GLINT 300SX

Core Architecture Manual

Dave Baldwin

Amendment History

Date	Issue	Name	Action
	1	DB	First Draft.
	2	DB	Final version for first spin of GLINT 300SX

WARNING

Contents

Change History

1. Architecture Overview

1.1. The GLiNT concept

The overall architecture of the GLiNT chip is best viewed using the software paradigm of a message passing system. In this system all the processing blocks are connected in a long pipeline with communication with the adjacent blocks being done through message passing. Between each block there is a small amount of buffering, the size being specific to the local communications requirements and speed of the two blocks.

The message rate is variable and depends on the rendering mode. The messages do not propagate through the system at a fixed rate typical of a more traditional pipeline system. If the receiving block can not accept a message, because its input buffer is full, then the sending block stalls until space is available.

The message structure is fundamental to the whole system as the messages are used to control, synchronise and inform each block about the processing it is to undertake. Each message has two fields - a 32 bit data field[1] and a 9 bit tag field. The data field will hold colour information, coordinate information, local state information, etc.. The tag field is used by each block to identify the message type so it knows how to act on it.

Each block, on receiving a message can do a number of things:

- Not recognise the message so it just passes it on to the next block.

- Recognise it as updating some local state (to the block) so the local state is updated and the message terminated, i.e. not passed on to the next block.

- Recognise it as a *processing* action, and if appropriate to the unit, the processing work specific to the unit is done. This may entail sending out new messages such as Colour and/or modifying the initial message before sending it on. Any new messages are injected into the message stream before the initial message is forwarded on. Some examples will clarify this.

  The Depth Block will receive a message 'new fragment[2]' and will calculate the corresponding depth and do depth the test. If the test passes then the 'new fragment' message is passed to the next unit. If the test fails then the message is modified and passed on. The temptation is not to pass the message on when the test fails (because the pixel is not going to be updated) however other units down stream need to keep their local DDA units in step.

  The Texture Read Unit will get a 'new fragment' message and, assuming this unit is enabled, will calculate the texture map addresses and will provide 1, 2, 4 or 8 texels to the next unit together with the appropriate number of interpolation coefficients.

[1]This is the minimum width guaranteed but some local block to block connections may be wider to accommodate more data.

[2]The messages are being described in general terms so as not to be bogged down in detail at this stage so what the 'new fragment' message actually specifies (i.e. coordinate, colour information) is left till later.

In general, the term *pixel* will be used to describe the picture element on the screen or in memory. The term *fragment* is used to describe the part of a polygon or other primitive which projects onto a pixel. Note that a fragment may only cover a part of a pixel.

Each unit and the message passing are conceptually running asynchronous to all the others, but in practice is synchronous because of the common clock.

How does the host process send messages? The message data field is the 32 bit data written by the host and the message tag is the bottom 9 bits of the address (excluding the byte resolution address lines). Writing to a specific address causes the message type associated with that address to be inserted into the message queue.

The message throughput is 50M messages per second and this gives a fragment throughput of up to 50M per second, depending on what is being rendered. For Gouraud shaded, Depth buffered pixels the fragment throughput drops to 12.5M per second.

1.2. Linkage

The following block diagram shows how the units are connected together. Some general points are:

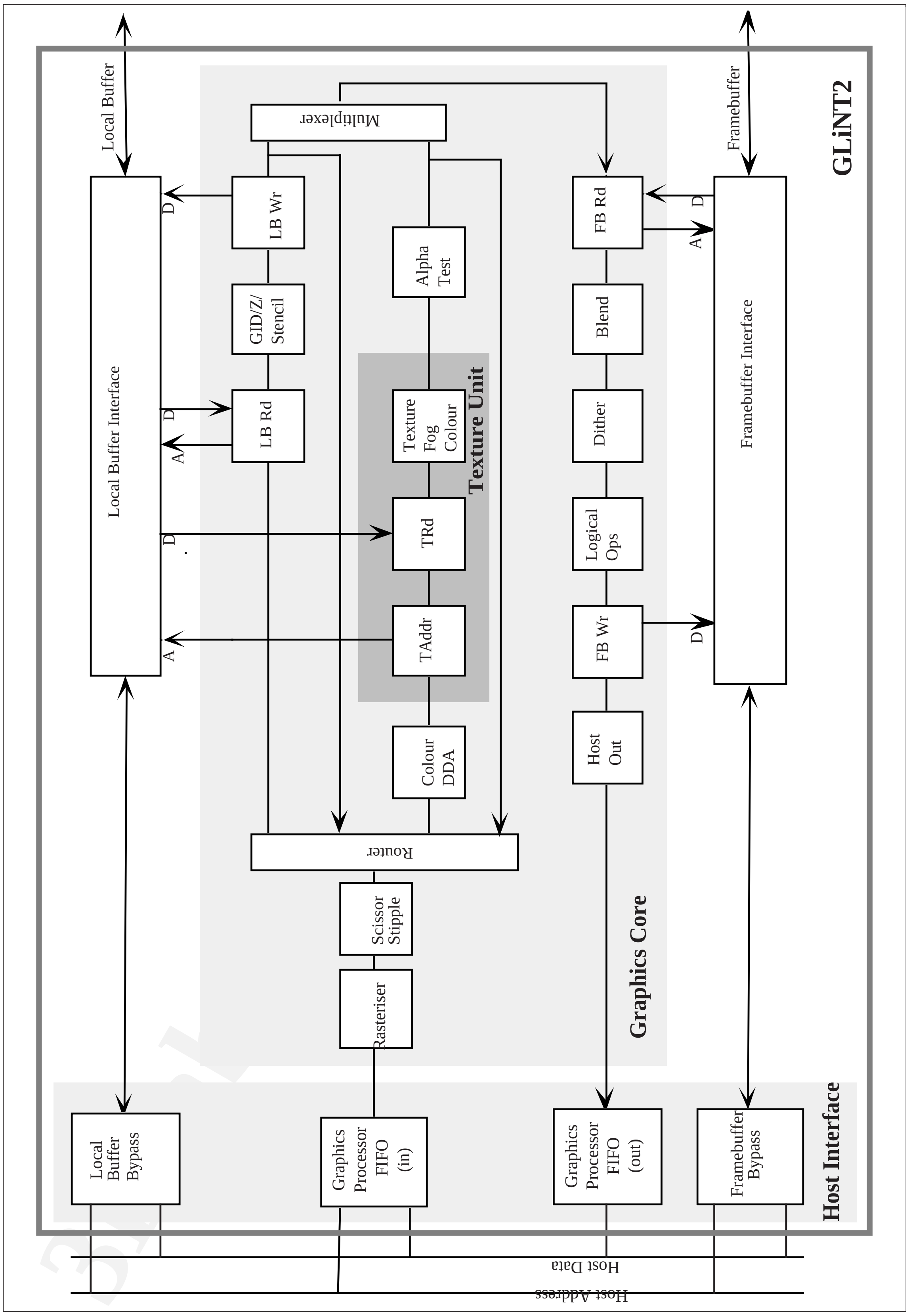
- This diagram is for GLiNT2. The following functionality is missing from GLiNT1:

  The Texture Address (TAddr) and Texture Read (TRd) Units are missing.

  The router and multiplexer are missing so the unit ordering is Scissor/Stipple, Colour DDA, Texture Fog Colour, Alpha Test, LB Rd, etc..

- The order of the units can be configured in two ways. The most general order (Router, Colour DDA, Texture Unit, Alpha Test, LB Rd, GID/Z/Stencil, LB Wr, Multiplexer) and will work in all modes of OpenGL. However, when the alpha test is disabled it is much better to do the Graphics ID, depth and stencil tests before the texture operations rather than after. This is because the texture operations have a high processing cost and this should not be spent on fragments which are later rejected because of window, depth or stencil tests.

- The loop back to the host at the bottom is to provide a simple synchronisation mechanism. The host can insert a Sync command and when all the preceding rendering has finished the sync command will reach the bottom host interface which will notify the host the sync event has occurred.



1.3. Benefits

The benefits this architecture give are all due to the very modular nature of it. Each unit lives in isolation from all the others and has a very well defined set of input and output messages.

This allows the internal structure of a unit (or group of units) to be changed to make algorithmic/speed/gate count trade-offs.

The isolation and well defined logical and behavioural interface to each unit allows much better testing and verification of the correctness of a unit.

The message passing paradigm is easy to simulate with software and the hardware design is nicely partitioned. The architecture is self synchronising for mode or primitive changes.

The host can mimic any block in the chain by inserting messages which that block would normally generate. These message would pass through the earlier blocks to the mimicked block unchanged and from then onwards to the rest of the blocks which cannot tell the message did not originate from the expected block. This allows for an easy work around mechanism to correct any flaws in the chip. It also allows other rasterisation paradigms to be implemented outside of the chip, but still use the chip for the low level pixel operations.

1.4. A day in the life of a triangle

Before we get too detailed in what each unit does (in the remainder of this document) it is worth while looking in general terms how a primitive (e.g. triangle) passes through the pipeline, what messages are generated and what happens in each unit. Some simplifications have been made in the description to avoid detail which would otherwise complicate what is really a very simple process. The primitive we are going to look at is the beloved Gouraud shaded Z buffered triangle, with dithering. It is assumed any other state (i.e. depth compare mode) has been set up but rather than emunerate it all here I will introduce it where necessary.

- The application generates the triangle vertex information and makes the necessary OpenGL calls to draw it.

- The OpenGL server/library gets the vertex information, transforms, clips and lights it. It calculates the initial values and derivatives for the values to interpolate (X_{left}, X_{right}, red, green, blue and depth) for unit change in dx and $dxdy_{left}$. All these values are in fixed point integer and have unique message tags. Some of the values (the depth derivatives) have more than 32 bits to cope with the dynamic range and resolution so are sent in two halves Finally, once the derivatives, start and end values have been sent to GLiNT the 'render triangle' message is sent.

- On GLiNT: The derivative, start and end parameter messages are received and filter down the message stream to the appropriate blocks. The depth parameters and derivatives to the Depth Unit; the RGB parameters and derivative to the Colour DDA Unit; the edge values and derivatives to the Rasteriser Unit.

- The 'render triangle' message is received by the rasteriser unit and all subsequent messages (from the host) are blocked until the triangle has been rasterised (but not necessarily written to the frame store). A 'prepare to render' message is passed on so any other blocks can prepare themselves.

- The Rasteriser Unit walks the left and right edges of the triangle and fills in the spans between. As the walk progresses messages are send to indicate the direction of the *next* step: StepX or StepYLeftEdge. The data field holds the *current* (x, y) coordinate. One message is sent per pixel within the triangle boundary. The step messages are duplicated into two groups: an active group and a passive group. The messages always start off in the active group but may be changed to the passive group if this pixel fails one of the tests (e.g. depth) on its path down the message stream. The two groups are distinguished by a single bit in the message tag. The step messages (in either form) are always passed throughout the length of the message stream and are used by all the DDA units to keep their interpolation values in step. The step message is effectively the fragment and any other messages pertaining to this fragment will always precede the step message in the message stream.

- The Scissor and Stipple Unit. This unit does 4 tests on the fragment (as embodied by the active step message). The screen scissor test takes the coordinates associated with the step message, converts them to be screen relative (if necessary) and compares them against the screen boundaries. The other three tests (user scissor, line stipple and area stipple) are disabled for this example. If the enabled tests pass then the active step is forwarded onto the next unit, otherwise it is changed into a passive step and then forwarded.

- The Colour DDA unit responds to an active step message by generating a Colour message and sending this onto the next unit. The active step message is then forwarded to the next unit. The Colour message holds, in the data field, the *current* RGBA value from the DDA. If the step message is passive then no Colour message is generated. After the Colour message is sent (or would have been sent) the step message is acted on to increment the DDA in the correct direction, ready for the next pixel.

- Texturing, Fog and Alpha Tests Units are disabled so the messages just pass through these blocks.

- In general terms the Local Buffer Read Unit reads the Graphic ID, Stencil and Depth information from the Local Buffer and passes it onto the next unit. More specifically it does:

  1. If the step message is passive then no further action occurs.

  2. On an active step message it calculates the linear address in the local buffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.

  3. The addresses are passed to the Local Buffer Interface Unit and the identified local buffer location read. The write address is held for use later.

  4. Sometime later the local buffer data is returned and is formatted into a consistent internal format and inserted into a 'Local Buffer Data' message and passed on to the next unit.

  The message data field is made wider to accommodate the maximum Local Buffer width of 52 bits (32 depth, 8 stencil, 4 graphic ID, 8 frame count) and this extra width just extends to the Local Buffer Write block.

  The actual data read from the local buffer can be in several formats to allow narrower width memories to be used in cost sensitive systems. The narrower data is formatted into a consistent internal format in this block.

  After this block the message stream looks like this for each pixel:

first	Colour
	LB Data
last	Active Step message

- The Graphic ID, Stencil and Depth Unit just passes the Colour message through and stores the LBData message until the step message arrives. A passive step message would just pass straight through.

When the active step message is received the internal Graphic ID, stencil and depth values are compared with the ones in the LBData message as specified by this unit's mode information. If the enabled tests pass then the new local buffer data is sent in the LBWriteData message to the next unit and the active step message forwarded. If any of the enabled tests fail then an LBCancelWrite message is sent followed by the equivalent passive step message. The depth DDA is stepped to update the local depth value.

- The Local Buffer Write Unit performs any writes which are necessary.

  The LBWriteData message has its data formatted into the external local buffer format and this is posted to the Local Buffer Interface Unit to be written into the memory (the write address is already waiting in the Local Buffer Interface Unit).

  The LBWriteCancel message just informs the Local Buffer Interface Unit that the pending write address is no longer needed and can be discarded.

  The step message is just passed through.

