<< MARKUP COPY >> J. Eyles

III.4.1. RENDERER FUNCTIONAL DESCRIPTION

A Renderer occupies one full-size board, with two ports on the Ring.

Each Renderer contains an array of 64 *enhanced memory chips* (EMC's), an array of 64 MegaBit Video-RAM's (VRAM's), and several controllers. The EMC's form an array of 128^2 (16,384) bit-serial pixel processing elements, which operate in Single-Instruction/Multiple DataStream (SIMD) fashion. Each pixel processor consists of a bit-serial pixel-ALU operating at 40 MHz, 208 bits of local memory, and a connection to a distributed *quadratic expression evaluator* (QEE) which supplies a local value of the bi-quadratic expression $Q(x,y) = Dx^2 + Exy + Fy^2 + Ax + By + C$ bit-serially at 40 MHz. In typical system operation, these 128^2 processing elements are mapped onto a 128 x 128 pixel region on the display, one processing element per pixel, with the local values of Q(x,y) corresponding to a processing element's *x,y* pixel address. The pixel-ALU's operate on the Q(x,y) data from the QEE and pixel-memory, storing their results back into pixel-memory. This array of pixel processors, also known as *enhanced memory*, is key to Pixel-planes' unique power. However, although we will refer to the processing elements as pixel processors, there may not always be a one-one correspondence between processors and display pixels; in fact, the pixel processors may sometimes contain data totally unrelated to pixel values, and the EMC's may be configured in a different way, so that Q(x,y) is computed over a different set of *x,y* values.

The array of pixel processors is tightly coupled to a Backing Store composed of video-RAM chips. The Backing Store provides additional memory, organized as 128 32-bit registers *per pixel processor*; that is, there are 128 *sectors* of Backing Store, each sector arranged as a 128 x 128 array of 32-bit values. The total size of a Renderer's backing store is equal to 128^3 words = 2^{21} words = 2 MegaWords = 8 MegaBytes. Each *sector* contains 16,384 words.

The function of a Renderer can be divided into 7 types of operations:

1) instructions are executed bit-serially in the 128^2 SIMD array of pixel processors
2) data is transferred between pixel processors and sectors of Backing Store
3) data is received from other devices on the Ring and written into Backing Store
4) data from Backing Store is sent to other Ring devices
5) status messages are generated (in response to incoming requests)
6) error messages are sent to the Host Interface, if an error occurs
7) two semaphores are P'ed and V'ed, for instruction stream synchronization

The Renderer has two ports on the Ring: the Image Generation Controller (IGC) Port and the

Backing Store (BS) Port. Each port processes a stream of commands. The IGC Port receives SIMD instructions for the pixel processors, and commands to transfer data between the pixel processors and backing store. The Backing Store Port receives messages containing data from other Ring devices to be written into backing store, and messages which cause backing store data to be sent to other Ring devices.

Both ports also receive commands to control a pair of hardware semaphores, and commands to send status messages; the semaphores and/or outgoing status messages can be used to synchronize the two command streams.

III.4.1—1 IGC Port

The IGC Port receives only one type of message, consisting of a stream of commands for the IGC; the destination address is ignored, except for the LSB (see below). The command stream may include regular IGC commands (executed SIMD in the 128x128 processors), transfer commands (causing transfers between the EMC's and Backing Store), send-status commands (causing a status message to be sent onto the Ring from the IGC Port), semaphore commands, and commands to initialize the IGC and the EMC's. The commands for the IGC Port are described in Section III.4.2.

The Stream Parser strips the destination address from an incoming message and then begins parsing the command stream. It passes the IGC commands which form the body of the message to the IGC, generating the correct IGC register addresses and doing the required hand-shaking with the IGC. The message body length may vary between 0 words and a maximum of 1023 words. The IGC commands are 1 to 8 words long and must be properly formatted as described below. If the Stream Parser is in the midst of processing a command when the end of the message is reached (actually the beginning of the next message), an error is generated and the Stream Parser begins parsing the next message from the beginning, and the command that was "left hanging" is probably lost. This means that IGC commands are not allowed to cross message boundaries.

If the LSB of the destination address is 1, the message is considered to be "flush-able". This means that if AEO (the all-enables-are-off signal for the pixel processors) is asserted when processing of the message begins, the message will be discarded. The commands in a flushed message will not be parsed, so any formatting errors will go undetected. Flushing is an efficiency feature; time is still required to read the message from the FIFO, but WrEnLLPR and GoLLPR are never asserted into the IGC, so there is no execution time. A flush-able message must not contain any commands which might cause AEO to be asserted, that is, which might

this stuf delete, or put as a aside; it confuses cause some of the pixel-processors to be re-enabled; furthermore, the flush-able message probably ought not to contain commands whose execution is independent of the state of the Enable registers, such as Backing Store Transfer commands. Finally, the message immediately prior to a flush-able message must have at least 4 instructions (NO-OPs if necessary) after the last instruction which affects the Enable registers, to ensure that the AEO signal has settled by the time processing of the flush-able message begins. Also, if the message prior to a flush-able message has a formatting error (ends in the middle of a command), operation of the flushing feature will be unpredictable.

III.4.1—2 Backing Store Port

The Backing Store Port receives just one type of message, containing formatted Backing Store commands. Each command consists of an opcode followed by 0 or more arguments. The number of arguments must be precisely as specified below. The destination address is treated as the opcode of the first command, since the bits of the opcode corresponding to the node address are not used. If this is not desired, set the first command of a message to be a BS_NOOP (see below). The command set for the BS Port is much smaller and simpler than that for the IGC Port, so it is fully described in this section.

Commands for the BS Port are invoked using a set of macros contained in the file *bs_commands.h*. For each macro, the first argument, 'p', is assumed to be a pointer to a message buffer containing the message destined for the BS Port of a Renderer. Each macro computes the opcode and arguments for the command, and places them into the message buffer, incrementing 'p' as appropriate.

The BS macros are as follows:

BS_NOOP (p)
A no-op.

BS_SENDSTATUS (p, id, retaddr)
Causes a status message to be transmitted from the Backing Store Port. The outgoing status message is sent to Ring address 'retaddr'; the body of the status message consists of an ID word given by the argument 'id', followed by the Renderer status word itself. See Section III.4.1-5 for more information on status messages.

BS_RECEIVE (p, sector, scanline, no_scanlines, ptr)

Causes incoming data to be written into backing store. The argument 'ptr' points to an integer array containing the data to be transferred. The argument 'sector' (range 0 to 127) specifies the sector in backing store to which the data is to be written. Data is written in scanline order (within the 128 x 128 pixel region), beginning with the scanline specified by the argument 'scanline' (range 0 to 127). The amount of data must be 128 * 'no_scanlines' words, that is, a whole number of scanline segments (where a scanline segment is defined as a row of 128 pixel values within a sector). Minimum value for 'no_scanlines' is 1, maximum is 128 (an entire sector) but subject to meximum message size restrictions (see Section III.4.1-3 below). If 'no_scanlines' goes beyond the end of the sector, it wraps around to scanline 0 of the *same* sector.

BS_TRANSMIT (p, sector, scanline, no_scanlines, ret_addr)

Causes a message containing backing store data to be transmitted from the Backing Store Port. The destination address for the outgoing message is given by 'retaddr'. The body of the outgoing message consists of 128 * 'no_scanlines' words of data from backing store; as with the **BS_RECEIVE** command, the arguments 'sector' and 'scanline' specify the backing store data to be transmitted. The maximum length for the outgoing message depends on the maximum message size specification for the destination device.

BS_PBS (p)

Causes a "P" operation to be performed on the BS Semaphore. The P operation waits for the BS semaphore counter to be non-zero, and then decrements the counter. The BS Semaphore is incremented (V'ed) by an **IGC_VBS()** or **IGC_VBSlater()** command to the IGC Port.

