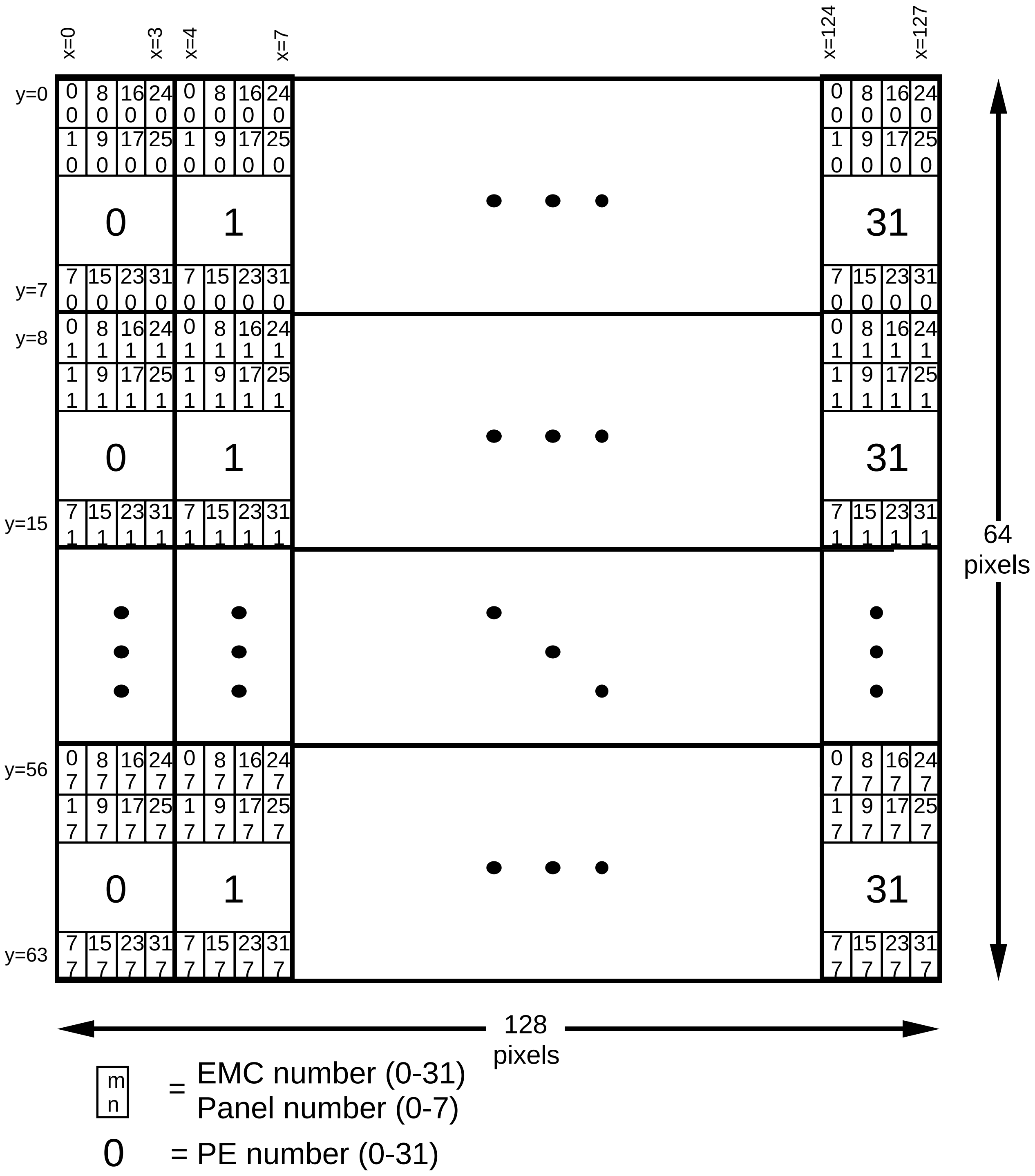


Figure 5A: Screen-space organization of EMCs for a single-sampled 128x64 pixel region - for display.

In the single-sampled organization of Figure 5A, a given panel on a given EMC represents every fourth pixel on a given scanline. Four panels, one from each of 4 different EMCs, represent an entire scanline.

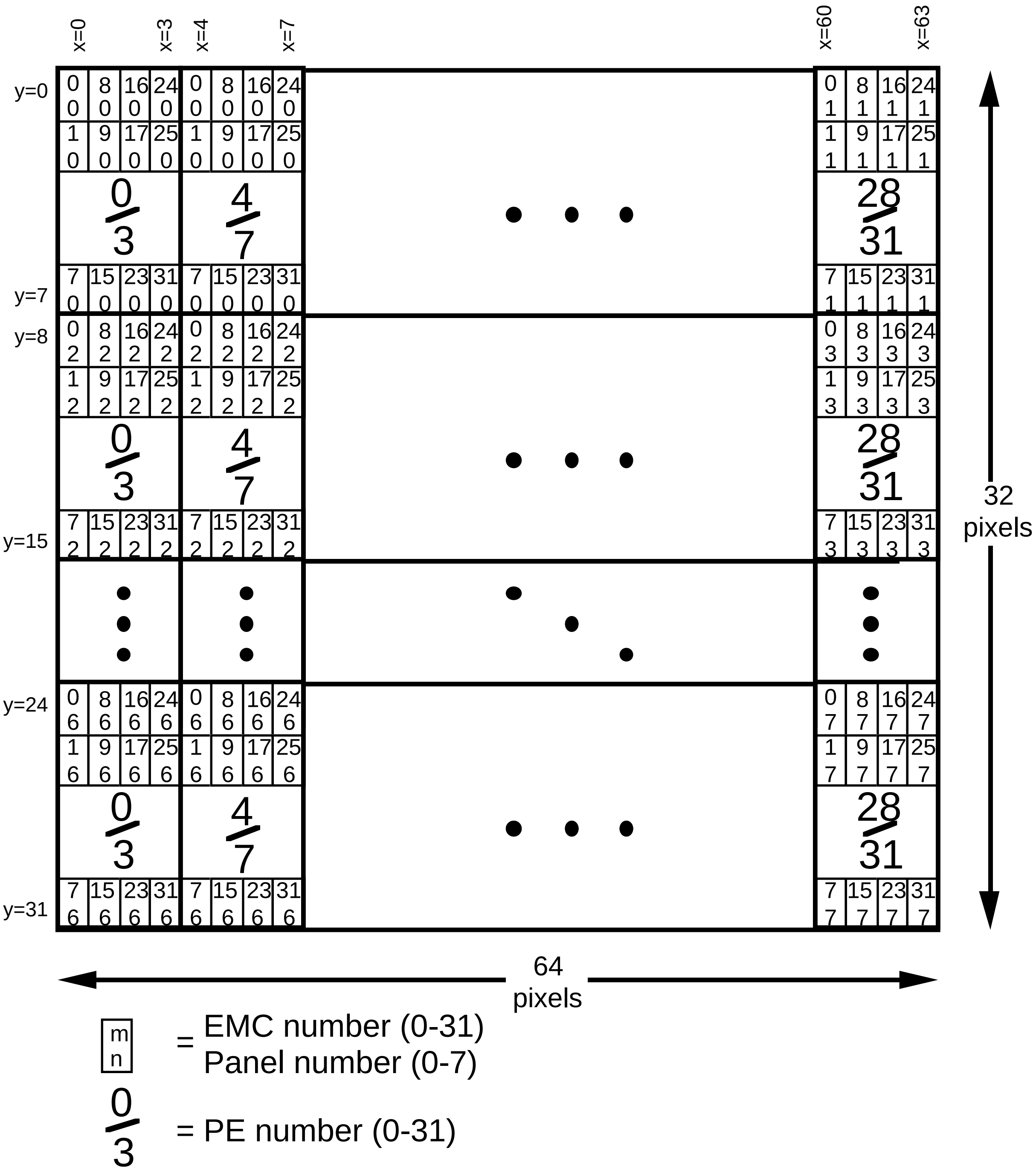


Figure 5B: Screen-space organization of EMCs for a 4 sample-per-pixel 64x32 pixel region - for display.

In the 4-sample per pixel organization of Figure 5B, the PEs on an EMC are arranged so that the even-numbered panels cover the left-half of the region and the odd-numbered panels cover the right-half of the region. Four neighboring PEs on a panel represent the 4 samples of a pixel, so each panel contains 16 pixels.

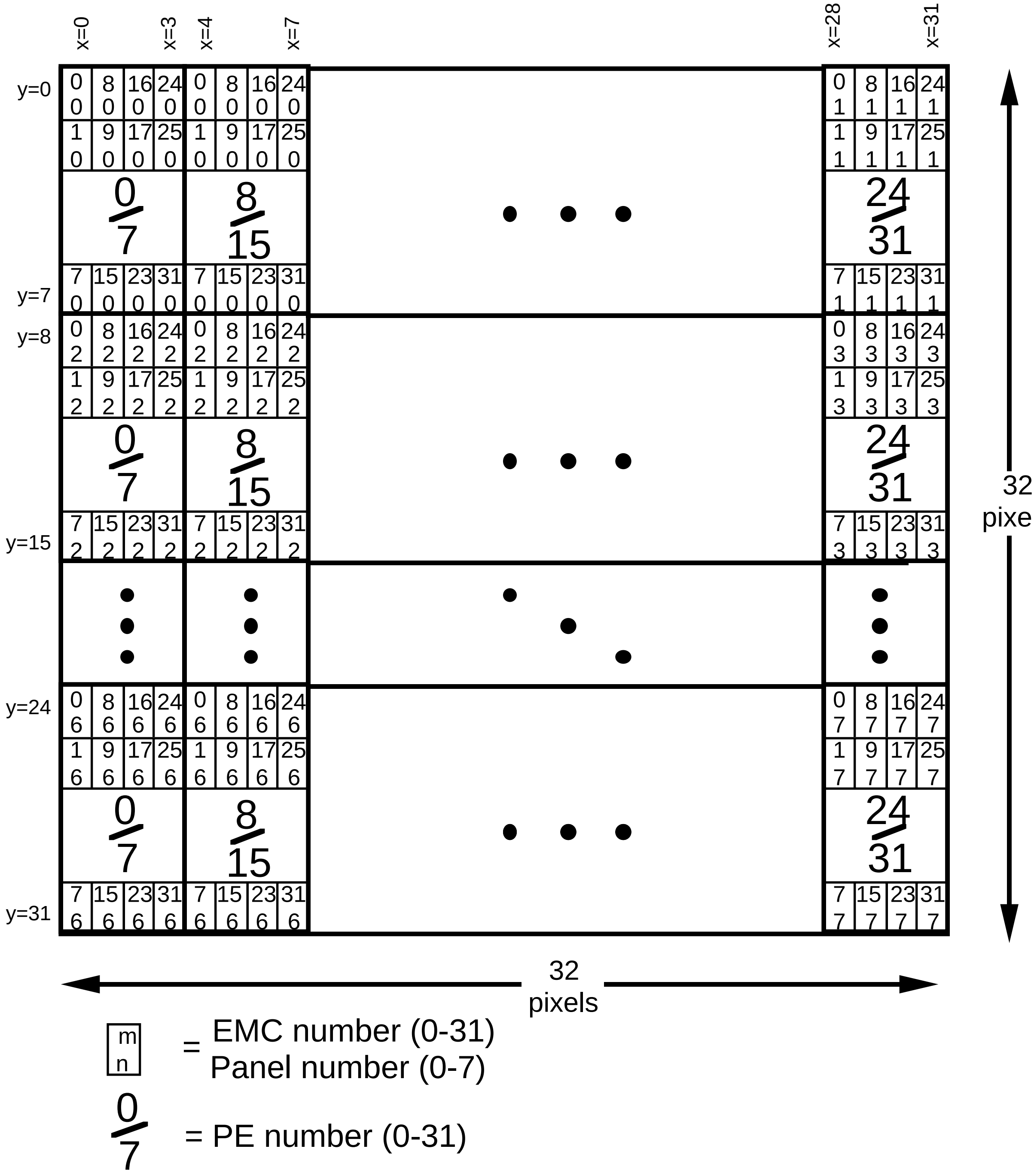


Figure 5C: Screen-space organization of EMCs for an 8 sample-per-pixel 32x32 pixel region - for display.

In the 8-sample per pixel organization of Figure 5C, the PEs on an EMC also are arranged so that the even-numbered panels cover the left-half of a region and the odd-numbered panels cover the right-half of a region. Eight neighboring PEs on a panel represent the 8 samples of a pixel, so each panel represents 8 pixels.

The primary difference between the standard rendering configurations and the texture map configurations is that for rendering texture maps, 4 different panels of one EMC must be clustered into a 2x2 grid to facilitate writing to texture memory (this has to do with the order in which PEs from a given EMC are accessed - see the section on the TASICs).

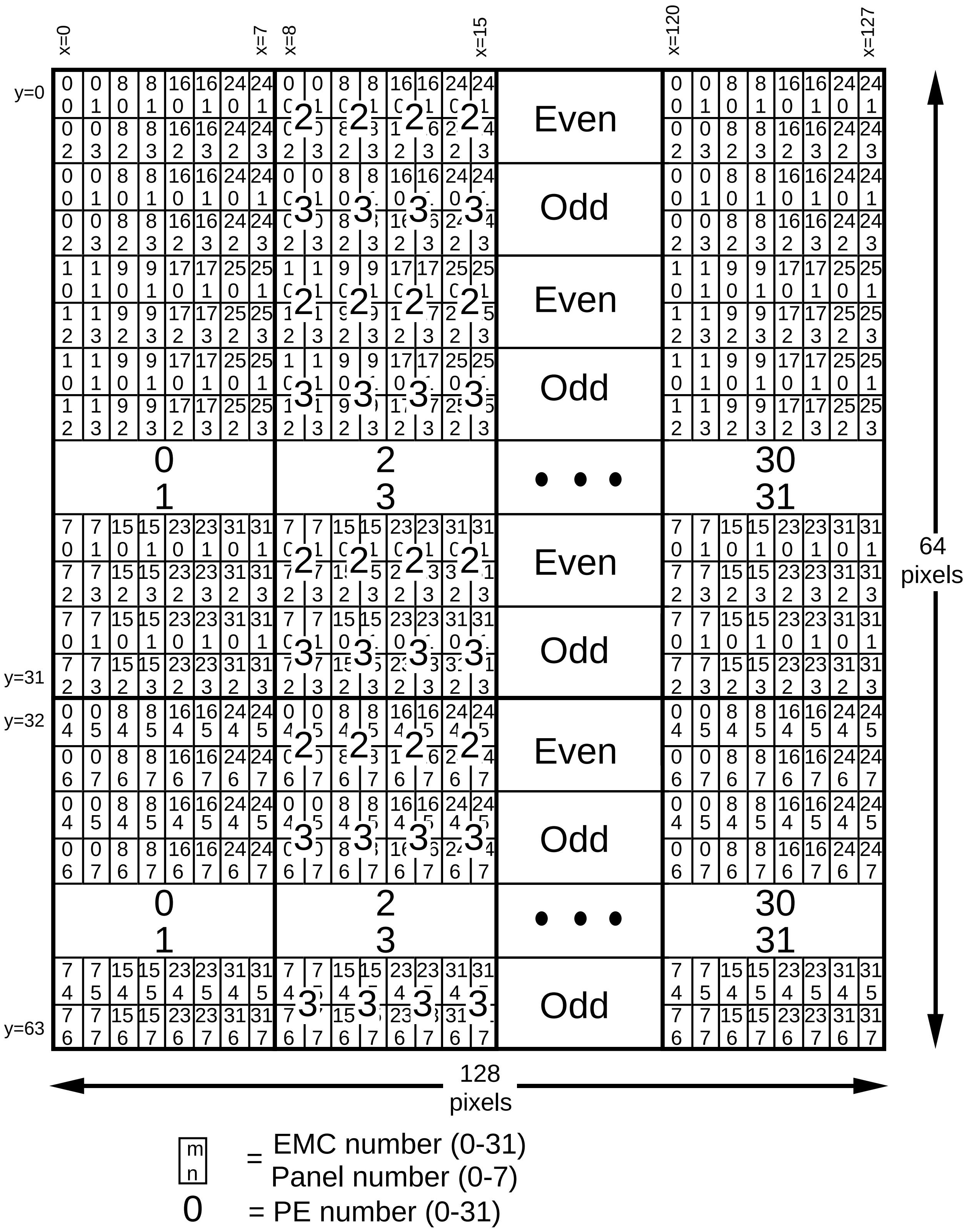

x=0
x=7
x=8
x=15
x=120
x=127
y=0
y=31
y=32
y=63
Even
Odd
Even
Odd
0
1
2
3
• • •
30
31
Even
Odd
Even
Odd
0
1
2
3
• • •
30
31
Odd
64 pixels
128 pixels
m
n
= EMC number (0-31)
Panel number (0-7)
0
= PE number (0-31)

Figure 5D: Screen-space organization of EMCs for a single-sampled 128x64 pixel region - for texture maps.

In the single-sampled organization of Figure 5D, the top half of the region is covered by panels 0-3 of all EMCs while the bottom half is covered by panels 4-7. The EMCs are interleaved as before, except now the panels are interleaved in *x* and *y* as well: 2 interleaved panels from each of the (interleaved) EMCs form a single scan-line.

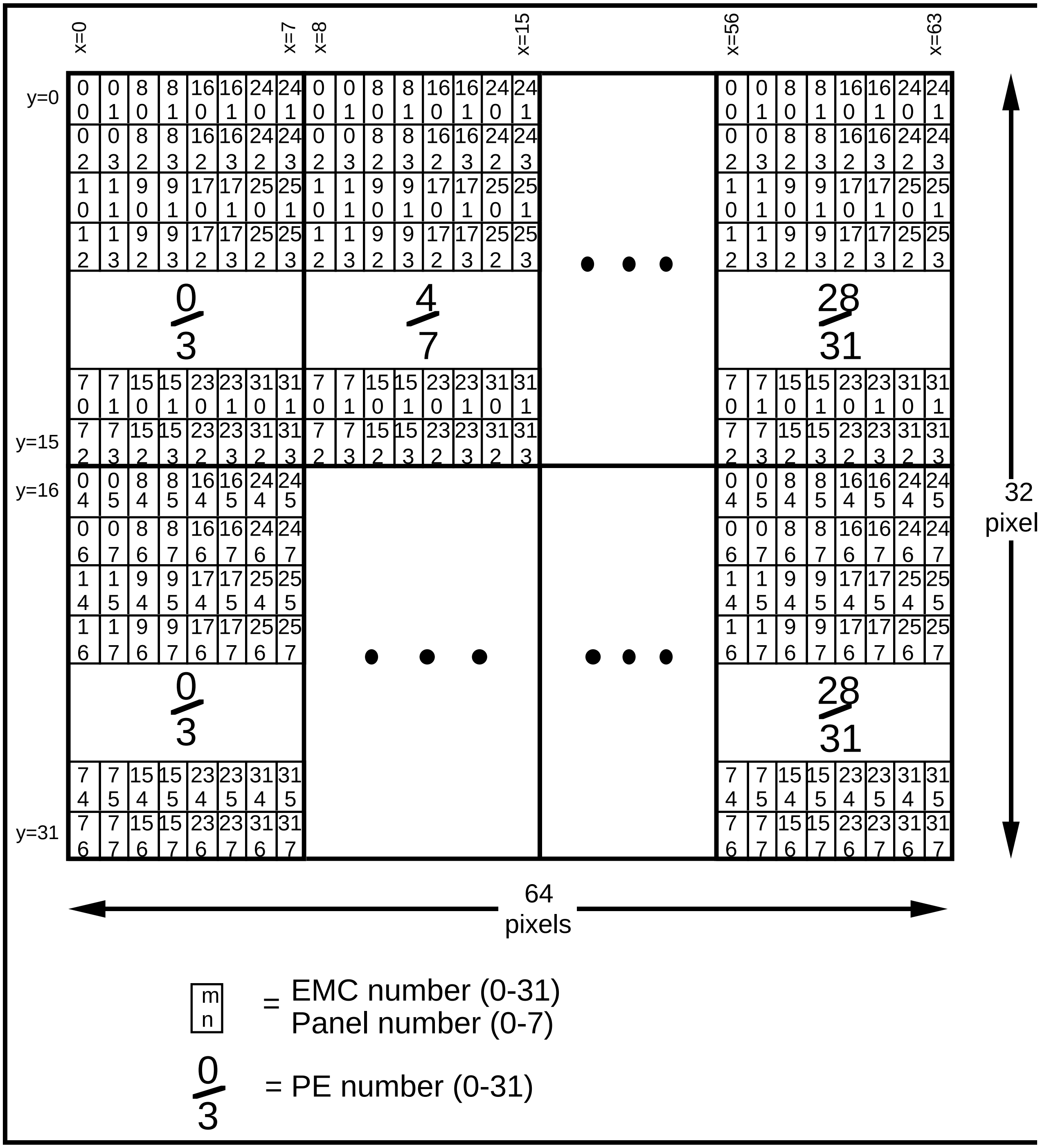


Figure 5E: Screen-space organization of EMCs for a 4 sample-per-pixel 64x32 pixel region- for texture maps.

The 4 sample-per-pixel organization of Figure 5E is similar to the single sample configuration, except now 4 adjacent PEs within a panel form one sample for a pixel.

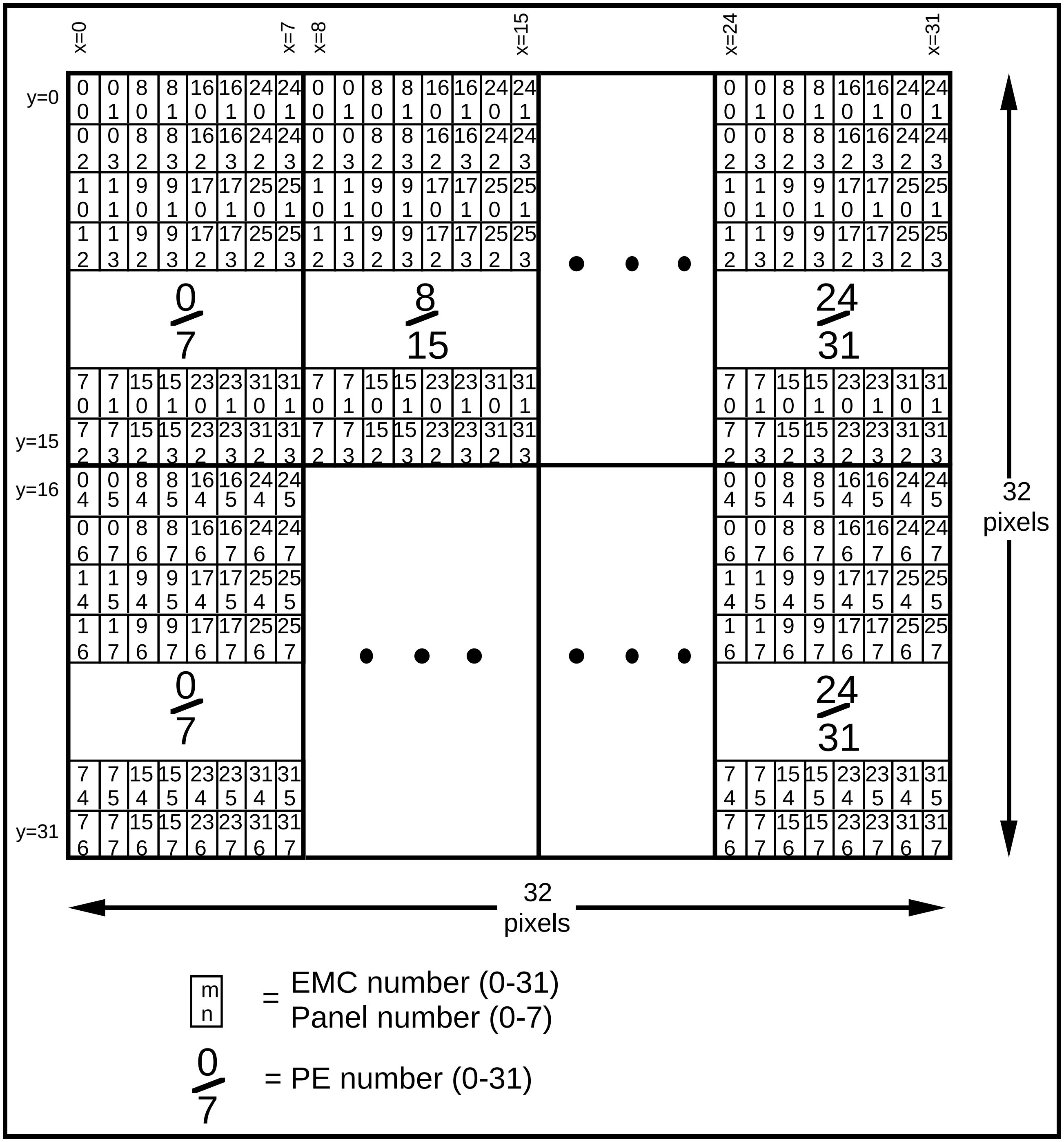


Figure 5F: Screen-space organization of EMCs for an 8 sample-per-pixel 32x32 pixel region- for texture maps.

Again, the 8 sample-per-pixel organization of Figure 5E is similar to the 4 sample configuration, except now 8 adjacent PEs within a panel form one sample for a pixel.

EMCs are grouped into 4 *modules*, containing 8 chips each; modules 0-3 contains EMCs 0-

7, 8-15, 16-23, and 24-31, respectively. Because of the EMC arrangement within each 4 x 8 block of pixels seen in Figures 5A-C, then means that each module represents the pixels on every 4'th column in the region.

II.4 TASIC Array

The array of 8 *Texture ASICS* or *TASICS* implement a data-parallel communication interface between pixel memory in the EMCs, the Geometry Network Interface (GNI) chip, texture/video memory, and optional video circuitry. They perform the buffering and data conversion required to read and write SDRAM memories and contain internal counters to refresh a video display or read video data from a frame grabber.

Although there are two physical TASIC chips per module, the two function together as a single *logical* TASIC. Each physical TASIC represents one bit-slice of two: the even bits of all external datapaths connect to one physical TASIC and the odd bits of all external datapaths connect to the other. The 8 physical (4 logical) TASICs are divided among the 4 modules, so that each module consists of eight EMCs and two TASICs. Figure 6 shows the EMCs and TASICs one module and the various data connections to/from the TASICs.

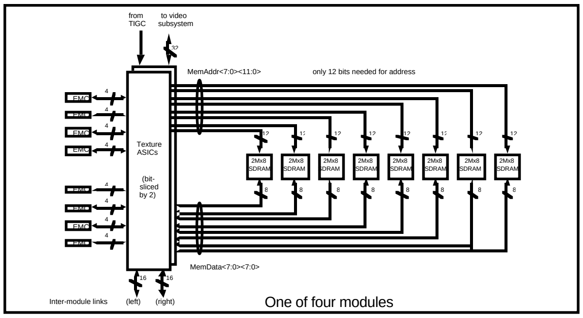


Figure 6: Interconnection between components in a module (one of four modules).

The TASICs of one module connect to the local ports of the module's eight EMCs in bit-slice fashion. They also connect to the corresponding TASICs of the other modules via a bit-sliced 16-bit ring network composed of *Inter-module TASIC Links*. This ring network provides communication between modules and the GNI, allowing pixel data to be shared between modules, and allowing the GP to participate in pixel calculations or to access pixel memory for diagnostic purposes.

Both the EMC-to-TASIC connections and Inter-Module TASIC links operate at 200 MHz, bidirectionally.

The TASICs of each module also contain a 32-bit bidirectional interface to optional video circuitry. This port, which also is bit-sliced by 2, is clocked with an external clock that can run up to 100 MHz. The port is used to send pixels from DRAM memory to video DACs (when refreshing a screen) or to read pixels from video ADCs (on a frame grabber) into texture/video memory.

Figure 7 shows a block diagram of a physical TASIC chip. Internally, it is composed of three dual-ported RAMs, several configurable datapaths, and a number of control and address registers and counters. One dual-ported RAM buffers SDRAM memory addresses (the *Address Corner Turner* or *ACT*); one buffers SDRAM data (the *Data Corner Turner* or *DCT*); the third buffers video data to/from the video port (the *Video FIFO* or *VFIFO*). Cycle-by-cycle operation of the TASICs is controlled by the TIGC Sequencer, which also controls the EMCs' local ports (since these interface directly to the TASICs).