- In general terms the Framebuffer Read Unit reads the colour information from the framebuffer and passes it onto the next unit. More specifically it does:

  1. If the step message is passive then no further action occurs.
  2. On an active step message it calculates the linear address in the framebuffer of the required data. This is done using the (X, Y) position recorded in the step message and locally stored information on the 'screen width' and window base address. Separate read and write addresses are calculated.
  3. The addresses are passed to the Framebuffer Interface Unit and the identified framebuffer location read. The write address is held for use later.
  4. Sometime later the colour data is returned and inserted into a 'Frame Buffer Data' message and passed on to the next unit.

  The actual data read from the framestore can be in several formats to allow narrower width memories to be used in cost sensitive systems. The formatting of the data is deferred until the Alpha Blend Unit as it is the only unit which needs to match it up with the internal formats.

  In this example no alpha blending or logical ops are taking place so reads are disabled and hence no read address is sent to the Framebuffer Interface Unit. The Colour and step messages just pass through.

- The Alpha Blend Unit is disabled so just passes the messages through.
- The Dither Unit stores the Colour message internally until an active step is received. On receiving this it uses the least significant bits of the (X, Y) coordinate information to dither the contents of the Colour message. Part of the dithering process is to convert from the internal colour format into the format of the framebuffer. The new colour is inserted into the Colour message and passed on, followed by the step message.
- The Logical Ops are disabled so the Colour message is just converted into the FBWriteData message (just the tag changes) and forwarded on to the next unit. The step message just passes through.
- The Framebuffer Write Unit performs any writes which are necessary.

The FBWriteData message has its data posted to the Framebuffer Interface Unit to be written into the memory (the write address is already waiting in the Framebuffer Interface Unit).

The step message is just passed through.

- The Host Out Unit is mainly concerned with synchronisation with the host so for this example will just consume any messages which reach this point in the message stream.

This description has concentrated on what happens as one fragment flows down the message stream. It is important to remember that at any instant in time there are many fragments flowing down the message stream and the further down they reach the more processing has occurred.

2. GLiNT1 Structure

This chapter details how the units are connected together and some general services and functions which pertain to all the units in the core.

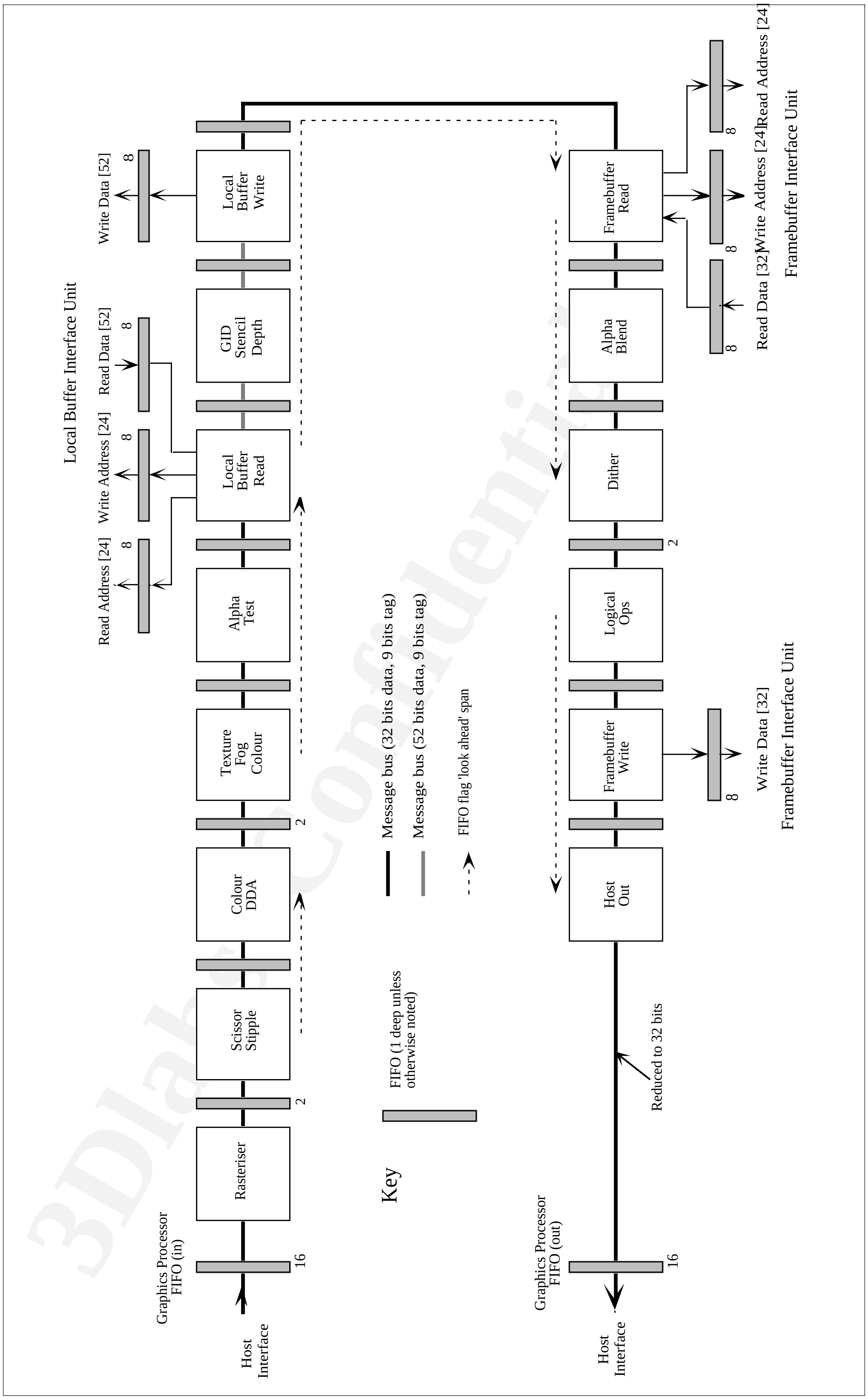
The description is specific to GLiNT1, a subset of GLiNT2, which was featured in the block diagram in the Overview section.

The following block diagram shows how the core is configured in GLiNT1, together with all the FIFOs between the blocks and between the core and the I/O (Host PCI Interface, Local Buffer Interface and Framebuffer Interface). This diagram omits the readback paths (see a later diagram), low level FIFO handshaking signals, reset and the clock for clarity.

All the core units and FIFOs are synchronous and designed to run at a maximum frequency of 50M Hz. There is no dynamic logic so the lower frequency is DC.

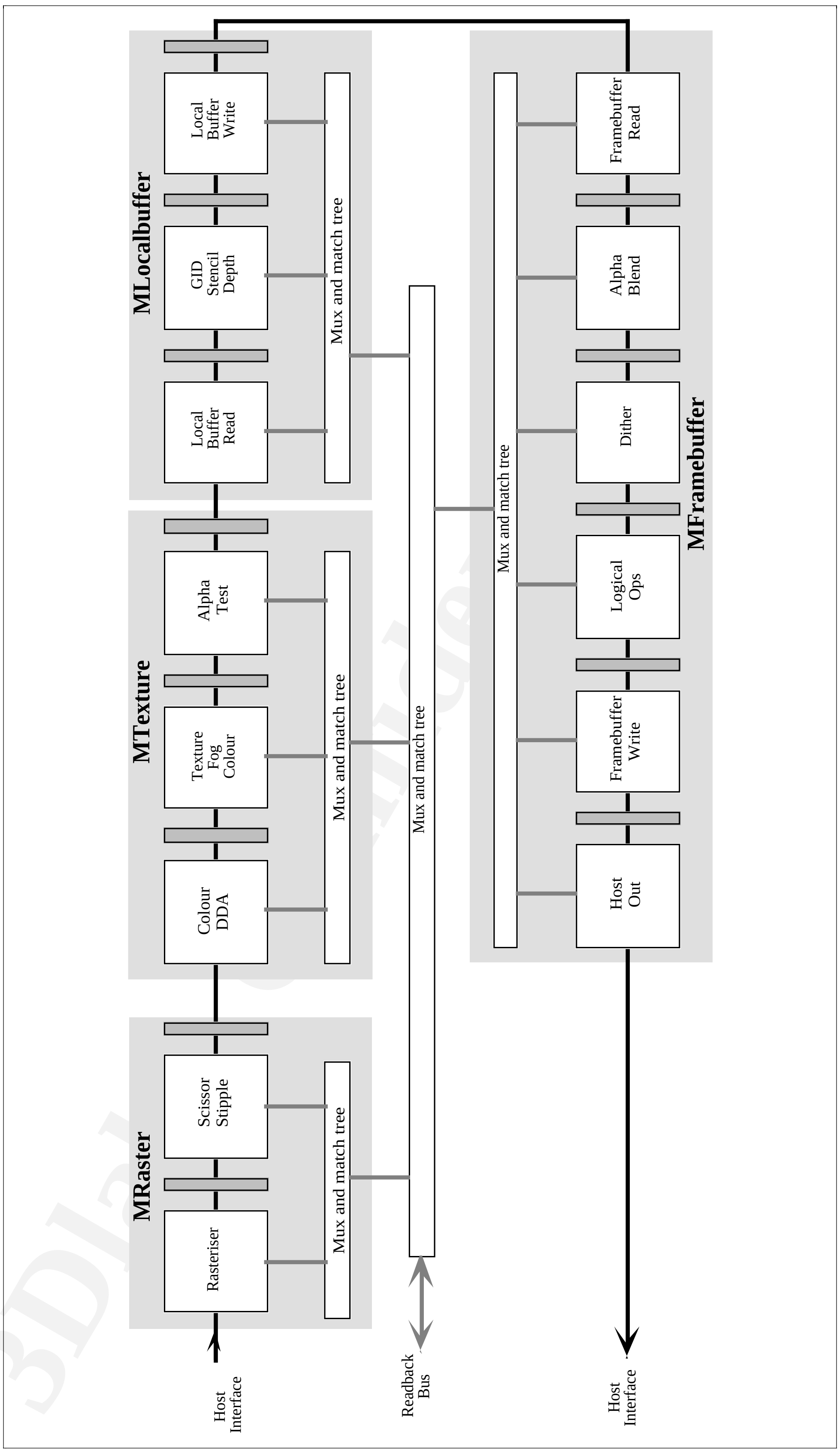
The Graphics Processor FIFOs (in and out) are considered part of the Host Interface and separate the PCI clock domain from the core clock domain so are responsible for resynchronisation of the message stream.

The Read Address, Write Address and Write Data FIFOs are considered part of the Local Buffer and Framebuffer Interface Units, i.e. they belong to the receiver.



Host Interface
Graphics Processor FIFO (in)
16
Rasteriser
2
Scissor Stipple
Colour DDA
2
Texture Fog Colour
Alpha Test
Local Buffer Read
GID Stencil Depth
Local Buffer Write
Read Address [24]
8
Write Address [24]
8
Read Data [52]
8
Write Data [52]
8
Local Buffer Interface Unit
Framebuffer Read
Alpha Blend
Dither
2
Logical Ops
Framebuffer Write
Host Out
Reduced to 32 bits
Graphics Processor FIFO (out)
16
Host Interface
Read Address [24]
8
Write Address [24]
8
Read Data [32]
8
Framebuffer Interface Unit
Write Data [32]
8
Framebuffer Interface Unit
Key
FIFO (1 deep unless otherwise noted)
Message bus (32 bits data, 9 bits tag)
Message bus (52 bits data, 9 bits tag)
FIFO flag 'look ahead' span

The following block diagram shows the read back path hierarchy. The units have been grouped like this for convenience in testing of the external memory interfaces and to allow the easy addition of the routing capability of GLiNT2. The name of each group is shown, the M (for Master) prefix denotes the clock domain in the VHDL hierarchy. The readback mechanisms are described later.



2.1. Unit attributes and facilities

2.1.1. Reset and initialisation

The units and FIFOs can be reset under software control or by a hardware reset signal, usually as a result of power-on.

During reset all the inter unit FIFOs, the FIFOs between the core and the local buffer, framebuffer and host interface are all made empty. Some of the units (Local Buffer Read and Framebuffer Read) also have internal FIFOs and these are made empty as well.

All the state machines in each unit are forced into their idle state so this together with the FIFOs being empty guarantees a safe start when the first message is received.

A reset does *not* change the contents of any state information which can be readback. After a power-on reset *all* these registers must be initialised by software to place them in a well defined state *before* any rendering is done. Units are *not* automatically disabled on a reset.

2.1.2. Readback

The readback path provides a way of reading back all the user visible registers in the core. This facility if provided for diagnostics/debugging and to allow the state to be read out for use in state save and restore when task switching (the restore is done via the message stream). The registers and the associated messages which can be read back are identified in the Message Tags Appendix.

The read back path is asynchronous for all the registers and does not rely on the message stream.

Many of the registers are not the full 32 bits in width and any unused bits are always read back as zero.

To read back a register the host software just reads the same address it would have written to, to send this message. Recall that the lower address bits are just the message tag. Care does need to be taken when using readback to ensure the register is up to date. Writing to a register via a message and then immediately reading back the same register is unlikely to return the data you thought you had just previous written. The time for the write message to reach the unit where it will update the register depends on how many messages are queued in the Graphics Processor Input FIFO and what the rasteriser and other units are doing. The only reliable way to ensure a register contain the most recent value before trying to read it back is to send a Sync message and wait for it to appear on the Graphics Processor Host Out FIFO.

The 'Readback Bus' consists of the following signals:

RbTag	This identifies the register to read back and is presented in parallel to all the units through the 'mux and match tree'. When no read back is being requested the tag is set to zero.
RbMatch	The unit which recognises the tag as its own (any action tags such as ActiveStep are not recognised by any unit for the purpose of readback) so holds the associated data will assert this signal.
RbOut	The unit which responds to the tag will assert the data onto this 'data' bus.