BS_VIGC (p)

Causes a "V" operation to be performed on the IGC Semaphore. The V operation simply increments the IGC semaphore counter. The IGC Semaphore is tested and decremented (P'ed) by an **IGC_PIGC()** command to the IGC Port.

Execution of each BS macro places a certain number of words into the message buffer. The number of words in each command (including the opcode) is as follows:

COMMAND	SIZE
BS_NOOP	1
BS_SENDSTATUS	3
BS_RECEIVE	1 + (128 * no_scanlines)
BS_TRANSMIT	2
BS_PBS	1
BS_VIGC	1

The Backing Store Port transmits two kinds of messages, status messages, and messages containing backing store data. These outgoing messages are generated by incoming commands as described above. It is important to realize that the Backing Store Port never processes two commands simultaneously. That is, when a **BS_SENDSTATUS** or **BS_TRANSMIT** command arrives, the Backing Store Port transmits the indicated outgoing message before processing the next incoming message. Similarly, a **BS_PBS** command must complete before the next command is executed. Thus, any of these 3 commands will halt the incoming message stream until the outgoing message has been transmitted, or until the semaphore P operation is complete. This can potentially lead to a deadlock situation, so the programmer must beware.

III.4.1—3 Sizing of Messages

No message, to either the IGC Port or to the BS Port, should exceed the maximum message size for that port. This maximum message size is equal to half the depth of the FIFO for that port. FIFOs for both the BS and IGC Ports will be 2Kwords deep (perhaps upgraded to 4K later), so the maximum message size will be 1024 words. This includes the destination address, so the message body is limited to 1023 words. If any message violates these rules, the receive FIFO (RxFIFO) for the port may be unable to absorb the message; this will generate an error message, since some of the incoming message will be lost, thereby corrupting the command stream.

Since IGC Port commands may not cross message boundaries, this means that the message must be terminated at the proper point so that the next command does not exceed the maximum message length.

BS Port commands are only a few words in length, except for the **BS_RECEIVE** command.

For this command, the maximum message length restriction of 1024 words puts a limit of 7 on the number of scanlines, since a message containing one **BS_RECEIVE** command with 8 scanlines of data would be 8 * 128 + 1 = 1025 words in length. This also means that any other commands in the message must not cause the total packet length to exceed 1024 words. So a sector might be divided into 32 sub-sectors of 4 scanlines each, and a message containing one **BS_RECEIVE** command sent for each sub-sector. Unfortunately this requires 32 messages just to send one sector of data to a Renderer.

For the BS Port, it is possible to exceed the maximum message length, if it can be guaranteed that the command stream can be processed at the full 20 MegaWord/sec rate supported by a Ring channel. Specifically, the Renderer can swallow **BS_RECEIVE** commands at full speed (subject to some small fixed per-command overheads). However, other BS Port commands, like **BS_SENDSTATUS**, **BS_TRANSMIT**, and **BS_PBS**, cannot be processed at a guaranteed rate, since they depend on external events such as availability of an outgoing Ring channel and receiver. So if it can be guaranteed, probably in the high-level software, that the BS Port of a given Renderer is processing only **BS_RECEIVE** commands at a given time (with perhaps a few **BS_VIGC** commands sprinkled in), then the maximum message length restriction can be ignored.

For example, a full sector of backing store data could be sent to a Renderer's BS Port in 3 messages, as follows. The first two messages would each contain one **BS_RECEIVE** command, with 4 scanlines (512 words) or data. The third message would contain the remaining 15K words of data. Since the combined length of the first two messages is greater than one half the FIFO depth, this guarantees that when RxReady is asserted to enable the third (over-sized) message, the BS Port is already processing the first message. Thus it is guaranteed that the command stream can be processed at full speed since the FIFO contains only **BS_RECEIVE** commands at this point. Of course, this also assumes that no other device is simultaneously sending messages to this Renderer's BS Port. Another scheme would be to send a **BS_SENDSTATUS** command, and then send the entire sector in one huge **BS_RECEIVE** command when the returned status message is received.

III.4.1—4 Using the Semaphores

The two hardware semaphores are used to achieve synchronization between the two Renderer command streams: that is, between IGC commands (especially transfer commands, which move data between backing store and the pixel processors) and incoming or outgoing backing store data.

Each semaphore consists of an 8-bit counter with overflow and zero-detection, and arbitration logic on the increment and decrement inputs. Both counters are zeroed when the Ring is reset. The BS Semaphore is incremented by a **IGC_VBS** command to the IGC Port, and tested and decremented by a **BS_PBS** command to the BS Port. Similarly, the IGC Semaphore is incremented by a **BS_VIGC** command to the BS Port, and tested and decremented by a **IGC_PIGC** command to the IGC Port.

For example, the BS Port command stream might include **BS_RECEIVE** commands containing a sector of backing store data, followed by a **BS_VIGC** command; the IGC Port command stream would include a **IGC_PIGC** command, followed by a command to transfer the new data into the pixel processors. So, after the entire sector of BS data had been received from the Ring, the IGC Semaphore would be incremented, causing the **IGC_PIGC** command to complete, and allowing the transfer of the new data into the pixel processors.

The contents of the semaphore counters are also visible in status messages, so the semaphore counters could be used in other ways as well. For example, the IGC Semaphore counter might be used to count the number of regions processed by that Renderer, and this value read back into a GP in a status message.

If either semaphore counter overflows, an error mesage is generated and one of the sticky error bits is set. The count value just wraps around to zero.

III.4.1—5 Status and Error Messages

Status messages are generated in response to commands to either the BS Port or the IGC Port. A **BS_SENDSTATUS** command to the BS Port (described above in Section III.4.1-2) generates an outgoing status message from the BS Port; an **IGC_SENDSTATUS** command to the IGC Port (described below in Section III.4.2) generates an outgoing status message from the IGC Port.

The format for the outgoing status message from the two ports is very similar. The outgoing status message contains 3 words: (1) the destination address, (2) the status message ID, (3) and the Renderer status word itself. The destination address is supplied as an argument to the command generating the message. The status message ID differs slightly for the 2 ports: for a BS status message, the status message ID is supplied as an argument to the **BS_SENDSTATUS** command; for an IGC status message, the upper 3 bytes of the status message ID is supplied as an argument to the command, and the lower byte is a unique board ID number blown into a PAL when the Renderer board is assembled. The Renderer status word is similar for the two ports, and is formatted as follows:

0-3	inverses of AEO signals from the pixel processors
4	<< RESERVED >>
5	<< RESERVED >>
6	TxReady asserted on BS Port
7	TxReady asserted on IGC Port
8-15	Contents of BS Semaphore counter (LSB at bit 8)
16-23	Contents of IGC semaphore counter (LSB at bit 16)
24 *	BS Semaphore overflowed (ERROR)
25 *	IGC Semaphore overflowed (ERROR)
26 *	BS RxFIFO was overwritten (ERROR)
27 *	IGC RxFIFO was overwritten (ERROR)
28 *	formatting error or bogus command in BS cmd stream (ERROR)
29 *	formatting error in IGC command stream (ERROR)
30 *	IGC timeout (64K 20 MHz cycles without a new cmd) (ERROR)
31 *	IGC "indicator" signal (on/off under IGC control)

The bits marked with '*' are available only in the status message transmitted from the IGC Port; they do not appear in the status word from the BS Port.

The protocol for generating the status message differs slightly for the two ports. When the BS Port receives a **BS_SENDSTATUS** command, it immediately attempts to acquire the outgoing port and receiver and sends the message. No further processing occurs at the BS Port until the status message has been sent. When the IGC receives a **IGC_SENDSTATUS** command, it (1) waits for any pending status or error message transmission to complete, (2) latches the return address, status message ID, and status word into the status transmitter, and (3) tells the transmitter to begin transmitting the status message. Thus, the **IGC_SENDSTATUS** command returns without waiting for successful completion of the outgoing status message (although the status information at the time of execution of the **IGC_SENDSTATUS** command is preserved), and the IGC may continue executing additional instructions . However, the *next* **IGC_SENDSTATUS** command will await successful completion of the pending status transmission.