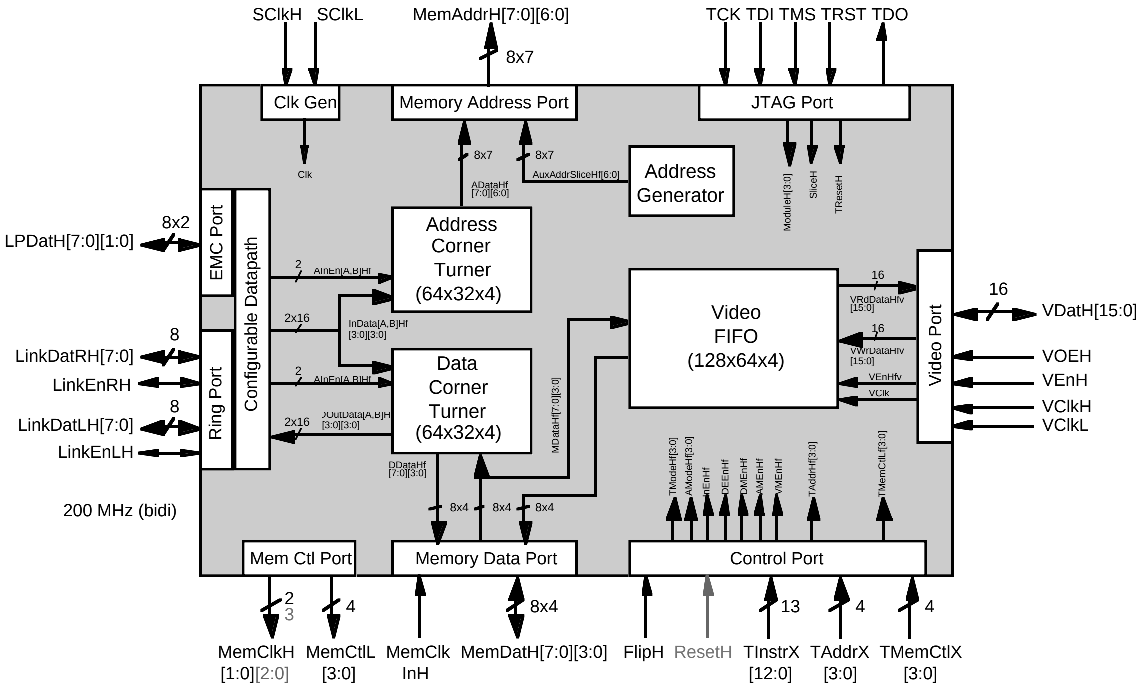


Figure 7: Block diagram of Texture ASIC (TASIC). Each physical TASIC (pictured here) represents one of the two bit-slices in a logical TASIC.[1]

Throughout the remainder of the document, unless otherwise indicated, we will refer to *logical* TASICs (both bit slices), not physical TASICs.

Address Corner Turner. The *Address Corner Turner* (*ACT*) is a dual-ported memory that "corner-turns" serial addresses arriving from the EMCs or GNI and buffers them up

[1] Two revisions of the TASIC have been fabricated, Rev1.0 and Rev2.0. Rev2.0 is almost identical functionally to Rev1.0, but has two extra signal pins (shown in gray), a more general SDRAM clock generator, and different semantics for 1to1 CommPort transfers (see below). A complete list of the differences between revs can be found in the *PixelFlow TASIC Functional Description*.

into the parallel format required at the Address Port. It is composed of a 32x64x8 dual-ported memory, as shown in Figure 8. Address data from the eight EMCs in the module stream into this memory from the left. The eight addresses from a single EMC are read out from below. Separate write and read pointers keep track of where in the ACT addresses are currently being written or read.[1]

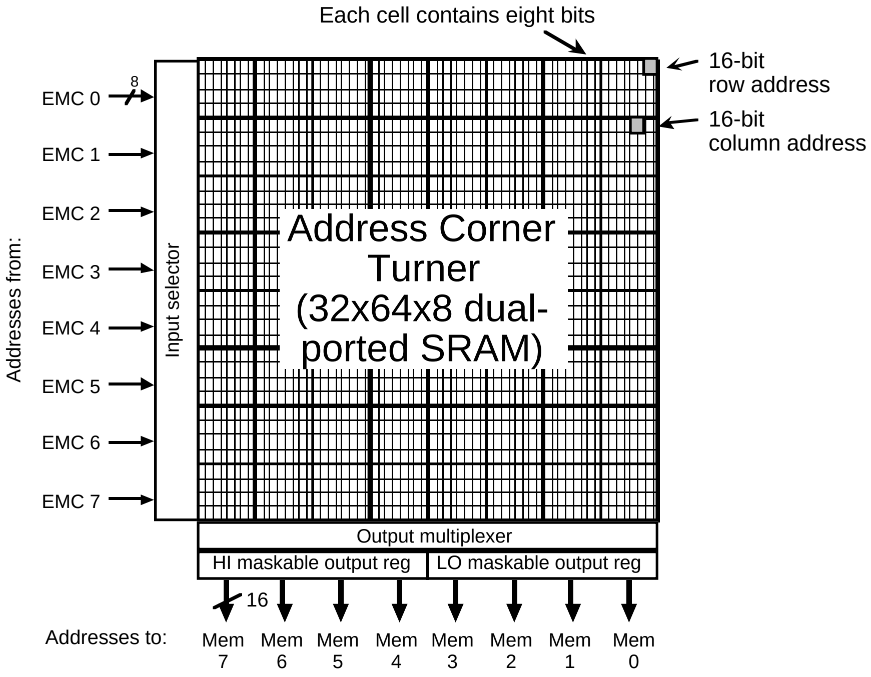


Figure 8: Address Corner Turner.

The ACT can be configured to mask outputs to the low or high sets of four SDRAMs. Commands to configure the ACT are described in Section V.2.

Data Corner Turner. The *Data Corner Turner* (*DCT*) is similar to the ACT, except it buffers data rather than addresses and can transfer data in both directions, from EMCs (or GNI) to SDRAMs, or vice versa. Like the ACT, it is composed of a 32x64x8 dual-ported memory, as shown in Figure 9. When writing textures to SDRAM memory, it receives data from eight EMCs from the left. The eight data values from a single EMC are read out from below. Separate write and read pointers keep track of where in the DCT data values are currently being written or read.

[1] Access patterns for these pointers and detailed timing for the ACT and DCT can be found in the *PixelFlow TASIC Functional Description.*

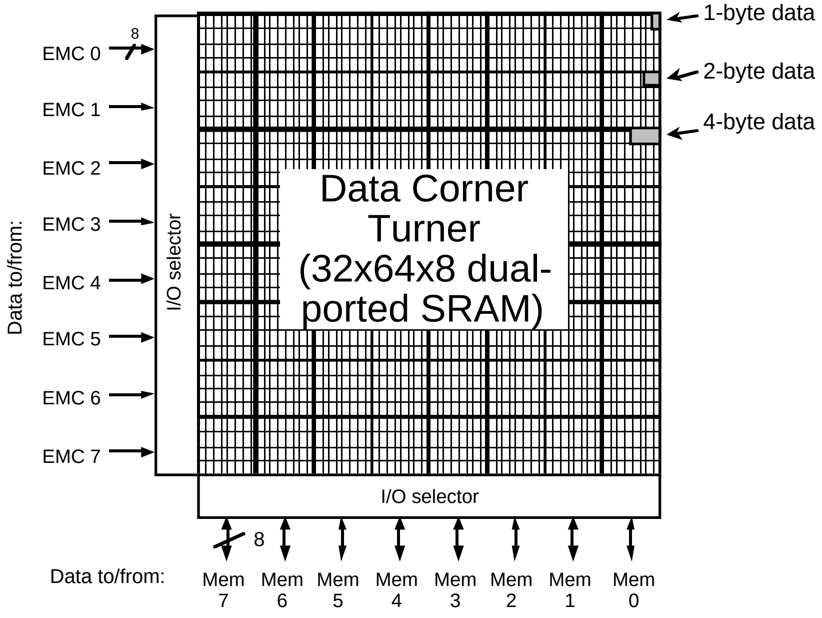
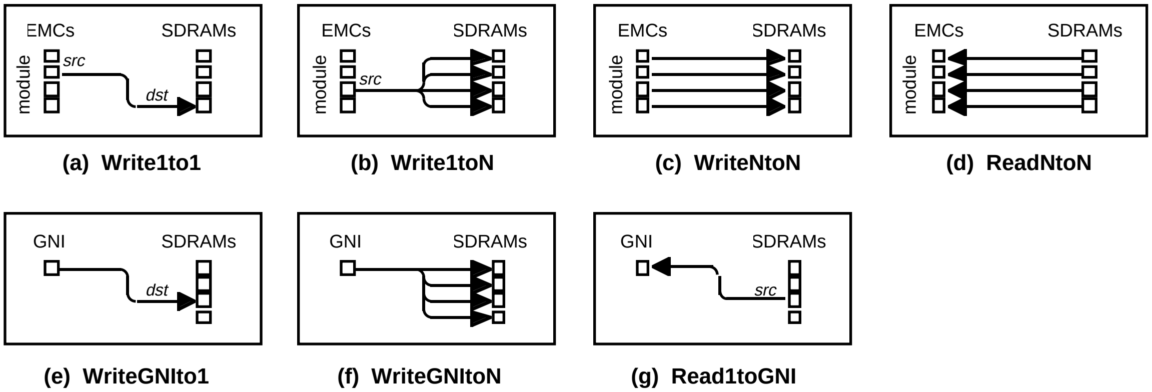


Figure 9: Data Corner Turner.

The DCT can be configured to send data in either direction and to transfer either 1, 2, or 4 byte data types. There are no commands to do this explicitly; rather DCT configuration is done at the beginning of data transfer commands (described in Section V.3).

Configurable Datapath. The datapath between the EMC port, the inter-module TASIC links, and the ACT/DCT is configurable to allow data to be transferred in a variety of directions, both between the modules (and GNI) of a rasterizer and within a given TASIC.

Options for transferring data between modules (and GNI) using the inter-module links are shown in Figure 10:



(a) Write1to1

(b) Write1toN

(c) WriteNtoN

(d) ReadNtoN

(e) WriteGNIto1

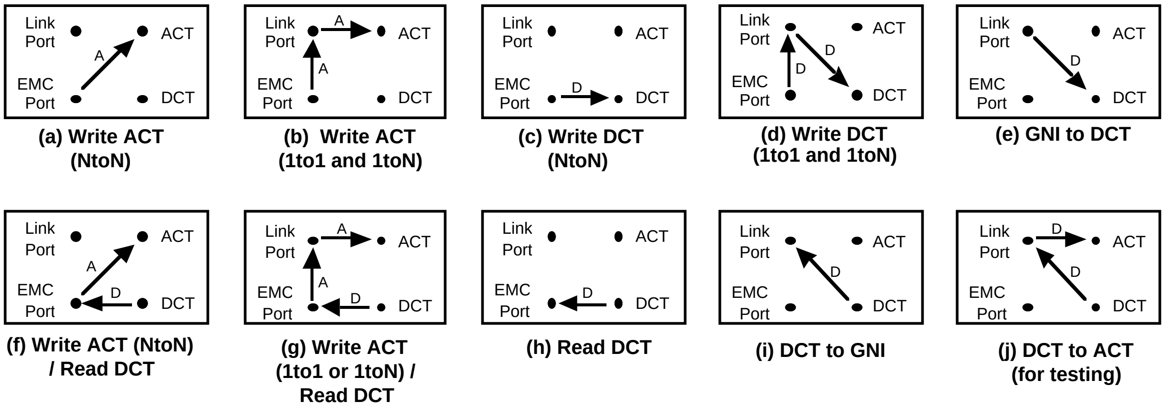
(f) WriteGNItoN

(g) Read1toGNI

Figure 10: Options for transferring data between modules (and GNI).

Data can be sent point-to-point from the EMCs of one module to the SDRAMs of another module. It can be broadcast to the SDRAMS of all modules. It can be written in parallel for each of the modules. Similarly, data can be sent point-to-point from the GNI to one module's SDRAMs or broadcast to every modules' SDRAMs. Reads can be done in parallel for all modules, or read data from one module can be sent to the GNI.

Possible datapath configurations within a TASIC are shown in Figure 11:



(a) Write ACT (NtoN)

(b) Write ACT (1to1 and 1toN)

(c) Write DCT (NtoN)

(d) Write DCT (1to1 and 1toN)

(e) GNI to DCT

(f) Write ACT (NtoN) / Read DCT

(g) Write ACT (1to1 or 1toN) / Read DCT

(h) Read DCT

(i) DCT to GNI

(j) DCT to ACT (for testing)

Figure 11: TASIC internal datapath options.

Portions of the datapath configuration are done implicitly as part of data transfer commands. Explicit setup commands are used to perform other parts of configuration. These commands are described in Section V.2.

Internal Address Generator. Under some circumstances, such as many texture writes and transfers of texel data to/from the GNI, it is impractical to provide memory addresses from the EMCs. To support these kinds of operations, the TASIC contains an internal address generator. It has three main parts:

1) A set of eight immediate address registers.

2) A 32-bit presettable up-counter followed by a programmable crossbar.

3) Multiplexing circuitry at the memory address outputs that can select and combine addresses from different sources.

The immediate address registers are 16-bit registers that can be set to arbitrary values via the TIGC. The value in a particular register can be sent directly to all eight TIGC address outputs or it can be xor'ed with other address sources.

A 32-bit presettable up-counter followed by a crossbar is available to provide incrementing addresses. The crossbar allows allows address bits to be interchanged, so fairly general addressing patterns are available.

Multiplexing circuitry at the memory address outputs selects among the different possible address sources. Some address sources can be xor'ed together as well, providing even more flexibility when generating addresses. Commands to configure the address generation logic are described in Section V.2.

Video FIFO and Video Port. On video boards (frame-buffer or frame-grabber boards), SDRAM memory is used to store pixels, making it unnecessary to provide additional frame-buffer memory for this purpose. Access to SDRAM memory is time-shared on video boards between video reads/writes and normal texture/backing-store accesses.

Since the video port runs asynchronously with respect to the rest of the system, some buffering is needed between the SDRAMs and the video port. This buffering is provided by the *Video FIFO* (*VFIFO*), another dual-ported memory. It is a 256x32x8 memory, which can load (or store) eight pixels simultaneously from the SDRAMs while asynchronously transmitting (or receiving) a stream of pixels over the video port (Figure 12). The TIGC controls the loading and unloading of pixels from the SDRAM end of the VFIFO, while an external video controller controls the other end via the video port.[1]

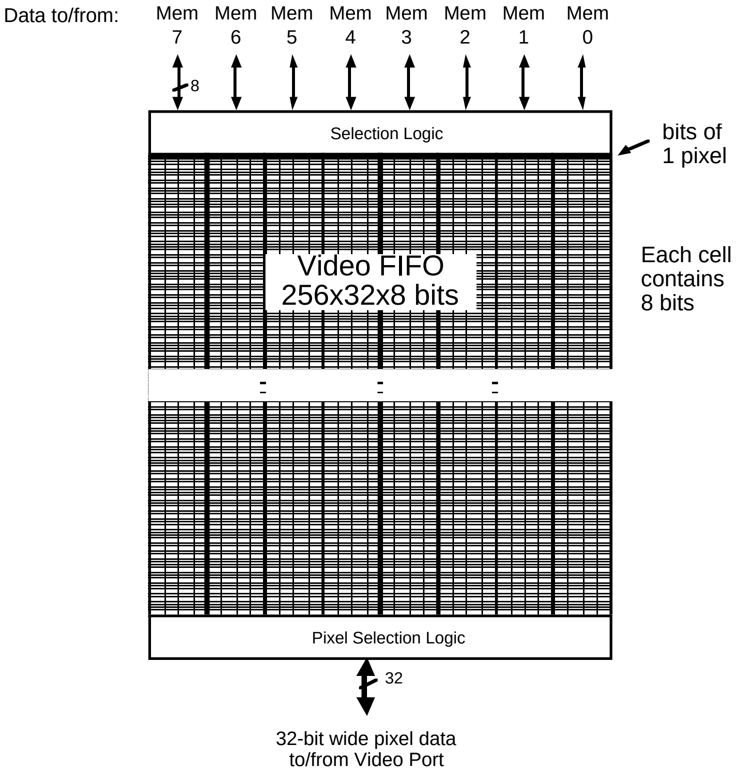


Figure 12: Video FIFO.

[1] The memory access pattern and detailed timing info for the VFIFO can be found in the *PixelFlow TASIC Functional Description*.

In addition to buffering pixels, the TASICs also have internal address registers for keeping track of pixel row/column addresses for the starting scan line of up to eight independent video fields, plus counters for the address of the current scan line and pixel. It also contains registers which specify the order fields are to be displayed, so interleaved and/or stereo displays can be refreshed continuously without intervention by the GP. Special TIGC Sequencer instructions update these address address and field registers. These are used to swap buffers when double-buffering and to synchronize GP operation with video scanout when desired.

Commands to configure and operate the VFIFO and Video Port are described in Section V.5.

II.5 Texture/Video Memory

As described above, two TASICs in each module connect to eight 2-M x 8 (16 megabit) synchronous DRAM (SDRAM) memories, a total of 32 SDRAM chips per system board. These memories are used to store image-based textures, provide inter-pixel communication for image-warping operations, serve as backing store for memory-intensive rendering algorithms, and store video frames in frame-buffer or frame-grabber boards.

The 8 SDRAMs in a module are arranged as shown in Figure 6 above. Each SDRAM chip contains 4K x 512 bytes of storage. The chips are controlled globally by the TIGC Sequencer (all eight do a memory operation at the same time), but addresses for each bank are independent. The total storage per module is 16 MBytes. Each SDRAM can read or write data to/from a random location in memory at a peak rate of 100 Mbytes/sec, so the raw memory bandwidth per module is 8 • 100 MBytes/sec = 800 MByte/sec (the attainable memory bandwidth for four-byte reads/writes is approximately 580 MBytes/sec).

Addresses for texture reads/writes can come from one of three sources: 1) computed on EMCs and sent over local-port to TASICs, 2) sent to TASICs over geometry network, 3) generated on the TASICs themselves using a simple 32-bit presettable counter followed by a crossbar that allows permutations of address bits within the 32-bit word.

The external memories on both the shader and video boards are dynamic memories—they 'forget' data that is stored within them if they are not refreshed periodically. Refreshing consists of visiting every row in the memory array of every memory chip at least every 8 msec. This is done by a TIGC Sequencer microcode subroutine that performs refresh cycles. Calls to this subroutine are in TIGC sequencer microcode routines so that refresh cycles are guaranteed to occur sufficiently often. Section V.2 describes this refreshing method in more detail, and special precautions that must be taken when loading microcode.

II.6 Video Interface

The optional video controller multiplexes pixel data to/from the four modules and converts between digital and analog formats. It contains its own pixel clock and counters for the number of pixels in a scanline, the number of scanlines in a field, and the fields in a frame (we define a frame as one or more fields that are displayed consecutively without requiring intervention by the GP). The video controller may contain an auxiliary frame buffer (for X overlays, for example) whose output is merged with the video signal from PixelFlow.

The basic strategy is to make the video controller the master of all video processing and make TIGC and TASIC video ports synchronize to it. This allows external gen-locking and prevents TIGC failures (hanging, for example) from disrupting all video.

The pixels in a scan line are interleaved over the modules, so that every 4 (contiguous) pixels on a horizontal scan line come from different modules. The SDRAMs provide storage for 4Kx4K 32-bit pixels, sufficient to double-buffer images for any display up to 4Kx2K pixels. The video port can be clocked at up to 100 MHz. By multiplexing the outputs of the four modules together, as shown in Figure 13, the overall pixel rate can be as high as 4 • 100 Mpixels/sec = 400 Mpixels/sec, sufficient to update a 2K x 2K monitor at 60 Hz. Portions of memory not needed for storing images can store texels for image-based texturing or other data as in any other system board.

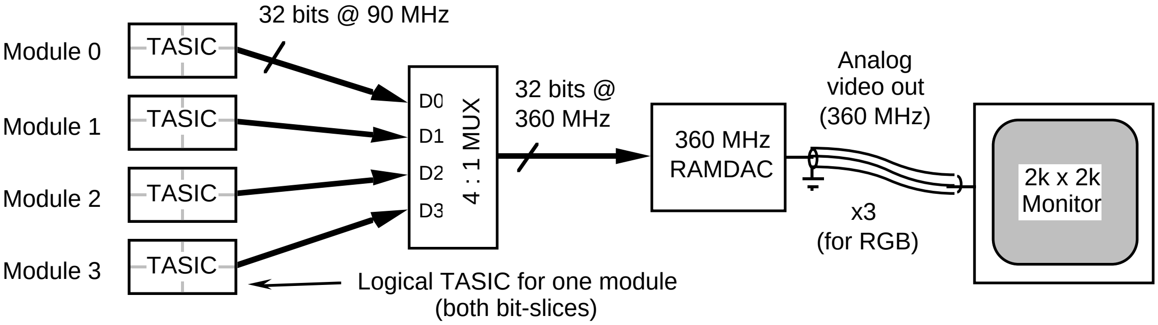


Figure 13: Video connections for a 2K x 2K-pixel frame-buffer board.

The TIGC Sequencer polls a video request input and reads or writes batches of pixels from/to SDRAM memory when indicated. Video pixels are buffered in the TASICs' internal VFIFO, which can store up to 4096 pixels (enough for two scan lines of a 2K x 2K display. The video controller handshakes with the TIGC sequencer so that the video buffer is always partially full when scan lines may be requested.

Figure 14 shows the connections between a sample video controller and rasterizer. *TASStrobe* and *TASStrobeDel* communicate status information from the TIGC to the GP (*TASStrobeDel* outputs are delayed by the tree latency). *GPStrobe* communicates status information from the GP to the TIGC. One of eight internal status bits is multiplexed onto *GPStrobe* under control of *TASMuxSel*, generated by the TIGC. *VidReq* conveys video requests to the TIGC. *VidStat* indicates the type of request (0x0 = unused, 0x1 = new line, 0x2 = new field, 0x3 = new frame). *VidClk* is the video subsystem clock (independent of the normal rasterizer clock). *VidEn* is the pixel read/write enable for the TASICs' video port. *VidRst* resets and initializes the video controller. Note that this is only one example implementation of a video interface. The encodings of video status and use of TIGC

ExtOP signals are programmable—the video controller, rasterizer glue chip, and TIGC video microcode just have to agree.

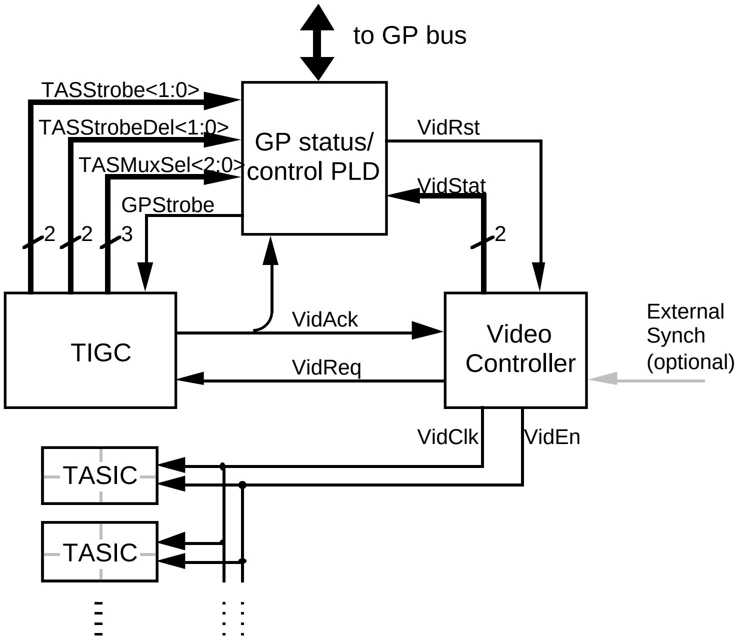


Figure 14: Video control interface (example).

The counters and finite-state machines in the Video Controller orchestrate video operation. Each time a new scanline, new field, or new frame is needed, the video controller asserts *VidReq*. The TIGC polls the *VidReq* input and reads the request type from *VidStat* via the GP status register using *TASMuxSel* to select the appropriate status bits. When the TIGC has recognized a request, it asserts *VidAck*, which clears *VidReq* and *VidStat* until the next video request. The GP status register PLD monitors *VidAck* to ensure that the TIGC is responding to video requests. If no *VidAck* occurs during a specified time interval, it can be inferred that the TIGC is operating incorrectly or is hung, necessitating a GP interrupt to reset the video subsystem.

The TIGC uses *VidStat* information to synchronize fields and frames with the video controller and to ensure that each frame is displayed at least once.

The video interface is left unconnected on non-video boards.

III GENERATING RASTERIZER COMMAND INPUT

When generating Rasterizer command input, the two IGCs are though of as a single logical entity. IGC commands consist of three types:

- **EIGC commands.** These perform computations on the SIMD array of pixel ALUs on the EMCs, and do setup for the two I/O port operations.
- **TIGC commands.** These control the TASICs, the EMC local port, and external DRAM memory.
- **RT Controller commands.** These control the operation of the semaphores which interlock the FIFOs and Sequencers, and operation of the Image Composition Controller. They pass through the input FIFOs and are intercepted by the RT Controller, but they are not executed by either Sequencer.

The commands of each type are defined in header files *<root>_opcodes.h* and *<root>_commands.h.* For EIGC commands, <*root*> = "EMC"; for TIGCcommands,<*root*> = "TAS". For RT Controller commands, <*root*> = "EMC" or "TAS", depending upon which semaphore is being controlled.

The opcode for each command is defined by a macro **I_<root>_<cmdname>** in the header file <*root*>*_opcodes.h*. Some EMC and TIGC sequencer commands require a supplementary opcode, which is defined by an additional macro **P_<root>_<cmdname>**, also in the header file<*root*>*_opcodes.h*. EIGC commands may also include either one or three coefficients (either just *C*, or *A*, *B*, and *C*); TIGC commands may include a single coefficient (*C*). The opcode and supplementary opcodes are each 32-bit quantities; the coefficients may each be either 32-bit (one word) or 64-bit (two word) quantities.

The <*root*>*_commands.h* header file contains C++ function definitions that generate entire IGC commands. The use of these commands is described in the following section. The <*root*>*_opcodes.h* and <*root*>*_commands.h* header files are generated by the PixelFlow IGC microcode assembler from microcode source (written by the hardware designers) contained in the file <*root*>*.ucode*. The assembler also generates files containing microcode: *EMC_ucode.h* contains microcode for the IGC sequencer, and *TAS_<type>_ucode.h* contains microcode for the TIGC sequencer (<type> refers to the particular video-port configuration on a given board, since TIGC microcode differs slightly depending on whether a board is a frame-buffer, frame-grabber, etc.). The microcode for the respective sequencer is declared as an initialized static array **unsigned EMC_ucode[]** or **unsigned TAS_<type>_ucode[]**. Portions of these files ar generated by hand, which specify the RT Controller commands.

III.1 Generating Commands with Inline C++ Functions

To generate IGC commands from within a C++ program, the user need not be concerned with the exact formatting of commands. The header files <*root*>*_commands.h* provide an easy way of generating command input. For each command **<root>_<cmdname>** described in the following sections, <*root*>*_commands.h* contains a C++ function:

void <root>_<cmdname>(*stream*, args,...)

These functions are normally declared as inline functions, so there is no procedure-call overhead for their execution. These functions generate all the instruction words for the command.

The first argument to each of these functions, *stream*, is a reference to an element of a special C++ class called **IGCStream**, defined in the header file *IGCStream.h***.** **IGCStream** contains routines for managing IGC command input buffers.[2] The *stream* argument must be defined prior to calling IGC command functions. When an IGC command function is called, it writes data to the stream, appending it to the current command buffer. When a buffer is filled, handlers within **IGCStream** automatically process the filled buffer and allocate a new one. This makes it possible to use IGC commands without worrying about block boundaries. **IGCStream** is implemented so that inline functions handle all routine operations. The only time a procedure call is needed is when a buffer fills and must be flushed (IGCStream is patterned after Unix *stdio* in this respect).

The remaining arguments to each IGC command are command-specific. They may be addresses into pixel memory, operand lengths, *C* or *ABC* coefficients, or other quantities as described in the command synopses below.

Error Checking

The IGC command functions and **IGCstream** provide several types of error checking and exception handling, which can be disabled under certain circumstances to increase speed. The three types of checking are:

1) **Need checking (buffer-space checking).** Ensuring that commands do not append data past the end of the current command buffer.
2) **Alignment checking.** Ensuring that double-word coefficients are aligned to even-word boundaries in the command buffer.
3) **Argument checking.** Ensuring that command functions are called with valid arguments.