The intermediate 'mux and match trees' will use the match signals from each unit to route the data to the Host Interface. If no match is found then RbOut is undefined and the RbMatch signal will not have been asserted. The Host Interface records this fact in a register and presents data set to zero back to the software.

The Readback bus interface is the same at every level of the hierarchy from the Host interface to the individual units.

The Host Interface must wait 30ns after presenting a new RbTag before it samples the state of the RbMatch and RbOut signals.

2.1.3. Undefined states

Many of the messages contain mode type information where a multi-bit field does not have all its combinations enumerated. For these undefined values the action is undefined, however in order to keep the C simulation and VHDL consistent the undefined action always defaults to that associated with the value zero.

2.1.4. General disable philosophy

Most units can be disabled. The general philosophy when a unit is disabled is that it still reacts to messages which update the internal state. This applies not only to messages which normally originate from the host (such as delta values), but also to internally generated messages (e.g. Colour).

Messages which cause some action (mainly the active step messages) are ignored when a unit is disabled.

2.1.5. Flow Through

In the PCI register space there is a flow through register. This register has a bit allocated per core unit, and when this bit is set *all* messages will just flow through and not update any internal state, or cause any actions to occur in that unit. FIFO blocking is still done as the normal transport mechanism is still used. When this bit is reset the unit behaves as described in its behavioural model.

2.1.6. Performance Metrics

The absolute minimum throughput of fragments the core units must maintain, when not limited by host or I/O considerations, is 12.5M fragments per second, except when texture mapping or high quality antialiasing is enabled.

The maximum number of message any fragment can take (again excluding texture mapping and some unusual situations) is 3, as measured through one block:

For graphics: ActiveStep, Colour, LBData or FBData.
For copies: ActiveStep, LB/FBData and LB/FBSourceData)

The performance a unit delivers should be limited by the number of messages needed to calculate/modify the output result and not the calculation time. The number of messages will vary from 1 to 3.

Pipelinning in a unit is kept to a minimum to conserve gates, minimise the latency and reduce complexity, within the bounds of meeting the above performance goals. Keeping the latency low is especially important in the Local Buffer group of units, but less important in the Framebuffer group of units where the normal operation is to write only.

Performance of the Local Buffer group and its interaction with the memory is paramount to the 3D graphics performance of the chip, but the FIFOs and page breaks of the memory make this one of the most difficult areas of GLiNT to be sure the performance is adequate. Simulation of the performance in this area will be necessary the verify the design goals have been met.

Where ever possible the memory systems (especially the Local Buffer) should be the limiting factor in the core's performace to make best use of this important resource.

2.2. Inter-unit FIFOs

The inter-unit FIFOs in the core all have a depth of one, except where noted otherwise. The preceding unit monitors the empty flag and the next unit monitors the full flag. The FIFO is made empty on reset.

The flags are part synchronous and part asynchronous to ensure that messages will propagate continuously with no bubbles in the stream (as some of the FIFOs are only a single entry deep). The FIFO flag asynchronous look ahead paths are shown as dotted arrows in the earlier GLINT1 structure diagram. The asynchronous path lengths are limited to 2 or 3 block and are isolated from each other by deeper FIFOs as shown, or by the M FIFOs in the Local Buffer Read Unit and Framebuffer Read Unit.

3. Rasteriser Unit

3.1. Description

The rasteriser unit scan converts the given primitive into a list of pixel coordinates which meet the rasterisation rules of OpenGL, X and NT[3]. In addition to generating the coordinates, the order the pixels are visited in is also defined (by the walk message types) so the local DDA units in the Texture, Colour, Fog and Depth units can incrementally keep in step.

When a primitive is antialiased the percentage coverage of the primitive within the scan converted pixels is calculated, for later use in the alpha blend unit. The same method of antialiasing is used for all primitives. The basic idea is to scan convert the primitive to a higher resolution (e.g. 4x4 sub samples per pixel) and count the number of sub pixel sample points covered. The ratio of covered sample points to total number of sample points gives the coverage weighting to adjust the colour by. The rasterisation process steps through in Y and calculates the two intersection point for this scanline. For normal rasterisation the pixels between these two intersection points are filled in. During antialiasing a step of Y/4 (for example) is used and within each scan line four pairs of intersections are calculated per scanline. The coverage for each of the four sub pixel scanline makes in a pixel (on this scanline) are calculated and summed.

The coordinates passed to the rasteriser can be window relative or screen relative. The rasteriser treats both the same and it is only in the Local Buffer and Framebuffer Read Units where they get converted to memory addresses.

The rasteriser is not concerned whether the origin is the bottom left or top left and again it is the Local Buffer and Framebuffer Read Units which take this into account when calculating the memory address. Obviously if the direction of scan conversion is important then the parameters must match up with the origin definition to give the desired effect.

Long term mode information is held in the RasteriserMode message and short term mode information (which only applies to this primitive being rasterised) is passed with the Render message.

3.1.1. Points

Points represent the easiest of all primitives to scan convert, however there are a number of cases which need to be handled differently. The main cases are whether the point is antialiased or not, and its size.

All the DDA related parameters are held constant over a point (a point may cover many pixels), and between points in a Begin/End set. Before any point rasterisation is done the host must have set up the Texture, Colour, Fog and Depth units so they maintain a constant value and don't increment between pixels in a point.

In OpenGL no stipple operations are defined for points so stippling must be disabled. This can be done by changing the stipple mode (see Stipple Unit) or by setting the stipple operation in the Render (or PrepareToRender) message to 'none'. This later method is much easier for the software to use.

[3]X accelerator chips such as the Weitek 'Power 9000' are also good NT accelerators so it looks as if by meeting the X rules we will also meet the NT rules.

3.1.1.1. Aliased Points (OpenGL)

The size of an aliased (or non-antialiased!) point is rounded to the nearest integer and clamped to some implementation dependent value.

When the integer size is unity only one pixel is rendered. The host just sends any of the Active walk messages with the (X, Y) position encoded in the data field for each point to render. Note the comments about stippling above.

When the size is non-unity the point is rendered as a square with sides of length size. The software will, in this case, use the polygon rendering methods to draw the points.

3.1.1.2. Aliased Points (X)

X only has single pixel sized points so these are rendered by just sends any of the Active walk messages with the (X, Y) position encoded in the data field for each point to render.

3.1.1.3. Antialiased (OpenGL)

Antialiased points are scan converted as circles, with the coverage of the boundary pixels ranging from 0% to 100%. The only size which *must* be supported is 1.0, however the sample server supports point sizes from 0.5 to 10.0 in steps of 0.125 and this may well set the expectation level of ISVs.

The hardware will support point diameters in the range 2 sub scanlines to 64 sub scanlines. This translates to pixel sizes 0.5 to 16.0 in steps of 0.25 for 4x4 antialiasing.

To antialiase a point we walk around the boundary in sub scanline steps. Given the sub scanline intersections on the boundary of the circle the same techniques are used to calculate the coverage value as for polygons. The sub scanline intersections are calculated incrementally using a small table. The table holds the change in X for a step in Y and this is added to the current X position every time we change sub scanlines. Symmetry is used so the table only needs to hold the delta values for one quadrant. The table is initialised by the OpenGL server as a function of the point size.

3.1.1.4. Antialiased (X)

There are no antialiased points in X.

3.1.2. Lines

There are two accepted way of drawing lines: using a DDA and Bresenham's algorithm. The only advantage Bresenham's algorithm has over the DDA one is no divide is necessary as part of the set-up. In the 3D case the cost of the divide is inconsequential given all the other transformation and perspective calculations needed, however in the 2D case this may not be true[4].

For OpenGL we will always use the DDA method because the cost of the divide is acceptable and is needed to calculate the gradient of any colour or depth change. Lines are specified by their end points (accurate to 4 bits of sub pixel position) and rate of change in X and Y per step along the major axis of the line.

[4]This depends on how fast the host machine is in doing the divide. Also doing a table lookup of the reciprocal and then a multiply may be faster than a divide.

3.1.2.1. Stipple Lines (OpenGL)

Line stippling (for aliased lines) is handled in the Stipple unit. All the rasteriser has to do is produce all the pixels on the line and the Stipple unit will effectively delete any unwanted portions of the line. Control of the stippling is described in the Stipple Unit section.

3.1.2.2. Stipple Lines (X)

The standard OpenGL method of stippling lines can be used in X for the more restricted case where the mark/space ratio of the stipple is the same. X allows an arbitrary stipple pattern to be defined and the way this is supported is using the Bitmap facility. Here the host provides a number of 32 bit mask words where each bit corresponds to one pixel in the line. The state of this bit determines whether the associated pixel is generated or skipped.

3.1.2.3. Aliased Lines (OpenGL)

Single pixel wide aliased lines are drawn using a DDA algorithm so all it needs by way of input data is StartX, StartY, dX, dY and length. The algorithm just calculates:

```
while (length--)
{
        X = X + dx
        Y = Y + dy
        plot ((int)X,  (int)Y)
}
```

The variables X, Y, dx and dy are all fixed point numbers (probably 16.16 format is the easiest). The conversion to memory address using the X, Y coordinate is done in the memory read units.

3.1.2.4. Aliased Lines (X)

Single pixel wide aliased lines are drawn using a DDA algorithm so all it needs by way of input data is StartX, StartY, dX, dY and length.

3.1.2.5. Aliased Wide Lines (OpenGL)

There is no direct support for wide lines. The OpenGL server has two options:

1. Wide lines can be drawn by repeating a single pixel wide line, but offset by one pixel in X for X major lines or one pixel in Y for Y major lines. Any values interpolated along the line (e.g. colour) will need to be re-initialised at the start of each individual line. This is easily done by the Render message.

2. Wide lines can be converted to parallelograms (the ends of a wide line are parallel to the edge of the screen in OpenGL) and then rendered as polygons.

As you might expect neither method is the best in all cases. For vertical or near vertical lines method 2 will cause less page breaks in memory so should be faster, however if there is any stippling then method 1 is likely to be much faster. Method 1 is the simpler method and is the preferred implementation.

3.1.2.6. Aliased Wide Lines (X)

Individual wide lines in X have square ends and multiple connected wide lines have a range of joint styles. X will need to either convert the wide lines to polygons, or a series of spans to achieve the desired effect.

3.1.2.7. Antialiased Lines (OpenGL)

Antialiased lines, of any width, are drawn as antialiased polygons. If stipple is enabled then the line is drawn as a series of polygons to match up with the stipple parameters.

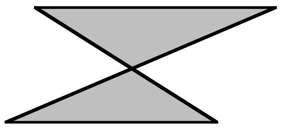
3.1.2.8. Antialiased Lines (X)

There are no requirements in X for antialiased lines.

3.1.3. Polygons

The only polygons the rasteriser handles are screen aligned trapezoids. These are characterised by having the top and bottom edges parallel to the X axis. The side edges may be vertical, but in general will be diagonal. The top or bottom edges can degenerate into points in which case we are left with flat topped or flat bottomed triangles. Any polygon can be decomposed into this shape, however the sample OpenGL server always decomposes polygons[5] into triangles because the interpolation of values over non-triangular polygons is ill defined.

The rasteriser does handle vertical 'bow tie' polygons.



X has a much more sophisticated definition of what a polygon is compared to OpenGL. It can be concave and self intersecting. In the non convex case the best thing is for X to do is to decompose the polygon into a series of spans and render them as 1 pixel high rectangles. For any convex polygons X can decompose them into screen aligned trapezoids as a further optimisation over just using spans.

The rasterisation rules of OpenGL and X both state than butting polygons must only touch pixels along the common boundary once (when not antialiasing). The simplest way to ensure this is never to plot the pixels along the top and right edges of a polygon. The GLiNT rasteriser will not generate fragments along the right hand edge, but the host must set up the start and end scanlines as required to miss either the top or bottom edges.

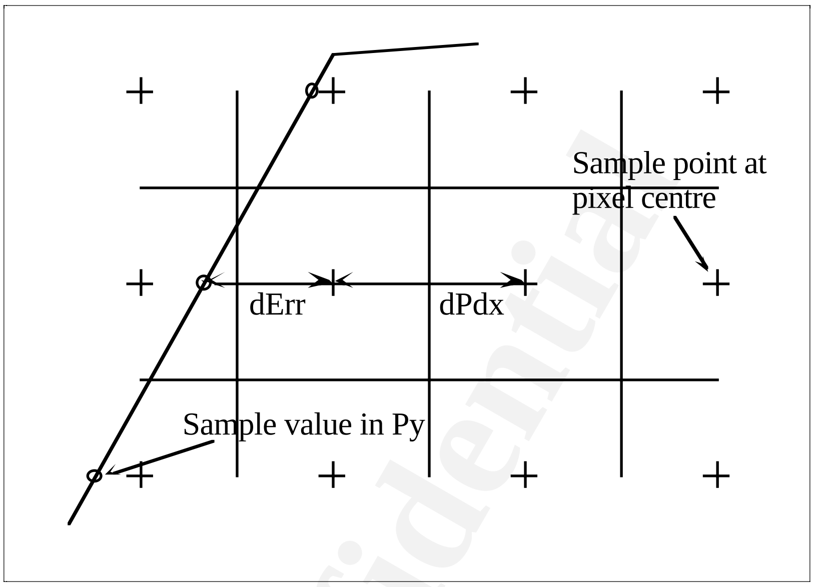
As part of the rasterisation process a number of parameters (colour, depth, fog and texture) are calculated for each fragment generated. These are calculated in DDA unit down stream under the guidance of the rasteriser step messages. The ideal way to calculate these values is to use the fragments XY coordinate and substitute this into the plane equation for each parameter in turn. This technique gives the best result, however it is computationally expensive so it is normal to use an incremental method such as a DDA to approximate to it. The DDA method introduces some errors of its own:

- An incremental error due to the finite precision of the delta values. To overcome this source of errors enough fractional bits are used so that the error cannot propagate into the actual bit range of the DDA where the parameter value is extracted from.
- The start value for a parameter, P, can be nearly dPdx (one step in the X direction) out because the value calculated as a result of a Y step (shown as a circle in the following diagram) corresponds to the value of the sample on the edge and not at the centre of the

[5] Excluding the special case of screen aligned rectangles.

first fragment to be drawn. It is necessary to correct for this error to eliminate bright edge artefacts and achieve high quality rendering.