Error messages are spontaneously generated when an error condition occurs on the Renderer, and are always transmitted from the IGC Port. An error message has the same format as a status message, but is always sent to Ring address 0. The upper 24 bits of the status message ID for an error message are garbage, while the lower byte indicates which board sent the message. The bits marked "ERROR" in the status word format given above represent various types of errors which can occur in the Renderer. When a given type of error occurs, an error strobe is asserted for one or more cycles. This error strobe causes the corresponding bit in the status word to be set, and to remain set until Reset is asserted. The logical-OR of the error strobes causes an outgoing error message to be generated from the IGC Port. If an error occurs, it will wait for any status message to be sent out for which the **IGC_SENDSTATUS** command has completed, before sending the error message. However, if an **IGC_SENDSTATUS** command is executing, it will be interrupted and that status message will be lost. Any additional errors which occur while the first error message is being transmitted will cause additional error messages to be generated.

The Renderer also has a bank of 16 LEDs on the front panel. Illumination of the each LED indicates the following conditions:

0	global all enables are off (AEO) signal is asserted
1	BusyHSPR signal from IGC is asserted
2	BS Semaphore is non-zero
3	IGC Semaphore is non-zero
4	BS Port receive FIFO is not empty
5	IGC Port receive FIFO is not empty
6	TxReady is asserted on BS Port
7	TxReady is asserted on IGC Port
8	BS Semaphore counter overflow error
9	IGC Semaphore counter overflow error
10	BS RxFIFO overwritten error
11	IGC RxFIFO overwritten error
12	formatting error or bogus command in BS cmd stream
13	formatting error in IGC command stream
14	IGC timeout error
15	IGC "indicator" signal is asserted

III.4.1—6 "All Enables Are Off" Signal

The "all enables are off" mechanism provides a global logical-NOR of the Enable registers in the array of 128^2 pixel processors. This feature can be used for several purposes, using several mechanisms.

It can be used as an efficiency enhancement: if all of the pixel processors are disabled, there is no point in executing additional IGC commands of the type which only affect Enable'd pixel processors. Another example would be collision detection,where the Enable register represents "contact" at a particular pixel or voxel. Another might be computing maxima and minima over the array of pixel processors or subsets thereof.

"All enables are off" can be used via 4 hardware mechanisms.

First, one of the LED indicators lights when AEO (global logical-NOR of Enable registers) is asserted. This will be useful for debugging purposes, and perhaps for a crude measure of IGC utilization on certain algorithms.

Second, the Renderer status word contains 4 versions of the AEO signal, each representing 32 columns in the 128x128 array. AEO0 (bit 0 of the status word) is the logical-OR of the Enables for pixels in columns 0-7, 32-39, 64-71, and 96-103 of the Renderer region (that is, 4 evenly spaced 8-pixel wide vertical stripes). Similarly, AEO1 is the logical-OR of the Enables for columns 8-15, 40-47, 72-79, and 104-111, and so on.

Third, messages to the IGC Port can be made "flush-able" by setting the LSB of the destination address. For a flush-able message, if AEO is asserted at the time processing of the message begins, then the entire message will be flushed. This is explained in detail in Section III.4.1-1. Note particularly that in any message prior to a flush-able message, the last command which might affect AEO (those in Section III.4.2.5.1-2) must be followed by at least 4 additional commands (IGC_NOOPs if necessary).

Fourth, the IGC can branch conditionally based on AEO, so certain IGC commands can execute conditionally based on AEO. This can be used to implement certain functions such as maximum and minimum.

III.4.2. IGC COMMAND INPUT

The first 2 sub-sections are for reference purposes. Section III.4.2.3 is for reference and for use by assembly language programmers. Section III.4.2.4 describes the macros used for generating IGC command input from 'C' programs.

III.4.2.1—Format of the IGC Command Stream

Messages received by the IGC Port are parsed into IGC commands by the Stream Parser. These commands consist of an opcode, followed by optional supplementary opcode and up to 6 coefficients (for the QEE). The coefficients to be expected are defined by 4 bits in the opcode which control the Stream Parser. The optional arguments, and the order in which they occur, are defined as follows:

ARGUMENTS	Bit 31	Bit 21	Bit 20	Bit 19
none	0	0	0	X
P	1	0	0	X
C	0	0	1	X
P,C	1	0	1	X
A,B,C	0	1	0	X
P,A,B,C	1	1	0	X
D,E,F,A,B,C	0	1	1	1
P,D,E,F,A,B,C	1	1	1	1
C,...,C	0	1	1	0
P,C,...,C	1	1	1	0

Bit 31 indicates that a supplementary opcode (P word) follows; bits 20 and 21 are known as the *CoefMode* field of the opcode and indicate that either 00: no coefficients follow; 01- only the constant coefficient C follows; 10- only the linear coefficients A, B, and C follow; or 11- all 6 coefficients A-F follow (unless bit 19 = 0).

Normally, bit 19 is ignored by the Stream Parser, and is part of the QEEMode setting (see below). However, when bits 20 and 21 are set, bit 19 is normally required to be set; otherwise, this illogically indicates that all 6 quadratic coefficients are sent, but the quadratic expression evaluator is in *no_coeffs* or *constant* mode and therefore would use at most the C coefficient. This combination would not, and in fact *must* not, be used for normal instructions. If bits 20 and 21 are 1, but bit 19 is 0, a special mode is invoked in which one opcode actually generates many IGC instructions, one for every C coefficient which follows; the C coefficients must have bit 31 = 0, except that the last coefficient is flagged with bit 31 = 1. This format allows compact representation of a sequence of commands using the same opcode but different C coefficients; it was designed to be used for lookup tables.

III.4.2.2—Format of IGC Command Data Words

As seen above, each IGC command has two components: the opcode and possible supplementary opcode, and the coefficients A-F (or some subset thereof); each command has two sorts of arguments, those which are encoded in the opcode and supplementary opcode, and the coefficients.

The opcode and supplementary opcode can be generated using macros contained in the header file *igc_opcodes.h* (created by the IGC microcode assembler *asmpp5*).

Opcode. For a given command, identified as **COMMANDNAME** (arg1, arg2, ...) in the command synopses, evaluation of the corresponding macro **I_COMMANDNAME()** in *igc_opcodes.h* generates most of the opcode. However, two 2-bit fields of the opcode must be specified by the user.

The *QEEMode* field, bits 18 and 19 of the opcode, defines operation of the QEE, as follows:

Bit 19	Bit 18	Mode	QEE function
0	0	——	QEE not used
0	1	constant	$Q(x,y) = C$
1	0	linear	$Q(x,y) = Ax + By + C$
1	1	quadratic	$Q(x,y) = Dx^2 + Exy + Fy^2 + Ax + By + C$

In *quadratic* mode, the full bi-quadratic expression is computed in the QEE. If *linear* mode is specified, the QEE's compute just the linear portion, Ax + By + C, and the D, E, and F coefficients are ignored; the D,E, and F registers need not be explicitly zero-ed, and their previous values will still be in the registers if it is desired to re-use them when running the QEE in quadratic mode in a subsequent instruction. Similarly, *constant* mode causes the QEE to just compute the constant C, without requiring 0's to be written to the other 5 coefficient registers and without disturbing the contents of those registers. The QEEMode field is 00 in the macro generated opcode. If the instruction does not use the QEE, this field must not be modified; if the instruction does use the QEE, it must be set to a non-zero value according to the desired QEE mode.