All of these types of checking are performed by default. They can be disabled at compile-time by defining an appropriate symbol (described below) with a **#define** statement, prior to including the IGC command-function header file(s).

Need checking

Need checking or buffer-space checking, prevents command functions from appending data past the end of the current command buffer. They do this by comparing the number of words to be written with the number of free words left in the buffer. If there is not enough space, the flush and alloc handlers are called to create new space.

Performing "need" checking within each command function ensures that buffer overflows will never occur, but performs checks very frequently (every 1 – 5 words, on average). In some cases, it may be desirable to defeat need checking within command functions and to perform it for blocks of commands at a time. Need checking is disabled by inserting the line '**#define IGCNOCHKNEED**' prior to including the IGC command-function header file(s). In this case, the code that calls the IGC command functions must contain a statement of the form **s->need(*nwds*)** before the IGC command functions are called. Here **s** is a pointer to the current IGC stream, and ***nwds*** is the number of words that will be appended by all of the IGC commands to be called before the next **s->need** statement.

As an aid in determining ***nwds***, the header file **<root>_commands.h** defines macros for the number of words in each command. These have the form:

<root>_<cmdname>_<suffix>_len

For example, the macro for the number of words in the **EMC_TreeEqZero** command with ABC floating-point coefficients is:

```
#define  EMC_TreeEqZero_Lf_len      4
```

If alignment checking (see below) is enabled, commands with double coefficients may actually require one additional word, if a no-op needs to be added. The need checking code does this automatically.

Alignment checking

Alignment checking ensures that the coefficients of commands with double-word coefficients are aligned to even-word boundaries in the command buffer. This may or may not be necessary, depending on the operand alignment requirements of the graphics processor and IGC input structure. If alignment checking is enabled (default), a test is made within every IGC command function which uses double-word coefficients to determine the alignment of the current buffer address. If the address is odd-word aligned and the command has a two word opcode (I and P words), or if the address is even-word aligned and the command has a one-word opcode, then a special "ignore" instruction is placed in the buffer ahead of the current instruction, to align it the coefficients to an even-word boundary; this dummy instruction is ignored by the Stream Parsers.

Alignment checking is disabled by inserting the line '**#define IGCNOCHKALIGN**' prior to including the IGC command-function header file(s). In this case, the commands are placed in the buffer without alignment restrictions, and no dummy instructions are added.

Argument checking

Argument checking performs range checking on the arguments to IGC command functions. It ensures that pixel addresses and lengths are valid according to the command specification, but increases the run-time cost for executing each command.

Argument checking can be disabled by inserting the line '**#define IGCNOCHKARGS**' prior to including the IGC command-function header file(s). This eliminates the run-time cost of argument checking, but can cause unpredictable operation of the IGC (including hanging) if command functions are called with invalid arguments. We anticipate enabling argument checking during code development, and disabling it for production code.

TFIFO Commands

So far we have assumed that commands are directed to the RFIFO, the FIFO for normal rendering commands. Most commands can also be directed to the TFIFO, the FIFO for commands which move data into the image-composition buffers and initialize image-composition operations. These commands are generated by functions with a special T_ prefix, or having the form **T_<root>_<cmdname>(*stream*, args,...)**. These commands are used in the system software that operates the image-composition network. Not every command has a corresponding version with a **T_** prefix (particularly those which use the coefficient arguments *A*, *B*, and *C*, which cannot be stored in the TFIFO). The command descriptions below indicate which commands have **T_** versions and which do not; in particular, none of the commands which use the linear expression evaluator have a **T_** version, and some others also do not, as noted.

RT Controller Commands

RT Controller commands must be directed to the correct FIFO on the correct IGC, for the P or V function they are to perform. These are described below in Section VI.

III.2 Generating Commands with *opcodes.h* Macros

The functions in *<root>_commands.h* append command words to an IGC stream. For some purposes, such as assembly language programs, these functions cannot be used. The *<root>_commands.h* functions are built on top of a set of macros defined in *<root>_opcodes.h*. These macros define the command opcodes, and can be used in C or C++ programs to generate IGC command words explicitly (providing more flexibility), or in assembly language programs, where there is no alternative.

For a given command **<root>_<cmdname>**, defined in *<root>_commands.h*, *<root>_opcodes.h* contains a macro definition: **I_<root>_<cmdname>(args,...)**. This macro generates the opcode or I word for the command. Some commands require a supplementary opcode or P word. For these there is an additional macro definition, **<root>_P_<cmdname>(args,...)**, which generates this supplemental opcode. The I word is always the first word of a command. The P word, if necessary, is the second word. The opcode(s) may be followed by *C* or *ABC* coefficients. The I word contains bit fields which indicate the format of the remainder of the command.

Generating the command words using these macros is accomplished using the following steps:

1) Generate the main (I word) opcode, by evaluating the appropriate macro of the form **I_<root>_<cmdname>()**.

2) If the command is intended for the TFIFO, set bit 30 (the *TCmd* bit) of the opcode. Note that some commands, including those which use the tree coefficients, cannot be sent to the TFIFO.

3) Check bit 28 of the opcode, (the *Long* bit), or alternatively, check to see if the macro **P_<root>_<cmdname>()** is defined in <*root*>*_commands.h*,; if so, evaluate **P_<root>_<cmdname>()** to generate the supplementary opcode.

4) Add the coefficients if required; first check bit 27 (the *Coef* bit) of the I opcode, to see if the instruction uses coefficients; if so, check bit 26 (the *Linear* bit), to see whether one (*C*) or three (*A*, *B*, and *C*) coefficients are required, bit 25 (the *Double* bit) to see if the coefficients are 32- or 64-bit, and bit 24 (the *Float* bit) to see if the coefficients are integer or floating-point; finally, add the coefficients words. Note that some commands use the C coefficient in a way that is invisible to the user, so the above opcode bits must be checked even if the command is not an LEE command.

It is important that the command words be ordered as described above: *I-word*, *P-word*, *A-lsw*, *A-msw*, *B-lsw*, *B-msw*, *C-lsw*, *C-msw* (the ordering of the two words for 64-bit A, B, and C is reversed if the "endian" bit in the Interface Control Register is set , see **EMC_IFSpec** instruction below).

Some of the RT Controller commands require padding with no-op instructions in order to function correctly; these are inserted within the inline functions in*commands.h* , but must be explicitly inserted when using the *opcodes.h* macros.

More details on the meaning of the bits of the I-word and P-word opcodes are found in the IGC documentation.

IV EIGC COMMANDS

Commands for the EMC Sequencer can be divided into several categories:

1) Commands for the SIMD processor array
2) Commands for configuring the linear expression evaluator (LEE)
3) Commands for configuring the image-composition port
4) Commands for configuring the local port
5) Miscellaneous EIGC Sequencer commands

These are described in the following sections.

IV.1 Commands for the SIMD Processor Array

These commands are executed in parallel on the SIMD array of PEs: each PE performs the same operation at the same time. Each can write results only into its own pixel memory. PEs can communicate with their two neighbors within a panel, via a simple inter-ALU path. Each PE ALU contains a number of registers, of which the following are visible to the programmer (also see Figure 4 in Section I):

- **Enable Register.** Allows conditional writes into pixel-memory. Most instructions which generate arithmetic/logical results write those results into pixel memory only at *enabled* PEs (those PEs where the Enable register contains a '1').
- **Carry Register.** A one-bit register used for passing results between commands. It is not like the Carry register in a conventional ALU, since it does not reliably contain overflow information from arithmetic instructions and it need not be explicitly cleared before operations such as addition. In the instruction descriptions, the 'C' column indicates the effect on the Carry register: an 'X' indicates that the instruction corrupts the Carry register, and a '√' indicates that the instruction leaves a well-defined result in the Carry register (but only at Enabled PEs). A few instructions affect the Carry at *all* PEs (regardless of Enable), and these are explicitly mentioned.
- **S Register.** An 8-bit register used for accumulating intermediate results (mainly in more complex arithmetic calculations like division). In the instruction descriptions, the 'S' column indicates the effect on the S register: an 'X' indicates that the instruction corrupts the S register, and a '√' indicates that the instruction leaves a well-defined result in the S register. The S register is affected at all PEs.

All PEs execute the same instruction on each cycle, but memory writes for arithmetic operations are conditioned by each PE's Enable register (this is indicated by the symbol '≅' in the command synopses below). A PE whose Enable register has been cleared can be thought of as *disabled* or "turned off." For example, the standard polygon algorithm scan-converts a polygon by disabling pixels outside the polygon before loading color information to shade the polygon.

Specifying Memory Segments

The instruction set for the SIMD array of PEs resembles that of a simple microprocessor. Operands may include: arbitrary length signed or unsigned integers in pixel-memory; constant or linear expressions from the LEE; the Enable register, the Carry register, and the S register. Since many instructions invisibly corrupt the Carry and S registers, they may be used only when explicitly stated for calculations which require more than one instruction.

Integers defined in pixel-memory have their least significant byte at their lowest address. Memory segments are identified with the notation **mem[lsb : len]**. For example, a 32-bit (4-byte) integer in the memory segment at bytes 24 through 27 (with LSByte at byte 24) is denoted **mem[24 : 4]**. Contents of memory segments may represent unsigned or two's-complement signed integers. For many instructions, the computation is the same whether the contents of the memory segment are treated as signed or unsigned; for others it does matter, and these are noted. Each memory segment must lie wholly within one of the five partitions of pixel-memory (main memory, the two local port buffers, and the two image-composition port buffers). Maximum length of memory segments is 8 bytes for most instructions; it is greater than 8 for a few instructions noted below. Minimum length is 1 unless otherwise specified.

A few instructions operate on individual bits within pixel memory. Bits are identified with the notation **membit[byte : bit]**, where *byte* indicates the byte and *bit* is between 0 and 7, with 0 indicating the least significant bit and 7 indicating the most significant bit.

A section of the pixel memory address space, addresses 448-511, is reserved for use by the pixel-memory address base register. This register is loaded using the **EMC_PMABase** command, described below. Any address in the range 448-511 has 448 subtracted from it, and then is added to the base register contents.[1]

Specifying Tree Results

The LEE can be used in several different modes. In *constant* mode the LEE result is just F(*x*, *y*) = C; in *linear* mode, the LEE result is F(*x*, *y*) = A*x* + B*y* + C. The *x*,*y* values are in pixlets. Thus, for the multiple-sample-per-pixel screen organizations used for sample-parallel rendering (Figures 5B-5C), the A,B,C coefficients computed in terms of pixels must be converted to pixlet dimensions before being used as instruction arguments; this is done by multiplying the coefficient values by the pixel dimension in pixlets, normally 8.

The coefficients for the LEE expression can be: 32- and 64-bit signed integers, single-precision (32-bit) IEEE floating-point numbers, and double-precision (64-bit) IEEE floating-point numbers. For linear mode, coefficients also can be 32- or 64-bit fixed-point numbers. There is a separate command for each of these 10 combinations of coefficient type and tree mode, all having the same base command name but with a different two-letter suffix.[2] The LEE result is treated the same in each case, and is simply denoted as **tree** in the command set description, but its actual form is specified by the user according to which specific command is invoked. The tree result is computed for some fixed number of bytes specified as an argument to the instruction; it is identified with the notation **tree[len]**, where *len* is the number of bytes. If the tree result is less than *len* bytes in magnitude, it is sign-extended to *len* bytes; if it is greater than *len* bytes in magnitude, the upper bytes are

[1] It is also possible to specify distinct base addresses for the different types of operands; contact Eyles for details. The pixel-memory address refresh counter can also be under program control, and retains its value between instructions; contact Eyles for details.

[2] The fixed-point types are not yet implemented for most of the instructions. See Eyles if they are needed.

discarded. Maximum value for the *len* argument for commands which use the LEE is 8; minimum value for *len* is 1.[1]

The LEE performs only fixed-point calculations. If the LEE coefficients are floating-point, they are converted within the IGC. For constant LEE mode instructions, the number of fraction bytes used in the fixed-point representation is 0. For linear LEE mode instructions, the number of fraction bytes is user-specified, as the argument **FB** to the instruction, in the range 1 - 3; the maximum value of 3 provides the greatest precision, while smaller values allow faster instruction execution, but with some loss of precision, depending on the magnitude of the coefficient. If the LEE coefficients are fixed-point, the number of fraction bytes is user-specified in the range 1 - 3, but the fixed-point number is assumed to have one additional fraction byte; for example if the FB argument is 1, and the instruction uses 32-bit fixed-point coefficients, then the coefficients are assumed to be of the form 16.16 (16 integer bits, 16 fractional bits). This extra fraction byte is ignored for the A and B coefficients, but for the C coefficient it is used to add precision to calculations with subpixel offsets, as described below.

The LEE result, F(*x*, *y*) = A*x* + B*y* + C, is computed from the integer or fixed-point coefficients. For contant mode instructions and/or instructions with integer coefficients, the coefficients are truncated to integers and an exact calculation is performed. For linear mode instructions with floating- or fixed-point data types, the A and B coefficients have FB fraction bytes, and the C coefficient has FB+1 fraction bytes; the A*x* + B*y* + C calculation is exact to FB+1 fraction bytes. This extra byte of calculation is free, and adds precision for situations where the subpixel offset is non-zero. Finally, the LEE result is truncated to an integer; this trunction is always downward, so negative values are truncated away from zero. After truncation, the len least significant bytes of this integer are used in the instruction as **tree[len]**.

Several types of error can be introduced in this process, and the user must be familiar with them. Floating-point coefficients with magnitude smaller than 2-*FB* will be converted to 0, as expected, since the fixed-point equivalent *is* 0; denormalized numbers are also converted to 0. Floating-point exceptions such as infinities and NaN's are converted to 0 as well. When the magnitude of the LEE result is too large to be represented by a *len* byte integer, the upper bytes of the LEE result will be discarded and the sign of **tree[len]** will generally be incorrect; in the extreme, very large LEE results computed using floating-point coefficients will degenerate to zero when all the significant bits of the mantissa are shifted 8**len* bits to the left of the radix point and the *len* least significant bytes are zero. The truncations can have more subtle effects as well. For example, the linear expression *x*+*y*+1.99 will evaluate to the integer '1' at pixel *x*=0, *y*=0, while *x*+*y*-1.99 will evaluate to the integer '-2'. Coefficient truncation can also give suprising results; for example, the linear expression 0.1*x* + *y* + 0 will evaluate to the integer 0 at *x*=10, *y*=0, since 0.1 truncates to the fixed-point number 6553/216, so the LEE result at pixel (10,0) is 65530/216, and this is truncated to 0. On the other hand, -0.1*x* + *y* + 0 evaluates to the integer -1 at *x*-10, *y*=0. Since the LEE supports a maximum screen-size of 16k x 16k

[1] It is possible to provide LEE commands that allow tree results up to 16 bytes in length, at some expense. Contact Eyles for details.

pixlets, two fractional bytes of precision gives a maximum error in A*x* + B*y* + C equal to $2^{14} \bullet 2^{-16} + 2^{14} \bullet 2^{-16} + 2^{-16} = 2^{-1}$; this is less than the quantitization error introduced when the LEE result is truncated to an integer.

The LEE result can be viewed as an immediate operand. Some instructions which require only a single byte of data do not use the LEE. Their single-byte operand is identified with the notation **byte_data**. These instructions are useful because **T_** forms are defined (unlike instructions which use the LEE, which cannot be placed in the TFIFO), and because they conserve command words.

Command Prefixes

For most of the commands described below, two functions are defined in *EMC_commands.h*, corresponding to the ordinary (RFIFO) and TFIFO versions of the command. For example, the command denoted **SetEnab()** in the instruction descriptions has two functions defined:

EMC_SetEnab (*p*)

T_EMC_SetEnab (*p*)

Each function simply places the opcode for **SetEnab** in the IGC stream indicated by *'p'* (the argument *'p'* is omitted from the command descriptions to save space). Were **SetEnab** an instruction which requires the double IP opcode, the P-word of the opcode would be placed in the stream next. The macro **T_EMC_SetEnab (*p*)** sets the *TCmd*bit of the opcode so that the command is placed into the TFIFO, rather than the RFIFO.

Command Suffixes

Commands which use the LEE have names with the suffix **_Mt** in the instruction descriptions. This suffix is expanded to produce ten distinct functions in *EMC_commands.h*, corresponding to the different LEE **M**odes and coefficient **t**ypes: '**M**' can be either 'S', for constant mode, or 'L', for linear mode; '**t**' can be 'i', 'l,' 'f', 'd', 'p', or 'q', for 32-bit integer, 64-bit integer, 32-bit float, 64-bit float, 32-bit fixed-point, and 64-bit fixed-point, respectively (but **_Sp** and **_Sq** are not defined). For example, the command denoted **MemPlusTree(dst,src,len[,A,B],C)** in the instruction descriptions has ten inline functions defined in *EMC_commands.h* [4] :

EMC_MemPlusTree_Si (p,DST,SRC,LEN,C)

[4] Since 64-bit integers are not supported on platforms on which the simulator is currently implemented, the **_Sl** form of the macro is actually **EMC_MEMplusTREE_Sl(p,DST,SRC,LEN,Clo,Chi)**, where **Clo** and **Chi** are the low-order and high-order 32-bits of the C coefficient respectively. Similarly, the **_Ll** form is **EMC_MEMplusTREE_Sl(p,DST,SRC,LEN,Alo,Ahi,Blo,Bhi,Clo,Chi)**, and the **_Lq** form is **EMC_MEMplusTREE_Sl(p,DST,SRC,LEN,FB,Alo,Ahi,Blo,Bhi,Clo,Chi)** .

EMC_MemPlusTree_Sl (p,DST,SRC,LEN,C)

EMC_MemPlusTree_Sf (p,DST,SRC,LEN,C)

EMC_MemPlusTree_Sd (p,DST,SRC,LEN,C)

EMC_MemPlusTree_Li (p,DST,SRC,LEN,A,B,C)

EMC_MemPlusTree_Ll (p,DST,SRC,LEN,A,B,C)

EMC_MemPlusTree_Lf (p,DST,SRC,LEN,FB,A,B,C)

EMC_MemPlusTree_Ld (p,DST,SRC,LEN,FB,A,B,C)

EMC_MemPlusTree_Lp (p,DST,SRC,LEN,FB,A,B,C)

EMC_MemPlusTree_Lq (p,DST,SRC,LEN,FB,A,B,C)

Note that the argument lists are slightly different for the different functions. The **_S** types have only the C coefficient, whereas the **_L** types have A, B, and C coefficients. All the **_L** types, except **_Li** and **_Ll**, have an additional *FB* argument to specify the number of fractional bytes of precision; *FB* can be in the range 1 to 3.

Commands which use the LEE coefficients cannot be directed to the TFIFO (the TFIFO word is wide enough for only the I and P opcode words). Hence no commands of the form **T_EMC_<cmdname>_<Mt>()** are defined. A few other commands use the LEE coefficients for special purposes, rather than for LEE operation; these also do not have TFIFO forms, as specified in the comments or notes associated with the command.

Commands for the SIMD array can be divided into several categories: commands to modify the Enable register, commands to store the Enable register, arithmetic/logical commands, commands for advanced arithmetic operations, global commands, special-purpose commands, inter-pixelcommands, and miscellaneous commands.

Unpredictable behavior may occur if illegal arguments are given. Argument checking can be invoked at some cost in performance (see Section III above).

Commands to Modify the Enable Register

These commands alter the contents of the Enable register. Remember that most of the arithmetic/logical commands, to be described below, affect only pixels whose Enable register is set.

Command:		Synopsis:	S	C	Note:
ClrEnab	()	enable = 0			
SetEnab	()	enable = 1			
EnabInv	()	enable = !enable			
SetEnabPixel	(x,y)	enable = this is pixel(x,y)			25, 29
EnabPixel	(x,y)	enable &&= this is pixel(x,y)			25, 29
MemIntoEnab	(byte, bit)	enable = membit [byte:bit]			
CryIntoEnab	()	enable = carry			
BitTstHi	(src, bytedata)	enable &&= (mem[src :1] &bytedata)==bytedata			
BitTstLo	(src, bytedata)	enable &&= (~mem[src:1] &bytedata)==bytedata			
TreeEqZero_Mt	(len,[A,B,]C)	enable &&= (tree[len] ==0)			
TreeGEZero_Mt	(len,[A,B,]C)	enable &&= (tree[len] >=0)			
TreeLTZero_Mt	(len,[A,B,]C)	enable &&= (tree[len] <0)			
SNETree_Mt	[A,B,]C)	enable &&= (S_register !=tree(1))			
Mesh_Mt	(bits,[A,B,]C)	enable &&= ((tree[[len] %2^bits)==0)	X		19
MemEqByte	(src, byte_data)	enable &&= (mem[src:1]==byte_data)			
MemEqZero	(src, len)	enable &&= (mem[src : len] == 0)			
MemEqOnes	(src, len)	enable &&= (mem[src : len] == ~0)			
MemNEZero	(src, len)	enable &&= (mem[src : len] !=0)			
MemNEOnes	(src, len)	enable &&= (mem[src : len] != ~0)			
MemEqMem	(src0, src1,len)	enable &&= (mem[src0 : len] == mem[src1 : len])			
MemNEMem	(src0, src1,len)	enable &&= (mem[src0 : len] != mem[src1 : len])		X	
MemGEMem	(src0, src1,len)	enable &&= (mem[src0 : len] >= mem[src1 : len])		X	1
MemGTMem	(src0, src1,len)	enable &&= (mem[src0 : len] > mem[src1 : len])		X	1
Mem2GEMem2	(src0, src1,len)	enable &&= (mem[src0 : len] >= mem[src1 : len])		X	2
Mem2GTMem2	(src0, src1, len)	enable &&= (mem[src0 : len] > mem[src1 : len])		X	2
MemEqTree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] == tree[len])			
MemNETree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] != tree[len])		X	
MemLETree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] <=tree[len])		X	3
MemLTTree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] < tree[len])		X	3
MemGETree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] >=tree[len])		X	3
MemGTTree_Mt	(src, len,[A,B,]C)	enable &&= (mem[src : len] > tree[len])		X	3
EnabOrEqMem	(byte,bit)	enable ||= membit [byte : bit]			
EnabXorEqMem	(byte,bit)	enable ^^= membit [byte : bit]		X	
EnabAndEqCry	()	enable &&= carry			

Commands to Store the Enable Register

These commands store the Enable register into memory or the Carry register. Unlike most memory writes and Carry register loads, these writes occur regardless of the contents of the Enable register.

Command:		Synopsis:	S	C	Note:
EnabIntoCry	()	carry = enable		√	
EnabIntoMem	(byte,bit)	membit [byte: bit] = enable			
MemOrEqEnab	(byte,bit)	membit [byte: bit] ||= enable			
MemAndEqEnab	(byte,bit)	membit [byte: bit] &&= enable			

Arithmetic and Logical Commands

These commands operate on signed LEE results and signed or unsigned integers in pixel memory.

Commands may have up to 3 possible pixel-memory operands: the destination operand *dst*, and a source operand, *src*, or two source operands, *src0* and *src1*. Unless otherwise noted, the destination and source operands need not be distinct, but they must not *partially* overlap; that is, if they overlap at all, their LSBs must align. Unless otherwise noted, *dst* and *src0* must have the same length, *dlen*.. The source *src1* may have a different length, *slen*.. The following default rules apply to commands which have separate length arguments:

dlen	==	*slen*	:	overflow and underflow are discarded
dlen	>	*slen*	:	carry/borrow is rippled through all bytes of destination; *src* may be considered unsigned or signed, as noted
dlen	<	*slen*	:	higher order bytes of src are ignored

As described above, minimum and maximum values for the *len*, *dlen* and *slen* arguments are 1 and 8, respectively (unless otherwise noted). Segment lengths must be contained within one of the partitions of pixel memory (the 256-byte main partition, or one of the 32-byte communication-buffer partitions).

Writing of the result of these commands is conditioned by the Enable register; that is, no result is written to memory unless the Enable register contains a 1 prior to the instruction (this is indicated by the "≅" symbol in the instruction synopsis). For commands which have a defined effect on the Carry register, it is correct only for Enabled pixels, although it may be affected at all pixels. For commands which affect the S register, it is affected regardless of the Enable register setting.

Command:		**Synopsis:**			**S**	**C**	**Note:**
LoadPixel	(x, y, dst, len, value)	at pixel(x,y) mem[dst : len]	 ≅	 value			24, 25
Clear	(dst, len)	mem[dst : len]	≅	0			
Set	(dst, len)	mem[dst : len]	≅	~0			
BitClr	(dst, byte_data)	mem[dst : 1]	&≅	~byte_data			
BitSet	(dst, byte_data)	mem[dst : 1]	|≅	byte_data			
BitXor	(dst, byte_data)	mem[dst : 1]	^≅	byte_data			
ByteIntoMem	(dst, byte_data)	mem[dst:1]	≅	byte_data			
TreeIntoMem_Mt	(dst, len,[A,B,]C)	mem[dst : len]	≅	tree [len]			
TreeClmpIntoMem_Mt	(dst,dlen,s len,[A,B,]C)	mem[dst : dlen]	≅	tree [slen]		X	17
TreeIntoS	((A,B,)C}	S_register	=	tree{1}	√		
MemIntoS	(src)	S_register	=	mem[src : 1]	√		
Copy	(dst, src, len)	mem[dst : len]	≅	mem[src : len]			12, 21
Swap	(src0,src1, len)	mem[src0 : len]	<->	mem[src1 : len]			18, 23
Inc	(dst, src, len)	mem[dst : len]	≅	mem[src : len] +1		√	22
Dec	(dst, src, len)	mem[dst : len]	≅	mem[src : len] -1		X	
Merge	(dst, src, mask)	mem[dst:1]	≅	(mem[dst:1] & ~mask) |(mem[src:1] & mask)			20
LSL	(dst, src, len)	mem[dst : len]	≅	mem[src : len] << 1		√	5
LSL4	(dst, src, len)	mem[dst : len]	≅	mem[src : len] << 4	X		5
LSR	(dst, src, len)	mem[dst : len]	≅	mem[src : len] >> 1		√	5
LSR4	(dst, src, len)	mem[dst : len]	≅	mem[src : len] >> 4	X		5
ASR	(dst, src, len)	mem[dst : len]	≅	mem[src : len] >> 1 (signed)	X	√	6
ASR4	(dst, src, len)	mem[dst : len]	≅	mem[src : len] >> 4 (signed)	X	X	6
ROL	(dst, src, len)	mem[dst : len]	≅	mem[src : len] << 1 w/ carry		√	4
ROR	(dst, src, len)	mem[dst : len]	≅	mem[src : len] >> 1 w/ carry		√	4
Invert	(dst, src, len)	mem[dst : len]	≅	~mem[src : len]		X	
Negate	(dst, src, len)	mem[dst : len]	≅	-mem[src : len]		X	2
AbsVal	(dst, src, len)	mem[dst : len]	≅	|mem[src : len] |	X	X	2
MemPlusMem	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0 : dlen] +mem[src1:slen]		√	1, 7, 22
MemClmpPlusMem	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0 : dlen] +mem[src1:slen]		√	1, 7,15, 22
MemMinusMem	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0 : dlen] - mem[src1:slen]		X	1, 7
MemClmpMinusMem	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0 : dlen] - mem[src1:slen]		X	1, 7, 27
MemPlusMem2	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0: dlen] +mem[src1 : slen]	X	X	2, 8
Mem2ClmpPlusMem2	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0: dlen] +mem[src1 : slen]	X	X	2, 8,16
MemMinusMem2	(dst, src0, src1, dlen, slen)	mem[dst : dlen]	≅	mem[src0 : dlen] -mem[src1:slen]	X	X	2, 8
MemAndMem	(dst, src0, src1, len)	mem[dst : len]	≅	mem[src0:len] &mem[src1 : len]			
MemOrMem	(dst, src0, src1, len)	mem[dst : len]	≅	mem[src0:len] |mem[src1 : len]			
MemXorMem	(dst, src0, src1, len)	mem[dst : len]	≅	mem[src0:len] ^mem[src1 : len]			
MemPlusTree_Mt	(dst, src, len,[A,B,]C)	mem[dst : len]	≅	mem[src : len] + tree [len]		X	

TreeMinusMem_Mt	(dst, src, len,[A,B,]C)	mem[dst : len] ≅ tree[len] - mem[src : len]		X	
MemAndTree_Mt	(dst, src, len, [A,B,]C)	mem[dst : len] ≅ mem[src : len] & tree [len]			
MemOrTree_Mt	(dst, src, len, [A,B,]C)	mem[dst : len] ≅ mem[src : len] | tree [len]			
MemXorTree_Mt	(dst, src, len, [A,B,]C)	mem[dst : len] ≅ mem[src : len] ^ tree [len]			
Min	(dst, src0, src1, len)	mem[dst : len] ≅ MIN(mem[src0:len], mem[src1:len])		X	1
Min2	(dst, src0, src1, len)	mem[dst : len] ≅ MIN(mem[src0:len], mem[src1:len])		X	2
Max	(dst, src0, src1, len)	mem[dst : len] ≅ MAX(mem[src0:len], mem[src1:len])		X	1
Max2	(dst, src0, src1, len)	mem[dst : len] ≅ MAX(mem[src0:len], mem[src1:len])		X	2
OvFix	(dst, len)	if (carry) mem[dst : len] ≅ ~0			
ClrCry	()	carry ≅ 0		√	
CryIntoMem	(byte,bit)	membit [byte : bit] ≅ carry			
MemIntoCry	(byte,bit)	carry ≅ membit [byte : bit]		√	

Advanced Arithmetic Commands

These commands support advanced arithmetic operations such as multiplication, division, and square roots.