This correction is needed for every scanline. A similar correction is needed at the start of the primitive because the parameter value at the start vertex is unlikely to lie on the horizontal centre of a pixel so need adjustment in Y. This correction is handled by software.



If dErr is the distance the edge is away from the pixel's centre (must be < 1) and dPdx is the change in P for unit change in x then the correct value at the first sample point is:

$$Px = Py + dErr * dPdx$$

The distance dErr is sent by the rasteriser in the SubPixelCorrection message and whenever this message is received by a DDA unit the above equation is implemented (in an incremental form to avoid the multiply operation). The correction dErr is sent in sign magnitude format with 4 bits of magnitude. and a sign.

Sub pixel correction must be enabled by the SubPixelCorrectionEnable bit in the Render message if it is required.

Antialiasing presents a much more complex problem to solve in that the sample point for the parameters must be inside the boundary of the fragment, but this may not be the centre of the pixel anymore. Near horizontal edges can give rise to a dErr value which approaches the width of the screen (or window). Two methods can be used to overcome this:

- The sample point can be moved to be within the boundary by 'micro nudging' the DDAs in X and Y.
- The parameter being interpolated can be integrated over the interior sub pixel sample points and then divided by the number of interior points (this is the method in the OpenGL spec).

In both these cases the changes to the DDA units are too extensive given the other problems this method of antialiasing has (the coverage calculation doesn't take into account sub pixel visibility and doesn't work well with a depth buffer). No sub pixel corrections are done for antialised primitives.

In OpenGL, when antialiasing is on, any pixels on a shared edge are touched twice, but the sum of the coverage values must not exceed 100%.

A triangle has three edges: a *dominant* edge and two *subordinate* edges. The dominant edge is the edge which covers the maximum Y extent. We always scan convert from the dominant edge.

The triangle is further decomposed (if it has a knee) into two triangles with flat tops or bottoms to facilitate the edge walking stage.

What parameters should the host give to define a triangle? There are several choices:

- The three vertices and the depth, texture, colour and fog parameter. This is the nicest interface but requires the chip to calculate the derivatives for each parameter per edge which requires divides and multiplies in (ideally) floating point. This is not feasible because of the gate count it would take.

- The three vertices and edges and all the derivatives in one 'go'. In this case the chip would sequence the third edge when it was time to render the second triangle.

- The three vertices and edges and all the derivatives given in two stages.

This last method is the preferred one because it is more general (it extends naturally to more than just triangles) and only presents a small overhead over the 'all in one go' case. This paradigm also fits in with the idea of having a dedicated Geometry Engine (which would do the slope and derivative calculations amongst other things) feeding messages to GLiNT, rather than a general purpose processor such as the i860.

The sequence of actions the host needs to do to render a triangle (with a 'knee') are:

- Load up the edge parameters and derivatives for the dominant edge and the first subordinate edges in the first triangle.

- The Render message starts the scan conversion of the first triangle, working from the dominant edge. This means that for triangles where the knee is on the left we are scanning right to left, and for triangles where the knee is on the right we are scanning more conventionally left to right.

- Load up the edge parameters and derivatives for the remaining subordinate edge in the second triangle.

- The ContinueNewSub message starts the scan conversion of the second triangle.

The order the fragments are produced in is dependent on the parameters. If the origin is in the bottom left corner and the Y derivative is both positive then the scan conversion proceeds from the bottom towards the top. If the derivative is negative then scan conversion proceeds in the opposite direction. The X direction is always from the dominant edge. For a rectangle both vertical edges are equally dominant so you can choose which edge to render from so effecting the scanning direction.

3.1.3.1. Stipple (OpenGL)

No stipple processing is needed in the Rasteriser as all of OpenGL's needs are accommodated in the Stipple Unit.

3.1.3.2. Stipple (X)

The stippling requirements for X which cannot be met by the Stipple Unit include:

- Arbitrary stipple on lines.
- Arbitrary stipple on polygons, especially rectangles.

The bit mask unit in the rasteriser (normally used for characters) can be used to give any arbitrary stipple to any primitive. The stipple pattern required is loaded into the bitmask register 32 bits at a time, in the order the pixels in the primitive are generated in. The state of a bit in the bitmask determines if an active pixel is generated or a passive one. One bit in the stipple sequence is required for each pixel in the primitive.

This stippling method is independent of the Stipple Unit and can replace its function or be used as a second level of stippling.

3.1.3.3. Aliased Polygons (OpenGL)

The OpenGL rasterisation rule are followed.

3.1.3.4. Aliased Polygons (X)

The X rasterisation rules are followed.

3.1.3.5. Antialiased Polygons (OpenGL)

Antialiasing of polygons (or more precisely, screen aligned trapezoids) are scan converted by walking the trapezoid's edges to a higher resolution (x4, say). The coverage for a specific pixel is calculated by summing the coverage each of the sub scanlines contributes. More specific details are given in the implementation section. Care needs to be taken when the trapezoids (from the same polygon) met part way through a scan line. The span of pixels cannot be generated until the second trapezoid is available as it will contribute to the coverage in this scanline. If, on the last trapezoid, the scan line is only part covered then a 'flush' message is needed to cause the coverage for these pixels to be generated as there is no follow on trapezoid.

3.1.3.6. Antialiased Polygons (X)

There are no requirements in X for antialiased polygons.

3.1.4. Image Download, Upload and Copy

The OpenGL Image functions come under the name of 'Pixel Rectangle rasterisation functions' and allow the user to specify rectangular regions using pixel coordinates, rather than the normal 3D coordinates. As far as the chip is concerned this is no different to line or polygon rasterisation because these are specified in pixel coordinates by the time they reach the chip, however there are some additional considerations.

This type of rendering in OpenGL cannot be stippled or antialiased. In X it can be stippled but this falls naturally out of the design.

The Pixel Rectangles rasterisation is used to support the following OpenGL functions:

- Image download (DrawPixels). This function provides the host with a method of placing data into window relative rectangular regions in any of the buffers (depth, stencil, colour). The host could access the memory directly, however the host doesn't know (or usually care) where the window is and it may move during the update. Also window clipping and fragment processing still need to be done.

  The rasteriser supports this function by scan converting the rectangle (so the host doesn't need to generate X, Y coordinates). The rasteriser doesn't free run and generate fragments as normal but waits for a Depth, Stencil or Colour message from the host before moving on to the next pixel. In other words is runs synchronous to the host for the duration of this primitive. The SyncOnHostData bit in the Render message enables this behaviour.

The bit mask mode can also be enabled during this function so arbitrary stippling can be done on the image being downloaded (useful in X). The bit mask register is loaded whenever the BitMask message is received. This is slightly different[6] to the way it works when the rasteriser is not in Image download mode. The BitMaskPattern message must be interleaved correctly with the image data to ensure the new mask is available immediately after the last bit in the current mask has been used. It this sequence is not correct then all subsequent fragments until the new mask is received will be generated as passive ones.

There is the potential for the host to send too few Colour (Depth, Stencil or FBData) messages for the size of primitive it has defined. Rather than have GLiNT hang because it is waiting for messages which will never arrive, any message other than Colour, Depth, Stencil, FBData or BitMask will cause this primitive to be terminated.

This functionality of running synchronous to the host has been described in association with this command, but it is available on all primitives.

Note that the API to the DrawPixel function in OpenGL provides a rich variety of ways to order and format the data. It is the hosts job to format the data to one suitable for GLiNT before passing it down.

- Image upload (ReadPixels). This function provides the host with a method of reading back a window relative rectangular region of any of the buffers (depth, stencil, colour). The host could access the memory directly, however the server doesn't know (or usually care) where the window is and it may move during the read.

  The rasteriser supports this function by scan converting the rectangle and sending the active walk messages. The Local Buffer Read Unit or the Framebuffer Read Unit will have already been set up to do the read and generate the appropriate LBDepth, LBStencil or FBColour message, which will collected by the Host Interface Unit (Out) and passed back to the host.

  Note that the API to the ReadPixels function in OpenGL provides a rich variety of ways to order and format the data. It is the hosts job to format the data after it has read it from GLiNT.

- Image copy (CopyPixels). This function provides a method where window relative rectangular regions can be copied within a buffer (depth, stencil, or colour). The rasteriser just needs to scan convert the destination rectangle and the copy part of the operation is handled in the Local Buffer Read Unit and/or the Framestore Read Unit. There are two complications to be aware of: Firstly, if the source and destination rectangles overlap then the direction of the scan conversion is important and must be set up correctly by the host. Secondly, the API for this function allows for the data to be re-formatted and this must be done outside of the chip.

3.1.5. BitMaps

A Bitmap primitive, in OpenGL and X, is a rectangle of ones and zeros which control which fragments are generated. Only fragments where the corresponding Bitmap bit is set as

[6]This change is necessary to prevent a dead lock situation arrising if too many Colour messages (for example) are sent before the next BitMask message is due.

appropriate are submitted for drawing. The normal use for this is in drawing characters, although the mechanism for implementing this facility is available in all primitives. All the parameters (e.g. depth, colour and texture) are held constant for each fragment within the Bitmap, although the chip doesn't enforce this.

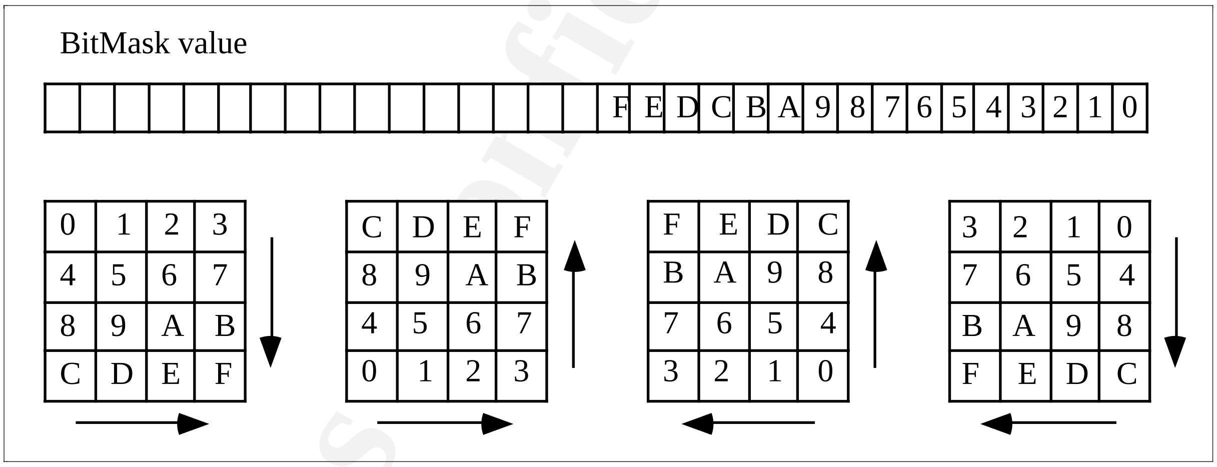
The bitmask is sent in the BitMaskPattern message and the pattern can be optionally invert before it is tested. The optional inversion is useful when two passes are needed to draw the primitive, for example to draw the foreground pixels using a different logical operation to the background pixels for a character.

As each pixel in the primitive is generated one bit of the bitmask is consumed. The bits can be consumed from the most significant end towards the least significant end, or vice versa depending on the state of the MirrorBitMask bit in the RasteriserMode message.

Each Bitmap has the following data associated with it:

- Origin X, Y coordinate (bottom left for OpenGL, top left for X)
- Width
- Height
- Bitmap data

The Bitmap data is packed into 32 bit words so that rows are packed adjacent to each other. The relationship between bits in the mask and the scanning order, assuming the MirrorBitMask bit is set to 0, is shown below:


BitMask value
F E D C B A 9 8 7 6 5 4 3 2 1 0
0 1 2 3 / 4 5 6 7 / 8 9 A B / C D E F
C D E F / 8 9 A B / 4 5 6 7 / 0 1 2 3
F E D C / B A 9 8 / 7 6 5 4 / 3 2 1 0
3 2 1 0 / 7 6 5 4 / B A 9 8 / F E D C

The rasteriser scan converts the given primitive and tests the least significant or most significant bit in the bitmask. If the bit is set (after optionally being inverted) them an active step message is issued, otherwise a passive one is. The bitmask is then rotated one place to the left or right depending on the MirrorBitMask bit.

When the bitmask word has been exhausted and pixels in the primitive still remain then rasterisation is suspended until the next Bitmap data word is received.

The aliased trapezoid rasterisation rule (for point sampling) state that pixels on the right hand edge are not plotted so these pixels are not present in the Bitmap, even though the size of the trapezoid given to the rasteriser includes them. For example, consider a 10x12 character. There are 120 pixels in the character so 120 bits worth of bitmap must be provided. The left edge of rectangle to scan convert starts at *x*, say, and the right hand edge is at *(x + 10)* . Remember these edges are inclusive so 11 pixels are present on a scanline, but because the point sampling rule misses out the right most pixel only 10 pixels are in fact plotted.