The *CoefMode* field, bits 20 and 21, together with bit 31 (the *long/short* bit), define which coefficients and possible supplementary opcode are to follow the opcode, as described above. The CoefMode field in the macro-generated opcode is always 00. The user must set this field according to which coefficients are to follow the instruction, as shown in the table above. For instructons which use the QEE, CoefMode is usually set to the same value as QEEMode, unless it is desired to re-use previously sent coefficient values. For instructions which do not use the QEE, but which do use the C coefficient as an integer scalar, CoefMode is set to 01 (C only), unless the previous C value is to be re-used, in which case CoefMode is set to 00 (no coeffs sent).

Note also that the combination CoefMode = 11, QEEMode = 0x (illogical in the normal usage since it specifies sending all 6 coefficients for a command which does not use the QEE in quadratic mode) flags the special lookup table mode (see Section III.4.2.1 above). This mode is used only for non-QEE instructions, usually instructions which use the C coefficient as a scalar. It generates an IGC instruction for each C coefficient which follows the opcode, ending when the first coefficient is read for which bit 31 = 1.

Supplementary Opcode. Bit 31 in the macro-generated opcode indicates whether or not the command requires a supplementary opcode. This bit should not be modified. The user can test this bit, or test to see if **P_COMMANDNAME** is defined in *igc_opcodes.h.* If so, the macro **P_COMMANDNAME** is evaluated to generate the supplementary opcode. The user need never modify the supplementary opcode; however, in rare cases, efficiency concerns may make it desirable to alter FBITS as part of a non-QEE instruction, rather than by sending a separate FBITS instruction. To do this, the FBITS field (bits 23-30) is modified; see details in IGC Microcode Assembler documentation.

Coefficients. The format for the quadratic expression evaluator coefficients (A,B,C,D,E,F) is the IEEE standard single-precision floating-point format: bits 0-22 represent the fractional portion of the mantissa, with an understood 1 to the left of the radix point; bits 23-30 represent the exponent in excess-127 form; and bit 31 is the sign-bit (the representation is sign/magnitude). Valid values for the coefficients are with exponents in the range -FBITS to 63-FBITS (FBITS is the number of fractional bits carried in the coefficients, see below). Coefficients with magnitudes outside this range are treated as 0; this includes very large and very small numbers, zero, and the exceptions defined in the IEEE standard.

For some instructions, the C coefficient is treated as a 32-bit signed integer, referred to as the 'scalar' in descriptions of the command set below. The C coefficient uses the same hardware register if it is used as a QEE coefficient or as a scalar, so C cannot be re-used by a QEE instruction if it has been overwritten by a scalar instruction, or vice versa.

III.4.2.3—Generating the IGC Command Stream

The above 2 sections show how to generate the IGC command data and how it should be formatted in the command stream. This section summarizes that information and describes another set of macros which facilitates generating IGC command input from assembly language programs.

The file *igc_opcodes.h* also contains a set of macros of the form **Ix_COMMANDNAME.** These generate the entire I opcode, including the bits QEEMode (bits 18-19) and CoefMode (bits 20-21) fields.

IGC commands can be divided into three categories:

1) those which use neither the quadratic expression evaluators (QEE) nor the scalar argument (SCA)
2) those which use the QEE ("tree" appears in the command synopsis)
3) those which use the scalar argument ("sca" appears in the command synopsis)

Those in the first category have just one Ix_ macro defined, Ix_COMMANDNAME().

Commands which use the QEE (these are identified by "tree" in the command synopsis) may operate the QEE is quadratic, linear, or constant mode, and have several options of how many coefficients to send with each mode. There are a total of nine permutations, so each command which uses the QEE has a total of nine Ix_ macros:

```
Ix_COMMANDNAME_C0 ()
Ix_COMMANDNAME_C1 ()
Ix_COMMANDNAME_L0 ()
Ix_COMMANDNAME_L1 ()
Ix_COMMANDNAME_L3 ()
Ix_COMMANDNAME_Q0 ()
Ix_COMMANDNAME_Q1 ()
Ix_COMMANDNAME_Q3 ()
Ix_COMMANDNAME_Q6 ()
```

where the letter following the command name, C, L, or Q, causes the QEEMode to be set to constant, linear, or quadratic mode respectively, and the digit following this letter causes the CoefMode to be set for 0 (no coefficients), 1 (C) coefficient, 3 (A,B,C) coefficients, or 6 (A,B,C,D,E,F) coefficients.

Commands which use the C coefficient as a scalar argument (these are identified by "sca" appearing in the commands synopsis) may send the C scalar, send no coefficient and use the previously sent C value, or use the lookup table mode, and send an array of C coefficients, each of which causes a separate execution of the instruction. Each command which uses the scalar has three Ix_ macros:

```
Ix_COMMANDNAME_S0 ()
Ix_COMMANDNAME_S1 ()
Ix_COMMANDNAME_TBL ()
```

where the _S0 form sets CoefMode to send no coefficient, _S1 sets CoefMode to send a scalar value, and _TBL sets the CoefMode to 11 which invokes the lookup table mode (since QEEMode is set to 00).

These macros can be used to generate a command as follows:

1) generate the opcode using the appropriate **Ix_COMMANDNAME** macro, according to the QEEMode desired and the number of coefficients to be sent, and add it to the command stream
2) test the opcode for bit 31 = 1, or optionally, test "ifdef P_COMMANDNAME"; if so, evaluate the macro **P_COMMANDNAME** to generate the supplementary opcode, and add it to the command stream
3) if the command is of the type that uses the QEE or scalar, add the specified coefficients to the command stream, according to the form of the Ix_ macro used:

_C0, _L0, _Q0, _S0	no coefficients (previously sent coefficients are used)
_C1, _L1, _Q1	C coefficient (_L1 and _Q1 use previous A,B,C,D,E)
_L3, _Q3	A,B,C coefficients (_Q3 uses previous D,E,F)
_Q6	A,B,C,D,E,F coefficients
_S1	C coefficient (interpreted as integer "scalar")
_TBL	a sequence of one or more C scalar coeffs, terminated by the first one with bit 31 = 1

It is important that the formatting of the command stream be precisely as specified, else incorrect results will result or the Renderer will hang.

III.4.2.4—The IGC Macros

The simplest way to generate IGC command data, from a 'C' program, is by using the set of macros contained in the file *igc_commands.h* (generated by the IGC microcode assembler). These macros are built on top of the Ix_, I_ and P_ macros described above.

For every command described under Section III.4.2.5 below, *igc_commands.h* contains either 1, 3, or 9 macros. Each macro represents one of several forms of the command. The arguments to the IGC macro are a pointer to a message buffer to write the command stream, followed by the arguments to the command as described in Section III.4.2.5, followed by possible QEE coefficients or scalar. When evaluated, the IGC macro places a number of 32-bit words into the message buffer. The user must declare and initialize the pointer.

There is no error checking on the arguments to these macros; the user must insure that the arguments are legal, according to the rules described below in the description of the command set.

IGC commands can be divided into three categories:

1) those which use neither the quadratic expression evaluators (QEE) nor the scalar argument (SCA)
2) those which use the QEE ("tree" appears in the command synopsis)
3) those which use the scalar argument ("sca" appears in the command synopsis)

Those in the first category just have one IGC macro. For example, the command SETENABS() has one IGC macro, IGC_SETENABS (p). This macro simply places the opcode for SETENABS at the location pointed to by 'p', and bumps 'p'. Were SETENABS an instruction which uses the supplementary opcode, it would be placed at the next location and 'p' bumped again.