Command:		Synopsis:			S	C	Note:
SLoad	(byte_data)	S_register	=	byte_data	√		
WriteS	(dst)	mem[dst:1]	≅	S_register			
MulUUn	(dst,src0,src1,dlen,slen)	mem[dst:dlen]	≅	mem[src0:1]*mem[src1:slen]	X		1, 9, 11
MulUSn	(dst,src0,src1,dlen,slen)	mem[dst:dlen]	≅	mem[src0:1]*mem[src1:slen]	X	X	9, 14
MulSSn	(dst,src0,src1,dlen,slen)	mem[dst:dlen]	≅	mem[src0:1]*mem[src1:slen]	X	X	9, 14
SqRoot	(dst, src)	mem[dst:1]	≅	square_root(mem[src:2])	X	X	1, 23, 25
RootStep1	(dst, src, len, bit)	carry	≅	mem[dst:len] >= mem[src:len] | (1<<bit)		√	1, 23,28
RootStep2	(dst, src, len, bit)	if (carry) mem[dst:len] mem[src:len]	 -≅ |≅	 mem[src:len] | (1<<bit) 1<<(bit+1)		X	1,23, 25,28
Divide	(dst, src0, src1, len)	mem[dst+1:len-1] mem[dst:1]	≅ ≅	mem[src0:len]/mem[src1:1] remainder	X	X	1, 7
DIvStep1	(dst, src, dlen, slen)	carry	≅	mem[dst:dlen] >= mem[src:slen]		√	1, 23, 26
DivStep2	(dst, src, len, aux, bit)	S_register if (carry) mem[dst:len] mem[aux:1]	= -≅ |≅	carry mem[src:len] 1 << bit	√	X	1, 23
InvSqStep	(dst, src, dlen, slen)	if (S_register & 1) mem[dst:dlen]	 +≅	 mem[src:slen]		X	1, 7, 23
DivAddSub	(dst,src0,src1,dlen,slen)	mem[dst:dlen]	≅	mem[src0:dlen] +/- mem[src1:slen]		X	1, 7
DivShift	(dst, src, len)	carry S_register mem[dst:1] mem[dst:len]	≅ ≅ ≅ <<≅	! ((mem[dst+dlen-1:1] >>7)&1) (S_register<<1)|carry mem[src:1] 1	√	√	1, 7
ClampFix	(dst, len, left)		≅				1
ByteToShort	(dst, src)	mem[dst:2]	≅	mem[src:1]<<4	X		1

Some of these instructions use the S register to accumulate results of complex arithmetic operations, such as divides, that require more than one instruction to implement. **SLoad** and **WriteS** are used for initializing and writing a result. Care must be taken to avoid mistakenly corrupting the S register. Similarly, the Carry register can also be used to pass information between instructions, so care must be taken not to corrupt its value.

The **Mul**n** instructions perform various flavors of 1-byte by N-byte multiplication. The first **U** or **S** indicates that the 1-byte operand is **U**nsigned or **S**igned, respectively; the second **U** or **S** indicates that the N-byte operand is **U**nsigned or **S**igned. To multiply two N-byte unsigned numbers, **MulUU** is called N times (using add instructions and temporary buffers to form the final product). To multiply two N-byte signed numbers, **MulUS** is used N-1 times and **MulSS** is used once (to multiply the MSByte of one operand times the other operand). The *dst* and *src0* operands must not overlap at all; *dst* and *src1* may not partially overlap, and *src0* and *src1* may overlap in any way.

SqRoot finds the 8-bit square root of a 16-bit integer. **RootStep1** and **RootStep2** are designed to be used in square root routines for longer operands.

Divide divides an N-byte unsigned integer by a 1-byte unsigned integer. It assumes that the quotient will fit in N-1 bytes (no overflow). It can be used alone, or to get an approximate quotient to begin an iterative algorithm.

DivStep1 and **DivStep2** are for a compare-then-subtract division algorithm. **DivStep1** will zero-extend *dst* if necessary; **DivStep2** need not do this, since the test in **DivStep1** will pass only if the higher-order bytes of *src* were zero. **InvSqStep** is meant to facilitate an inverse square root algorithm.

DivAddSub and **DivShift** are for a subtract-then-correct division algorithm. **DivAddSub** adds if the LSB of the S register and the Carry register are both 0 on input, or subtracts if the LSB of the S register and the Carry register are both 1 on input. **DivShift** copies the byte at address *src* into the LSByte of *dst*, andthen shifts *dst* left one bit; the original MSB of *dst* is inverted and put into the Carry register; also, the S regsiter is shifted left one bit and the new Carry put into its LSB.

Enable Stack Commands

It is useful to maintain a "stack" of Enable register values. There is no hardware Enable stack; but it is possible to use a single byte of pixel memory as a 256-deep Enable stack, if it is assumed that each time this stack is "pushed" that *fewer* pixels will be enabled. Thus the value in each pixel's stack counter says how many times the stack must be "popped" for this pixel to be Enabled. The following instructions are used for manipulating this Enable stack:

Command:		***Synopsis:***	***S***	***C***	***Note:***
ResetEnab	(addr)	enable = 1 mem[addr:1] = 0x00			
PushEnab	(addr)	if(enable) mem[addr:1] = mem[addr:1]+1			
PopEnab	(addr)	enable = (mem[addr:1]!=0x00) mem[addr:1] ≅ mem[addr:1]-1 enable = ! enable			
RestoreEnab	(addr)	enable = (mem[addr:1]==0x00)			
XorEnab	(addr)	enable = (! enable) && (mem[addr:1]==0x00)			
BreakEnab	(addr,n)	if(n>0) mem[addr:1] ≅ n-1 enable = 0			

These compute global maxima and minima over *all* enabled

Global Commands

These commands use the global-OR signal to perform global computations; these are computations that are performed over all pixels in the rasterizer region, rather than locally at each individual pixel.

Command:		***Synopsis:***		***S***	***C***	***Note:***
GMax	(dst, src, len)	mem[dst : len]	≅ MAX{mem[src : len] }	X	√	1, 23
GMin	(dst, src, len)	mem[dst : len]	≅ MIN{mem[src : len] }	X	√	1, 23

These compute global maxima and minima over *all* enabled pixels in the SIMD array. **EMC_GMax** computes maximum value of **mem[src:len]** over all *enabled* pixels (treating it as an unsigned value) and writes this maximum into **mem[dst:len]** for all Enabled pixels. **EMC_GMin** (not yet implemented) behaves similarly. The Enable register is not disturbed. Operands may not overlap at all. On exit, Carry=1 at pixels which had the maximum value (Carry=0 at other pixels); Carry is not disturbed at pixels which were not Enabled.

ALU Register Save/Restore

These commands are used to save the ALU state to pixel-memory, and restore it.

Command:		***Synopsis:***	***S***	***C***	***Note:***
ALUSave	(dst)	// Save ALU state into mem[dst:6] // (destroys ALU state)	X	X	
ALURstr	(dst}	// Restore ALU state from mem[dst:6]	√	√	

EMC_ALUSave and **EMC_ALURstr** must be used in sequences on commands in the TFIFO. This is analogous to saving and restoring processor state in an interrupt service routine. A sequence of TFIFO commands may be thought of as an interrupt for the EMC sequencer, and is guaranteed to be executed conitguously, without being interrupted by RFIFO commands, provided there is no command which can block the TFIFO. The beginning of such a sequence must include an **EMC_ALUSave** command, to save the states of the pixel-ALUs in some dedicated 6-byte area of pixel-memory. Saved information includes the S register contents, and the Carry and Enable registers (M and R registers and ALU condition codes are also saved, but these are invisible to the user anyhow). Just before a sequence of TFIFO commands can be interrupted by RFIFO commands (the TFIFO is blocked by a **WaitXfer** or a **T_EMC_P*** command) , an **EMC_ALURstr** command is used to restore the ALU state. TFIFO interrupt sequences are described in detail in Section VI.

Special-Purpose Commands

These commands support special tasks for Rendering.

Command:		Synopsis:			S	C	Note:
Sample	(base,xreg, xsub, yreg, ysub)						
FEdge_Mt	(len,[A,B,]C)	enable carry	= ≅	1 (tree[len] < 0)		√	
MEdge_Mt	(len,[A,B,]C)	enable carry	&&= ≅	carrybar (tree[len] < 0)		√	
ZCmp_Mt	(src, len,[A,B,]C)	enable carry	&&= ≅	carrybar (mem[src : len] >= tree[len])		√	3
FLoad_Mt	(dst, len,[A,B,]C)	enable mem[dst : len-1] Sregsiter	&&= ≅ ≅	carry first len-1 bytes of tree [len] MSByte of tree[len]	√		
MLoad_Mt	(dst, len,[A,B,]C)	mem[dst-1:1] mem[dst : len-1] Sregsiter	≅ ≅ ≅	S_register first len-1 bytes of tree [len] MSByte of tree[len]	√		
MLoad1_Mt	(dst,[A,B,]C)	mem[dst-1:1] Sregsiter	≅ ≅	S_register tree[1]	√		
LLoad_Mt	(dst, len, [A, B,] C)	mem[dst-1:1] mem[dst : len]	≅ ≅	S_register tree [len]	X		
TblStep_Mt	(dst, len, [A, B,] C)	mem[dst:len] enable S_register	≅ &&= =	tree[len] (S_register!=0x00) S_register - 1	√	X	
TblEntry_Mt	(dst, src, dlen, slen,[A,B,]C)	if (mem[src : slen] ==tree[slen] mem[dst : dlen] = tree[slen+dlen] >>8*slen					13

EMC_Sample is used in applications for which multiple samples of an anti-aliasing kernel reside in pixel-memory simultaneously. It changes the pixel-memory address base register and and region/subpixel offset, thereby combining the functions of **EMC_PMABase** and **EMC_Offset**, which are described in the following sections, into a single command, to save execution cycles and input bandwidth.

EMC_[FM]Edge, EMC_ZCmp and **EMC_[FML]Load** are hand-tuned commands for drawing convex polygons very rapidly. These commands save cycles by pipelining micro-operations between commands, and should allow the rasterizer to process up to 3 million triangles per second. **EMC_FEdge** and **EMC_MEdge** behave similarly to **EMC_TreeGEZero**, except that **EMC_FEdge** sets the Enable register prior to evaluating the sign of the LEE result (so it's used for the first edge of a convex polygon) and the sign-bit of the current instruction's LEE result is saved in the Carry register and the Enable register is updated at the beginning of the *following* instruction. **EMC_ZCmp** behaves like **EMC_MemGETree** and is used for the Z comparison, but it also updates the Enable register based on the sign-bit contained in the Carry register for the last **EMC_MEdge** instruction. **EMC_FLoad**, **EMC_MLoad** and **EMC_LLoad** behave like **EMC_TreeIntoMem** and are used for loading the color and Z buffers, except they leave the last byte of the LEE result in the S register and it is written into pixel memory at the beginning of the *following* instruction. Since the commands use the S and Carry registers to pass information between commands, they should be called in the following sequence; any intervening commands must not disturb the S and Carry registers:

EMC_FEdge
EMC_MEdge (one or more)
EMC_ZCmp
EMC_FLoad
EMC_MLoad (zero or more)
EMC_LLoad

The pixel memory operands for the sequence of **EMC_[FML]Load** commands must form an ascending sequence of contiguous addresses in pixel memory address space. Minimum value of the *len* argument for all these commands is 2, except it is still 1 for **EMC_LLoad**. Talk to Eyles for more details and assistance in changing or augmenting this set of commands.

EMC_TblStep and **EMC_TblEntry** are used for broadcasting lookup tables to the SIMD array. **EMC_TblStep** is much faster, but less general; to use it, the table key is loaded into the S Register (using **EMC_MemIntoS**), and then the table entries are sent in order using **EMC_TblStep_Si** or **EMC_TblStep_Sl**. This requires that there be a table entry for all possible values of the key, that they be sent in order, and that the key be no more than 8 bits. **EMC_TblEntry** is slower but much more general, since it sends a key value and an entry value each time it is called.

Inter-Pixel Commands

These commands allow communication among the 32 PEs in a panel. The relative screen positions of the PEs in a given panel depend upon the rasterizer configuration, as shown in Figures 5A-C. No communication between different panels is possible, except indirectly by use of the communications ports.

Command:		***Synopsis:***	***S***	***C***	***Note:***
PixSwap{1,2,3,...,7}	(src0, src1, len)	see below			21
PixCopyDn{1,2,4,8}	(dst, src, len)	see below			21
PixCopyUp{1,2,4,8}	(dst, src, len)	see below			21

The various versions of **EMC_PixSwapN** are used to exchange operands between PEs within a panel. The PEs to be swapped must N positions apart. The operands must occur in pairs, each pair consisting on a *src0*-operand and a *src1*-operand. A *src0*-operand is **mem[src0:len]** at a PE for which the Carry register is set; a *src1*-operand is

mem[src1:len] at a PE for which the Enable register is set; thus, all PEs for which either the Carry *or* the Enable register is set are affected. Each *src1*-operand must lie **N** PE positions higher than its corresponding *src0*-operand. Different pairs of operands can span overlapping ranges of PEs; however, if the end-points of the ranges touch, that is, if any *src0*-operands and *src1*-operands lie in the *same* PE, then **mem[src0:len]** and **mem[src1:len]** must not overlap at all (normally *src0* and *src1* may be identical), else unpredictable results will occur. The argument *len* can lie in the range 1 - 32. The contents of the Enable and Carry registers are not affected.

Additional functionality is available upon request.

EMC_PixCopyDnN copy operands from PEs to other PEs which are N positions lower in the panel. **EMC_PixCopyUpN** copy operands from PEs to other PEs which are N positions higher in the panel.*Src*-operands need not be marked (eg. with the Carry or Enable register). *Dst* operands are marked with the Enable register; in other words, as usual, only Enabled PEs are affected. Any data shifted in from the end of a panel is all 0's. For example, if all PEs are Enabled and **EMC_PixCopyDn4** is executed, then **mem[dst:len]** at PE[i] gets the value of **mem[src:len]** from PE[i+4], for i = 0 - 27, and PEs 28 - 31 get 0's written into **mem[dst:len]**.

Notes:

1) The contents of the memory segment(s) are assumed to represent unsigned integers.

2) The contents of the memory segment(s) are assumed to represent two's-complement signed integers.

3) The contents of the memory segment is assumed to represent an unsigned integer, and an unsigned comparison is performed.

4) Rotate through carry (carry is shifted into MSB or LSB, LSB or MSB is shifted into carry).

5) Logical shift (zero is shifted into MSB or LSB, LSB or MSB is shifted into carry).

6) Arithmetic shift right (MSB is sign-extended). For EMC_ASR, LSB is shifted into carry.

7) The contents of the *src* memory segment is assumed to represent an unsigned integer; it is zero-extended if *dlen* > *slen*.

8) The contents of the *src* memory segment is assumed to represent a signed integer; it is sign-extended if *dlen* > *slen*.

9) Arguments must obey *dlen* > *slen*.

10) **Membit[byte:bit]** is written for all pixels, regardless of the value of the Enable register. *Dst* and *src* may point to the same memory location.

11) Product is zero-extended if *dlen* > *slen* + 1.

12) The *dst* and *src* memory segments may overlap in any way.

13) Does lookup and write for a table entry in one instruction. If the *slen* LSBytes of **tree** match **mem[src : slen]**, then the next *dlen* bytes of **tree** are written into **mem[dst : dlen]**. Only affects enabled pixels, and the Enable register is not disturbed. Not allowed in TFIFO.

14) Product is sign-extended if *dlen* > *slen* + 1.

15) The result is clamped to all 1's if overflow occurs. There is a penalty in execution time.

16) The result is clamped to the maximum respresentable positive value if overflow occurs, to the mimimum respresentable negative value if underflow occurs. There is a penalty in execution time.

17) **Mem[dst:dlen]** is clamped to all 1's if **tree[slen]** is larger than 2**_dlen_ -1, to 0 if it is negative. Requires *slen* > *dlen*.

18) The contents of the two memory segments are interchanged. No temporary register is required.

19) Clears Enable unless *bits* LSBs of tree result are 0's. *Len* is implicitly *bits/8 + 1*. Range for *bits* is 1-63.

20) Bits in *dst* are replaced by corresponding bits from *src*. *Mask* defines which bits are replaced.

21) Maximum value for *len* is 32.

22) Carry is set if overflow occurred.

23) The memory operands may not overlap in any way.

24) The operation occurs only at pixel (x,y) and only if its Enable is set. Range for *len* is 1 - 4. *Value* must be a 32-bit integer and only the *len* least significant bytes are used. Enable is not disturbed.

25) No **T_** version (for the TFIFO) of this command exists. (This applies by default to all commands with the **_Mt** suffix).

26) The contents of the *dst* memory segment is assumed to represent an unsigned integer. Arguments must obey *slen* >= *dlen*; *dlen* is zero-extended if *slen* > *dlen*.

27) The result is clamped to 0 if underflow occurs. There is a penalty in execution time.

28) Range for *bit* is 0 - 7 for EMC_RootStep1 and 0 - 6 for EMC_RootStep2.

29) Pixel(x,y) is defined within context of how the tree is configured.

IV.2 Commands to Configure the Linear Expression Evaluator

When the linear mode version of an LEE instruction is used, the tree result **tree[len]** is computed as F(x,y) = A*x* + B*y* + C for each processor in the SIMD array. In order to process the entire display screen, it is necessary to move the rasterizer region to different portions of the screen. In order to sample geometry at sub-pixel offsets for anti-aliasing (if anti-aliasing is done using multiple passes), it is necessary to offset the region by fractions of a pixel.

These functions are accomplished using the instruction:

Command:		***Synopsis:***	***S***	***C***	***Note:***

EMC_Offset	(xreg,xsub,yreg,ysub	// Set rasterizer region to position given
		// by *xreg*, *yreg* with subpixel offset given
		// by *xsub*, *ysub* (in 64'ths of a pixel)			

Xreg and *yreg* specify the region-offset (the upper left-hand corner of the rasterizer region); values must lie in the range 0 - 16383 (values need not be multiples of the rasterizer region size, but normally they would be). *Xsub* and *ysub* specify the sub-pixel offset, in 64'ths of a pixel; values must lie in the range -127 to 127 (so offset is in the range -1 and 63/64 pixels to +1 and 63/64 pixels). For example, to position the rasterizer at pixel 512, 128, with sub-pixel offset -0.5, 1.625, the command:

EMC_Offset(*p*, 512, -32, 128, 104)

would be used.

Note that these commands, like all tree commands, do arithmetic on *pixlet* values, as opposed to *pixel* values. Thus, for the multi-sample-per-pixel organizations (Figures 5B-5C) used for sample-parallel rasterization, it must be remembered that a pixel equals 8 pixlets (in linear dimension), so the arguments to EMC_Offset must be 8-times the pixel values intended. Also, for sample-parallel rasterization,the *xsub* and *ysub* arguments normally would be set to 0.

The region- and sub-pixel offset values affect linear mode LEE instructions only; this is because the A and B coefficients are effectively zero for constant mode instructions, and the offset is accomplished by adding multiples of the A and B coefficients to the C coefficient.

Since EMC sequencer commands which use the LEE cannot be placed into the TFIFO, it makes no sense to reconfigure the LEE within the TFIFO. Consequently, no **T_EMC_Offset** command is defined.

Since the LEE includes logic on both the EMCs and the IGC, the EMC portion of the LEE must also be configured. This need normally be done only at system or application initialization time. The EMC portion of the LEE is configured using a set of special commands for loading the LEE configuration registers on the EMCs; these commands specify the pixels within the rasterizer region for which each EMC is responsible. The details for doing this are found in the EMC documentation and an IGC library function will be supplied. This syntax of these commands is given here for completeness; they are not intended to be used by most users of this document:

Command:		***Synopsis:***	***S***	***C***	***Note:***
EMC_CfgInit	(numemcs)	// Initialize the EMC ID registers (for *numemcs*)			
EMC_RegLoad	(chip, reg, value)	// Load specified register on specified EMC			

EMC_RegLoads0	(1stchip, reg, value, n)	// Load register *reg* on a sequence of *n*
		// EMCs starting with EMC # *1stchip*;
		// load register with *value*.			

EMC_RegLoads1	(1stchip, reg, 1stvalue, n)	// Load register *reg* on a sequence of *n*
		// EMCs starting with EMC # *1stchip*;
		// load register with sequence of values
		// starting with *1stvalue*.			

EMC_RegLoads2	(chip, 1streg, value, n1,N2)	// Load *n1* + *n2* registers on specified EMC
		// Loads *n1* registers, starting with *1streg*, with
		// *value*, then loads next *n2* registers
		// with *value* + 1.			

EMC_GRegLoad	(reg, value)	// Load register *reg* with *value*, on all EMCs.			

IV.3 Commands to Configure the Image-Composition Port

The following commands are used to configure the image-composition port:

Command:		***Synopsis:***	***S***	***C***	***Note:***
EMC_RevCopy	(dst, src, len)	// Copy mem[src:len] to mem[dst:len] // while reversing the byte-order.			23

EMC_ICEnds	(leftend, rightend)	// Specify if this is either end of machine
		// or in the middle
		// also initializes pixel offset/stride to 0			

EMC_ICInit	(leftend, rightend)	// same as EMC_ICEnds			

EMC_ICPort	(l2rmode,r2lmode,nbytes,zbytes)	// Initialize both IC paths, as shown.
		// Total number of bytes per pixel = *nbytes*,
		// z-bytes per pixel = *zbytes*			

EMC_L2RInit	(mode,nbytes,zbytes,offset,stride)	// Set IC L2R port as specified
		// Total number of bytes per pixel = *nbytes*,
		// z-bytes per pixel = *zbytes*			

EMC_IR2LInit	(mode,nbytes,zbytes,offset,stride)	// Set IC R2L port as specified
		// Total number of bytes per pixel = *nbytes*,
		// z-bytes per pixel = *zbytes*			

The image compostion network consists of two unidirectional pathways, the "left-to-right" (L2R) path and the "right-to-left" (R2L) path, implemented on a single physical daisy chain of simultaneous bi-directional signals. The two virtual uni-directional paths on a set of adjacent boards can be closed into a loop by specifying the leftmost and rightmost board in the set. This is done using **EMC_ICEnds**; *leftend* is set to 1 if this rasterizer is the left-end of the loop, and *rightend* is set to 1 if this rasterizer is the right-end of the loop; both arguments are set to 0 otherwise. Each **EMC_ICEnds** command must be accompanied by an **EMC_Alive** command with the same arguments (see Section VI). These commands are normally issued once, at machine initialization time, but the more twisted programmer can imagine reconfiguring the Image Composition network topology within an application. Also, at board reset, the "alive" and "ends" registers are all cleared (equivalent to issuing an **EMC_Dead** and **EMC_ICEnds**(0,0) command).

Prior to each transfer operation, the image-composition port is initialized, and its operating mode specified, using **EMC_ICPort**, **EMC_L2RInit**, and **EMC_R2LInit** commands. **EMC_ICPort** initializes both the L2R and R2L paths; if more flexibility is needed, **EMC_L2RInit** and **EMC_R2LInit** are used to initialize the two paths separately.

For **EMC_ICPort**, **EMC_L2RInit**, and **EMC_R2LInit**, *nbytes* specifies the total number of bytes per PE to be transferred; valid range is 1 to 32. *Zbytes* specifies the number of bytes in the Z-buffer (used in the compositing calculation, see below); *zbytes* should be 0 if *mode* does not specify a compositing operation (see below), otherwise, the valid range is 1 - 8 for **EMC_{L2R,R2L}Init**, or 3 - 6 for **EMC_ICPort**; also, *zbytes* must be less than or equal to *nbytes*.

The arguments *l2rrmode* and *r2lmode* for **EMC_ICPort**, or the argument *mode* for **EMC_{L2R,R2L}Init**, specify the mode for the transfer in each direction (the codes marked with * duplicate function of other codes, and are used only for testing):

Mode:	Buffer Write	Output Stream
0x0	—	input
0x1	input	input
0x2	composite	input
0x3 *	buffer	input
0x4 *	—	input
0x5 *	input	input
0x6 *	composite	input
0x7 *	buffer	input
0x8	—	composite
0x9	input	composite
0xA	composite	composite
0xB *	buffer	composite
0xC	—	buffer
0xD	input	buffer
0xE	composite	buffer
0xF *	buffer	buffer

Each mode is defined by (1) the output stream from the Image Composition network, and (2) the data (if any) written *into* the Image Composition buffer. The possibilities for the output stream are (1) *input* - the input stream, (2) *buffer* - the values read from the local image-composition transfer buffer, or (3) *composite* - the composited pixel values (from the input stream and the transfer buffer). For the buffer-write, there is a fourth choice, *null*, meaning that nothing is written back into the image-composition transfer buffer; the *buffer* choice wouldbe used only for testing since *null* has the same (non-)effect by writing the buffer contents back into itself.

The Image Composition network has a one-bit data path for each panel. Normally, all 32 PEs in each panel are acessed, in the order 0 through 31. It is possible to perform the transfer for only some of the PEs, and/or to access the PEs in a different pattern. If fewer than 32 PEs per panel are to be accessed, the *nbytes* argument to **EMC_InitXfer** (see Section VI) is set proportionately less than the *nbytes* argument to **EMC_ICPort** or **EMC_{L2R,R2L}Init**. For example, if the *nbytes* argument to **EMC_InitXfer** is half the *nbytes* argument to the initialization command, then PEs 0 - 15 are accessed.

EMC_{L2R,R2L}Init allow some flexibility in PE access pattern. The *offset* argument, in the range 0 - 31, specifies which PE is accessed first. The *stride* argument, in the range 0 - 3, specifies the stride between PEs accessed in log2 form (so 0, 1, 2, and 3 correspond to strides of 1, 2, 4, and 8 PEs, respectively). For example, if*offset* = 4, *stride* = 3, and the *nbytes* argument to **EMC_InitXfer** is set to access only 1/8'th of the PEs, then PEs 4, 12, 20, and 28 are accessed. Once **EMC_{L2R,R2L}Init** are used with non-default values for *stride* and *offset*, transfers subsequently initialized using **EMC_ICPort** commands will have the same *stride* and *offset* values, unless an **EMC_ICEnds** command is issued, which has the effect of restoring *offset* and *stride* to their default values.

The Z-buffer, containing the unsigned Z-value for the composite computation, must lie at byte addresses 0 to *zbytes* - 1 in the transfer buffer, and stored in reverse order, with MSByte at address 0 and LSByte at address *zbytes* - 1. The command **EMC_RevCopy** is provided for convenience in placing a byte-reversed Z value into the Image Composition buffer; normally *dst* is set to L2RBASE or R2LBASE and *len* is set to *zbytes*-1.

When doing the Z-comparison for a compositing operation, the smaller Z value wins; that is, the pixel with the smaller (unsigned) Z value is forwarded to output and/or written into the transfer buffer. In a tie, when the Z-values are the same, the pixel from the input stream "wins".

After the image-composition port is initialized using **EMC_ICPort** or **EMC_{L2R,R2L}Init**, the transfer must be initiated using an RT Controller command, such as **T_EMC_InitXfer** (see Section VI). [1] After the transfer is initiated, it actually begins at some non-deterministic time in the future, based on the status of the other boards involved in the transfer. It is important that once a transfer has been initiated, that the transfer buffers not be addressed, and that the configuration not be disturbed (by another **EMC_ICPort**, **EMC_{L2R,R2L}Init**, or **EMC_InitXfer** command), until the transfer has completed. This is done using semaphores and the **T_EMC_WaitXfer** command, using the protocol shown in Section VI.