This dialogue has assumed that the host provides the Bitmap data every time a character is drawn. It is not possible, without a significant number of additional gates, to store bitmaps in the local buffer. In the case of bitmaps we are likely to be GLiNT performance bound rather than host bound and this wouldn't change if the bitmaps were local.

One other option which has been considered is to use the block fill modes (of the framestore) with the pixel write masks. This in theory could give very good performance, however isn't very useful in OpenGL because it doesn't support GID, depth or stencil operations. Also our character performance[7] (166K/second) is adequate.

3.1.6. Fast Block Fill

The framebuffer has a fast block fill mode where a number of pixels can be written simultaneously. Pixel blocks of 8, 16 and 32 pixels are supported. The data bits are used to select which pixels in a block to update and all pixels will be updated using the same data. Write masks within a pixel still work. Each block must start on a address which is exactly divisible by the block size. For example up a 16 pixel block is aligned on a 16 pixel boundary (i.e. bottom 4 address bits are 0).

This fast block fill method does have some restrictions: None of the per pixel clipping, stipple, or fragment operations are available with the exception of write masks. Even with these restrictions this is still very useful for X, but of little use for OpenGL (except maybe for full screen demos!). One subtle restriction from this is that the block coordinates will be interpreted as screen relative and **not** window relative when the pixel mask is calculated in the Framebuffer Units. Calculating the pixel mask from window relative coordinates is much more expensive as the coordinates would first need to be converted to screen relative ones first.

To support fast fills the rasteriser needs two things:

1. At the start of every line send the FastBlockLimits message to the Framebuffer Write unit. The data field holds the left and right limits the fast fill is to occur between.

2. During span filling, increment X by 8, 16 or 32 as appropriate and send the FastBlockFill message. The data field holds the X, Y coordinate address.

The Framebuffer Write Unit will generate the appropriate pixel mask from this data and instigate the necessary VRAM memory cycles to do the block write.

Any screen aligned trapezoid can be fast filled and not just rectangles.

[7] 10x12 characters with a basic pixel rate of 20M per second ([10 * 2 + 5] / 10 cycles per pixel @ 50MHz). Also if half the pixels in the character are not written then the rate goes up to 291K. For comparison Leo achieves 75K characters per second.

3.2. Input Messages

Rasteriser Parameter Message Group		
Tag Mnemonic	Data Field	Description
RasteriserMode	See below	Defines the long term mode of operation of the rasteriser.
StartXDom	Fixed point 16.16 format	Initial X value for the dominant edge in trapezoid filling, or initial X value in line drawing.
dXDom	Fixed point 16.16 format	Value added when moving from one scanline (or sub scanline) to the next for the dominant edge in trapezoid filling. Also holds the change in X when plotting lines so for Y major lines this will be some fraction (dx/dy), otherwise it is normally ± 1.0, depending on the required scanning direction.
StartXSub	Fixed point 16.16 format	Initial X value for the subordinate edge.
dXSub	Fixed point 16.16 format	Value added when moving from one scanline (or sub scanline) to the next for the subordinate edge in trapezoid filling.
StartY	Fixed point 16.16 format	Initial scanline (or sub scanline) in trapezoid filling, or initial Y position for line drawing.
dY	Fixed point 16.16 format	Value added to Y to move from one scanline to the next. For X major lines this will be some fraction (dy/dx), otherwise it is normally ± 1.0, depending on the required scanning direction.
Count	16 bit integer	Number of pixels in a line. Number of scanlines in a trapezoid. Number of sub scanlines in an antialiased trapezoid. Diameter of a point in sub scanlines.
BitMaskPattern	32 bits as defined earlier	Value used to control the BitMask stipple operation (if enabled).
PointTable[0, 1, 2, 3]	Packed dx point data. The format is defined later.	Antialiase point data table. There are 4 words in the table and the message tag is decoded to select a word.

Rasteriser Control Message Group		
Tag Mnemonic	Data Field	Description
Render	See below	Starts the rendering process. This message and the data field are passed through to the other blocks as the PrepareToRender message. The data field defines the short term modes required by this primitive.
ContinueNewDom	16 bit integer	Allows the rasterisation to continue with a new dominant edge. The dominant edge DDA is reloaded with the new parameters. The subordinate edge is carried on from the previous trapezoid. This allows any convex polygon to be broken down into a collection of trapezoids and continuity maintained across boundaries. The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.
ContinueNewSub	16 bit integer	Allows the rasterisation to continue with a new subordinate edge. The subordinate DDA is reloaded with the new parameters. The dominant edge is carried on from the previous trapezoid. This is very useful when scan converting triangles with a 'knee' (i.e. two subordinate edges). The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.
Continue	16 bit integer	Allows the rasterisation to continue after new delta value(s) have been loaded, but doesn't cause either of the trapezoid's edge DDAs to be reloaded. The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register.

ContinueNewLine	16 bit integer	Allows the raterisation to continue for the next segment in a polyline. The XY position is carried on from the previous line, however the fraction bits in the DDAs can be kept, set to zero or half under control of the RasteriserMode. The data field holds the number of scanlines (or sub scanlines) to fill. Note this count does not get loaded into the Count register. *The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments.*
FlushSpan	Not used	Used when antialiasing to force the last span out when not all sub spans may be defined.

The format of the Render message data field is shown below:

Bit No.	Name	Description
0	AreaStippleEnable	This bit, when set, enables area stippling of the fragments produced during rasterisation in the Stipple Unit. Note that area stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no area stippling occurs irrespective of the setting of the area stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no area stippling for this primitive.
1	LineStippleEnable	This bit, when set, enables line stippling of the fragments produced during rasterisation in the Stipple Unit. Note that line stipple in the Stipple Unit must be enabled as well for stippling to occur. When this bit is reset no line stippling occurs irrespective of the setting of the line stipple enable bit in the Stipple Unit. This bit is useful to temporarily force no line stippling for this primitive.
2	ResetLineStipple	This bit, when set, causes the line stipple counters in the Stipple Unit to be reset to zero, and would typically be used for the first segment in a polyline. This action is also qualified by the LineStippleEnable bit and also the stipple enable bits in the Stipple Unit. When this bit is reset the stipple counters carry on from where they left off (if line stippling is enabled)
3	FastFillEnable	This bit, when set, causes the FastBlockLimits message to be sent on every scanline step. During the span fill the X value increments by the amount determined by FastFillIncrement and each fragment output uses the FastBlockFill message rather than the normal walk messages. When this bit is reset the normal rasterisation process occurs.
4, 5	FastFillIncrement	This two bit field selects the block size the framebuffer supports. The sizes supported and the corresponding codes are: 0 = 8 pixels, 1 = 16 pixels, 2 = 32 pixels.
6, 7	PrimitiveType	This two bit field selects the primitive type to rasterise. The primitives are: 0 = Line 1 = Trapezoid 2 = Point
8	AntialiaseEnable	This bit, when set, causes the generation of sub scanline data and the coverage value to be calculated for each fragment. The number of sub pixel samples to use is controlled by the AntialiasingQuality bit. When this bit is reset normal rasterisation occurs.
9	AntialiasingQuality	This bit, when set, sets the sub pixel resolution to be 8x8 When this bit is reset the sub pixel resolution is 4x4.

10	UsePointTable	When this bit and the AntialiasingEnable are set, the dx values used to move from one scanline to the next are derived from the Point Table.
11	SyncOnBitMask	This bit, when set, causes a number of actions: The least significant bit or most significant bit (depending on the MirrorBitMask bit) in the Bit Mask register is extracted and optionally inverted (controlled by the InvertBitMask bit). If this bit is 0 then any Active walk message is made into a Passive one, otherwise the message stays an Active one. After every fragment the Bit Mask register is rotated by one bit. If all the bits in the Bit Mask register have been used then rasterisation is suspended until a new BitMaskPattern message is received. If any other message is received while the rasterisation is suspended then the rasterisation is aborted. The message which caused the abort is then processed as normal for that message type. Note the behaviour is slightly different when the SyncOnHostData bit is set to prevent a deadlock from occurring. In this case the rasterisation doesn't suspend when all the bits have been used and if new BitMaskPattern messages are not received in a timely manner then the subsequent fragments will just reuse the bitmask.
12	SyncOnHostData	When this bit is set a fragment is produced only when one of the following messages have been received from the host: Depth, Stencil, Colour or FBColour. If SyncOnBitMask is reset then any message other than one of these three is received then the rasterisation is aborted. If SyncOnBitMask is set then any message other than one of these three or BitMaskPattern is received then the rasterisation is aborted. The message which caused the abort is then processed as normal for that message type. The BitMaskPattern message doesn't cause any fragments to be generated, but just updates the BitMask register.
13	TextureEnable	This bit, when set, enables texturing of the fragments produced during rasterisation. Note that the Texture Units must be suitably enabled as well for any texturing to occur. When this bit is reset no texturing occurs irrespective of the setting of the Texture Unit controls. This bit is useful to temporarily force no texturing for this primitive.
14	FogEnable	This bit, when set, enables fogging of the fragments produced during rasterisation. Note that the Fog Unit must be suitably enabled as well for any fogging to occur. When this bit is reset no fogging occurs irrespective of the setting of the Fog Unit controls. This bit is useful to temporarily force no fogging for this primitive.

15	CoverageEnable	This bit, when set, enables the coverage value produced as part of the antialiasing to weight the alpha value in the alpha test unit. Not that this unit must be suitably enabled as well. When this bit is reset no coverage application occurs irrespective of the setting of the AntialiasMode in the Alpha Test unit.
16	SubPixelCorrectionEnabl e	This bit, when set enables the sub pixel correction of the colour, depth, fog and texture values at the start of a scanline. When this bit is reset no correction is done at the start of a scanline. Sub pixel corrections are only applied to aliased trapezoids.

The format of the RasteriserMode message data field is shown below:

Bit No.	Name	Description
0	MirrorBitMask	When this bit is set the bitmask bits are consumed from the most significant end towards the least significant end. When this bit is reset the bitmask bits are consumed from the least significant end towards the most significant end.
1	InvertBitMask	When this bit is set the bitmask is inverted first before being tested.
2,3	FractionAdjust	These bits control the action of a ContinueNewLine message and specify how the fraction bits in the Y and XDom DDAs are adjusted. 0: No adjustment is done, 1: Set the fraction bits to zero, 2: Set the fraction bits to half. 3: Set the fraction to *nearly half*, i.e. 0x7fff
4,5	BiasCoordinates	These bits control how much is added onto the SartXDom, StartXSub and StartY values when they are loaded into the DDA units. The original registers are not effected. 0: Zero is added, 1: Half is added, 2: *Nearly half*, i.e. 0x7fff is added

3.3. Output Messages

Rasteriser 'Walk' message group		
Tag Mnemonic	Data Field	Description
PrepareToRender	See the Render message data field.	This message precedes any of the 'Walk' messages and allows the Units to get ready.
ActiveStepX	Current (X, Y). X in ls 16 bits, Y in ms 16 bits	Valid fragment at given coordinate. Next fragment at (X± 1, Y)
ActiveStepYDomEdge	Current (X, Y). X in ls 16 bits, Y in ms 16 bits	Valid fragment at given coordinate. Next fragment at Y+1 on left edge
PassiveStepX	Current (X, Y). X in ls 16 bits, Y in ms 16 bits	Failed fragment at given coordinate. Next fragment at (X±1, Y)
PassiveStepYDomEdge	Current (X, Y) X in ls 16 bits, Y in ms 16 bit	Failed fragment at given coordinate. Next fragment at Y+1 on left edge
SubPixelCorrection	Bits 3…0 hold the magnitude of the correction. Bit 4 holds the sign, 1 is negative.	Holds the amount of sub pixel correction to apply in any enabled DDA units down stream.

Rasteriser Misc message group		
Tag Mnemonic	Data Field	Description
FastBlockLimits	X left in ls 16 bits, X right in ms 16 bits	Limits of the pixels on a scanline to fast fill between in screen relative coordinates.
FastBlockFill	Current (X, Y). X in ls 16 bits, Y in ms 16 bits	The group of pixels to fast fill. The intersection of this group and the Fast Fill limits selects the range of pixels within a group to fill. In screen relative coordinates.
CoverageValue	Coverage value in ls 9 bits. 0 = 0% 256 = 100%	The coverage value calculated during antialiasing.

3.4. Message Sequences

This section gives the message sequences necessary to get the rasteriser to scan convert various primitive types. Any other messages necessary to set up the Colour DDA, for example, or any other mode information are not covered here.

In general the order of the messages is not important, but obviously as the Render message starts the rendering it should only be sent once any other parameters have been set up.

None of the registers are used destructively (except the BitMaskPattern) so only those parameters which have changed from the previous primitive need to be sent. The message sequences outline here send all the parameters.

The contents of the data field is enclosed in round brackets, however this should only be taken as a guide because for slope calculation the sub pixel position or snapping to the pixel's centre have not been taken into account.

The Render (or PrepareToRender) data field is tabulated. In this table:

Reset = 0, Set = 1 and Don't care = X

3.4.1. One Pixel Points

A series of one pixel points P(X_1, Y_1), P(X_2, Y_2) … P(X_n, Y_n) are required.

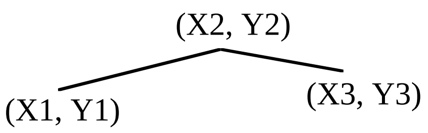
Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	2
FastFillEnable	0	FastFillIncrement	X	UsePointTable	X
AntialiaseEnable	X	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	X	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

```
StartXDom (X1)
StartY (Y1)
Render
StartXDom (X2)
StartY (Y2)
Render
…     …
…     …
StartXDom (Xn)
StartY (Yn)
Render
```

These points can be plotted more efficiently by using the PrepareToRender and ActiveStepX messages directly but this presents a less clean interface to the programmer.