Commands which use the QEE (these are identified by "tree" in the command synopsis) may operate the QEE is quadratic, linear, or constant mode, and have several options of how many coefficients to send with each mode. There are a total of nine permutations, so each command which uses the QEE can be invoked using one of nine different IGC macros. For example the command MEMpluseqTREE(dst,src,len)can be invoked using the following macros:

```
IGC_MEMpluseqTREE_C0 (p, dst, src, len)
IGC_MEMpluseqTREE_C1 (p, dst, src, len, C)
IGC_MEMpluseqTREE_L0 (p, dst, src, len)
IGC_MEMpluseqTREE_L1 (p, dst, src, len, C)
IGC_MEMpluseqTREE_L3 (p, dst, src, len, A, B, C)
IGC_MEMpluseqTREE_Q0 (p, dst, src, len)
IGC_MEMpluseqTREE_Q1 (p, dst, src, len, C)
IGC_MEMpluseqTREE_Q3 (p, dst, src, len, A, B, C)
IGC_MEMpluseqTREE_Q6 (p, dst, src, len, A, B, C, D, E, F)
```

In each IGC macro, the letter following the command name, C, L, or Q, indicates whether the QEE is to be operated in *constant* (Q(x,y) = C), *linear* (Q(x,y) = Ax + By + C), or *quadratic* ($Q(x,y) = Dx^2 + Exy + Fy^2 + Ax + By + C$) mode. The digit following this letter represents how many coefficients are to be sent with the instruction. As described above, with the quadratic mode, one may send all 6 coefficients, just send A, B, and C and use the previous values of D, E, and F, just send C and use the previous values of the other 5 coefficients, or send none of the coefficients and just use the previous values. Similarly, with the linear mode, one may send A, B, and C, just C, or none of the coefficients. With constant mode, one may send the C coefficient or re-use the previous C coefficient. (CAUTION: Previously sent coefficient values may not be re-used after changing using the FBITS() command, or after re-initializing or reloading the microcode store of the IGC).

Commands which use the C coefficient as a scalar argument (these are identified by "sca" appearing in the commands synopsis) can be invoked using one of three IGC macros. One may send the C scalar (the C coefficient interpreted as a 32-bit signed integer), send no coefficient and use the previously sent C value, or use the lookup table mode, and send an array of C coefficients, each of which causes a separate execution of the instruction. For example, the command MEMeqSCA(dst,dlen) can be invoked using these IGC macros:

```
IGC_MEMeqSCA_S0 (p,  dst, dlen)
IGC_MEMeqSCA_S1 (p, dst, dlen, S)
IGC_MEMeqSCA_TBL (p, dst, dlen, PTR)
```

The S0 form sends no scalar value, using the previously sent one. The S1 form sends a scalar value S. The TBL form uses the argument PTR as a pointer to an 'int' array containing a number of S values. The command is executed once for each value of S in the PTR array, up to and including the first value with MSB = 1. Note that this TBL form can only be used in siuations where only the 31 LSB's of the scalar are used, since the MSB is used as a flag indicating the final value. Also, the scalar uses the same IGC register as the C coefficient for QEE expressions. Thus a scalar value will be overwritten by a QEE instruction which writes a C coefficient, and vice versa.

III.4.2.5—The IGC Command Set

The IGC command set can be divided into several categories: commands that are executed SIMD in the 128x128 array of pixel-processors in the enhanced-memory chips (EMC's); commands to transfer data between the EMCs and the VRAM backing store; commands to initialize and read/write the microcode store of the Image Generation Controller; commands to configure the quadratic expression evaluators (QEE's) of the EMC's; and commands to control the hardware semaphores.

III.4.2.5.1—Commands for the SIMD Pixel-Processor Array

The array of pixel processors execute the same instruction stream in parallel. Pixels cannot communicate with each other; each can write results only into its own pixel-memory. Each pixel processor consists of the following:

Memory:
208 bits of random access memory. The programmer usually divides this memory into segments of various lengths, each of which contains an unsigned or two's complement signed integer. This segmentation is the same at each pixel.

ALU:
(Arithmetic and Logic Unit) performs operations on memory segments, the Enable register, the Carry register, QEE results, and the scalar. Results may be written into memory segments, the Enable register, and the Carry register.

Enable Resister:
Used to condition most writes into pixel-memory. In particular, most instructions which generate arithmetic/logical results write those results into pixel memory only at *enabled* pixels, meaning those pixels where the Enable register contains a '1'.

Carry Register:
May contain the carry result from the last arithmetic operation, for both enabled and disabled

pixels. However, this Carry register was *not* designed to be used in the conventional way, to convey information fom one IGC instruction to a later one; it does not reliably contain overflow/underflow information from an instruction, unless the instruction is specifically designed in this way. Instructions which use the Carry register automatically clear it before execution of the instruction begins, so it need not be explicitly cleared before an operation. *(At some point, commands which disturb the CRY register should be flagged).*

All pixel-ALU's execute the same instruction on each cycle, but memory writes on arithmetic operations are conditioned by each pixel's Enable register. A pixel 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.

The instruction set for the SIMD pixel-processor array resembles that of a simple micro-processor. Operands may include: arbitrary length signed or unsigned integers in pixel-memory; the constant, linear, or quadratic QEE result computed from floating-point coefficients of the instruction; the scalar; the Enable register, and the Carry register.

Integers defined in pixel memory have their LSB at their lowest address. Memory segments are identified with the notation **mem[lsb, len]**. For example, an 8-bit integer in the memory segment at bits 24 through 31 (with LSB at bit 24) is denoted **mem[24 : 8]**. Contents of memory segments may represent unsigned or two's-complement signed integers. For many instructions, it does not matter if the contents of the memory segment are treated as signed or unsigned; for others it does, and these are noted. Each memory segment must, of course, be wholly containedwithin the 208 bits of pixel-memory. Maximum length of memory segments is at least 40 bits in all cases, but actually depends upon the specific instruction, as described below.

The QEE result ($Dx^2 + Exy + Fy^2 + Ax + By + C$) is always two's-complement signed. It may either be fully computed to its sign-bit (identified with the notation **tree**), or some fixed number of bits of the QEE result may be computed (identified with the notation **tree[n]**); when **tree[n]** is specified, the QEE result is always sign-extended to **n** bits, regardless of its actual length. The QEE result is treated the same in an instruction, regardless of whether the full quadratic form, the linear form ($Ax + By + C$), or the constant form (C), is generated; it is simply denoted as **tree** in the command set description, and its actual form is specified by the user when specifying each instance of the command.

Before computing the QEE result, the floating-point coefficients are converted to fixed-radix form, with a number of fractional bits given by the setting for FBITS; the coefficients are truncated, and this truncation is always towards zero. The QEE result is computed exactly from these truncated coefficients, and then the QEE result is truncated to an integer; unlike the coefficient truncation, this truncation is always downward, hence negative values are truncated *away* from zero. 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'. The coefficient truncation can also give suprising results; for example, with an FBITS setting of 10, the linear expression 0.1x + y + 0 will evaluate to the integer 0 at x=10, y=0, since 0.1 is truncated to 102/1024, the QEE result at pixel (10,0) is 1020/1024, and this is truncated to 0. On the other hand, -0.1x + y + 0 evaluates to the integer -1 at x-10, y=0. These surprising and mildly disconcerting results may be described by the following general rule:

> max. error in a linear QEE result is 1 if FBITS = $\log_2$ (screen dimension) + 1.

Similarly, for full quadratic expressions:

> max. error in a quadratic QEE result is 1 if FBITS = 2* [$\log_2$ (screen dimension)+1].

Since an error of 0 is impossible, an error bound smaller than 1 makes no sense; this is because, although FBITS fractional bits of the coefficients are generated, only the integer portion of the QEE results are available at the pixel-ALU and pixel-memory. In summary, two truncations occur when floating-point coefficients are supplied to the EMC's, which are essentially integer devices: (1) the coefficients are truncated, towards zero, during the conversion to fixed-point, (2) the QEE result compued from these fixed-point coefficients is truncated, downwards, to an integer.

For some instructions, the C coefficient is treated as a 32-bit integer, denoted **sca**, rather than as a coefficient for the QEE. Its value is the same at all pixel-processors. Note that this is different from using the QEE in constant (Q(x,y) = C) mode, where C is a floating-point number. This scalar is denoted **sca[n]** when only its **n** bits are to be used; if **n** > 32, the value is sign-extended. Use of **tree** and **sca** in an instruction are mutually exclusive.