IV.4 Commands to Configure the Local Port

The local port consists of the local input port and the local output port; these two ports may operate simultaneously and independently. Each must be initialized, prior to exercising it with TAS commands. This is done using the following commands:

Command:		***Synopsis:***	***S***	***C***	***Note:***
EMC_LPortIn	(nbytes, mode)	// Initialize local input port			
EMC_LPortInLoop	(nbytes, mode)	// Initialize local input port, for loopback mode			
EMC_LPortOut	(nbytes, mode)	// Initialize local output port			
EMC_LPWeave	(dst, src0, src1, len)	// Weave two segments of memory into one	X	X	
EMC_LPUnWeave	(dst0, dst1, src, len)	// Unweave a segment of memory into two	X	X	

For each command, *nbytes* specifies the total number of bytes per PE to be transfered; valid range is 4 to 32 (values less than 4 can be used with special TAS microcode, talk to Eyles

[1] Under certain conditions, it is not necessary to issue initialization commands prior to each transfer. This is because the PE counters wrap around; so if they wrap around exactly to 0, and the operating modes are to be the same, then no initialization command is needed. The saved overhead can be significant for some algorithms. See Eyles for details.

and Molnar for details). The argument *mode* specifies whether the local port accesses PEs in panel-major (*mode*=0) or panel-minor (*mode*=1) order.

Execution of an **EMC_LPortIn[Loop]** or **EMC_LPortOut** command initializes the local port controller, and sets the input or output "mark" register to the *current* value of the pixel-ALU Enable register. The appropriate **TAS_** command(s) must then be executed to exercise the local port and input or output data (See Section V below). Once either port is initialized and TAS commands begin exercising the port, the port's buffer in pixel memory must not be accessed by **EMC_** commands, nor may the port be re-initialized, until the port operation is complete; this interlocking is accomplished using the semaphores (see Section VI below). The input and output ports can be configured and used completely independently.

The mark registers specify a subset of PEs to be involved in the local port operation. For example, if a triangle is scan-converted, so that only the pixels within that triangular region are enabled, and then an **EMC_LPortOut** command is issued, any subsequent local port output will use only the pixels within the triangular region, that is, only the PEs which were enabled at the time **EMC_LPortOut** was executed. If the local output port is exercised after all marked PEs have been accessed, zeroes are output; if the local input port is exercised after all marked PEs have been accessed, nothing happens (no data is written into the buffer). Most of the **TAS_** commands which use the local port check an "active" signal from the EMCs and terminate after all marked pixels have been accessed.

With *mode*=0, the local port accesses the PEs in panel-major order; that is, all marked PEs in Panel 0 are accessed, in PE order (PEs 0 through 31), then all marked PEs in Panel 1, and so on through Panel 7. Normally, this means that the marked pixels are accessed in scan-line order (pixel-major, sample-minor), with the PEs representing all samples of a pixel accessed together. When *mode*=1, the PEs are accessed in panel-minor order. <<STEVE, FILL THIS IN >>

The **EMC_LPortInLoop** command configures the local input port, similarly to **EMC_LPortIn**, except that the port is set for "loopback" mode, in which the output port is connected to the input port. This command is normally used only for chip test; it could also be used to transfer pixel data among the panels within each EMC.

EMC_LPWeave takes segments *mem[src0:len]* and *mem[src1:len]* and weaves them into a single segment *mem[dst:2*len]*. *Src0* is packed into the even bits of *dst*, *src1* is packed into the odd bits of *dst*.. **EMC_LPUnWeave** undoes this process. It takes the even bits of *mem[src:2*len]* and creates *mem[dst0:len]*; similarly, the odd bits of *src* are packed together to form *mem[dst1:len]*. << JGE, THESE SEEM TO BE BROKEN NOW >>

For more information on using the local port, see the description of TIGC command set in Section V below.

IV.5 Miscellaneous EMC Sequencer Commands

Miscellaneous Commands

The following are miscellanous commands that execute on the EMC Sequencer:

Command:		***Synopsis:***	***S***	***C***	***Note:***
EMC_Ignore	()	// No operation, does not get loaded into FIFO			
EMC_NoOp	()	// No operation for sequencer (goes thru FIFO)			
EMC_NoOp2	()	// Same as NoOp, with 2 word (I and P) opcode)			
EMC_MetaNoOp	()	// No operation, but flagged as *meta* instruction			
EMC_EMCInit	()	// Initialize EMCs (for simulations)			
EMC_RefClr	()	// Set pixel-memory refresh counter to zero			
EMC_PMABase	(base_address)	// set pixel-memory address base register to // specified value			
EMC_PipeFlush	()	// Idle for ?? cycles (to flush the EMC // control and LEE pipelines)			
EMC_Hang	()	// Hangs the sequencer in a tight loop // (for debugging purposes)			
EMC_FlogDBus	()	// Toggles memory data busses on all PE's in // the SIMD, for worst-case power			
EMC_EOrWait	()	// Waits fo EOrH to settle			
EMC_EOrTest	()	// Waits fo EOrH to settle, tests it, // sets appropriate bit in CSR			

EMC_NoOpand **EMC_NoOp2** are no-operation commands for the EIGC. Both take one cycle to execute, but perform no action. **EMC_PipeFlush** is essentially a long NOOP. It inserts enough idle cycles into the EMC control and LEE pipelines to clear them of any previous instructions or data. It is used to ensure that a set of EIGC commands have actually executed, prior to initiating TIGC or Image-Composition Network operations which use data.

EMC_PMABase sets the pixel-memory address base register. This offset is applied to any pixel-memory addresses in the range 448-511 (after 448 is subtracted). For example, **EMC_PMABase**(100) followed by **EMC_TreeIntoMem**(460,2) writes to *mem[112:2]*.

<< **PRELIMINARY>> EMC_EOrTest** is used to sample EOrH (the global-OR of the Enable register). It inserts enough idle cycles to allow EOrH to settle based on the results of the previous commands. It then tests EOrH, and sets the appropriate "sticky" bit in the GP's CSR. This bit must be cleared by the GP prior to issuing a successive **EMC_EOrTest** command.

Commands to Initialize the EIGC Sequencer

The EIGC Sequencer must be initialized after power-up or if the Rasterizer is reset (this may be necessary if either IGC hangs due to faulty microcode or an invalid opcode, or if it is waiting on an external handshake signal). The initialization sequence involves putting the sequencer into a special mode (RMode), loading the microcode store, setting the program counter to 0, and exiting the special mode. The user of this docuement need not be concerned with the details of doing this, which are given in the IGC documentation; an IGC library function will be supplied to initilaize the sequencers. The commands are described here for completeness:

Command:		***Synopsis:***	***S***	***C***	***Note:***
EMC_IFSpec	(Rlim,Tlim,Endian)	// Set interface control register			
EMC_RModeOn	()	// Put the sequencer in RMode			
EMC_RModeOff	()	// Cause the sequencer to exit RMode, // and set sequencer program counter to 0			
EMC_MCWrite	(addr, low32, high32)	// Load the 64-bit word (specified by two 32-bit // words) into the specified microcode location.			

EMC_MCRead	(addr)	// Set the sequencer program counter to 'addr'.
		// This command is also used to read
		// microcode memory during chip testing,
		// but this function is not available during
		// normal operation)			

On power-up, the interface control register is undefined; it should be initialized, using **EMC_IFSpec**, before any other commands are sent, else the FIFO flags will behave unpredictably. The arguments*Rlim* and *Tlim* define the high-water marks for the R and T FIFOs. The status flag *ERFullH* is asserted whenever the number of commands in the RFIFO is greater than *Rlim*; similarly, *ETFullH* is asserted whenever the number of commands in the TFIFO is greater than *Tlim*. *Endian* is set to 0 if the low-order 32-bit word of a 64-bit coefficient is input into the IGC before the high-order word; it is set to 1 if the high-order word comes first.

To load the sequencer microcode store, it first must be put into *RMode*, either by doing a Rasterizer reset (which places both IGCs into *RMode* by default) or by using the **EMC_RModeOn()** command. Next, an **EMC_MCWrite()** command is used to load each 64-bit word of microcode required. Finally, the **EMC_RModeOff()** command is used to reset the sequencer's program counter to 0 and put the sequencer in normal mode.

When the sequencer is in *RMode*, it can acccept only commands which do not execute microcode, such as **EMC_RModeOn**, **EMC_MCWrite**, **EMC_MCRead**, **EMC_RModeOff**, **EMC_RefClr**, **EMC_Offset**, **EMC_PMABase**, **EMC_IFSpec** and the various “meta” commands. Other commands cause undefined results, including the possiblity of hanging the sequencer. The **EMC_MCWrite()** command must never be issued except when the sequencer is in *RMode*.

The standard location for the EMC sequencer microcode is in the file *EMC_ucode.h*, in an the initialized array **static unsigned EMC_ucode[]**; this file is generated by the EMC microcode assembler *asmEMC*, from microcode source provided by the IGC hardware designer, as described above.

These commands can also be used to reload the EMC sequencer microcode on-the-fly, if it becomes necessary to use more than one version of the microcode in the same application.

Initialization of the rasterizer also requires initializing the TIGC Sequencer and configuring the linear expression evaluators on the EMCs. Initializing the TIGC Sequencer is accomplished using precisely the same command sequences described in this section, but with the TIGC sequencer versions of the commands: **TAS_RModeOn**, **TAS_MCWrite**, **TAS_MCRead**, and **TAS_RModeOff.** Configuring the LEE is described above in Section IV.2.

No **T_** version of **EMC_MCWrite** and **TAS_MCWrite** exists, since these commands use the C coefficent. Thus microcode loading cannot be done via the TFIFO.

IV.6 Commands for the Rasterizer Glue Chip

The following commands are used for manipulating the “scratch registers” on the Rasterizer glue-chip.

The most important of these can be used for handshaking with the GP:

Command:		***Synopsis:***	***S***	***C***	***Note:***
SetScratch{1,2}	()	// Set glue-chip sync bit which tells the GP to // do something			
WaitScratch{1,2}	()	// Wait for the GP sync bit to be cleared			

EOrReadMem	(src , len)	// Read *len* bytes of pixel memory through EOr
		// syncing with the GP after every byte
		// Assumes that exactly one PE is Enabled			

EOrReadTree_Mt	(len [, A, B] , C])	// Read LEE result through EOr			

IV.7 Test Commands

A number of other commands exist; these were used for hardware debug and/or test vector generation. The intrepid heart will have to look at the IGC microcode source files to be enlightened about these commands. They are summarized here.

The following are primarily for simulation and fault-coverage of the IGC chip: **EMC_TESTSEQ1**, **EMC_TESTSTIN**, **EMC_TESTLoopCount**, **EMC_PMATest**,

EMC_PMABase3, **EMC_TESTRefCnt**, **EMC_FIFOEntry**, **EMC_MCReadC**, **EMC_TreeBigMem**, **EMC_MemPlusTree3**.

The following are primarily for simulation and fault coverage of the EMC chip: **EMC_EMCInit**, **EMC_SimBoot**, **EMC_FlogEOr**, **EMC_ALUTest**, **EMC_EOrMemTst**, **EMC_EOrMemTstPartial**, **EMC_FlogDBus**.

The following commands are obsolete. They have been replaced by equivalent commands with better names:

EMC_EnabAndEqMem	replaced by	EMC_BitTstHi
EMC_EnabAndEqMemBar	replaced by	EMC_BitTstLo
EMC_PixCopy{1,2,4,8}	replaced by	EMC_PixCopyDn{1,2,4,8}

V TIGC COMMANDS

The command set for the TIGC is similar to that of the EIGC, with two major exceptions: (1) *A*, *B*, *C* coefficients cannot be not used (TASICs have no linear expression evaluator) and (2) TIGC commands generally take many more cycles to execute than EIGC commands.

TIGC commands execute in parallel on the array of TASICs to perform data-transfer operations between the EMCs' local port buffers, the GP bus, and external texture/video memory. Data transfers can be local within module-local (*i.e.* data is transferred between the EMCs and SDRAMs of each module) or they can be global (*i.e.* data is sent from one module to other modules over the inter-module TASIC ring).

TIGC commands can be divided into the following categories:

1) Commands for reading/writing external memory
2) Commands for communicating with the GP
3) Commands for configuring the video port
4) Miscellaneous TIGC commands

These are described in the following sections.

V.1 External Memory Organization

Before describing the TIGC command set, we first give further details on the organization of the external memory (texturing) subsystem of the rasterizer board. As described in Section II.4, the rasterizer is divided into four modules containing eight EMCs, two TASICs (one logical TASIC), and eight SDRAM memories. The TASICs' and SDRAMs' main purpose is to perform high-speed image-based texturing and other table-lookup operations which are difficult or time-consuming on the EMCs. The driving problem is to perform mipmap texturing on all the pixels in a 128x64-pixel region in a small multiple of the time required to composite the region's pixels.

Mipmap textures require eight lookups per pixel: four from each of two adjacent resolution levels in the mipmap pyramid. This can be accomplished by providing eight independent memories: any lookup will access each memory exactly once if they are interleaved 2x2 at each resolution level (two resolution levels (even/odd) • 2x2 interleaves at each resolution level = eight independent banks or memories).

Although only a modest amount of texture storage is needed (16 or so MBytes), the bandwidth required to texture at these speeds requires a large number of memories. 32 SDRAMs were judged to provide a reasonable balance between bandwidth and cost/space/power requirements, etc. The 32 SDRAMs provide an aggregate peak bandwidth of 32 • 100 MBytes/sec = 3.2 GByte/sec. This is sufficient to look up eight four-byte values per each pixel in a 128x64-pixel array in about 140 µsec. Equivalently, about 7,000 regions can be mipmapped per second.

Normally, texture data will be stored redundantly in each of the four modules (since any pixel generally must be able to access any texel). Replicating the memory in this fashion increases lookup bandwidth four-fold, but unfortunately, does not increase texture storage correspondingly.

The unit of operation in the memory subsystem is for each pixel in a module to look up a value from each of the eight SDRAM memories in the module. Initially, each pixel processor calculates eight independent row and column addresses for the eight memories in its module and copies these into its local port buffer. The texture subsystem reads these addresses from the local-port and applies them to the eight SDRAM memories, then reads the eight corresponding data values and loads them into the pixel-processors' local-port input buffer. These operations are depicted schematically in Figure 15.

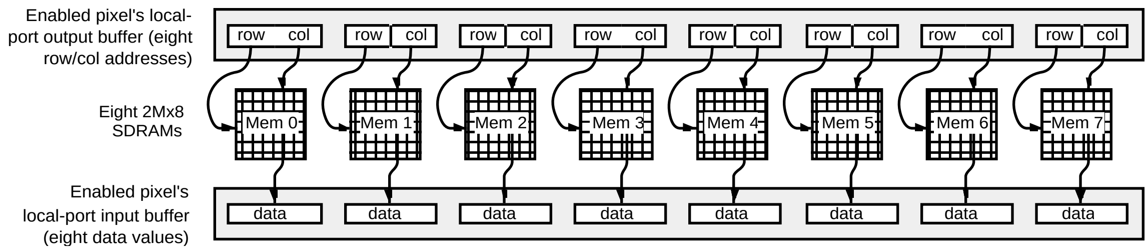

Enabled pixel's local-port output buffer (eight row/col addresses)
row
col
Eight 2Mx8 SDRAMs
Mem 0
Mem 1
Mem 2
Mem 3
Mem 4
Mem 5
Mem 6
Mem 7
Enabled pixel's local-port input buffer (eight data values)
data

Figure 15: Logical operation of an external memory read operation. Eight data values are read corresponding to the eight row/column addresses stored in each enabled pixel in the local-port output buffer.

Specifying independent row/column addresses for each memory allows complete flexibility in addressing: the eight memory chips can be viewed as disjoint memories or as interleaved memories in a variety of patterns (1x8, 8x1, 2x4, 4x2, 2x2x2). For mipmap textures, frame-buffer storage, etc., it is most natural to consider the eight chips as implementing a pair of 2x2-interleaved memory systems or *panels* (not to be confused with an EMC panel), as shown in Figure 16. If each panel stores a 2D array of texels, a filtered-texture lookup requires precisely one access to each of the four banks ina panel, no matter where the texture is sampled. Having two panels allows two filtered texture lookups to be performed simultaneously. This is desirable for MIP-map texturing. For simpler forms of texturing, a pair of lookups can be done simultaneously (perhaps for two supersamples or pixels in different regions).

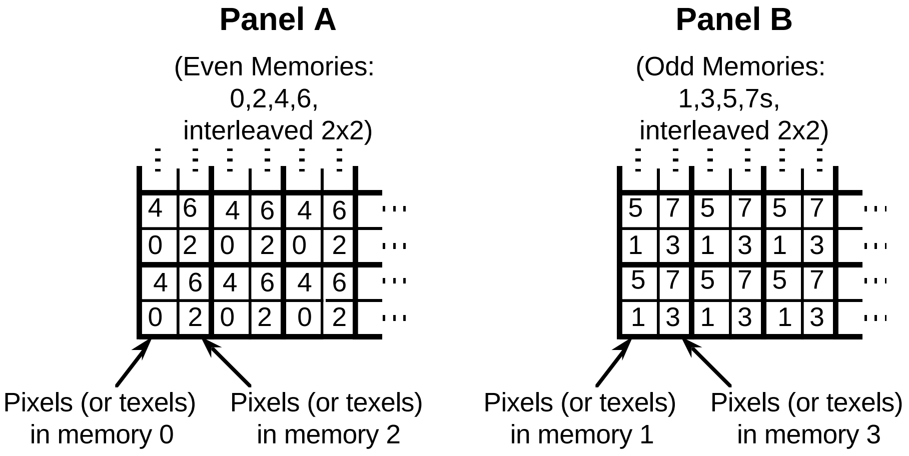

Panel A
(Even Memories: 0,2,4,6, interleaved 2x2)
4 6 4 6 4 6 ...
0 2 0 2 0 2 ...
4 6 4 6 4 6 ...
0 2 0 2 0 2 ...
Pixels (or texels) in memory 0
Pixels (or texels) in memory 2
Panel B
(Odd Memories: 1,3,5,7s, interleaved 2x2)
5 7 5 7 5 7 ...
1 3 1 3 1 3 ...
5 7 5 7 5 7 ...
1 3 1 3 1 3 ...
Pixels (or texels) in memory 1
Pixels (or texels) in memory 3

Figure 16: Double 2x2 interleaving organization of external memory. Each memory in a panel stores every second pixel (texel) in every second row.

The SDRAMs contain one other complicating feature. Each SDRAM memory contains two internal memory banks. Each bank can be accessed in a method analogous to fast-page mode of a conventional DRAM: a row address is given first followed by potentially multiple column addresses. The SDRAMs contain internal column address counters that permit burst accesses: a single column address is given which initiates a burst of writes or reads to/from consecutive memory locations. The two-bank design of the SDRAMs allows the row address command to be overlapped with burst reads/writes to/from the other bank, essentially hiding the row access time. If banks are accessed alternately, this can nearly double the memory bandwidth of each chip. One way to guarantee that banks will be hit alternately is to store data redundantly in the two banks on each chip and have successive lookups access the data in whichever bank is next. The most significant bit of each memory's row address indicates one of two banks on each SDRAM chip. The TASICs allow this bit to be manipulated independently of the other address bits.

For 16 Mbit SDRAMs, row addresses are 12 bits long and column addresses are 9 bits long.[1] Address bit 11 selects which of the two banks in each SDRAM is to be accessed. This means the effective size of the texture array in each module is 2^{12} = 4096 rows by 2^7 = 128 columns per memory chip (assuming 4-byte texels). If the memories are organized as two panels of 2x2 interleaved memories, as described above, the size of the overall texture array, therefore, is 2x8192x256 texels. If we shuffle address bits around to make the panels as square as possible, this provides two 2048x1024 panels of 4-byte texels, as shown in Figure 17.

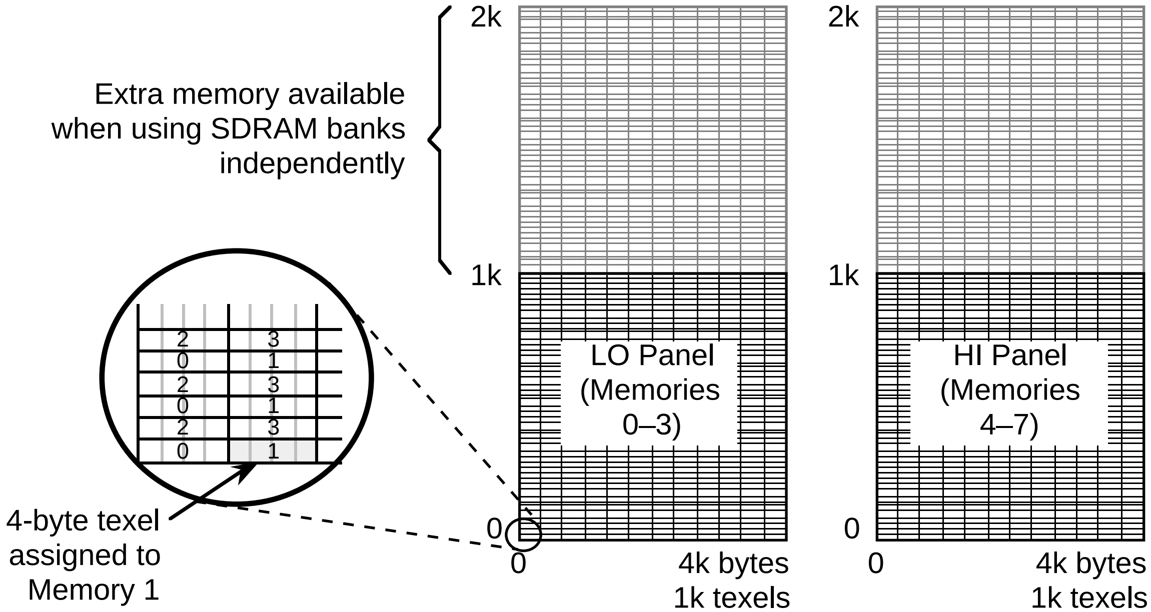


Figure 17: Dimensions of external memory arrays allocated as two 2x2-interleaved panels. Each cell represents a logical 4-byte texel.

[1] For 64-Mbit SDRAMs, which may be used in the future, row addresses are 14 bits (bits 13 and 12 specify 1 of 4 banks); column addresses are 9 bits.

Data alignment. Due to the bit-sliced nature of the TASICs and for simplicity in allocating external memory, external memory is allocated in 4-byte quantities (called *logical texels*), with the 4 bytes stored at adjacent column addresses in the same memory, as shown in Figure 17. Data types shorter than 4 bytes are supported by overlaying multiple texture (or pixel) maps whose data lengths sum to 4 bytes. For example, a texture map with two-byte texels might occupy bytes 0 and 1 of each logical 4-byte texel, while a second texture map with one-byte texels occupies byte 2 of the same texel.

This leads to the following alignment restrictions:

- 4-byte data must be aligned to 4-byte boundaries (two low col. address bits = 0)
- 3-byte data must be aligned to 4-byte boundaries
- 2-byte data must be aligned to 2-byte boundaries
- There is no alignment restriction for 1-byte data

Address Sources. As mentioned before, textures are normally stored redundantly on each of the four modules, so pixels in any module can look up any texture coordinates. Texture memory within each chip is addressed to the individual byte. A texel is specified by the row address and by the column address of its least-significant byte. These texture addresses are applied to the SDRAMs by the TASICs and can be generated in one of three ways:

1) The EMCs may calculate the addresses and send them to the TASICs over the local port (normally done for texture reads).
2) The GNI can send the addresses to the TASICs using the inter-module TASIC ring.
3) The internal address generator on the TASICs can provide the addresses (normally done for texture writes and video reads/writes). (more details below)

The EMCs must generate addresses when doing random texture lookups (only the PEs know from what location a texel is to be fetched). They would be used to generate addresses for writes as well, except this would exact a performance penalty. The EMC local-port bandwidth is matched with the SDRAM bandwidth when doing texture reads. However, when doing texture writes, addresses and data would have to flow the same direction. Furthermore, the local-port output buffer is not big enough to store 8 addresses and 8 data values. So, instead, it is possible to have the TASICs generate the addresses, allowing data to flow at full bandwidth.

The GNI can send addresses to the TASIC by means of the inter-module TASIC ring. This is useful when the GP desires to retrieve data from texture memory. The TASIC address generator could be used for this purpose as well.

When addresses are sent to the TASICs, they are stored in the Address Corner Turner (ACT). An address for each SDRAM is stored for each EMC. This implementation is transparent in some instructions, but can be used explicitly in others to supply a base address which is modified by the M register. The M register is essentially an up counter, and thus can generate sequences of addresses. More detail is provided below.

Address Output Modes. The address supplied to each SDRAM depends on the TASIC address port mode. The address for each panel is formed by one of the following:

1) The ACT output

2) The M register output

3) The ACT output xored with the M register output

This is shown in Figure 18.

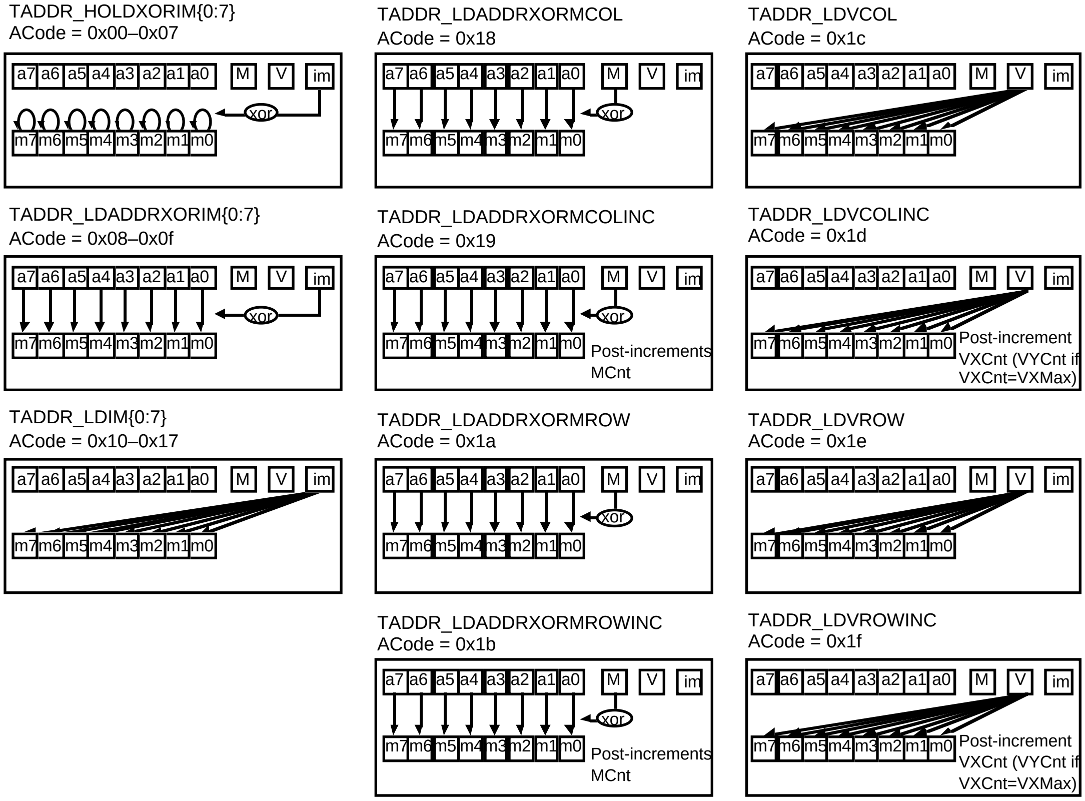


Figure 18: Address output modes. ***a7-a0*** **refer to the eight ACT outputs.** ***m7-m0*** **refer to the eight SDRAM output addresses.** ***M*** **refers to the output of the** ***MCnt*** **register and crossbar.** ***Im*** **refers to the set of eight immediate address registers.** ***V*** **refers to the** ***VYCnt***/***VXCnt*** **registers (for video address generation—will be described later).**

Different modes are used by different data transfer commands. The particular mode for each command is described in Section V.3 below.

Data Replication. As mentioned above, to read (or write) SDRAMs as rapidly as possible, the two banks within each SDRAM chip must be accessed alternately. Since alternate accesses to the same memory will be from different pixels, the only way to guarantee that banks can be hit alternately is to replicate the data yet again. The simplest way to do this is to store the same data in both banks of the same chip. A more complex method, but one that makes texture writing more efficient, is to store identical data in bank 0 of one panel and bank 1 of the other panel[1]. We have adopted this approach. We call replicated texture maps that are stored in bank 0 of Panel A and bank 1 of Panel B *even* and texture maps that are stored in bank 1 of Panel A and bank 0 of Panel B *odd*.