3.4.2. Aliased PolyLine (OpenGL or simple stipple X)

A two segment polyline from (X_1, Y_1) to (X_2, Y_2) to (X_3, Y_3) is required. Both segments are X major so



$$\text{abs}(X_{n+1} - X_n) > \text{abs}(Y_{n+1} - Y_n)$$

Note that for individual line segments or the first line segment in a polyline the line stipple is reset (as shown).

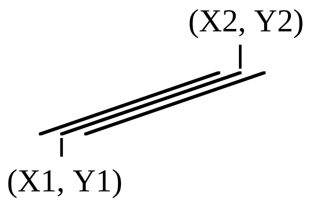
Render Data Field					
AreaStippleEnable	0	LineStippleEnable	1	PrimitiveType	0
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	1
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

```
StartXDom (X1)
dXDom (±1.0)
StartY (Y1)
dY ((Y2- Y1)/(X2 - X1))
Count (abs (X2 - X1))
Render
dXDom (±1.0)
dY ((Y3- Y2)/(X3 - X2))
ContinueNewLine (abs (X3 - X2))
```

The use of ContinueNewLine is not recommended for OpenGL because the DDA units will start with a slight error as compared with the value they would have been loaded with for the second and subsequent segments. The fractional bits of the DDA can be forces to zero or half on the ContinueNewLine action.

3.4.3. Aliased Wide Line (OpenGL)

A single wide line from (X_1, Y_1) to (X_2, Y_2) is required. The line is 3 pixels wide. The line is X major so abs $(X_2 - X_1) >$ abs $(Y_2 - Y_1)$.



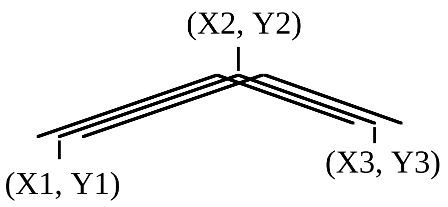
Render Data Field					
AreaStippleEnable	0	LineStippleEnable	1	PrimitiveType	0
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	1
SyncOnBitMask	0	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

```
StartXDom (X1 - 1)
dXDom (±1.0)
StartY (Y1)
dY ((Y2- Y1)/(X2 - X1))
Count (abs (X2 - X1))
Render
StartXDom (X1)
Render
StartXDom (X1 + 1)
Render
```

3.4.4. Aliased Wide Polyline (OpenGL)

A wide line from (X_1, Y_1) to (X_2, Y_2) and then (X_3, Y_3) is required. The line is 3 pixels wide. The lines are X major so:

abs (X_2 - X_1) > abs (Y_2- Y_1) and
abs (X_3 - X_2) > abs (Y_3- Y_2)

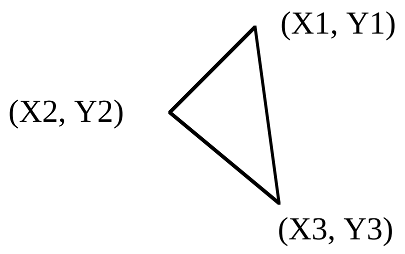


Render Data Field					
AreaStippleEnable	0	LineStippleEnable	1	PrimitiveType	0
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	0
SyncOnBitMask	0	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	0

```
StartXDom (X1 - 1)
dXDom (±1.0)
StartY (Y1)
dY ((Y2- Y1)/(X2 - X1))
Count (abs (X2 - X1))
LoadLineStippleCounters (0)    // Reset stipple counters
Render                         // First segment
StartXDom (X1)
LoadLineStippleCounters (0)    // Reset stipple counters
Render
StartXDom (X1 + 1)
LoadLineStippleCounters (0)    // Reset stipple counters
Render
SaveLineStippleState
StartXDom (X2 - 1)
dXDom (±1.0)
StartY (Y2)
dY ((Y3- Y2)/(X3 - X2))
Count (abs (X3 - X2))
LoadLineStippleCounters (1)    // Restore stipple counters
Render                         // Second segment
StartXDom (X2)
LoadLineStippleCounters (1)    // Restore stipple counters
Render
StartXDom (X2 + 1)
LoadLineStippleCounters (1)    // Restore stipple counters
Render
```

3.4.5. Aliased Triangle

The triangle looks like this and we will render from top to bottom. The origin is assumed to be bottom left:



Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	0	SubPixelCorrectionEnable	1

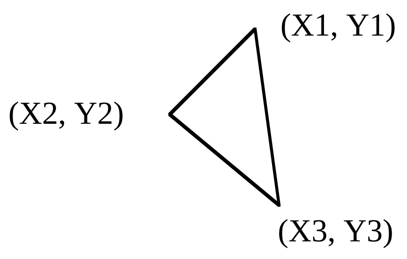
```
StartXDom ($X_1$)
dXDom (($X_3$- $X_1$)/($Y_3$ - $Y_1$))
StartXSub ($X_1$)
dXSub (($X_2$- $X_1$)/($Y_2$ - $Y_1$))
StartY ($Y_1$)
dY (-1.0)
Count ($Y_1$ - $Y_2$)
Render                          // Top half
StartXSub ($X_2$)
dXSub (($X_3$- $X_2$)/($Y_3$ - $Y_2$))
ContinueNewSub ($Y_2$ - $Y_3$)        // Bottom half
```

Note that if both edges need to be reloaded to continue on with the bottom half of the polygon then issue ContinueNewSub (0) and then ContinueNewDom (count). The ContinueNewSub (0) will just update the DDA with the new start value but not draw any scanlines. Alternatively , if the accuracy of the DDA end values is good enough and can be used as the start values for the next trapezoid then the delta values can be updated and the Continue message used.

The sub pixel correction is only needed if colour, depth, fog or texture interpolation is being used.

3.4.6. Antialiased Triangle

The triangle looks like this and we will render from top to bottom. The origin is assumed to be bottom left and we are using 4x4 antialiasing quality:



Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	1	AntialiasingQuality	0	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	1
FogEnable	1	CoverageEnable	1	SubPixelCorrectionEnable	1

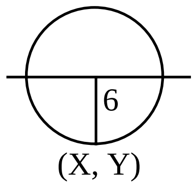
```
StartXDom (X1)
dXDom ((X3- X1)/(4 * (Y3 - Y1)))
StartXSub (X1)
dXSub ((X2- X1)/(4 * (Y2 - Y1)))
StartY (Y1)
dY (-1.0/ 4.0)
Count ((Y1 - Y2) * 4)
Render                          // Top half
StartXSub (X2)
dXSub ((X3- X2)/(4 * (Y3 - Y2)))
ContinueNewSub ((Y2 - Y3) * 4)  // Bottom half
FlushSpan ()
```

Note that the DDA units will need to have their sample point biased from the centre of the pixel to the lower edge of the pixel so the DDA units can be tracked properly with the walk messages. This can be done by calculating the start values for integer Y values rather than at Y+0.5 as would normally be done.

The sub pixel correction is only needed if colour, depth, fog or texture interpolation is being used.

3.4.7. Antialiased Point

The point looks like this and we will render from bottom to top. The origin is assumed to be bottom left and we are using 4x4 antialiasing quality. The point's diameter is 3 pixels, or 12 sub scanlines. The point table is assumed to be set up already.

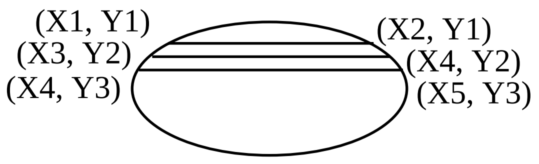


Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	1
AntialiaseEnable	1	AntialiasingQuality	0	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	X	TextureEnable	1
FogEnable	1	CoverageEnable	1	SubPixelCorrectionEnable	0

```
StartXDom (X)
StartXSub (X)
StartY (Y)
dY (1.0/ 4.0)
Count (12)
Render
FlushSpan ()
```

3.4.8. Group of Spans (X)

This demonstrates how X may scan convert a shape which is more complicated than GLiNT can handle. X will convert the shape into a series of spans (only 3 are shown here) and GLiNT will fill in the spans.

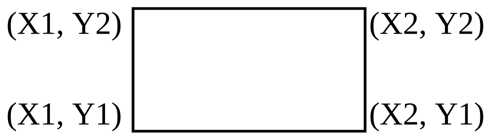


Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	0
FogEnable	0	CoverageEnable	0	SubPixelCorrectionEnable	0

```
StartXDom (X1)
StartXSub (X2)
StartY (Y1)
Count (1)
Render                          // 1st span
StartXDom (X3)
StartXSub (X4)
StartY (Y2)
Render                          // 2nd span
StartXDom (X5)
StartXSub (X6)
StartY (Y3)
Render                          // 3rd span
```

3.4.9. Fill Rectangle (i.e. clear)

The rectangle looks like this and the origin is assumed to be bottom left:



Render Data Field					
AreaStippleEnable	1	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	0	TextureEnable	0
FogEnable	0	CoverageEnable	0	SubPixelCorrectionEnable	0

To fill top to bottom and left to right:

```
StartXDom (X1)
dXDom (0)
StartXSub (X2)
dXSub (0)
StartY (Y2)
dY (-1.0)
Count (Y2 - Y1)
Render
```

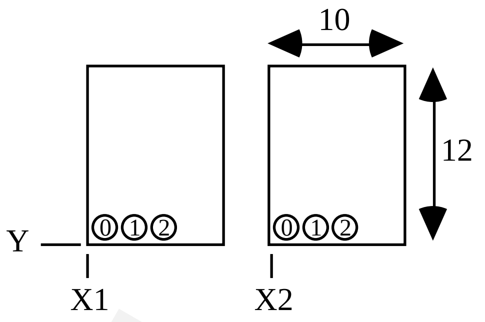
To fill bottom to top and right to left:

```
StartXDom (X2)
dXDom (0)
StartXSub (X1)
dXSub (0)
StartY (Y1)
dY (1.0)
Count (Y2 - Y1)
Render
```

The other two fill directions are simple permutations of these two cases.

3.4.10. Character String

The two characters looks like this and the origin is assumed to be bottom left. The relationship between the bits in the BitMaskPattern and pixels in the character cell are shown by the numbers (0 being the ls bit). The pixels will be plotted bottom to top and left to right.



Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	1	SyncOnHostData	0	TextureEnable	0
FogEnable	0	CoverageEnable	0	SubPixelCorrectionEnable	0

In OpenGL the AreaStippleEnable would always be 0, but in X may be enabled or disabled.

```
StartXDom (X1)
dXDom (0)
StartXSub (X1+ 10)
dXSub (0)
StartY (Y)
dY (1.0)
Count (12)
Render                  // Start first character
BitMaskPattern ()
BitMaskPattern ()
BitMaskPattern ()
BitMaskPattern ()
StartXDom (X2)          // Note only two parameters needed
StartXSub (X2+ 10)
Render                  // Start second character
BitMaskPattern ()
BitMaskPattern ()
BitMaskPattern ()
BitMaskPattern ()
```

Note that the rasteriser overscans the character because the right hand edge is not plotted and the BitMaskPattern doesn't include these pixels

3.4.11. Image Download

The image download rectangle looks like this and the origin is assumed to be bottom left. The host provides the data in top to bottom, left to right order. Colour data will be provided. There are *n* pixels in the rectangle.

(X1, Y2)
(X2, Y2)
(X1, Y1)
(X2, Y1)

Render Data Field					
AreaStippleEnable	0	LineStippleEnable	0	PrimitiveType	1
FastFillEnable	0	FastFillIncrement	X	UsePointTable	0
AntialiaseEnable	0	AntialiasingQuality	X	ResetLineStipple	X
SyncOnBitMask	0	SyncOnHostData	1	TextureEnable	0
FogEnable	0	CoverageEnable	0	SubPixelCorrectionEnable	0

In OpenGL the AreaStippleEnable would always be 0, but in X may be enabled or disabled.

```
StartXDom (X1)
dXDom (0)
StartXSub (X2)
dXSub (0)
StartY (Y2)
dY (-1.0)
Count (Y2 - Y1 + 1)   // Width of image
Render
Colour (P0)           // Pixel 0
Colour (P1)
Colour (P2)
…
…
Colour (Pn)
```

Note that the rasteriser overscans the rectangle because the right hand edge is not plotted and the downloaded image doesn't include these pixels