Maximum length for memory segments used in operations that do not use the QEE result, denoted *dlen* and *slen* in the instruction syntax, is 128; this includes operations that use the scalar coefficient **sca**, although **sca** is sign-extended if the computation proceeds beyond 32 bits. Maximum length of memory segments for instructions which use the QEE result, denoted *len*, is 73-FBITS (always at least 43).

Minimum value for *len*, *dlen*, and *slen* is 1, unless otherwise noted.

No error checking is done on arguments to the instructions. Unpredictable behavior may occur if illegal arguments are given.

(If necessary, microcode could be written to test for length arguments of 0, with the penalty that the maximum for 'dlen' and 'slen' be reduced to 127. See the IGC hardware designer if you require zero-checking).

Instructions for the SIMD array can be divided into several categories: the FBITS instruction, instructions to modify the Enable register, instructions to store the Enable register, arithmetic/logical instructions, global instructions, and special-purpose instructions.

IV.4.2.5.1—1 Changing Number of Fractional Bits

The coefficients (A,B,C,D,E, and F) sent with a command to be used by the quadratic expression evaluator registers are IEEE-standard single-precision floats. Since the QEE computes in integer arithmetic, these floating-point coefficents nust be converted to a bit-serial 2's complement fixed-point representation. The number of fractional bits in the fixed-point representation (FBITS) is variable between 0 and 30. The command

FBITS(N)

changes the number of fractional bits to N.

For instructions which follow an FBITS instruction, all QEE coefficients must be sent again, even if they are the same as previously loaded values.

The user may decrease FBITS to speed up instruction execution, or increase FBITS to provide greater precision. It is important to remember that floating-point coefficients are truncated to FBITS fractional bits, and therefore precision can be lost, depending on the magnitude of the coefficient and the setting for FBITS. (See the discussion above).

Additionally, since the number of bits of coefficients that the hardware can support is fixed, the maximum allowable coefficient exponent is dependent on the value of FBITS: any coefficient with an exponent larger than 63-FBITS is treated as zero. Also, the maximum value of the *len* argument, in instructions which use the QEE result, is equal to 73-FBITS.

For instructions which do not use the QEE result, including instructions which interpret the C coefficient as a *scalar*, the setting of FBITS affects neither the speed of execution, nor the precision of the results, nor the valid range for the instruction arguments.

(It may also be possible to change FBITS as part of a regular command, if that command does not use the QEE result. This may be done if efficiency concerns dictate. See the above sections and/or the IGC hardware designer for details.)

III.4.2.5.1—2 Instructions to Modify the Enable Register

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

instruction		*synopsis*			*see note*
CLRENABS	()	enable	=	0	
SETENABS	()	enable	=	1	
ENABINV	()	enable	=	!enable	
MEMIntoENAB	(src)	enable	=	mem [src : 1]	
TREEeqZERO	()	enable	&&=	(tree == 0)	
TREEgeZERO	()	enable	&&=	(tree >= 0)	
TREEltZERO	()	enable	&&=	(tree < 0)	
MESH	(len)	enable	&&=	(tree[len] == 0)	
GRID	(len)	enable	&&=	(~tree[len] == 0)	
MEMeqZERO	(src, slen)	enable	&&=	(mem [src : slen] == 0)	
MEMeqONES	(src, slen)	enable	&&=	(mem [src : slen] == ~0)	
MEMneZERO	(src, slen)	enable	&&=	(mem [src : slen] != 0)	
MEMeqSCA	(src, slen)	enable	&&=	(mem [src: slen] == sca [slen]	
MEMgeSCA	(src, slen)	enable	&&=	(mem [src: slen] >= sca [slen]	
MEMgtSCA	(src, slen)	enable	&&=	(mem [src: slen] > sca [slen]	
MEMeqMEM	(lsrc, src,slen)	enable	&&=	(mem [lsrc : slen] == mem [src : slen])	
MEMneMEM	(lsrc, src,slen)	enable	&&=	(mem [lsrc : slen] != mem [src : slen])	
MEMgeMEM	(lsrc, src,slen)	enable	&&=	(mem [lsrc : slen] >= mem [src : slen])	1
MEMgtMEM	(lsrc, src,slen)	enable	&&=	(mem [lsrc : slen] > mem [src : slen])	1
MEM2geMEM2	(lsrc, src,slen)	enable	&&=	(mem [lsrc : slen] >= mem [src : slen])	2
MEM2gtMEM2	(lsrc, src, slen)	enable	&&=	(mem [lsrc : slen] > mem [src : slen])	2
MEMeqTREE	(src, len)	enable	&&=	(mem[src : len] == tree[len])	
MEMneTREE	(src, len)	enable	&&=	(mem[src : len] != tree[len])	
MEMleTREE	(src, len)	enable	&&=	(mem [src : len] <= tree)	3
MEMltTREE	(src, len)	enable	&&=	(mem [src : len] < tree)	3
MEMgeTREE	(src, len)	enable	&&=	(mem [src : len] >= tree)	3
MEMgtTREE	(src, len)	enable	&&=	(mem [src : len] > tree)	3
ENABandeqMEM	(src)	enable	&&=	mem [src:1]	
ENABandeqMEMBAR	(src)	enable	&&=	~mem [src:1]	

ENABoreqMEM	(src)	enable	|=	mem [src:1]
ENABxoreqMEM	(src)	enable	^^=	mem[src:1]
CRYIntoENAB	()	enable	=	carry
ENABoreqCRY	()	enable	|=	carry

III.4.2.5.1—3 Instructions to Store the Enable Register

These instructions are used to store the Enable register into memory, or the Carry register. These writes occur regardless of the contents of the Enable register.

instruction		*synopsis*		
ENABIntoMEM	(dst)	mem [dst: 1]	=	enable;
MEMoreqENAB	(dst)	mem [dst: 1]	||=	enable;
MEMandeqENAB	(dst)	mem [dst: 1]	&& =	enable;
ENABIntoCRY	( )	carry	=	enable;

III.4.2.5.1—4 Arithmetic and Logical Instructions

These arithmetic and logical instructions operate on signed QEE results and signed or unsigned integers in pixel memory.

Instructions may have up to 3 possible pixel-memory operands: the destination operand *dst*, and 2 source operands, *lsrc* and *src*. Unless otherwise noted, the operands need not be distinct, but *dst* must not partially overlap *lsrc* or *src*. Unless otherwise noted, *dst* and *lsrc* must have the same length , *dlen*. The source *src* may have a different length, *slen*.The following rules apply to instructions which have separate length arguments:

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

As described above, maximum value for the *dlen* and *slen* arguments is 128, maximum value for the *len* argument is 73-FBITS, and minimum value for all length arguments is 1 (unless otherwise noted). Segment lengths must be contained within the 208 bits of pixel memory.