Replicated textures are slower to write than non-replicated textures, since the data must be stored twice. They also reduce the amount of texture storage by a factor of two, so they should only be used when needed.

Non-replicated textures do not require the even/odd distinction.

V.2 Configuration Commands

The following commands set up the various TASIC address and control registers in preparation for data transfer commands.

Commands to Configure the Address Generator Output

These commands configure the address generator by enabling/disabling the Address Corner Turn (ACT) and M crossbar register for all combinations of panels.

[1] When writing texture memory, it is desirable to keep both panels busy. If textures are replicated in both banks of the same chip, writing a given texture will only affect one of the two panels. To keep the other panel busy, two texture maps would have to be paired together and both written at the same time. This has ugly software implications. With the method we have adopted, the data is replicated across panels, so a single texture write affects both panels; the redundant data can be written in a single pass without wasting one panels’ memory cycles.

Command:		***Synopsis:***
TAS_SetAPortModeAll		Configure address generator, enable ACT and M for both panels
TAS_SetAPortModePanA		Configure address generator, enable ACT and M for panel A only
TAS_SetAPortModePanB		Configure address generator, enable ACT and M for panel B only
TAS_SetAPortModeACT		Configure address generator, enable ACT for both panels
TAS_SetAPortModeACTPanA		Configure address generator, enable ACT for panel A only
TAS_SetAPortModeACTPanB		Configure address generator, enable ACT for panel B only
TAS_SetAPortModeM		Configure address generator, enable M for both panels
TAS_SetAPortModeMPanA		Configure address generator, enable M for panel A only
TAS_SetAPortModeMPanB		Configure address generator, enable M for panel B only
TAS_SetAPortModeNone		Configure address generator, disable ACT and M

Commands to Select the Xfer Mode

The inter-module TASIC links and the configurable datapath on the TASICs allow transfers to be done in several ways (refer to Figure 11 for a pictorial description of these modes):

- ***Xfer1to1* mode:** Point-to-point transfers from the EMCs of module *src* to the SDRAMs of module *dst*.
- ***Xfer1toN* mode:** Broadcasts from the EMCs of module *src* to the SDRAMs of all modules.
- ***XferNtoN* mode:** Parallel transfers from the EMCs to the SDRAMs of all four modules.
- ***XferGto1* mode** Point-to-point transfers from GNI to the SDRAMs of module *dst*.
- ***XferGtoN* mode:** Broadcast from the GNI to the SDRAMS of all modules.
- ***Xfer1toG* mode:** Point-to-point transfer from EMCs of module src to the GNI.

Commands to select one of these transfer modes are as follows:[1]

[1] The 1to1 and Gto1 modes operate differntly in TASIC Rev1.0 and Rev2.0. In the Rev1.0 TASIC, these modes perform writes to all four modules, but zero the data going to the ACT/DCT of the non-dst modules. In Rev2.0, these modes perform writes only to the ACT/DCT of the dst module. The address pointers of the ACT/DCT in non-dst modules are not incremented, so the ACT/DCT's of the four modules are will not be "in synch" until four such transfers have been performed (one to each module). This change was made to allow more efficient texture paging from the GNI.

Command:		***Synopsis:***
TAS_SetXferMode1to1	(*src*,*dst*)	// Select *Xfer1to1* mode, specify *src* and *dst* modules
TAS_SetXferMode1toN	(*src*)	// Select *Xfer1toN* mode, specify src module
TAS_SetXferModeNtoN	( )	// Select *XferNtoN* mode
TAS_SetXferModeGto1	(*dst*)	// Select *XferGto1* mode, specify *dst* module
TAS_SetXferModeGtoN	( )	// Select *XferGtoN* mode
TAS_SetXferMode1toG	(*src*)	// Select *Xfer1toG* mode, specify *src* module

Commands to Set the *MCnt* and *MSel* Registers

The *MCnt* register is a 32-bit presettable up-counter, which allows an ordered sequence of addresses to be generated beginning at any desired value. It is followed by a programmable 64 to 32 crossbar switch, which allows *MCnt* bits to be permuted when forming row and column addresses. This counter and crossbar are shown in Figure 20.

The crossbar is configured by means of a 192-bit register called *MSel*. *MSel* is divided into 28 6-bit chunks. Each chunk corresponds to one row or column address bit and selects the source of that bit from one of the 64 inputs. *MSel*<5:0> selects the source for column address bit 0, *MSel*<11:6> selects the source for row address bit 0, and so forth.

The reason the crossbar has 64 instead of 32 inputs is that the upper 16 inputs are hardwired to 1 and the next 16 inputs are hardwired to 0. Selecting these inputs allows certain address bits to be held constant regardless of the contents of *MCnt* .

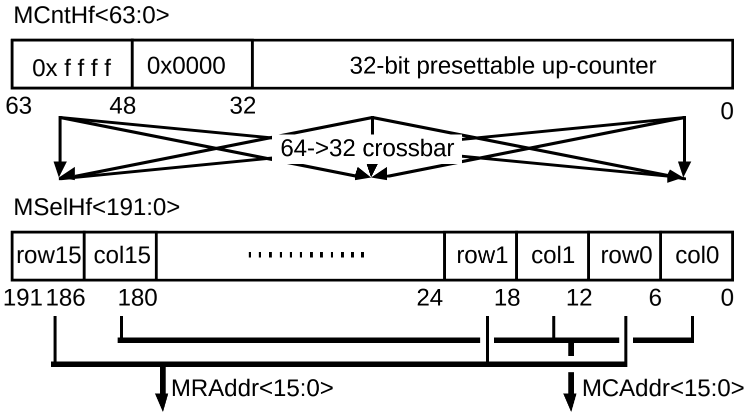


Figure 20: *MCnt* and *MSel* Registers.

The following commands set the *MCnt* and *MSel* registers:

Command:		***Synopsis:***
TAS_SetMCnt	(*val8*)	// Set the 8 LSBs of the *MCnt* register, first shifting left 8 bits
TAS_SetMSel	(*rsel, csel*)	// Set the 12 LSBs of the *MSel* register, first shifting left 12 bits

TAS_SetMCnt shifts the 32-bit *MCnt* register left 8 bits and sets the 8 LSBs of the register to *val8* (8 bits). This command must be repeated four times to set the entire *MCnt* register.

The **TAS_SetMSel** commands shift the 192-bit *MSel* register left 12 bits and sets the 12 LSBs of the register to (*rsel* << 6) | *csel.* The 6-bit values *rsel* and *csel* select the source of one row and one column address bit within the extended *MCnt* register. This command must be repeated 14 times to set the entire *MSel* register. If only 12 address bits are needed (as for 16 Mbit SDRAMs), it need only be executed 12 times.

Command to Preload the ACT

The block read/write and interleaved read/write commands described later may use the ACT as a source of texture read/write addresses. The following command loads addresses from the EMCs into the ACT:

Command:		***Synopsis:***
TAS_LoadACT	()	// Load addresses from the EMC local port into the ACT

The **TAS_LoadACT** command transfers octets of row/column addresses from the local-port output registers of marked pixels in each EMC and loads them into the ACT. A single octet from an EMC may come from one or more PEs, depending on the way the local-port output buffer is configured. Exactly 32 bytes per EMC is transferred. If an EMC is marked to send more than 32 bytes, the PEs which were not accessed remain marked after the instruction has finished. If an EMC is marked to send fewer than 32 bytes, then 0s are transferred for the remaining addresses.

Commands to Initialize the SDRAMs Memory System

The following commands are used to initialize and configure the SDRAM external memory system:

Command:		***Synopsis:***
TAS_Init	()	// Initialize the SDRAMs

TAS_Init configures the TASICs to drive the SDRAM memories at 100 MHz, precharges both banks of each SDRAM memory chip and sets the mode register on each SDRAM to its default setting[1].

Command to Refresh Memory

The SDRAMs that compose external memory are dynamic memory devices and must be refreshed periodically. Each of the 2048 rows of the two banks on each SDRAM chip must be accessed every 64 msec. The following command performs one refresh cycle for both banks, using the internal refresh counter in the SDRAM chips to keep track of the current refresh row:

Command:		***Synopsis:***
TAS_MemRefr	( )	// Do one refresh cycle

This command seldom needs to be called explicitly. Rather, an opportunistic refreshing scheme is used. This command is placed at microcode location 0, meaning that it executes by default when no other command is pending; ie. when the TIGC sequencer is idle, external memory is refreshed continuously.

The TIGC sequencer may be busy for long periods, however. To prevent the refresh interval from being exceeded, every command that is more than one or two cycles long branches to **TAS_MemRefr** when it ends. Routines that are longer than the refresh interval have internal refresh cycles. In this manner, refreshing is performed sufficiently often, no matter what the sequence of TAS commands and/or idle periods.

There is one exception to this: when the sequencer is in RMODE it is inactive and cannot perform refresh cycles. Care must be taken to ensure that the TIGC sequencer is not in RMODE for longer than ?? nsec., and that a **TAS_MemRefr** is executed immediately after leaving RMODE. This can be done by placing the **TAS_RModeOn**, **TAS_RModeOff**, and intervening instructions in a packet of the following form:

```
TAS_SetScratch2(s);            // Synchronize with GP
TAS_WaitScratch2(s);
TAS_RModeOn(s);                // Enter RMODE
(Microcode read/write commands);
TAS_RModeOff(s);               // Exit RMODE
TAS_MemRefr(s);                // Do refresh cycle immediately
```

The GP ensures that this packet executes without interruption by flushing the IGCStream buffer after the **TAS_MemRefr**, waiting for the *Scratch2* bit in the rasterizer glue chip to

[1] Its default setting is: CAS_latency = 3, burst_mode = 'sequential', burst_length = 4. A few TAS commands change the mode register setting to a burst length less than 4. These commands restore the mode register to its default setting before they complete.

be set, then clearing the *Scratch2* bit. The entire packet of instructions must take less than ?? nsec to execute. Note that these precautions are only necessary if the contents of texture memory are to be preserved. During startup, the contents of the external memories are undefined, so a lapse in refreshing will do no harm.

V.3 Commands for Reading and Writing External Memory

This section describes commands for reading and writing external memory. These commands are divided into two classes: *block* commands and *scatter* commands. Block commands assume that blocks of data can be read from or written to memory locations that share the same row address, and therefore can take advantage of fast-page mode accesses to the SDRAMs. Scatter commands perform a full row/column access for each data item transferred. They allow successive data items to be read from/written to arbitrary locations in texture memory.

Block commands generally use the *MCnt*/*MSel* registers to generate addresses (though the ACT can be used to "tweak" addresses if it has been preloaded using the **TAS_LoadACT** command). Scatter commands use addresses computed by the EMCs and transferred to the TASICs along with the data.

The number of data items transferred is governed by the number of PEs with their mark flags enabled (see Section II.3), so their execution time is variable.

Block Read Commands

The following commands transfer blocks of data between SDRAM memory and the EMCs' local-port input buffers:

Command:		***Synopsis:***
TAS_MemRdBlock1	()	// Block read, 1-byte texels
TAS_MemRdBlock2	()	// Block read, 2-byte texels
TAS_MemRdBlock4	()	// Block read, 4-byte texels
TAS_MemRdBlock1Ileave16	()	// Interleaved block read, 1-byte texels
TAS_MemRdBlock2Ileave32	()	// Interleaved block read, 2-byte texels
TAS_MemRdBlock4Ileave64	()	// Interleaved block read, 4-byte texels

(Outlined commands are currently unimplemented)

Each of these commands reads the designated number of bytes from each of the eight SDRAM memories in the respective module and loads this data into the local-port input buffer of each marked pixel. The suffix '1', '2' or '4' specifies the number of bytes read per memory (*i.e.* the texel size).

Addresses for the SDRAM reads are generated using the *MCnt*/*MSel* address generator xor'ed with the ACT. This behavior can be modified by first issuing the appropriate **TAS_SetAPortMode*** command. *MCnt* is incremented once for each data item transferred (from all the SDRAMs in parallel). The contents of the ACT are not modified during these commands.

Data read from the SDRAMs is loaded 8 bytes at a time (1 byte per SDRAM) into the TASICs' DCTs. When a "batch" of 32 bytes has been read from each of the eight SDRAMs, the DCT begins transferring this data to the EMCs, while loading a new "batch" from the SDRAMs. (You want to consult the drawings of the ACT/DCT above (or the more detailed drawings in the *PixelFlow TASIC Functional Description*) to understand how these commands move data within the TASIC).

Block read commands **(TAS_MemRdBlock*n*)** perform 16 read operations between precharges of the SDRAM. Hence, all 16 consecutive data items read from a single memory must lie in the same memory row. Also, reads must begin on block boundaries.

Interleaved block read commands (**TAS_MemRdBlock*n*ILeave*16n***) are similar, except consecutive blocks must lie in alternating SDRAM banks. The row access for one block can be overlapped with the data reads from the previous block, making these commands approximately ?? % faster than block read commands. To use the interleaved block read commands, bit 11 of the row address and bit 9 of the column address must toggle between alternate blocks.

In both types of commands, number of data items transferred is governed by the number of PEs whose mark register is set. If some EMCs have more marked PEs than others, data values will continue to be read until all EMCs are finished. The data will be discarded on EMCs with no remaining marked PEs.

Prior to issuing one of these commands, the EMC local-port input and output buffers must be configured as follows:

For each marked pixel:

- The local-port input buffer must be configured to receive the correct number of bytes per PE using the **EMC_LPortIn** command. This number *k* must be a multiple of the size of the data items transferred.

Figure 21 shows a typical usage of these commands. We will assume that each marked PE is to receive eight 2-byte data items, hence a **TAS_MemRdBlock2** command is used. The local-port input buffer is configured to receive 16 bytes. When the command completes, the local-port input buffer contains eight 2-byte values, as shown.

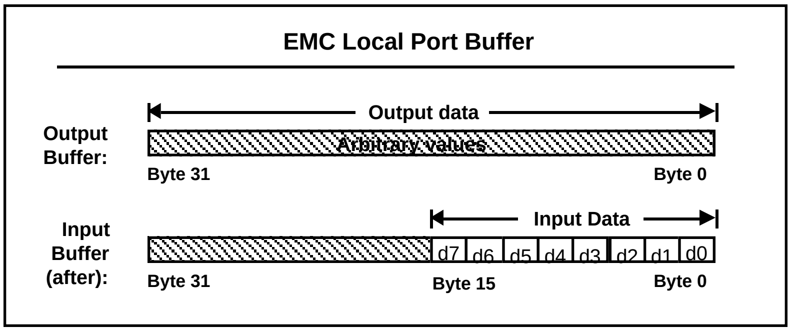


Figure 21: Contents of EMC local-port buffers before and after a block read command.

The ACT and DCT address pointers at the start of Block commands will be reset to zero. At the conclusion of a Block command, the contents of the*MCnt* register will be undefined. The contents of the local-port output buffer are unchanged after any of these commands, as are the unwritten bytes of the local-port input buffer.

Block Write Commands

The following commands transfer blocks of data between the EMCs' local-port output buffers and the SDRAMs of one or all modules:

Command:		*Synopsis:*
TAS_MemWrBlock1	()	// Block write, 1-byte texels
TAS_MemWrBlock2	()	// Block write, 2-byte texels
TAS_MemWrBlock4	()	// Block write, 4-byte texels
TAS_MemWrBlock1Ileave16	()	// Interleaved block write, 1-byte texels
TAS_MemWrBlock2Ileave32	()	// Interleaved block write, 2-byte texels
TAS_MemWrBlock4Ileave64	()	// Interleaved block write, 4-byte texels

(Outlined commands are currently unimplemented)

Each of these commands writes the designated number of bytes from the EMC local-port output buffers of marked pixels to the eight SDRAM memories in the designated module or modules. The suffix '1', '2' or '4' specifies the number of bytes written per memory (*i.e.* the texel size).

These commands are similar to the corresponding block read commands (except data flows in the opposite direction). There are three other considerations specific to writes:

- Writes can be destined to one or all modules.
- All writes occur—even ones resulting from unmarked PEs (if other EMCs still have

marked PEs). These "invalid" writes generally are to SDRAM address 0.

- The local-port output buffer is used for outgoing data; the local-port input buffer is unused.

Unlike reads, in which data from a module's SDRAMS is always returned to the EMCs of that module, write commands can send data to other or all modules. The commands **TAS_SetWriteMode{1to1,1toN,NtoN}** can be used to specify the desired source and destination module(s).

With write commands there is no provision for discarding data for invalid writes (unlike read commands, in which data destined for an EMC with no marked PEs simply is ignored). Write operations are always performed so care must be taken to enable the same number of PEs in all EMCs. Otherwise, spurious writes will occur (addresses will continue to be generated as before, but zeroes will be written, since there is no valid data).

Prior to issuing a block write command, the EMC local-port output buffers must be configured as follows:

For each marked pixel:

- The local-port output buffer must be configured to transmit the correct number of bytes per PE using the **EMC_LPortOut** command. This number *k* must be a multiple of the size of the data items transferred.

Figure 22 shows a typical usage of one of these commands, in this case **TAS_MemWrBlock4**. The local-port output buffer is configured to send 32 bytes. The local-port input buffer is unused.

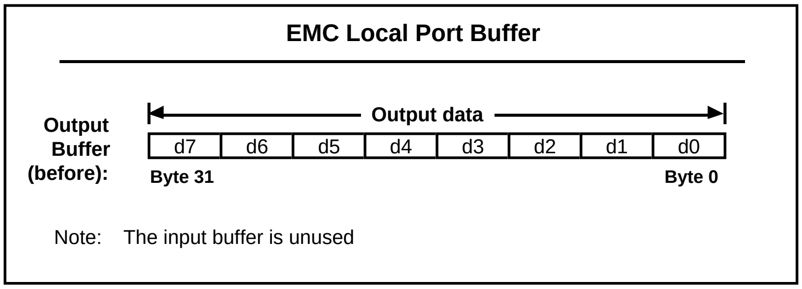


Figure 22: Contents of EMC local-port buffers before a block write command.

Scatter Read Commands

The following commands transfer data from arbitrary locations in SDRAM memory into EMCs' local-port input buffers:

Command:		Synopsis:
TAS_MemRdScatterEven1	( )	// Scatter read even, 1-byte texels
TAS_MemRdScatterOdd1	( )	// Scatter read odd, 1-byte texels
TAS_MemRdScatterEven2	( )	// Scatter read even, 2-byte texels
TAS_MemRdScatterOdd2	( )	// Scatter read odd, 2-byte texels
TAS_MemRdScatterEven4	( )	// Scatter read even, 4-byte texels
TAS_MemRdScatterOdd4	( )	// Scatter read odd, 4-byte texels

(Outlined commands are currently unimplemented)

Each of these commands reads the designated number of bytes from each of the eight SDRAM memories in the respective module and loads this data into the local-port input buffer of each marked pixel. Unlike the block read commands, addresses for these commands come from the EMCs and can access data in random locations in SDRAM memory. Again, the suffix ‘1’, '2' or '4' specifies the number of bytes read per memory (*i.e.* the texel size).

Data flow for these commands is more complicated than for the block read commands. In addition to moving data from SDRAMs to EMCs (vias the DCTs), addresses are simultaneously transferred from the EMCs through the ACTs, and then are applied to the SDRAMs.

To maximize read bandwidth, these commands read consecutive data items from each SDRAM from different internal banks. The ‘Even’ and ‘Odd’ versions of the commands access bank 0 first and bank 1 first, respectively and correspond to even and odd replicated texture maps, as described in Section V.1. The commands toggle address bit 11 (the bank select bit) automatically, so address bit 11 should not be set in either row or column addresses).

Prior to issuing one of these commands, the EMCs must compute 16-bit row and column addresses for each marked pixel and store them into the local-port output buffer as shown in Figure 23 below. The local-port input and output buffers for these pixels must then be configured as follows:

- The local-port output buffer must be configured to transmit 4 bytes of (address) data for every data item to be received using the **EMC_LPortOut** command.
- The local-port input buffer must be configured to receive the correct number of bytes per PE using the **EMC_LPortIn** command.

Figure 23 shows a typical usage of these commands. We will assume that each marked PE is to receive eight 2-byte data items, hence **TAS_MemRdScatterEven2** or

TAS_MemRdScatterOdd2 is used. Eight row and eight column addresses are loaded into the local-port output buffer (address bit 11 must be zero for each of these). The local-port output buffer is configured to send 32 bytes and the local-port input buffer is configured to receive 16 bytes. When the command completes, the local-port input buffer contains eight 2-byte values, as shown.

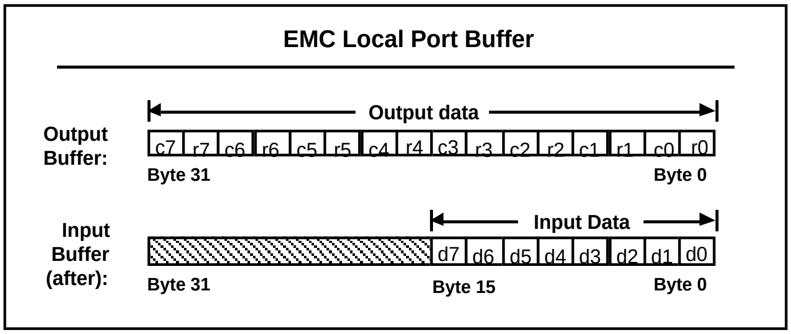


Figure 23: Contents of EMC local-port buffers before and after a scatter read command.

These instructions have the side effect of setting the address port mode as if the instruction **TAS_SetAPortModeACT** was issued.

Note that when using scatter read commands for unreplicated textures, half of the data values returned will be bogus and can simply be discarded.[1]

[1] Note also that when performing mip-map texture reads, the addresses and data for a particular mip-map resolution level will alternate between Panel A and Panel B in successive pixels. Thus, a swazzling step may be needed to align addresses and data before further computations can begin.

Scatter Write Commands (currently unimplemented)

The following commands allow data to be written to arbitrary locations in SDRAM memory:

Command:		Synopsis:
TAS_MemWrScatterEven1A	( )	// Scatter write even, 1-byte texels, panel A
TAS_MemWrScatterEven1B	( )	// Scatter read even, 1-byte texels, panel B
TAS_MemWrScatterOdd1A	( )	// Scatter write odd, 1-byte texels, panel A
TAS_MemWrScatterOdd1B	( )	// Scatter read odd, 1-byte texels, panel B
TAS_MemWrScatterEven2A	( )	// Scatter write even, 2-byte texels, panel A
TAS_MemWrScatterEven2B	( )	// Scatter read even, 2-byte texels, panel B
TAS_MemWrScatterOdd2A	( )	// Scatter write odd, 2-byte texels, panel A
TAS_MemWrScatterOdd2B	( )	// Scatter read odd, 2-byte texels, panel B
TAS_MemWrScatterEven4A	( )	// Scatter write even, 4-byte texels, panel A
TAS_MemWrScatterEven4B	( )	// Scatter read even, 4-byte texels, panel B
TAS_MemWrScatterOdd4A	( )	// Scatter write odd, 4-byte texels, panel A
TAS_MemWrScatterOdd4B	( )	// Scatter read odd, 4-byte texels, panel B

(Outlined commands are currently unimplemented)

These commands correspond to the Scatter Read commands described above, except that values can only be written to one panel (four SDRAM memories) in each module at a time.[1] Each of these commands transfers the designated number of bytes from the EMC local-port output buffers of marked pixels to the designated panel (four SDRAM memories) in the designated module or modules. In these commands, both addresses and data are provided by the EMCs. The suffix ‘1’, '2' or '4' specifies the number of bytes read per memory (*i.e.* the texel size).

The Even/Odd suffix designates whether an even or odd texture map is to be written. The A/B suffix designates whether data is to be written to panel A or B. These commands automatically enable the appropriate panel.

As with block writes, these commands have the following properties:

- Writes can be destined to one or all modules by configuring the TASIC datapaths with the **TAS_SetXferMode{1to1,1toN,NtoN}** commands (with TASIC Rev2.0, **1to1** commands need additional thought).
- All writes occur—even ones resulting from unmarked PEs (if other EMCs still have

[1] The reason for this is two-fold: The local-port output buffer of one PE is not large enough to store both the addresses and data for eight SDRAMS. More importantly, the EMC to TASIC bandwidth can only support half-speed writes

marked PEs) or to the masked panel. These "invalid" writes are to SDRAM address zero.

- The local-port output buffer is used for outgoing data; the local-port input buffer is unused.

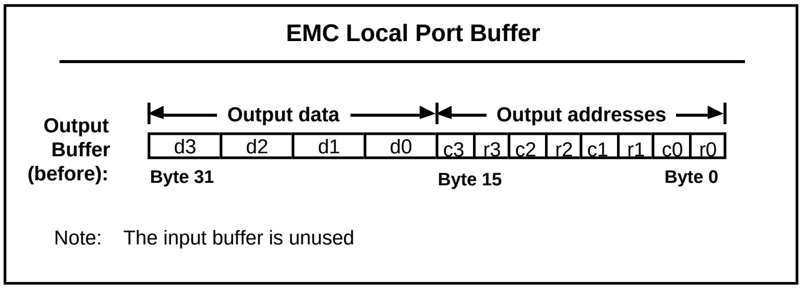
Prior to issuing a block write command, the EMC local-port output buffers must be configured as follows:

For each marked pixel:

- Addresses and data must be stored in the local-port output buffer. The four 4-byte addresses must occupy bytes 0–16. The four data values must occupy the next 4, 8, or 16 bytes (for 1, 2, and 4-byte writes, respectively).

- The local-port output buffer must be configured to transmit the correct number of address+data bytes per PE using the **EMC_LPortOut** command.

Figure 24 shows a typical usage of one of these commands, in this case **TAS_MemWrScatterEven4**. The local-port output buffer is configured to send 32 bytes. The local-port input buffer is unused.



Note: The input buffer is unused

Figure 24: Contents of EMC local-port buffers before a scatter write command.

These instructions have the side effect of setting the address port mode as if the instruction **TAS_SetAPortModeACTPan*X*** , (where ***X*** is depends on which panel the instruction writes to) was issued.

Scatter Command Notes. As mentioned above, the scatter commands perform SDRAM accesses which implicitly alternate between banks. Because of the way the corner turns are filled and accessed, the EMCs are processed in round robin order. What this implies is that with an even scatter instruction, the even EMCs always access the even bank while the odd EMCs always access the odd bank (vice versa for the odd instructions). The duration of a texture access is as long as the EMC with the most data. For example, if all EMCs except for are finished, the EMCs are still processed in round robin order (accessing the appropriate even/odd banks), except that the data for the finished EMCs is ignored.

Setup and synchronization. All of the commands above require setup by the EMCs prior to execution and processing of results after they complete. Figure 25 shows a typical use of the **TAS_MemRdScatterEven4** command. The other commands (including write commands, described below) are used similarly.

```
EMC commands to enable a desired set of pixels;
EMC commands to calculate memory addresses;
EMC commands to copy addresses into  the local port  input buffer;
EMC_LPortIn();                      // Configure the local input port
EMC_LPortOut();                     // Configure the local output port
EMC_VTas();                     // Make sure local port is configured
TAS_PEmc()                      //        before TAS command begins
TAS_MemRdScatterEven4();  // Do memory read
(Optional EMC commands);
TAS_VEmc();                         // Prevent EMC commands from executing
EMC_PTas();                     //        until TAS read completes
EMC commands to copy data from the local port output buffer;
EMC commands to operate on the data;
```

Figure 25: Typical use of TIGC memory commands.

The EMCs must prepare addresses, copy them into local buffer memory, and configure the local port prior to beginning the memory read. The **EMC_VTas** and **TAS_PEmc** commands ensure that all setup calculations complete before the **TAS_MemRd** command begins. Similarly, the **TAS_VEmc** and **EMC_PTas** commands ensure that the read operation completes before EMC commands which require the data can execute. Since read operations take many cycles, it is often desirable to perform unrelated EMC operations while the read commands are executing. These can be placed between **TAS_MemRd** and the first **TAS_VEmc**. Section VI.1 gives further details on the use of semaphores to synchronize the processing of EMC and TAS commands.

TAS_MemWr commands do not affect the contents of either the local-port output or input buffers. Synchronizing with the EMC sequencer is only needed before a **TAS_MemWr** command is executed (unlike **TAS_MemRd** commands, which require synchronizing before and after the command).

V.4 Commands to Communicate with the GP

This section describes commands used to transfer data between PixelFlow PA-8000-based GPs and the texture subsystem, primarily for loading (and paging) textures, and retrieving frame buffer data. The current commands assume the presence of a GNI and are a first implementation of this interface. Many alternatives are possible.