3.5. Behavioural Model

The behavioural model for the rasteriser is too complex to describe on one page so it has been split into 'functions' to make the description more manageable. Each 'function' definition and usage has been highlighted by using a bold font so they may be easily seen.

```
main
{
   Wait for input message;
   {
      switch (on input message)
      {
         case ParameterGroup:
               Update the selected register;
               Flush the input message;
               break;
         case ControlGroup:
               switch (on input message)
               {
                  case Render:
                        ProcessRenderMessage;
                        break;
                  case ContinueNewDom:
                        ProcessContinueNewDomMessage;
                        break;
                  case ContinueNewSub;
                        ProcessContinueNewSubMessage;
                        break;
                  case Continue;
                        ProcessContinueMessage;
                        break;
                  case ContinueNewLine;
                        Copy count in data field to counter;
                        Adjust the Xdom and Y DDA fraction bits
                           depending on AdjustFractions field in
                           RasteriserMode message;
                        ProcessLinePrimitive;
                        break;
                  case FlushSpan:
                        ProcessFlushSpanMessage;
                        break;
               }
               break;
         default:
               Forward input message;
               break;
      }
   }
}
```

```
ProcessRenderMessage
{
    if (PrimitiveType == undefined)
        return
    Copy data field into RenderData register;
    Flush the input message;
    Issue start command to the Y, XDom and XSub DDA units and
        addjust depending on BiasCoordinates;
    Copy Count value to counter;
    Reset the Bit Mask Unit;
    Set RadiusCount = 0 and RadiusDirection = up;
    Wait for room in the receiving queue;
    Send PrepareToRender message using the RenderData;
    if (PrimitiveType == Point)
    {
        X = integer part of XDom DDA output;
        Y = integer part of Y DDA output;
        GenerateMessage (StepX);
    }
    else
    {
        if (PrimitiveType == Line)
        {
            ProcessLinePrimitive;
        }
        else
        {
            if (AntialiasingEnabled == 1)
            {
                Clear subspan counter;
                ProcessAntialiasTrapezoidPrimitive;
            }
            else
            {
                ProcessTrapezoidPrimitive;
            }
        }
    }
}
```

```
ProcessLinePrimitive
{
    while (counter != 0)
    {
        X = integer part of XDom DDA output;
        Y = integer part of Y DDA output;
        GenerateMessage (StepYDomEdge);
        Step Xdom and Y DDA units;
        Decrement counter;
    }
}
```

```
ProcessTrapezoidPrimitive
{
  while (counter != 0)
  {
    Copy Xdom and Xsub data into the Span Fill Unit;
    FillSpan;
    Step the X DDAs using the step constant mode;
    Step the Y DDA unit;
    Decrement counter;
  }
}
```

```
FillSpan
{
   Y = integer part of Y DDA output;
   if (Xcounter != XStop)
   {
      if (FastFillEnable == 1)
      {
         if (Xcounter <= XStop)
         {
            Send FastBlockLimits message with Xleft = Xcounter and
               Xright = XStop;
            while (Xcounter < XStop)
            {
               Send FastBlockFill message;
               switch (on FastFillIncrement)
               {
                  case 0: Increment Xcounter by 8; break;
                  case 1: Increment Xcounter by 16; break;
                  case 2: Increment Xcounter by 32; break;
               }
            }
            Send FastBlockFill message;
         }
         else
         {
            Send FastBlockLimits message with Xleft = XStop and
               Xright = Xcounter;
            while (Xcounter >= Xstop)
            {
               Send FastBlockFill message;
               switch (on FastFillIncrement)
               {
                  case 0: Decrement Xcounter by 8; break;
                  case 1: Decrement Xcounter by 16; break;
                  case 2: Decrement Xcounter by 32; break;
               }
            }
            Send FastBlockFill message;
         }
      }
      else
      {
         CalculateSubPixelCorrection;
         if (Xcounter < Xstop)
         {
            while (Xcounter < XStop)
            {
               X = Xcounter;
               GenerateMessage (StepX);
               Increment Xcounter;
            }
            GenerateMessage (PassiveStepYDomEdge);
         }
         else
         {
            GenerateMessage (PassiveStepX);
            Decrement Xcounter;
            while (Xcounter > XStop)
            {
               X = Xcounter;
               GenerateMessage (StepX)
               Decrement Xcounter;
            }
            X = Xcounter;
            GenerateMessage (ActiveStepYDomEdge);
            }
        }
    }
    else
    {
        GenerateMessage (PassiveStepYDomEdge);
    }
}
```

CalculateSubPixelCorrection

```
{
    if (SubPixelCorrection is enabled)
    {
        // almost one = 0.FFFF
        dErr magnitude = (almost one - Fract (XDom)) >> 12;
        if (Xcounter < XStop)
            dErr sign = positive;
        else
            dErr sign = negative;
        Wait for room in next unit;
        Send SubPixelCorrection message with dErr magnitude and
            dErr sign in the data field;
    }
}
```

```
ProcessAntialiaseTrapezoidPrimitive
{
   Reset sub span data complete flag;
   while (counter != 0)
   {
      LoadSubSpanData from XDom and XSub values;
      if (sub span data complete)
      {
         FillAntialiaseSpan;
         Clear subspan counter;
         Reset sub span data complete flag;
      }
      else
      {
         if (UsePointTable)
         {
            Extract and sign extend delta values from the point
               table at address RadiusCount;
            if (RadiusDirection = down)
            {
               left delta = table delta;
               right delta = -table delta;
            }
            else
               left delta = -table delta;
               right delta = table delta;
            }
            Step the X DDAs using the variable constant mode;
            switch (RadiusDirection)
            {
               case up:
                        Increment RadiusCount;
                        if (RadiusCount = Count / 2 - 1)
                           if (Count is even)
                              RadiusDirection = down;
                           else
                              RadiusDirection = level;
                        break;
               case level:
                        RadiusDirection = down;
                        break;
               case down:
                        Decrement RadiusCount;
                        break;
            }
         }
         else
         {
            Step the X DDAs uning the step constant mode;
         }
         Decrement counter;
         Step the Y DDA unit;
      }

   }
}
```

```
LoadSubSpanData
{
  if (subspan counter == 0)
  {
    Load scanline Y into scanline number register;
    Load XDom and XSub into the register file at address in the
      subspan counter;
    Load XCounter from XDom;
    Load XStop from XSub;
    switch (Compare XDom and XSub)
    {
      case XDom > XSub:
      case XDom = XSub:
                          scan direction = right to left;
                          break;
      case XDom < XSub:
                          scan direction = left to right;
                          break;
    }
    Increment subspan counter;
  }
  else
  {
    if (scanline Y is different to scanline number register)
    {
      // We have moved onto the next scanline so return
      // the fact that the sub scanline data for this span
      // is complete.
      // Don't use the XDom and XSub values this time.
      Set sub spanline complete flag;
    }
    else
    {
      // More sub span data for our scanline.
      switch (Compare XDom and XSub)
      {
        case XDom > XSub:
                            scan direction = right to left;
                            break;
        case XDom = XSub:
                            keep previous scanning direction;
                            break;
        case XDom < XSub:
                            scan direction = left to right;
                            break;
      }
      Load XDom and XSub into the register file at address
        in the subspan counter;
      if (scan direction is left to right)
      {
        Compare XDom with XCounter and if less than update
          XCounter;
        Compare XSub with XStop and if greater than update
          XStop;
      }
      else
      {
        Compare XDom with XCounter and if greater than update
          XCounter;
        Compare XSub with XStop and if less than update
          XStop;
      }
      Increment subspan counter;
      }
   }
}


FillAntialiaseSpan
{
   if (scanning direction is left to right)
   {
      while (Xcounter <= XStop)
      {
         Clear the accumulated coverage value;
         Clear sub scanline number;
         while (sub scanline number < subspan counter)
         {
            Fetch XDom and XSub from the register file at the
               address given by the sub scanline number;
            Compute the coverage for fragment Xcounter;
            Accumulate the coverage;
            Increment sub scanline number;
         }
         Y = scanline number register;
         X = Xcounter;
         GenerateCoverageMessage;
         Increment Xcounter;
      }
   }
   else
   {
      while (Xcounter >= XStop)
      {
         Clear the accumulated coverage value;
         Clear sub scanline number;
         while (sub scanline number < subspan counter)
         {
            Fetch XDom and XSub from the register file at the
               address given by the sub scanline number;
            Compute the coverage for fragment Xcounter;
            Accumulate the coverage;
            Increment sub scanline number;
         }
         Y = scanline number register;
         X = Xcounter;
         GenerateCoverageMessage;
         Decrement Xcounter;
      }
   }
}
```

```
GenerateCoverageMessage
{
  if (coverage != 0%)
  {
    if (Xcounter == XStop)
    {
      Send Coverage message to next unit;
      GenerateMessage (ActiveStepYDomEdge);
    }
    else
    {
      Send Coverage message to next unit;
      GenerateMessage (StepX);
    }
  }
  else
  {
    if (Xcounter == XStop)
    {
      GenerateMessage (PassiveStepYDomEdge);
    }
  }
}


GenerateMessage (step message)
{
  if (step message is passive)
    Send step message;
  else
  {
    // No bitmask bit or image data is expected for right hand
    // edge due to point sampling so is not consumed here.
    if (SyncOnHostData & SyncOnBitMask)
    {
      MGSyncOnHostDataAndSyncOnBitMask
    }
    if (SyncOnHostData & !SyncOnBitMask)
    {
      MGSyncOnHostData
    }
    if (!SyncOnHostData & SyncOnBitMask)
    {
      MGSyncOnBitMask
    }
    if (!SyncOnHostData & !SyncOnBitMask)
    {
      MGNoSync
    }
  }
}
```

```
MGSyncOnHostDataAndSyncOnBitMask
{
  Set fragment processed to false;
  while (fragment processed is false)
  {
    Wait for input message;
    switch (on input message)
    {
      case Colour:
      case Depth:
      case Stencil:
      case FBData:
           if (X == -ve or Y == -ve)
           {
             Send Passive version of step message;
           }
           else
           {
             Extract test bit from the mask register
               depending on MirrorBitMask;
             Optionally invert the test bit depending on
               InvertBitMask;
             if (test bit == 1)
             {
               Forward input message;
               Send Active version of step  message;
             }
             else
             {
               Send Passive version of step message;
             }
           }
           Flush input message;
           Rotate mask register depending on MirrorBitMask;
           Set fragment processed to true;
           break;
      case BitMaskPattern:
             Update Bit Mask Unit with new pattern;
             Flush input message;
             break;
      default:
           Abort rendering and process message as normal;
           break;
    }
  }
}
```

```
MGSyncOnHostData
{
   Wait for input message;
   switch (on input message)
   {
      case Colour:
      case Depth:
      case Stencil:
      case FBData:
            if (X == -ve or Y == -ve)
               Send Passive version of step message;
            else
            {
               Forward input message;
               Send Active version of step message;
            }
            Flush input message;
            break;
      default:
            Abort rendering and process message as normal;
            break;
   }
}


MGSyncOnBitMask
{
   if (bit mask expired)
   {
      Wait for input message;
      if (input message is BitMaskPattern)
      {
         Update Bit Mask Unit with new value;
         Flush input message;
      }
      else
      {
         Abort rendering and process message as normal;
      }
   }
   if (X == -ve or Y == -ve)
      Send Passive version of step message;
   else
   {
      Extract test bit from the mask register depending
         on MirrorBitMask;
      Optionally invert the test bit depending on InvertBitMask;
      if (test bit == 1)
         Send Active version of step  message;
      else
         Send Passive version of step message;
   }
   Rotate mask register depending on MirrorBitMask;
}
```

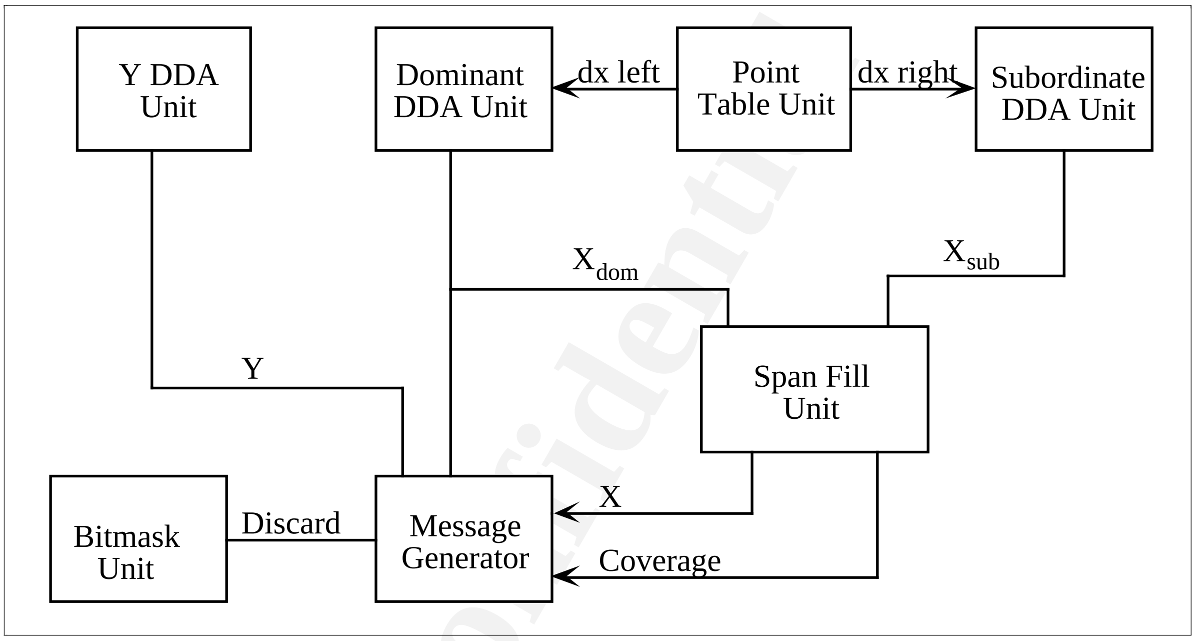
```
MGNoSync
{
  if (X == -ve or Y == -ve)
    Send Passive version of step message;
  else
    Send Active version of step  message;
}


ProcessContinueNewDomMessage
{
  Flush the input message;
  Issue start command to the XDom DDA unit and addjust
    depending on BiasCoordinates;
  Copy Count value to counter;
  if (AntialiasingEnabled == 1)
    ProcessAntialiasTrapizoidPrimitive;
  else
    ProcessTrapezoidPrimitive;
}


ProcessContinueNewSubMessage
{
  Flush the input message;
  Issue start command to the XSub DDA unit and addjust
    depending on BiasCoordinates;
  Copy Count value to counter;
  if (AntialiasingEnabled == 1)
    ProcessAntialiasTrapizoidPrimitive;
  else
    ProcessTrapezoidPrimitive;
}


ProcessContinueMessage
{
  Flush the input message;
  Copy Count value to counter;
  if (AntialiasingEnabled == 1)
    ProcessAntialiasTrapizoidPrimitive;
  else
    ProcessTrapezoidPrimitive;
}


ProcessFlushSpanMessage
{
  Flush the input message;
  FillAntialiaseSpan;
}
```

3.6. Implementation

The Rasteriser is one of the most complex and important units in the chip so will be covered in depth.