The arithmetic instruction writes are all conditioned by the Enable register, though the ALU actually carries out the instruction at each pixel, affecting the Carry register in the process. (It is possible to write these instructions so that the result is written into pixel-memory regardless of the setting of the Enable register. Normally, if this is desired, it is simple to first set the Enable register using a SETENABS instruction. However, Enable-independent instructions could be supplied. See the IGC hardware designer for details.)

instruction		*synopsis*			*see note*
CLEAR	(dst, dlen)	mem [dst : dlen]	=	0	
SET	(dst, dlen)	mem [dst : dlen]	=	~0	
CLRCRY	()	carry	=	0	11
TREEintoMEM	(dst, len)	mem [dst : len]	=	tree [len]	
TREEBARintoMEM	(dst, len)	mem [dst : len]	=	~tree [len]	
TREEclmpintoMEM	(dst, len)	mem [dst : len]	=	tree [len]	17
SCAintoMEM	(dst, dlen)	mem [dst : dlen]	=	sca [dlen]	
CPY	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen]	12
SWAP	(dst, src, dlen)	mem [dst : dlen]	<—>	mem [src : dlen]	6
INC	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] + 1	
DEC	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] - 1	
SHIFTL	(dst, src, dlen, n)	mem [dst : dlen]	=	mem [src : dlen] << n	4
SHIFTR	(dst, src, dlen, slen, n)	mem [dst : dlen]	=	mem [src : slen] >> n	1,5
INVERT	(dst, src, dlen)	mem [dst : dlen]	=	~mem [src : dlen]	
NEGATE	(dst, src, dlen)	mem [dst : dlen]	=	- mem [src : dlen]	2
MEMpluseqSCA	(dst, src, dlen)	mem [dst : dlen]	=	mem [src : dlen] + sca [dlen]	
MEMplusMEM	(dst, lsrc, src, dlen, slen)	mem [dst : dlen]	=	mem [lsrc : dlen] - mem[src:slen]	7
MEMpluseqMEM	(dst, src, dlen, slen)	mem [dst : dlen]	+=	mem [src : slen]	7
MEMclmppluseqMEM	(dst, src, dlen, tmp)	mem [dst : dlen]	+=	mem [src : dlen]	1,9,15
MEMminusMEM	(dst, lsrc, src, dlen, slen)	mem [dst : dlen]	=	mem [lsrc : dlen] - mem[src:slen]	7
MEMminuseqMEM	(dst, src, dlen, slen)	mem [dst : dlen]	-=	mem [src : slen]	7
MEMplusMEM2	(dst, lsrc, src, dlen, slen)	mem [dst : dlen]	=	mem[lsrc:dlen] - mem [src : slen]	8
MEMpluseqMEM2	(dst, src, dlen, slen)	mem [dst : dlen]	+=	mem [src : slen]	8
MEM2clmppluseqMEM2	(dst, src, dlen, tmp)	mem [dst : dlen]	+=	mem [src : dlen]	2,9,16
MEMminusMEM2	(dst, lsrc, src, dlen, slen)	mem [dst : dlen]	=	mem [lsrc : dlen] - mem[src:slen]	8
MEMminuseqMEM2	(dst, src, dlen, slen)	mem [dst : dlen]	-=	mem [src : slen]	8
MEMandMEM	(dst, lsrc, src, dlen)	mem [dst : dlen]	=	mem[lsrc:dlen] & mem [src : dlen]	
MEMandeqMEM	(dst, src, dlen)	mem [dst : dlen]	&=	mem [src : dlen]	
MEMorMEM	(dst, lsrc, src, dlen)	mem [dst : dlen]	=	mem[lsrc:dlen] | mem [src : dlen]	
MEMoreqMEM	(dst, src, dlen)	mem [dst : dlen]	|=	mem [src : dlen]	

MEMxorMEM	(dst, lsrc, src, dlen)	mem [dst : dlen]	=	mem[lsrc:dlen] ^ mem [src : dlen]
MEMxoreqMEM	(dst, src, dlen)	mem [dst : dlen]	^=	mem [src : dlen]
MEMpluseqTREE	(dst, src, len)	mem [dst : len]	=	mem [src : len] + tree [len]
TREEminusMEM	(dst, src, len)	mem [dst : len]	=	tree[len] - mem [src : len]
MEMandTREE	(dst, src, len)	mem [dst : len]	=	mem [src : len] & tree [len]
MEMorTREE	(dst, src, len)	mem [dst : len]	=	mem [src : len] | tree [len]
MEMxorTREE	(dst, src, len)	mem [dst : len]	=	mem [src : len] ^ tree [len]
CRYintoMEM	(dst)	mem [dst : 1]	=	carry

III.4.2.5.1—5 Global Instructions

These instructions use the AEO signal to perform global computations, that is, computations which use all or some of the processing elements in the Renderer.

GMAX	(dst, src, dlen, tmp)	mem [dst : len]	=	MAX { mem [src : len] }	9,14
GMIN	(dst, src, dlen, tmp)	mem [dst : len]	=	MIN { mem [src : len] }	9,14

III.4.2.5.1—6 Special Purpose Instructions

These instructions were written to support special tasks for Rendering.

FTECT	( )	enable	=	((tree[1]) == 1)	
FEDGE	( )	enable	=	(tree >= 0)	
SEDGE	(src)	enable	=	(mem[src:1] && (tree >= 0)	
FEDGEBAR	( )	enable	=	(tree < 0)	
SEDGEBAR	(src)	enable	=	(mem[src:1] && (tree < 0)	
EDGE2	( )	enable	&&=	(tree >= 0)	
		carry	&&=	(tree < 0)	
STRIPEDGE	(src, dst)	enable	&&=	(tree >= 0)	
		mem[dst:1]	=	mem[src:1] && (tree < 0)	10
MEMEDGE	(dst)	mem[dst:1]	=	(tree >= 0)	10
FCMEMA	(src, len)	enable	=	(mem [src : len] <= tree)	3
SCMEMA	(src, len, aux)	enable	=	mem[aux:1] && (mem [src:len]<=tree)	3
OVFIX	(dst, dlen, tmp)	if (carry)	then	mem[dst : dlen] = ~0	9
TBLENTRY	(dst, src, dlen, slen)	if (mem [src : slen] = index)	then	mem[dst : dlen] = entry	13
SPLAT	(dst, len, tmp)	enable = (tree >= 0)	and	mem[dst : len] = tree [len]	18

<u>Notes</u>:

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 it is zero-extended to the full length of the tree result.

4) These restrictions apply: 0 <= n < dlen

5) These restrictions apply: dlen >= slen - n , 0 <= n < slen.

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

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) 'Tmp' is a memory location for temporary use. It becomes undefined.

10) Mem[dst:1] is written for all pixels, regardless of the value of the Enable register. 'Dst' and 'src' may point to the same memory location.

11) it is not normally necessary to explicitly clear the Carry prior to an instruction; this instruction is executed regardless of the state of the Enable register

12) There is no restriction on overlapping of the 'dst' and 'src' memory segments.

add footnote for Carry result of add's, etc.

13) Does lookup and write for a table entry in one instruction. 'Index' is slen LSB's of scalar, 'entry' is next dlen bits of scalar. If index matches mem[src : slen], then entry is written into mem [dst : dlen]. Only affects enabled pixels, and the Enable register is not disturbed.

14) Computes global maximum and minimum over the Renderer region. GMAX computes maximum value of mem[src:dlen] over all Enabled pixels (treating it as an unsigned value) and writes this maximum into mem[dst:dlen] for all Enabled pixels. GMIN behaves similarly.

15) The result is clamped to all 1's if overflow occurs.

16) The result is clamped to the maximum respresentable positive value if overflow occurs, to the mimimum respresentable negative value if underflow occurs.

17) Mem[dst:len] is clamped to all 1's if the QEE result is larger than the maximum representable value, to 0 if the QEE result is negative.

18) Mem [tmp : len] is for temporary use; it becomes undefined. QEE result is loaded into mem [dst : len] only if it was non-negative.

III.4.2.5.2—Transfer Commands

Three commands are available for controlling transfers between the pixel processors and the backing store. They are executed independently of the state of the Enable register.

```
BSLOAD(sector)              /* initiate a "load" operation, transferring the specified      */
```

```
                      /*    sector of backing store into bits 0-31 of pixel memory      */
BSSTORE(sector)       /* initiate a "store" operation, transferring bits 0-31           */
                      /*    of pixel memory into the specified sector of backing store  */
BSWAIT()              /* wait for conclusion of the active transfer operation, used     */
                      /*    when execution of further instructions requires that a      */
                      /*    previously initiated transfer operation be complete         */
```

For **BSLOAD()** and **BSSTORE()**, the argument 'sector' must be in the range 0 - 127.

(Note, for the C simulator, the 'sector' argument is interpreted as the number of nibbles to transfer, unless the value is greater than 100 in which case the number of nibbles is 128 * (sector -100); in other words, the argument 116 causes an entire transfer of 4096 nibbles).