Commands for synchronization with the GP

The following commands support synchronization between the TIGC sequencer and program execution on the GP. They are useful, for example, to indicate to the GP that a particular set of commands have been executed by the TIGC sequencer. Two forms of synchronization are supported: interrupts and polling. The current software convention is to use *Scratch1* bit in the rasterizer glue chip for interrupt-based synchronization, and the *Scratch2* bit for polling-based synchronization.

Command:		**Synopsis:**
TAS_SetScratch1	()	// Set glue chip Scratch1 bit, causing a GP interrupt
TAS_SetScratch2	()	// Set glue chip Scratch2 bit, by convention polled by the GP
TAS_WaitScratch1	()	// Wait for the Scratch1 bit to be cleared
TAS_WaitScratch2	()	// Wait for the Scratch2 bit to be cleared

Commands for setting GNI registers

The following TIGC commands set GNI internal registers and are used to provide packet destination and length information when sending data from the rasterizer to GPs over the Geometry Network.

Command:		**Synopsis:**
TAS_SetGNIPktHdr	(*val*)	// Set the 8 MSbits of the GNI packet header register to *val*, // right shifting the remaining bits
TAS_SetGNIMsgHdr	(*val*)	// Set the 8 MSbits of the GNI message header register to *val*, // right shifting the remaining bits
TAS_SetGNIPktSize	(*bytes*)	// Set the packet size to *bytes* (valid range is 256 to 1024 in // multiples of 256)

See the GNI documentation for the appropriate use of these registers.

GP read/write commands

The following commands allow GPs to send and receive data to/from texture and EMC memory via the GNI and TASICs:

Command:		***Synopsis:***
TAS_XferEtoG	()	// Transfer 512 bytes from EMCs of selected module to GNI

TAS_XferGtoE	(*bytes*)	// Transfer *bytes* bytes of data from the GNIs to the EMCs of
		// one or all modules (depending on Xfer mode). Possible values for
		//*bytes* are: 256, 512, 768, 1024.

TAS_XferTtoG_Block4	()	// Transfer 512 bytes of data from SDRAMs of selected module to // GNI using MCnt register to generate addresses.

TAS_XferGtoT_Block4	(*bytes*)	// Transfer *bytes* bytes of data from the GNIs to the SDRAMs of one
		// or all modules (depending on Xfer mode). Mcnt register is used
		// to generate SDRAM addresses. Possible values for bytes
		// are: 256, 512, 768, 1024.

Before executing any of these commands, the TASIC transfer mode must be set appropriately using one of the **TAS_SetXferModeXXX** commands**.** Possible settings for each command are as described in the paragraphs below.

TAS_XferEtoG must be preceded by a **TAS_SetXferMode1toG(** ***src*** **)** command to configure the inter-module TASIC ring and select source module. It transfers a packet of 512 bytes from the SDRAMs of module *src* to the GNI.

TAS_XferGtoE must be preceded by either **TAS_SetXferModeGtoN()** or **TAS_SetXferModeGto1(** ***dst*** **)**. It transfers *byte* bytes from the GNI to the EMCs of all or a single module. In the GtoN case, GNI data is written to the EMCs of all four modules. In the Gto1 case, GNI data is written to the EMCs of the selected module, while zeroes are written to the EMCs of non-selected modules. (For TASIC Rev2.0, this command will have to be rewritten, since Gto1 transfers will only load the DCTs of one module).

TAS_XferTtoG_Block4 must be preceded by a **TAS_SetXferMode1toG(** ***src*** **)** command. It transfers a 512-byte packet of data from the SDRAMs of module *src* to the GNI. The SDRAM addresses used by this command are generated by the *MCnt* register. Thus, *MCnt* / *MSel* must be configured appropriately prior to issuing this command.

TAS_XferGtoT _Block4 must be preceded by either **TAS_SetXferModeGtoN()** or **TAS_SetXferModeGto1(** ***dst*** **)**. It transfers *byte* bytes from the GNI to the SDRAMs of all or a single module. In the GtoN case, GNI data is written to the SDRAMs of all four modules. In the Gto1 case, GNI data is written to the SDRAMs of the selected module, while zeroes are written to the SDRAMs of non-selected modules. (For TASIC Rev2.0, this command will have to be rewritten, since Gto1 transfers will only load the DCTs of one module). The SDRAM addresses used by this command are generated by the *MCnt* register. Thus, *MCnt* / *MSel* must be configured appropriately prior to issuing this command.

Note that all of these commands require careful synchronization with the GP (*e.g.* sending data to the GNI at the right time, use of synchronization commands). Misuse of these commands can cause the TIGC or GP to hang. Consequently, these commands should be encapsulated within tested system code and not made directly available to application code. For futher information on the use and operation of these commands, consult the comments in the microcode source file **TAS.ucode.pre** and the library routines that use them in **~pxfl/pbase/src**.

V.5 Video Control Commands

On boards with video input or output circuitry, the TASICs write or read video data to/from the SDRAM memories at the behest of the video controller. This means that they must generate the row and column addresses required for video scan-in (scan-out). The TASICs have *x* and *y* base address registers for eight independent fields (a field is an array of pixels that are scanned in or out in order). The fields can be chained together to accommodate display or input modes with multiple fields (stereo, interleaving, etc.). We define a *frame* as one or more fields that are displayed or acquired consecutively without requiring intervention by the GP.

Figure 28 shows the TASIC video field and video address registers.

VNxtFld0<2:0>	VNxtFrmFld0<2:0>
VNxtFld1<2:0>	VNxtFrmFld1<2:0>
VNxtFld2<2:0>	VNxtFrmFld2<2:0>
VNxtFld3<2:0>	VNxtFrmFld3<2:0>
VNxtFld4<2:0>	VNxtFrmFld4<2:0>
VNxtFld5<2:0>	VNxtFrmFld5<2:0>
VNxtFld6<2:0>	VNxtFrmFld6<2:0>
VNxtFld7<2:0>	VNxtFrmFld7<2:0>

VFld<2:0>

VXMax0<15:0>	VYBase0<15:0>	VXBase0<15:0>
VXMax1<15:0>	VYBase1<15:0>	VXBase1<15:0>
VXMax2<15:0>	VYBase2<15:0>	VXBase2<15:0>
VXMax3<15:0>	VYBase3<15:0>	VXBase3<15:0>
VXMax4<15:0>	VYBase4<15:0>	VXBase4<15:0>
VXMax5<15:0>	VYBase5<15:0>	VXBase5<15:0>
VXMax6<15:0>	VYBase6<15:0>	VXBase6<15:0>
VXMax7<15:0>	VYBase7<15:0>	VXBase7<15:0>

VYCnt<15:0>	VXCnt<15:0>

Figure 26: TASIC Video Field and Video Address Registers.

VFld stores the index of the active field (the field currently being scanned in or out). *VNxtFld*<*n*> and *VNxtFrmFld*<*n*> store the index of the next field and next frame field for each possible field value *n* = 0–7. The *VNxtFld* registers are used to chain fields into frames. For example, on an NTSC frame-buffer with an even and odd field *VNxtFld*<0> might be set to 1 and *VNxtFld*<1>, so that the two fields to alternate. The *VNxtFrmFld* registers store the first field of the next frame. Usually they are configured to point to the first field of the current frame. However, when a new frame is available (for example, when it is time to swap buffers in a double-buffered display), the *VNxtFrmFld* registers for the current frame's fields are set to point to the first field of the new frame.

VXBase<n> and *VYBase<n>* store starting *x* and *y* values for the video address counters, *VXCnt*, and *VYCnt*. These logical *x* and *y* addresses are mapped into physical SDRAM row/column addresses using a crossbar similar to the *MCnt*/*MSel* crossbar described in Section V.2. This crossbar is shown in Figure 29.

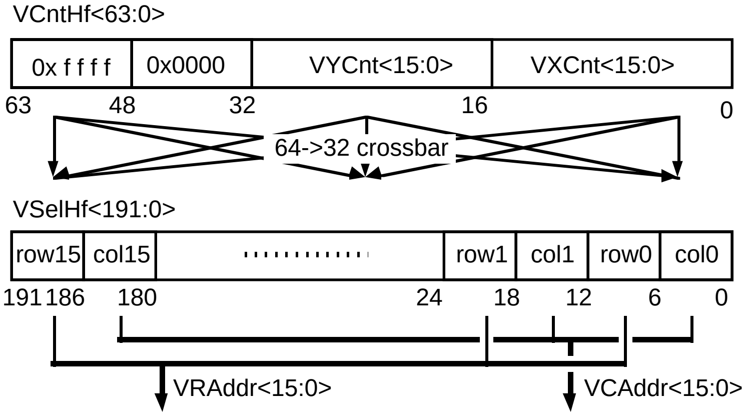


Figure 27: Video address counters and crossbar.

VXCnt is incremented by the video microcode routine for each pixel on a scanline. Since scanlines can contain arbitrary numbers of pixels, an additional set of registers *VXMax<n>*, is provided to specify a maximum *VXCnt* value for each field. When *VXCnt* reaches *VXMax<VFld>*, *VXCnt* is reset to *VXBase<VFld>* and *VYCnt* is incremented. The crossbar following *VXCnt* and *VYCnt* allows address bits to be permuted when generating actual SDRAM row/column addresses.

The TIGC monitors the VidReq input from the video controller. When a video request is received, it may be one of three types:

- Request new frame.
- Request new field.
- Request new scanline.

If a new frame request is received, *VFld* is set to *VNxtFrmFld<VFld>*. If a new field request is received, *VFld* is set to *VNxtFld<VFld>*. In either case, *VXCnt*/*VYCnt* are set to *VXBase*/*VYBase* for the new field, and the read/write pointers for the VFIFO are cleared. A new scanline request causes the TIGC sequencer to load/unload a new scanline of pixels to/from the VFIFO, using *VXCnt* /*VYCnt* to generate the addresses for these reads/writes. *VFld i*s not affected.

The video controller can assert VidReq at any time. The TIGC microcode for a particular video device ensures that this input is polled sufficiently often that new scanline requests can be serviced before the VFIFO runs dry. The video controller may request a new field or new frame at any time, even during the middle of a field. This allows the TASIC video interface to synchronize with the video controller or an external, genlocked video source.

Commands to Set the Video Base and Field Registers

The following commands are used to configure the various video control registers.

Command:		Synopsis:
TAS_SetVBase*n*	(*val8*)	// Load 8 LSBs of *val8* into *VBase*<*n*>, *n* = 0 to 7
TAS_SetVNxtFld*n*	(*fld*)	// Set *VNxtFld*<*n*> to *fld*
TAS_SetVNxtFrmFld*n*	(*fld*)	// Set *VNxtFrmFld*<*n*> to *fld*

TAS_SetVBase*n* loads the composite 48-bit video base address register *VXMax*<*n*>;*VYBase*<*n*>;*VXBase*<*n*>. It sets the eight LSBs of this composite register to *val8* and shifts the remaining bits of the register eight bits to the left. These commands must be called six times to initialize an entire *VBase* register.

TAS_SetVNxtFld*n* loads *VNxtFld*<*n*> with the argument *fld*. It is used to chain a sequence of one or more fields together into a frame.

TAS_SetVNxtFrmFld*n* loads *VNxtFFrmld*<*n*> with the argument *fld*. It is used to specify the first field of the next frame.

Commands to Operate the Video Controller

The following commands are used to initialize and operate the TASIC video port, once it has been configured using the commands above:

Command:		Synopsis:
TAS_SetVFld	(*fld*)	// Set*VFld* to *fld* ; no synchronization with video controller
TAS_SetVFldSynch	(*fld*)	// Wait until next-frame request; then Set VFld to fld
TAS_ClearEOF	()	// Clear the end-of-frame (EOF) sticky bit in the GP status register
TAS_WaitEOF	()	// Wait until end-of-frame (EOF) sticky bit is set

TAS_SetVFld simply sets the current field register *VFld* to *fld*. This is done regardless of the current state of the video controller. It can result in bogus pixel values being scanned in/out until the next-frame request is received from the video controller.

TAS_SetVFldSynch also sets *VFld* to *fld*, but attempts to do so more gracefully. It spin-locks until a next-frame request is received, servicing next-field and next-scanline requests in the meantime, but blocking subsequent TIGC commands. When a next-frame request is received, it initializes the VFIFO, sets *VFld* to *fld*, and initializes the *VXCnt* and *VYCnt* counters in preparation for the scanning in/out the new frame indicated by *fld*. It then acknowledges the next-frame request and allows normal TIGC operation to resume.

TAS_ClearEOF and **TAS_WaitEOF** are used together to block instructions from executing until a frame has finished. **TAS_ClearEOF** clears the end-of-frame (EOF) sticky bit in the GP status register[1]. **TAS_WaitEOF** spin-locks until the EOF bit is set, blocking subsequent TIGC commands, but processing all video controller requests in the meantime.

V.6 Video Control Examples

(This section written by Greg Welch and revised by Steve Molnar)

The video configuration and operation commands outlined in the preceding section can be used in a variety of ways to facilitate the smooth and correct updating, displaying, or acquiring of various types video frames. As stated earlier, we define a *frame* as one or more fields that are displayed or acquired consecutively without requiring intervention by the GP. What follows are descriptions of several possible scenarios surrounding the display or acquisition of video data, along with some sample code fragments which use the preceding commands.

Asynchronous Two-Buffer Frame Buffering

We first look at sample command streams for writing and displaying video frames using two buffers. In these examples we will consider the use of a double-buffered frame-buffer where each frame consists of a single field. Thus, in the following examples field 0 will be used for frame 0 and field 1 for frame 1.

Initially we consider the asynchronous update & display case, the case where it is known that frame updates can or will take longer than corresponding frame scan-outs. In this situation we say that the frame updates occur asynchronously with respect to frame scan-outs.

The startup code shown below includes commands to write the first frame of pixel data, to configure the video port, and to reset the controller.

```
Commands to write pixel data to frame 0 (field 0)
Commands to configure the video controller for single-field frames
TAS_SetVNxtFrmFld0(0);      // Chain frame 0 to itself
TAS_SetVFldSynch(0);        // Reset the video controller, making field 0 active
```

Once these commands have been issued, frames are repeatedly being scanned-out from field 0, by the video controller. Frame 1 can then be updated and the commands to make it

[1] The EOF bit in the GP status register is set whenever the video controller issues a next-frame request (*i.e.* after it has scanned out the last pixel of a frame). This bit is "sticky", meaning that once it is set, it remains set until explicitly cleared by using the **TAS_ClearEOF** command.

active can be issued as follows:

```
Commands to write pixel data to frame 1 (field 1)
TAS_SetVNxtFrmFld1(1);        // Chain frame 1 to itself
TAS_SetVNxtFrmFld0(1);        // Chain frame 0 to frame 1 (switch active buffers)
TAS_ClearEOF();               // Clear the sticky EOF bit in the GP status port
TAS_WaitEOF();            // Wait for frame 0 to be scanned out at least once
```

Note that the **TAS_ClearEOF** and **TAS_WaitEOF** commands are needed to prevent subsequent writes to frame 0 from occurring until frame 1 is active. They also guarantee that frame 0 has been scanned-out at least once.

In the normal steady state, there is a continuous synchronized series of writes and displays: when frame *i* is active, frame *i*+1 can be written; when frame *i*+1 has been written, the video port configuration can be changed so that frame *i*+1 becomes active at the next frame start; when the next-frame request associated with frame *i*+1 occurs frame *i*+2 can then be written and the corresponding video port configuration commands can be issued; etc.

An example of using the commands of the previous section to implement such a steady-state series of double-buffered updates and displays follows.

```
// Frame 1 is active here
while (TRUE)
{
    Commands to write pixel data to frame 0 (field 0)
    TAS_SetVNxtFrmFld0(0);        // Chain frame 0 to itself
    TAS_SetVNxtFrmFld1(0);        // Chain frame 1 to frame 0 (switch active buffers)
    TAS_ClearEOF();               // Clear the sticky EOF bit in the GP status reg
    Miscellaneous other commands
    TAS_WaitEOF();            // Wait for EOF bit (frame 1 to be inactive)

    Commands to write pixel data to frame 1 (field 1)
    TAS_SetVNxtFrmFld1(1);        // Chain frame 1 to itself
    TAS_SetVNxtFrmFld0(1);        // Chain frame 0 to frame 1 (switch active buffers)
    TAS_ClearEOF();               // Clear the sticky EOF bit in the GP status reg
    Miscellaneous other commands
    TAS_WaitEOF();            // Wait for EOF bit (frame 0 to be inactive)
}
```

The **TAS_ClearEOF** and **TAS_WaitEOF** commands prevent the active frame from being overwritten. If other commands are needed that will not corrupt the contents of the active frame, these can be sandwiched between **TAS_ClearEOF** and **TAS_WaitEOF** commands as shown.

For multi-field frames, the startup code is slightly different. The following example is similar to the one above, except that frames are assumed to consist of two fields, as in a double-buffered NTSC display. Frame 0 comprises fields 0 and 1. Frame 1 comprises fields 2 and 3:

```
Commands to write pixel data to frame 0 (fields 0,1)
Commands to configure the video controller for double-field frames
TAS_SetVNxtFld0(1);          // Chain field 0 to field 1
TAS_SetVNxtFld1(0);          // Chain field 1 to field 0
TAS_SetVNxtFld2(3);          // Chain field 2 to field 3
TAS_SetVNxtFld3(2);          // Chain field 3 to field 2
TAS_SetVNxtFrmFld0(0);       // Chain frame 0 to itself
TAS_SetVFldSynch(0);         // Reset the video controller, making field 0 active
```

The **TAS_SetVNxtFld** commands do not have to be repeated. The only difference in the steady-state loop is that to switch active buffers *VNxtFrmFld*[1] will be set to 2 and *VNxtFrmFld*[3] will be set to 0.

Synchronous Two-Buffer Frame Buffering

Next we consider the synchronous update and display case: the case where it is known that each frame update takes less time than the corresponding frame scan-out. In this situation we say that frame updates occur sychronously with respect to frame scan-outs.

Because frame updates occur in lock-step with frame scan-out, we can link each of the two fields directly to each other. In this situation, the startup code can be modified as shown below.

```
Commands to write pixel data to frame 0 (field 0)
Commands to configure the video controller for single-field frames
TAS_SetVNxtFrmFld0(1);       // Chain frame 0 to frame 1 (permanently)
TAS_SetVNxtFrmFld1(0);       // Chain frame 1 to frame 0 (permanently)
TAS_SetVFldSynch(0);         // Reset the video controller, making field 0 active
```

As a result of the fast frame-update capability, there is no longer a need to chain and unchain fields during the steady-state series of double-buffered updates and displays. Therefore the steady-state update code may now resemble the following simple loop:

```
// Frame 1 is active here
while (TRUE)
{
    Commands to write pixel data to frame 0 (field 0)
    TAS_ClearEOF();          // Clear the sticky EOF bit in the GP status port
    TAS_WaitEOF();       // Wait for frame 0 to be inactive

    Commands to write pixel data to frame 1 (field 1)
    TAS_ClearEOF();          // Clear the sticky EOF bit in the GP status port
    TAS_WaitEOF();       // Wait for frame 0 to be inactive
}
```

Again, by issuing a **TAS_ClearEOF** prior to the **TAS_WaitEOF**, we guarantee that the previously active frame becomes inactive before we update it. Note that this scheme is dangerous. It will fail if any frame cannot be updated in time.

Three-Buffer Frame Buffering

We now consider sample command streams for writing and displaying video frames when three buffers are available. Because there is more than one inactive buffer (two in fact) when any one buffer is active, we can alter the above two-buffer code slightly to gain some flexibility and potentially some efficiency.

In the following example we will only explore the case where frame updates take longer than frame scan-outs, *i.e.* the case where the frame updates occur asynchronously with respect to external display updates. If frame updates can be accomplished at a rate faster than the frame scan-out rate, then the frame updates can (if desired) be completely synchronized with the external display updates. This would be accomplished in a manner similar to that described above for the synchronous two-buffer frame buffering.

Specifically, let's look at the case where video frames are being generated from a traditional double-buffered stereo frame buffer. In this situation, while four physical fields would typically be used (two left fields and two right fields) one can think of such a system as actually using three *logical* buffers as follows. At any point in time, one each of the two left and two right physical fields will be chained together, by use of the **TAS_SetVNxtFld** commands, to be repeatedly displayed. These two chained fields can be thought of as one single logical buffer, the *active* logical buffer. Each of the remaining left and right fields would be considered its own single (inactive) logical buffer. This makes sense because either of these remaining inactive left and right fields can be individually updated and then exchanged (replaced in the chain) for one of the left or right fields in the active logical buffer. Thus the total number of logical buffers would be three: one active logical buffer with two physical fields, and two inactive logical buffers each with one physical field. Two such examples are shown in Figure 30 below.

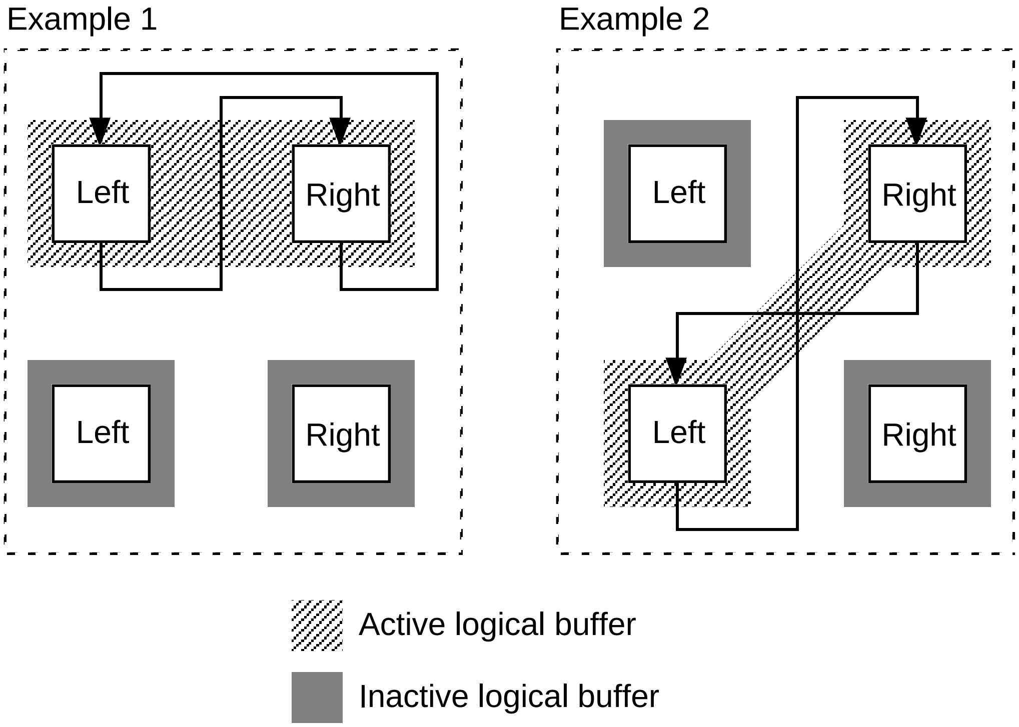


Figure 28: Two examples of logical buffer triples. The arrows denote the order of chaining of the physical fields.

Unlike the preceding two-buffer example, in this example we will not associate particular physical fields with particular buffers. Instead we will continue to refer to buffers as abstract logical buffers, numbered 0, 1 and 2. The physical fields associated with each of the three logical buffers might vary with the use of **TAS_VNxtFld** commands, and will not be described explicitly. We will assume that the controller is configured for the appropriate number of fields per frame, e.g. two fields per frame for a stereo buffer.

The initial startup code in this situation would be identical to that shown above for the asynchronous two-buffer case. Once these startup commands have been issued, frames are being generated (by the video controller) from logical buffer 0. One of the two free logical buffers, buffer 1 for example, can then be updated and the commands to make it active can be issued as follows.

```
TAS_ClearEOF();                    // Clear the sticky EOF bit in the GP status port
Commands to write pixel data to buffer 1 (field 1)
Commands to make buffer 1 active (field chain commands)
TAS_WaitEOF();                 // Wait for buffer 0 to have been scanned-out
```

Again the **TAS_ClearEOF** and **TAS_WaitEOF** commands are only necessary if writes to buffer 0 will immediately follow, as would normally be the case as we begin a steady-state process of alternating writes and displays.

Note in particular the repositioning of the **TAS_ClearEOF** command to before the buffer write and chain commands. This early placement provides a relatively long opportunity for the EOF bit to be set prior to the issuance of the **TAS_WaitEOF**. This then potentially decreases the amount of time that the **TAS_WaitEOF** will block subsequent commands. When the **TAS_WaitEOF** does unblock, we are no longer guaranteed that the currently active buffer (buffer 0 in our example) has become inactive, but since we have three buffers available and would typically be writing to the next inactive buffer anyway (buffer 2 in our example) this situation is acceptable and even desirable.

An example of using the commands of the previous section to implement a steady-state series of three-buffer updates and displays follows.

```
// Logical buffer 2 is active here
while (TRUE)
{
    TAS_ClearEOF();             // Clear the sticky EOF bit in the GP status port
    Commands to write pixel data to logical buffer 0
    Commands to make buffer 0 active (field chain commands)
    Miscellaneous other commands
    TAS_WaitEOF();          // Wait for buffer 2 to have been scanned out

    TAS_ClearEOF();             // Clear the sticky EOF bit in the GP status port
    Commands to write pixel data to logical buffer 1
    Commands to make buffer 1 active (field chain commands)
    Miscellaneous other commands
    TAS_WaitEOF();          // Wait for buffer 0 to have been scanned out

    TAS_ClearEOF();             // Clear the sticky EOF bit in the GP status port
    Commands to write pixel data to logical buffer 2
    Commands to make buffer 2 active (field chain commands)
    Miscellaneous other commands
    TAS_WaitEOF();          // Wait for buffer 1 to have been scanned out
}
```

One-Shot Frame Grabbing

The one-shot frame grab is somewhat analogous to displaying a single frame of video from a particular frame buffer. We present two examples for grabbing a single frame of video, both which accomplish the same task but in different fashions.

Our first example directly manipulates the video controller, stopping and starting the controller as needed to obtain a single frame of video. The video will be written to field 0 for this example.

```
Commands to stop the video controller
TAS_ClearEOF();
Commands to configure the video controller for single-field frames
Commands to configure the video controller for a one-shot frame grab
TAS_SetVFld(0);          // Select a destination field for the single frame
Commands to start the video controller
TAS_WaitEOF();
Commands to  read the  grabbed data from field 0
```

Our second example demonstrates a method that allows the video controller to run continuously in a free-running mode, but uses field chaining commands to divert a single frame of video to a particular buffer or field. For this example, we will assume that the controller is currently cycling through field 1, (*i.e.* field 1 is chained to itself, and the controller is free-running), and that we would like to store our single frame of video in field 0. We will also assume that the controller is configured for single-field frames.

```
TAS_VNxtFrmFld0(1);      // Make sure we leave  field 0 after grabbing frame
TAS_SetVFldSynch(0);     // Make field 0 active
TAS_ClearEOF();
TAS_WaitEOF();         // Wait for scan  in to occur and field 0 to become inactive
Commands to read  the grabbed data from field 0
```

The one-shot frame-grab is useful for a one-time acquisition of a single video frame, or for the repeated and ongoing acquisition of video in the situation where the time to process the grabbed data is longer than the time to acquire the data. Under the latter circumstances, the second method (above) may prove to be desirable depending on the amount of time required to stop and start the video controller. With the second method, it might at first seem to be a problem that while the video controller is already (presumably) synchronized with the external video source, one must wait for the beginning of each such frame before grabbing. On the other hand, with the first method one would still have to wait the worst-case time of a single frame anyway during each start-up of the video controller, to synchronize with the external video source.

Asynchronous Free-Running Frame Grabbing

This case is analogous to asynchronous two-buffer frame buffering. This example demonstrates the use of the TAS video commands to continuously grab frames from a free-running video controller in the case where the time to read and process each frame is greater than the time to scan-in each frame.