Execution of a **BSSTORE()** or **BSLOAD()** command merely *intiates* a Transfer operation. It may be followed by ordinary IGC commands, thus allowing overlapping of Transfer operations with execution of ordinary IGC commands. However, these commands must not access bits 0-31 of pixel-memory, which are involved in the Transfer operation. If these addresses are mistakenly accessed, the offending IGC command will not be correctly executed; however, the Transfer operation will not be disturbed, nor will the contents of pixel-memory be corrupted.

When it becomes necessary to access bits 0-31, the IGC must halt, pending completion of the Transfer operation; the **BSWAIT()** command causes the IGC to wait in a tight loop until any current Transfer operation is complete. If no Transfer operation is executing, **BSWAIT** returns immediately. No **BSWAIT** is needed prior to initiating another Transfer operation, since the **BSLOAD** and **BSSTORE** commands wait for completion of any pending Transfer before initiating another.

Other times, it may be necessary to wait for completion of a Transfer operation so that data in Backing Store can be transmitted to other Ring devices. Again, the **BSWAIT** command can used to wait for completion of the BS operation, followed by a command to do a V operation on the BS Semaphore. Alternatively, the command initiating the Transfer operation can be immediately followed by a **VBSlater** command, which causes a V operation on the BS Semaphore immediately upon completion of the Transfer operation. This means that the IGC command stream need not contain a **BSWAIT** and **VBS** to enable transmission of the data from the Backing Store Port. Of course, A **BSWAIT** is still required before any IGC instructions which access bits 0 - 31.

III.4.2.5.3—Commands for Initializing the IGC

On power-up, or if the Ring must be reset (possibly because the IGC is hung with faulty microcode, invalid opcodes, or waiting on an external handshake signal), it is necessary to initialize the IGC; this involves loading the microcode store and setting the FBITS register. It may also be desired to reload some or all of the microcode on-the-fly if more than one set of IGC microcode is to be used.

Microcode is loaded using the following commands:

```
RMODEON()          /* put the IGC in RMode                                     */
RMODEOFF()         /* cause the IGC to exit RMode, and prepare for normal input */
MCWRITE(addr)      /* load 'sca [32]' ino the specified address in microcode    */
MCREAD(addr)       /* set the IGC'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 Renderer operation) */
```

To load the microcode store, the IGC is first put into RMode using two **NOOP()** commands followed by an **RMODEON()** command (the **NOOP**'s are only needed for an "on-the-fly" microcode reload, to insure that earlier instructions are not corrupted). After a Ring reset, the IGC is guaranteed to be in RMode already. Next, for every address that is to be loaded, an **MCWRITE()** command is issued; note that **MCWRITE()** is a two word command, so bits 20 and 21 of the opcode should be set for "C only". Finally, a **NOOP()** or **MCREAD(0)** command is used to reset the IGC program counter, and RMode is exited using an **RMODEOFF()** or **FBITS()** command.

After RMode has been entered using **RMODEON()**, it is most important that no commands other than **MCWRITE**, **MCREAD**, and **NOOP** (but not **NOOP2**) be issued, until microcode loading is complete and RMode is exited using **RMODEOFF()**.

Very short initialization sequences (less than 20 words or so of microcode are loaded) after power-up or Ring reset may not allow the IGC logic sufficient time to flush itself out.

The standard location for the IGC microcode is in the file *igc_microcode.h,* in an the initialized array *int igc_microcode[]*; this file is generated by the IGC microcode assembler *asmpp5*, from microcode source provided by the IGC hardware designer.

Initialization after power-up or Reset should also include an **FBITS()** command to load the fractional bits register.

III.4.2.5.4—Commands for Configuring the Pixel Processors

Two special commands are used for configuring the quadratic expression evaluators of the EMCs. These are:

```
SUPCFG(bits)                /* shift the first 'bits'  bits of the 1's complement  of 'sca'      */
                            /*    into the configuration register of all EMCs                      */
SEPCFG(chip_no, bits)       /* shift the first 'bits' bits of 'sca'  into the                     */
                            /*    configuration register of the EMC indicated by 'chip_no'         */
```

Note that the data must be inverted for **SUPCFG()**. The argument 'chip_no' must be in the range 0 to 63. The argument 'bits' must be in the range 2 to 32.

The QEE configuration sequence must be terminated with two **NOOP()** commands, to insure that a new set of coefficients is not shifted into the QEE's prior to configuration being completed.

III.4.2..5—Semaphore Control Commands

These commands are used for controlling the 2 hardware semaphores, in conjunction with corresponding commands to the Backing Store Port.

```
VBS( )                      /* perform a V operation on the Backing Store Port Semaphore          */
                            /*    (increment the semaphore counter)                                */
VBSlater( )                 /* perform a V operation on the Backing Store Port Semaphore          */
                            /*    after any pending Transfer operation has completed               */
```

The VBS commands are used to increment the BS Semaphore; this usually is used to allow a BS_PBS command in the BS Port command stream to terminate, thereby allowing subsequent BS commands to execute.

VBS increments the BS Semaphore immediately.

VBSlater increments the BS Semaphore upon completion of any Transfer operation (initiated by a **BSLOAD** or **BSSTORE** command) which is in progress; the BS semaphore is incremented immediately if no Transfer operation is in progress. After a **VBSlater** is issued, no additional **VBS** or **VBSlater** commands may be issued until a pending Transfer operation is complete.

```
PIGC( )                    /* perform a P operation on the IGC Port Semaphore         */
                           /*    (wait for the semaphore counter  to be non-zero and then   */
                           /*     decrement it)                                        */
```

The **PIGC** command waits in a tight loop until the IGC Semaphore is non-zero, and then decrements the semaphore. The IGC Port Semaphore is V'ed by a **BS_VIGC** command. It would typically be used to force the IGC command stream to wait until data to be processed by the IGC has been loaded from the Ring into backing store, or until data from a backing store sector has been moved to another Ring device. The two commands immediately following a **PIGC()** command must not use the QEEs, because of the dynamic nature of the QEEs and the fact that the **PIGC** execution time is unbounded.

III.4.2.5.6—Miscellaneous Commands

The following are miscellanous commands.

```
NOOP  ( )                  /*  no operation                                              */
NOOP2  ( )                 /*  no operation (2 word version, includes supplementary opcode)  */
HANG  ( )                  /*  hangs the IGC in a tight loop (for debugging purposes)     */
SENDSTATUS(id)             /* send status message to return address given by 'sca [32]',   */
                           /*     'id' specifies status message ID word, except that the 8 LSBs */
                           /*     are replaced by the PAL-programmed board ID number        */
INDON ()                   /* turn on IGC "indicator" signal (it is initially off after system reset) */
INDOFF ()                  /* turn off IGC "indicator" signal                            */
```

III.4.2.6—Programming Hints and Pitfalls

The following cautions must be observed when generating Renderer commands. Most of these are mentioned above. This list summarizes potential mistakes.

1) No error checking is done for command arguments. If illegal values are used, unpredictable operation will result, including possible hanging of the Renderer.

2) Coefficients of QEE commands are truncated to fixed-point (number of fraction bits is set using the FBITS command); this truncation is towards 0. The tree result is computed using these truncated values. The tree result is then truncated to an integer; this truncation is down, not towards zero.

3) Tree coefficients may normally be re-used, that is, coefficients from the previously sent tree instruction need not be re-sent if the same coefficient value is to be used (see the IGC macros). However, coefficients must always be re-sent if microcode is reloaded, or if the value for FBITS is changed. Also, 'scalar' occupies the same hardware register as the C tree coefficient, so writing one will overwrite the other.

4) Instructions which use the QEE must not follow 1 or 2 instructions after any instruction which may take very long to execute, or which alters the QEE configuration; these include **BSSTORE, BSLOAD, BSWAIT, SENDSTATUS, PIGC, SUPCFG, SEPCFG** and others (this is mentioned in the description of such instructions).