In this situation the incoming video data can be written to two alternating fields, bouncing back and forth, but not on consecutive frame scan-outs. Instead, video port configuration commands must be used to periodically switch between fields that are otherwise normally chained to themselves. In this case since the frame processing is not fast enough, the frame grabbing cannot stay synchronized with the external video, *i.e.* not every external frame can be grabbed.

```
Commands to configure the video controller for single-field frames
Commands to configure the video controller for continuous frame grabbing
TAS_SetVNxtFrmFld0(0);     // Chain frame 0 to itself for reset
TAS_SetVFldSynch(0);       // Reset & synch the controller, making frame 0 active
TAS_ClearEOF();            // Clear the sticky EOF bit in the GP status reg

while (TRUE)
{
    TAS_SetVNxtFrmFld1(1); // Chain frame 1 to itself (protect frame 0 after scan-in)
    TAS_SetVNxtFrmFld0(1); // Specify switch from active field 0 to field 1
    TAS_WaitEOF();       // Wait for frame 0 to be grabbed & inactive
    Commands to read and process pixel data from frame 0
    TAS_ClearEOF();        // Clear the sticky EOF bit in the GP status reg

    TAS_SetVNxtFrmFld0(0); // Chain frame 0 to itself (protect frame 1 after scan-in)
    TAS_SetVNxtFrmFld1(0); // Specify switch from active frame 1 to frame 0
    TAS_WaitEOF();       // Wait for frame 1 to be inactive
    Commands to read and process pixel data from field 1
    TAS_ClearEOF();        // Clear the sticky EOF bit in the GP status reg
}
```

Notice that while incoming video field *i* is being read from frame 0, incoming video frames *i*+1 through some frame *j* are being written (over-written) in field 1. When the reading of frame *i* from field 0 completes (during frame *j*) the appropriate set of field chaining commands are issued. Then frames *j*+1 through some frame *k* are written to field 0 while frame *j* is being read from field 1, etc.

Notice that between the time frame *i* was read from field 0 and frame *j* was read from field 1, *j*-*i* frames of incoming video were lost. It is the need to protect one field from the incoming video while it is being read that is the main consequence of the asynchronous grabbing scheme. However such a scheme may be either necessitated by the long time required to read and process a single incoming video frame, or desired for other reasons.

Synchronous Free-Running Frame Grabbing

This case is analogous to synchronous two-buffer frame buffering. The following example demonstrates the use of the TAS video commands to continuously grab frames from a free-running video controller in the case where the time to process each frame is less than the time to scan-in each frame. In this situation the incoming video data can be written to two alternating fields, bouncing back and forth on consecutive incoming frames. If the frame processing is indeed fast enough, the frame grabbing can then remain synchronized with the external video, i.e. every incoming frame can be grabbed.

```
Commands to configure the video controller for single-field frames
Commands to configure the video controller for continuous frame grabbing
TAS_SetVNxtFrmFld0(1);      // Chain frame 0 to frame 1
TAS_SetVNxtFrmFld1(0);      // Chain frame 1 to frame 0
TAS_SetVFldSynch(0);        // Reset & synch the controller, making field 0 active
TAS_ClearEOF();             // Clear the sticky EOF bit in the GP status reg

while (TRUE)
{
    TAS_WaitEOF();          // Wait for frame 0 to be scanned-in & inactive
    Commands to read and process pixel data from field 0
    TAS_ClearEOF();         // Clear the sticky EOF bit in the GP status reg

    TAS_WaitEOF();          // Wait for frame 1 to be scanned-in & inactive
    Commands to read and process pixel data from field 1
    TAS_ClearEOF();         // Clear the sticky EOF bit in the GP status reg
}
```

V.7 Miscellaneous TIGC Sequencer Commands

Miscellaneous Commands

The following are TIGC Sequencer commands that operate in the same manner as the miscellaneous EIGC Sequencer commands listed in Section IV.5.

Command:		***Synopsis:***
TAS_NoOp	()	// No operation
TAS_NoOp2	()	// No operation (version with long 64-bit opcode)
TAS_Hang	()	// Hangs the sequencer in a tight loop (for debugging purposes)

Commands to Initialize the TIGC Sequencer

The TIGC Sequencer is initialized in the same manner as the EMC Sequencer (see Section IV.5). The following commands are used to write and read microcode and to cause the TIGC Sequencer to enter and exit *RMode*. The TASICs are initialized by each command that uses them, so no special commands are required for this purpose.

Instruction:		***Synopsis:***
TAS_IFSpec	(Rlim,Tlim,Endain)	// Set interface control register
TAS_RModeOn	()	// Put the sequencer in RMode
TAS_RModeOff	()	// Cause the sequencer to exit RMode, and prepare for normal input
TAS_MCWrite	(addr, word0, word1)	// Load 'word0' and 'word1' into the specified microcode location

TAS_MCRead	(addr)	// Set the sequencer's program counter to 'addr' (this command is
		// also used to read microcode memory during chip testing, but
		// this function is not available during normal operation)

The standard location for TIGC Sequencer microcode is in the file *TAS_<type>_ucode.h*, in an the initialized array **unsigned TAS_<type>_ucode[]**, where <*type*> is the mnemonic for the video configuration (see Section III).

VI RT CONTROLLER COMMANDS AND SYNCHRONIZATION

On each IGC, the command streams from the RFIFO and TFIFO merge at the RT Controller, which dispatches commands in a designated order to the Sequencer.

There are two threads of control for the Rasterizer, one is buffered in the RFIFO, and the other is buffered in the TFIFO. Each thread can contain both EIGC and TIGC commands. The two Sequencers can be interlocked within either thread, and the two threads can be interlocked for either Sequencer. This interlock is performed using semaphore control and preference commands, described in Section VI.1 and VI.2. Other RT Controller commands control operation of the Image Composition Controller; these are described in Section VI.3. Sections VI.4 and VI.5 show how these commands are typically used, by giving sample control algorithms for rasterizer and shader boards.

VI.1 Semaphore Commands

The interlock semaphores are summarized in the table:

Interlock	Semaphore Counter	Blocks	Command to P (wait and decrement)	Command to V (increment counter)
EMC_ commands wait for T_EMC commands	EIGC RFIFBLK	RFIFO on EIGC	EMC_PTfifo	T_EMC_VRfifo
T_EMC_ commands wait for EMC_ commands	EIGC TFIFBLK	TFIFO on EIGC	T_EMC_PRfifo	EMC_VTfifo
TAS_ commands wait for T_TAS_ commands	TIGC RFIFBLK	RFIFO on TIGC	TAS_PTfifo	T_TAS_VRfifo
T_TAS_ commands wait for TAS_ commands	TIGC TFIFBLK	TFIFO on TIGC	T_TAS_PRfifo	TAS_VTfifo
EMC_ commands wait for TAS_ commands	EIGC RSEQBLK	RFIFO on EIGC	EMC_PTas	TAS_VEmc
TAS_ commands wait for EMC_ commands	TIGC RSEQBLK	RFIFO on TIGC	TAS_PEmc	EMC_VTas
T_EMC_ commands wait for T_TAS_ commands	EIGC TSEQBLK	TFIFO on EIGC	T_EMC_PTas	T_TAS_VEmc
T_TAS_ commands wait for T_EMC_ commands	TIGC TSEQBLK	TFIFO on TIGC	T_TAS_PEmc	T_EMC_VTas

The first two semaphores interlock EIGC commands from the RFIFO and TFIFO. For example, if it is necessary to wait for a certain RFIFO EIGC command (**EMC_Foo**) to finish before a TFIFO EIGC command (**T_EMC_Foo**) can execute, then **EMC_Foo** is followed by **EMC_VTfifo**, and **T_EMC_Foo** is preceded by **T_EMC_PRfifo**.

The second two semaphores interlock TIGC commands from the RFIFO and TFIFO.

The third pair of semaphores interlocks EIGC and TIGC commands in the RFIFO stream, and the fourth pair of semaphores interlocks EIGC and TIGC commands in the TFIFO stream.

VI.2 FIFO Preference

On each IGC, either FIFO can have "preference". Typically, the TFIFO has preference. This means that commands are processed from the TFIFO until it is blocked, by either one of the two semaphore interlocks, or by a **WaitXfer** command (see below); even if the TFIFO becomes empty, it retains control and no instructions are read from the RFIFO. If preference shifts to the RFIFO, then the situation is reversed, and no commands are read from the TFIFO until the RFIFO becomes blocked, or until the RFIFO "yields" preference back to the TFIFO.

After reset, the TFIFO has preference. Either FIFO command stream can "grab" or "yield" preference. This is done using the following commands:

Interlock	Action
EMC_Grab	RFIFO becomes preferred on EIGC
EMC_Yield	TFIFO becomes preferred on EIGC
T_EMC_Grab	TFIFO becomes preferred on EIGC
T_EMC_Yield	RFIFO becomes preferred on EIGC
TAS_Grab	RFIFO becomes preferred on TIGC
TAS_Yield	TFIFO becomes preferred on TIGC
T_TAS_Grab	TFIFO becomes preferred on TIGC
T_TAS_Yield	RFIFO becomes preferred on TIGC

Note that a "grab" command obeys the preference rules like any other command. For example, if the next command in the RFIFO on the EIGC is **EMC_Grab**, but the TFIFO has preference, the "grab" command will not execute and cause the RFIFO to gain preference, until the TFIFO has become blocked (or if a **T_EMC_Yield** command is executed from the TFIFO).

VI.3 IC Controller Commands

After reset, a board is "dead". This means that it mindlessly propagates the *ready* and *go* signals, and the image-compostion data stream, through untouched; thus it is transparent to the transfer operations and does not participate. During system initialization, most or all of the boards are set to be "alive". An image-composition loop can be formed by specifying two "end" boards. For example, the left-most alive board is set as a "left-end" board, meaning that its left-hand output stream is connected to its left-hand input stream, and similarly a "right-end" board is specified. More than one (non-overlapping) loop can be formed by specifying more than one pair of "end" boards. A "dead" board cannot be specified as an "end" board.

Each IGC contains an Image Composition Controller, but only the one on the EIGC is used. The Image Composition Controller contains two flip-flops, one or both of which are set to initiate a transfer operation. If the *L2RXfer* flip-flop is set, the board will propagate the "left-to-right" *ready* signal from the downstream board (to its "right") to the upstream board (to its "left"); then, when it receives the *go* signal from the upstream board, it causes the EMCs to begin compositing and passes the *go* signal to the downstream board. Similarly, the *R2LXfer* flip-flop indicates that the board is ready to participate in a "right-to-left" transfer operation.

There is a set of commands for controlling the Image Composition Controller. The following table lists these commands:

Command:		***Synopsis:***
EMC_Dead	()	// Set this board to be "dead" (board is always dead after reset)

EMC_Alive	(left, right)	// Set this board to be "alive", and specify whether it lies at either
		// end of an image composition loop (must be accompanied by
		// EMC_ICEnds command)

EMC_WaitXfer	()	// Block until any pending Image Composition transfer is complete
EMC_InitXfer	(nbytes, l2rarm,l2rfst,l2rlst, r2larm,r2lfst,r2llst)	// Initiate an Image Composition transfer operation

EMC_Dead and **EMC_Alive** determine the connectivity of the image-composition network and are normally issued only at machine initialization time. **EMC_Dead** designates a board as transparent to the image composition network; *i.e.* all data, and the ready and go signals, pass through unchanged. **EMC_Alive** designates a board as *alive* or active, allowing it to participate in image-composition network operations; the arguments specify whether a board is at the left end (*left* = 1) or right end (*right* = 1) of a closed loop of the image composition network, or if it is interior (*left* = *right* = 0). Each **EMC_Alive** command must be accompanied by an **EMC_ICEnds** command with identical arguments (see Section IV.3).

The **WaitXfer** command acts somewhat like a wait-for-semaphore **_P** command; it causes the RT Controller to block the FIFO containing the **WaitXfer** command and wait for any previously initiated Image Composition transfer operation to complete.

The **InitXfer** command does several things. It sets one or both of the *L2RXfer* and *R2LXfer* flip-flops, to indicate that *this* board is ready for a transfer operation; the operation does not actually begin until all system boards involved in the operation are ready for the transfer (this interlock is performed by hardware, transparently to the programmer). Only when all boards are ready does the transfer actually begin, and only when the transfer is completed are the flip-flop(s) cleared, thereby enabling any **WaitXfer** command to complete.

The *nbytes* argument to **InitXfer** specifies the total number of bytes per pixel to be transferred; like the *nbytes* argument to **EMC_ICPort**, it must be in the range 1– 32. After the transfer actually begins, it takes 128**nbytes* cycles for it to complete.

All transfer operations involve a sequence of system boards which is a subset of the total system. There can be a separate sequence for the left-to-right ("l2r") and right-to-left ("r2l") directions, or a sequence may wrap-around and involve both directional paths; there can be several sequences involving disjoint sets of boards. The *l2rarm* argument is set to 1 to indicate that this board is participating in the left-to-right transfer, thereby setting the *L2RXfer* flip-flop; *l2rfst* is set to 1 indicates that this is the first board in the sequence of boards participating in the left-to-right transfer, and *l2rlst* is set to 1 if this is the last board in the chain. *L2rfst* and *l2rlst* must be zero if *l2rarm* is zero. The arguments *r2larm*, *r2lfst*, and *r2llst* specify the sequence of boards for the right-to-left transfer.

Partial transfers. If it is desired to transfer less than a full region of pixels, the *nbytes* argument to **InitXfer** can be set to a proportionately smaller value than the *nbytes* argument to **EMC_ICPort** or **EMC[L2R,R2L]Init**. For example, if *nbytes* is set to 16 in an **EMC_ICPort** command which configures the EMCs, but to 8 in the **InitXfer** command which initializes the transfer, then only half the pixels in the region will be composited. The composition occurs in PE-index order, in parallel for all 8 panels, so in this example, the left half of the region will be composited (?).

Care must be taken to ensure that all boards participating in a composition-network operation have been configured with an **EMC_ICPort** command. ransferan InitXfer command until all of the boards in s must only be given when a

VI.4 Rasterization Control Algorithm

We now show the basic rasterization control algorithm (the control algorithm on a Renderer board). We consider two cases:

a) *Rasterization into a scratch buffer.* Primitives are rasterized into a *scratch* buffer at a fixed location in pixel memory, then copied into a region buffer, where they remain until they are copied into the IC Port buffer for compositing.

b) *Multiple transfers per region.* Two or more composition cycles are required to transfer all of the data per pixel.

Figure 31(a) shows case (a) above, the most common of the three cases. First, **T_EMC_VRfifo** is used to preset the actual number of region buffers (the number is determined by the number of bits per pixel required for the rendering algorithm). The number of buffers can be changed on the fly using **T_EMC_VRfifo** and **EMC_PTfifo** appropriately.

After initializing the RT Controller, the first region is rasterized into the scratch buffer. **EMC_PTfifo** is then used to ensure that a region buffer is available (it blocks RFIFO commands on the EIGC, and thus any further rasterization, until a region buffer is available). The data is then copied from the scratch buffer into a region buffer and **EMC_VTfifo** is used to indicate that there is a region ready for compositing. The TFIFO command sequence contains a **T_EMC_PRfifo** command to insure there is a region ready for compositing, followed by a **T_EMC_WaitXfer** command to insure that the previous transfer operaion has completed. It then saves the pixel-ALU state, copies the region from the region buffer into one of the IC Port buffers, initializes the IC port using the **EMC_ICPort** command, and launches the transfer with the **IGC_InitXfer** command. It also issues a **T_EMC_VRfifo** command to indicate that a region buffer has been freed up.

```
// Initialize board; set # of free buffs
EMC_Alive;
EMC_ICEnds;
for number of pixel buffers nbuffs
   T_EMC_VRfifo;

// Loop for each region
for each region  r  {

   // RFIFO commands
   Rasterize -> scratch buff;
   EMC_PTfifo;
   Copy scratch -> buff[r%nbuffs];
   EMC_VTfifo;

   // TFIFO commands
   T_EMC_PRfifo;
   T_EMC_WaiXfer;
   T_EMC_ALUSave;
   Copy buff[r%nbuffs] -> Xfer buff;
   T_EMC_VRfifo;
   T_EMC_ICPort;
   T_EMC_ALURstr;
   T_EMC_InitXfer;
}
```

a) Basic algorithm for rasterizing into a scratch buffer.

```
// TFIFO commands
T_EMC_PRfifo;
T_EMC_WaitXfer;
T_EMC_ALUSave;
Copy buff[r%nbuffs] (1/2) -> Xfer buff;
T_EMC_ICPort;
T_EMC_ALURstr;
T_EMC_InitXfer;
T_EMC_WaitXfer
T_EMC_ALUSave;
Copy buff[r%nbuffs] (2/2)-> Xfer buff;
T_EMC_VRfifo;
T_EMC_ICPort;
T_EMC_ALURstr;
T_EMC_InitXfer;
```

b) TFIFO commands to perform two transfers per region.

Figure 29: Use of control commands for typical rasterization algorithms.

The sequence of TFIFO commands to initiate the transfer operation is critical. First, the data to be composited must be copied into the image-composition transfer buffers, and the compositor circuitry on the EMCs must be initialized using an **EMC_ICPort** (or **EMC_[L2R,R2L]Init**) command. Since the TFIFO command sequences are executed at unpredictable times relative to the RFIFO command stream, the state of the EMC pixel processors must be saved and restored at the beginning and end of each TFIFO command sequence, using the **EMC_ALUSave** and **EMC_ALURstr** commands; these commands require a 6-byte area of pixel-memory to save the state.

Figure 31(b) shows a modified TFIFO command sequence that performs two image-composition transfers per screen region. This sequence is used when a shader requires more than 256 bits per pixel/sample.

VI.4 Shading Control Algorithm

The control algorithm on Shader boards is similar to the rasterization control algorithm, except that regions of pixels are loaded, processed, and unloaded, rather than simply being unloaded onto the image-composition network. Figure 32 shows the basic shading control algorithm.

```
// Initialize board
EMC_Alive;
EMC_ICEnds;
EMC_VTfifo;

// Load first region
EMC_PTfifo;                 // RFIFO cmds
EMC_VTfifo;
T_EMC_PRfifo;                   // TFIFO cmds
T_EMC_WaitXfer
T_EMC_ALUSave;
T_EMC_VRfifo;
T_EMC_ICPort;
T_EMC_PipeFlush;
T_EMC_ALURstr;
T_EMC_InitXfer;

// Load second region
EMC_PTfifo;                 // RFIFO cmds
EMC_VTfifo;
T_EMC_PRfifo;                   // TFIFO cmds
T_EMC_WaitXfer;
T_EMC_ALUSave;
Copy Xfer buffer -> scratch;
T_EMC_VRfifo;
T_EMC_ICPort;
T_EMC_PipeFlush;
T_EMC_ALURstr;
T_EMC_InitXfer;

// Loop for remaining regions
for each region r > 2 {
    EMC_PTfifo;             // Rfifo cmds
    Shade pixels in scratch;
    EMC_VTfifo;
    T_EMC_PRfifo;               // TFIFO cmds
    T_EMC_WaitXfer;
    T_EMC_ALUSave;
    Copy Xfer buffer -> scratch;
    (Copy shaded pixels -> Xfer buffer;)
    T_EMC_VRfifo;
    T_EMC_ICPort;
    T_EMC_PipeFlush;
    T_EMC_ALURstr;
    T_EMC_InitXfer;
```

Figure 30: Use of control commands for typical shading algorithm.

The first two transfers prime the pipeline and load the first region of pixels into the shader. During each succeeding transfer, a region is loaded, a region is shaded, and (optionally) a region is unloaded. The last two transfers empty the pipeline and are not shown here.

Shading commands are loaded into the RFIFO, and copy commands are loaded into the TFIFO, the same as on Renderer boards. The semaphores provide the handshaking necessary to synchronize the operation of the two FIFOs.

APPENDIX A — IGC COMMAND EXECUTION TIMES

The following table provides execution times for the IGC command set. For EIGC and TIGC Sequencer commands this time refers to execution time within the specified sequencer. The formulas given in this column contain arguments from the argument list given in Sections IV and V.

A.1 EMC Command Execution Times

In general the execution times are pixel-memory bandwidth bound: the execution time equals the number of pixel memory accesses plus a few cycles of overhead. For instructions which use memory and the LEE, the LEE is "free", except for the *fbytes* cycles (see below). For LEE-only instructions, each byte of LEE result takes one cycle.

For instructions which use the LEE, the formulae also include the argument *fbytes*. *Fbytes* is 0 for versions which do not include an explicit *fbytes* argument (constant LEE mode and/or integer coefficients); for linear mode instructions with floating-point or fixed-point coefficients (command suffixes **_L[fdpq]**), *fbytes* is an instruction argument.

Note that all instructions require at least two cycles.

Meta instructions take 2 cycles, except for [EMC, TAS]_IFSPec and [EMC, TAS]_Ignore, which take 0 cycles. For instructions which may block a FIFO, these are minimum cycle counts.

EMC_ COMMAND	Cycle count
ClrEnab	2
SetEnab	2
EnabInv	2
SetEnabPixel	5
EnabPixel	5
MemIntoEnab	3
CryIntoEnab	2
BitTstHi	3

BitTstLo	3
TreeEqZero	len + 1 + fbytes
TreeGEZero	len + 1 + fbytes
TreeLTZero	len + 1 + fbytes
SNETree	3
Mesh	(N%8==0 ? 0 : N%8+2)+ (bits/8) + 3 + fbytes
MemEqByte	3
MemEqZero	len + 2
MemEqOnes	len + 2
MemNEZero	len + 2
MemNEOnes	len + 2
MemEqMem	2 * len + 2
MemNEMem	2 * len + 2
MemGEMem	2 * len + 2
MemGTMem	2 * len + 2
Mem2GEMem2	2 * len + 2
Mem2GTMem2	2 * len + 2
MemEqTree	len + 2 + fbytes
MemNETree	len + 2 + fbytes
MemLETree	len + 2 + fbytes
MemLTTree	len + 2 + fbytes
MemGETree	len + 2 + fbytes
MemGTTree	len + 2 + fbytes
EnabOrEqMem	4
EnabXoreqMem	5
EnabIntoCry	2
EnabIntoMem	4
MemOrEqEnab	3
MemAndEqEnab	3
LoadPixel	len + 5
Clear	len + 1
Set	len + 1
BitClr	3
BitSet	3
BitXor	3
ClrCry	2
ByteIntoMem	2
TreeIntoMem	len + 1 + fbytes
TreeClmpIntoMem	dlen + slen + 2 + fbytes
TreeIntoS	2
MemIntoS	2

Copy	MAX (3, 2 * len)
Swap	4 * len
Inc	2 * len + 1
Dec	2 * len + 1
Merge	5
LSL	MAX (3, 2 * len)
LSL4	6 * len + 1
LSR	MAX (3, 2 * len)
LSR4	MAX (6, 6 * len - 4)
ASR	MAX (4, 2 * len + 1)
ASR4	6 * len
ROL	MAX (3, 2 * len)
ROR	MAX (3, 2 * len)
Invert	2 * len + 1
Negate	2 * len + 1
AbsVal	2 * len + 3
MemPlusMem	2 * dlen + MIN(dlen,slen) + 1
MemClmpPlusMem	3 * dlen + MIN(dlen,slen) + 2
MemMinusMem	2 * dlen + MIN(dlen,slen) + 1
MemClmpMinusMem	3 * dlen + MIN(dlen,slen) + 2
MemPlusMem2	2 * dlen + MIN(dlen,slen) + 1 (add 2 if dlen>slen)
Mem2ClmpPlusMem2	3 * dlen + MIN(dlen,slen) + (dlen > slen ? 7 : 4)
MemMinusMem2	2 * dlen + MIN(dlen,slen) + 1 (add 2 if dlen>slen)
MemAndMem	3 * len + 1
MemOrMem	3 * len + 1
MemXorMem	3 * len + 1
MemPlusTree	2 * len + 1 + fbytes
TreeMinusMem	2 * len + 1 + fbytes
MemAndTree	2 * len + 1 + fbytes
MemOrTree	2 * len + 1 + fbytes
MemXorTree	2 * len + 1 + fbytes
Min	MIN (8, 5 * len + 2)
Min2	MIN (8, 5 * len + 2)
Max	MIN (8, 5 * len + 2)
Max2	MIN (8, 5 * len + 2)
OvFix	len + 2
CryIntoMem	4
MemIntoCry	2
SLoad	2
WriteS	2
MulUUn	9 * slen + dlen + 1

MulUSn	9 * slen + dlen + 1 (+1 more if dlen > slen+1)
MulSSn	9 * slen + dlen + 1 (+1 more if dlen > slen+1)
SqRoot	128
RootStep1	2 * len + 1
RootStep2	3 * len + 2
Divide	34 * (len-1) + 2
DivAddSub	3 * dlen + MIN(dlen,slen)
DivShift	MAX (4, 2 * len +1)
DivStep1	dlen + slen + 1
DivStep2	3 * len + 2
InvSqStep	2 * dlen + MIN (dlen, slen) + 1
ClampFix	len + 3
ByteToShort	7
ResetEnab	2
PushEnab	3
PopEnab	4
RestoreEnab	3
XorEnab	3
BreakEnab	4
GMax	258 * len + 2
Sample	2
FEdge	len + fbytes
MEdge	len + fbytes
ZCmp	len + fbytes (len + 1 for _S*, _Li, _Ll versions)
FLoad	len + fbytes
MLoad	len + fbytes
MLoad1	2
LLoad	len + 1 + fbytes
TblStep	len + 2
TblEntry	dlen + slen + 2 + fbytes
PixSwapN	2 * (N+1) * len + 2
PixCopyDnN	(N+1) * len + 2
PixCopyUpN	(N+1) * len + 2
ALUSave	16
ALURstr	6
Ignore	0
MetaNoOp	2
NoOp	2
NoOp2	2
PipeFlush	32 (may change later)
Hang	countably infinite

EOrWait	??
EOrTest	??
EorMemTst	24 * len + 2
Offset	2
RegLoad	3
RegLoads0	3 * n
RegLoads1	3 * n
GRegLoad	3
ICEnds	??
ICPort	?? (the transfer itself takes much longer, see text)
L2RInit	?? (the transfer itself takes much longer, see text)
R2LInit	?? (the transfer itself takes much longer, see text)
RevCopy	MAX (3, 2 * len)
LPortIn	21 (approx)
LPortInLoop	21 (approx)
LPortOut	21 (approx)
LPWeave	8 * len + 1
LPUnWeave	8 * len + 1
IFSpec	0
RModeOn	2
RModeOff	2
MCWrite	2
MCRead	2

A.2 TIGC Command Execution Times

The following table gives the execution time in rasterizer cycles for the TIGC commands. Some of the commands have fixed execution times. Texture read/write commands have execution times which depend on the number of pixels that have their local-port select flag set. For these commands, execution time is determined by the *worst-case* EMC, that is, the EMC which has the *most* pixels for which the select flag is set (designated *npix* in the table below; *npix* is in the range [0, 256]).

TAS_ Command	*Cycle count*
TAS_SetACTXXX	2
TAS_SetImAddr*n*	2
TAS_SetImAddr*n*Short	3
TAS_SetWriteModeXXX	2
TAS_SetMCnt	2
TAS_SetMSel	3
TAS_LoadACT	??
TAS_MemClk50	2 + refr
TAS_MemClk100	2 + refr
TAS_MemPreCharge	??
TAS_MemInitModeReg	??
TAS_MemRefr	7 + refr
TAS_MemRdBlock1	
TAS_MemRdBlock2	
TAS_MemRdBlock4	
TAS_MemRdBlockIleave1	
TAS_MemRdBlockIleave2	
TAS_MemRdBlockIleave4	
TAS_MemWrBlock1	
TAS_MemWrBlock2	
TAS_MemWrBlock4	
TAS_MemWrBlockIleave1	
TAS_MemWrBlockIleave2	
TAS_MemWrBlockIleave4	
TAS_MemRdScatterEven1 **TAS_MemRdScatterOdd1**	
TAS_MemRdScatterEven2 **TAS_MemRdScatterOdd2**	
TAS_MemRdScatterEven4 **TAS_MemRdScatterOdd4**	$npix = 0$: 14 (approx) $npix = 1$: 145 $npix \geq 2$: $90 + 55npix$

TAS_MemWrScatterEven1L TAS_MemWrScatterEven1H TAS_MemWrScatterOdd1L TAS_MemWrScatterOdd1H	
TAS_MemWrScatterEven2L TAS_MemWrScatterEven2H TAS_MemWrScatterOdd2L TAS_MemWrScatterOdd2H	
TAS_MemWrScatterEven4L TAS_MemWrScatterEven4H TAS_MemWrScatterOdd4L TAS_MemWrScatterOdd4H	
MemRefr	12
MemPreCharge	17
MemInitModeReg	19
GPWait	??
GPStrobe	??
GPWr	??
GPRd	??
SetVidModeIn	2
SetVidModeOut	2
SetVAddr*n*	2
SetVNxtFld*n*	2
SetVFld	2
NoOp	2
NoOp2	2
Hang	countably infinite
RModeOn	2
RModeOff	2
MCWrite	2
MCRead	2