The target fragment generation rate is no worse than one fragment every two clock cycles as this is the maximum rate the memories can work at.

The top level block diagram of the rasteriser is shown below:



An informal description of how to rasterise some typical primitives will now be covered to set each block in the above diagram into some context before the blocks are described in detail.

To draw an aliased line the following steps take place:

1. All the start and slope (derivative) registers are set up and the Render message (with the PrimitiveType set to line) has been received. The loop counter holds the number of pixels in the line to plot.

2. The state machine kicks in and initialise the DDA units with a start command. This causes the start coordinate to be copied to the output register in preparation for generating the first fragment.

3. The state machine then loops until the loop counter decrements to zero and each time around the loop it generates an ActiveStepXY message, filling in the data field with the X and Y coordinates from the Y DDA and X Dom DDA units. The DDA units are then given a step command to move them onto the next fragment.

To draw an aliased triangle the following steps take place:

1. All the start and slope (derivative) registers are set up for the first part of the triangle and the Render message (with the PrimitiveType set to trapezoid) has been received. The loop counter holds the number of scan lines the part triangle covers.

2. The state machine kicks in and initialise the DDA units with a start command. This causes the start coordinate to be copied to the output register in preparation for generating the first fragment.

3. The dominant X value, X_{dom}, and subordinate X value, X_{sub}, fragment positions (with subpixel resolution) are loaded into the Span Fill unit.

4. The Span Fill unit iterates from the X_{dom} to X_{sub} pixel positions on the scanline. Remember the right most fragment in the span is not really generated, however a passive message may be needed to keep the DDA units in sync. If the X_{dom} and X_{sub} integer values are equal then no fragment is generated, however a PassiveStepDominantEdge message is needed to keep the DDA units in sync.

5. Decrement the loop counter and move onto the next scan line and calculate a pair of new X_{dom} and X_{sub} fragment positions. If the loop counter is not zero then steps 3 and 4 are repeated.

6. Once the first half of the triangle is complete we need to reload the X_{sub} start value and derivative, and the number of scan lines the second half of the triangle cover (into the loop counter).

7. A ContinueNewSub message causes steps 3, 4 and 5 to be repeated.

To draw an antialiased triangle the following steps take place. We assume that 4 sub scanlines are sampled per scanline to build up the coverage value:

1. All the start and slope (derivative) registers are set up for the first part of the triangle and the Render message (with the PrimitiveType set to trapezoid and antialiasing enabled) has been received. The loop counter holds the number of sub scanlines the part triangle covers. The derivative values are for sub scanline steps rather than whole scanline steps.

2. The state machine kicks in and initialise the DDA units with a start command. This causes the start coordinate to be copied to the output register in preparation for generating the first fragment.

3. The four pairs of dominant and subordinate X values ($X0_{dom}$, $X0_{sub}$), ($X1_{dom}$, $X1_{sub}$), ($X2_{dom}$, $X2_{sub}$), ($X3_{dom}$, $X4_{sub}$) are calculated by stepping the X DDA units four times (once per sub scanline) and loaded into the Span Fill unit. For simplicity here we are ignoring that some of the sub scanlines at the top and/or bottom of the triangle may not be touched. The loop counter is decremented for each sub scanline.

4. When the Xn_{dom} and Xn_{sub} values are loaded into the Span Fill unit the maximum and minimum values are determined. When filling from left to right we find the minimum X_{dom} and the maximum X_{sub}. When filling from right to left we find the maximum X_{dom} and the minimum X_{sub}.

5. The Span Fill unit iterates from X_{dom} to X_{sub} *inclusive* and for each fragment calculate the coverage value. The coverage value in this fragment for each sub scanline is calculated and summed together. A new ActiveStepX message for each fragment except the last one when an ActiveStepDominantEdge message is generated. The coverage value for this fragment is sent in a separate message immediately following the Active walk message. If a coverage value of zero is calculated then no fragment is generated, however a passive message is needed to keep the DDA units in sync.

6. If the loop counter is not zero then steps 3, 4 and 5 are repeated.

7. Once the first half of the triangle is complete we need to reload the X_{sub} start value and derivative, and the number of sub scanlines in the second half of the triangle cover (into

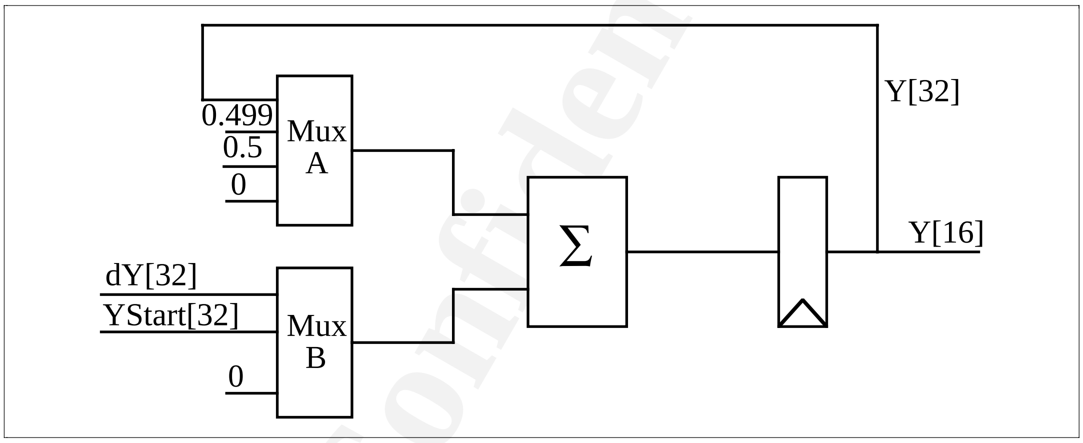
the loop counter). Note we have to handle the case where the knee occurs on a sub scanline such that some of the sub spans are from the first part of the triangle and some are from the second part. In this case we don't fill the incomplete scanline until the second subordinate edge is loaded.

8. A ContinueNewSub message causes steps 3, 4, 5 and 6 to be repeated.

9. Unless the second part of the triangle covers all four sub scanlines on the last scanline the last scanline of fragments will not be generated (because fragment generation only starts when all the sub scanlines are defined). The FlushSpan message causes the partial span to be generated.

3.6.1. Y DDA Unit

The Y DDA unit has two roles depending on whether we are drawing lines or trapezoids. In line drawing mode the unit is stepped after every pixel, while in trapezoid drawing mode it is stepped only when all the pixels on the scan line have been generated. The block diagram of the unit is shown below:

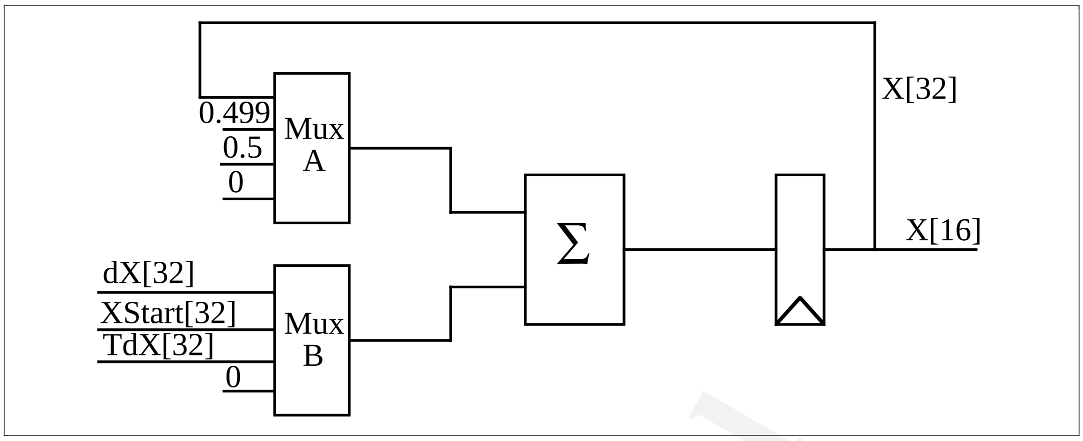


The unit responds to three commands (start, step and hold) and one of these is executed on every clock cycle. The only output value of the unit is the integer Y coordinate of the fragment or scan line we are on. Note that the direction of the step is determined by the sign of the delta Y value so it is possible to step from top to bottom, or vice versa (this flexibility is needed for line drawing and copy rectangles).

The number format is 1 bit sign, 15 bits integer and 16 bits fraction.

3.6.2. X DDA Units (Dominant and Subordinate)

The X DDA units trace out the dominant and subordinate edges of the trapezoids. The dominant unit is also used when drawing lines to provide the X coordinate. The block diagram of the unit is shown below:



It is identical to the Y DDA unit, except for the additional input to one of the multiplexers. This takes in a delta value derived from a table, rather than a constant from a register and is used when antialiasing points. This is covered later.

The unit responds to four commands (start, step constant, step variable and hold) and one of these is executed on every clock cycle. The only output value of the unit is the X coordinate (including fractional bits) of the fragment. Note that the direction of the step is determined by the sign of the delta dx/dy value so it is possible to step from left to right or vice versa (this flexibility is needed for line drawing and copy rectangles.

When we are antialiasing a primitive the delta values will be loaded with the change in X per sub scanline and stepped once per sub scanline.

3.6.3. Point Unit

The point unit is little more than a table which holds the offset in X (to sub pixel precision) to add to the current X value to trace the boundary of a circle. The table only holds the boundary steps for one quadrant and these steps are either added or subtracted from the preceding boundary position to trace out the boundary.

Consider what happens as we trace the boundary from the lowest scanline. Both the left and right X DDA units (we rename them from dominant and subordinate as these terms have no meaning for circles) start off with the same X value.

In the lower semicircle:

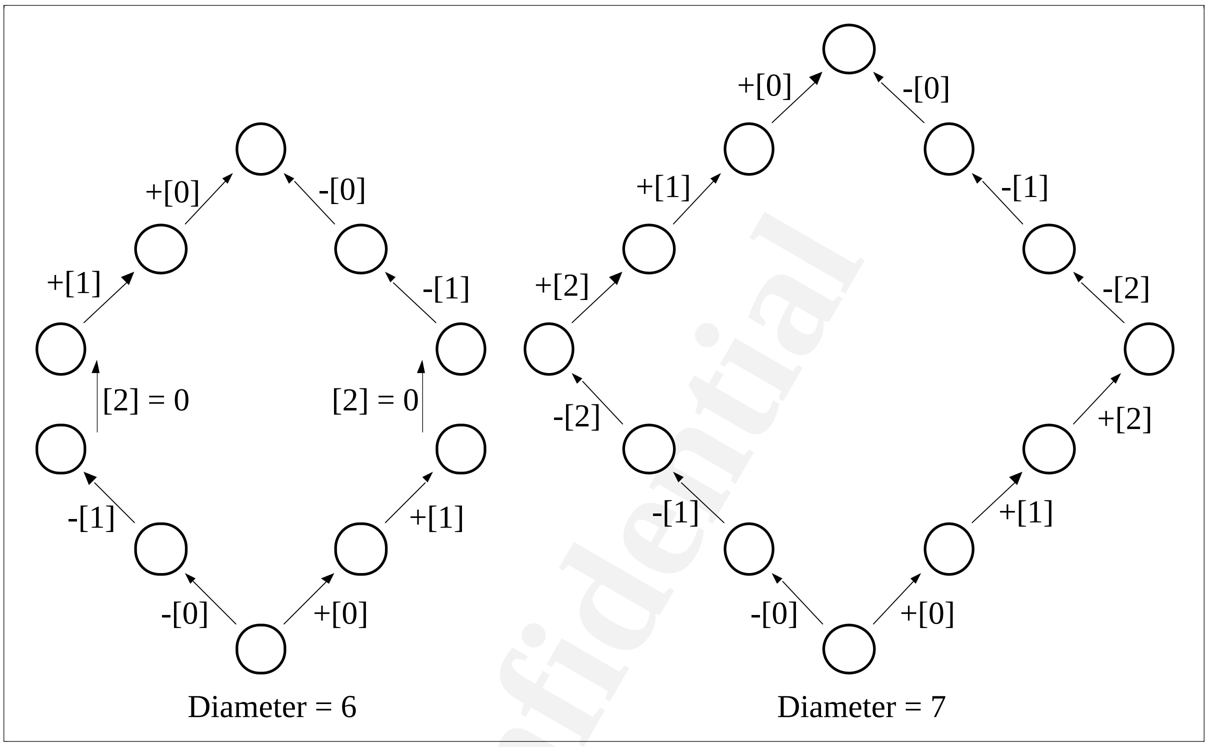
> As we step up through the scanlines the delta X value from the table is *subtracted* from the left X and *added* to the right X. When we move onto the next scanline we also *increment* the address in the table where the delta X value is taken from.

In the upper semicircle:

> As we step up through the scanlines the delta X value from the table is *added* to the left X and *subtracted* from the right X. When we move onto the next scanline we also *decrement* the address in the table where the delta X value is taken from.

The count value loaded by the Count message holds the diameter of the point to draw in sub scanline steps. For each sub scanline step the next entry in the table is accessed and the new boundary position calculated. When half the sub scanlines have been processed a flag is set which toggles the addition/subtraction and increment/decrement actions for each semicircle. This makes use of the vertical symmetry in a circle. Rather than be able to select whether the adders in the DDA units are adders or subtracters the actual data is complimented when it needs to be subtracted.

The pattern of table accesses, additions and subtractions are shown below for even and odd diameters. A diamond has been used to simplify the diagram. On the diagram the symbol ±[n] by an arrow indicates the contents of the table at address *n* are added/subtracted to move along the arrow.



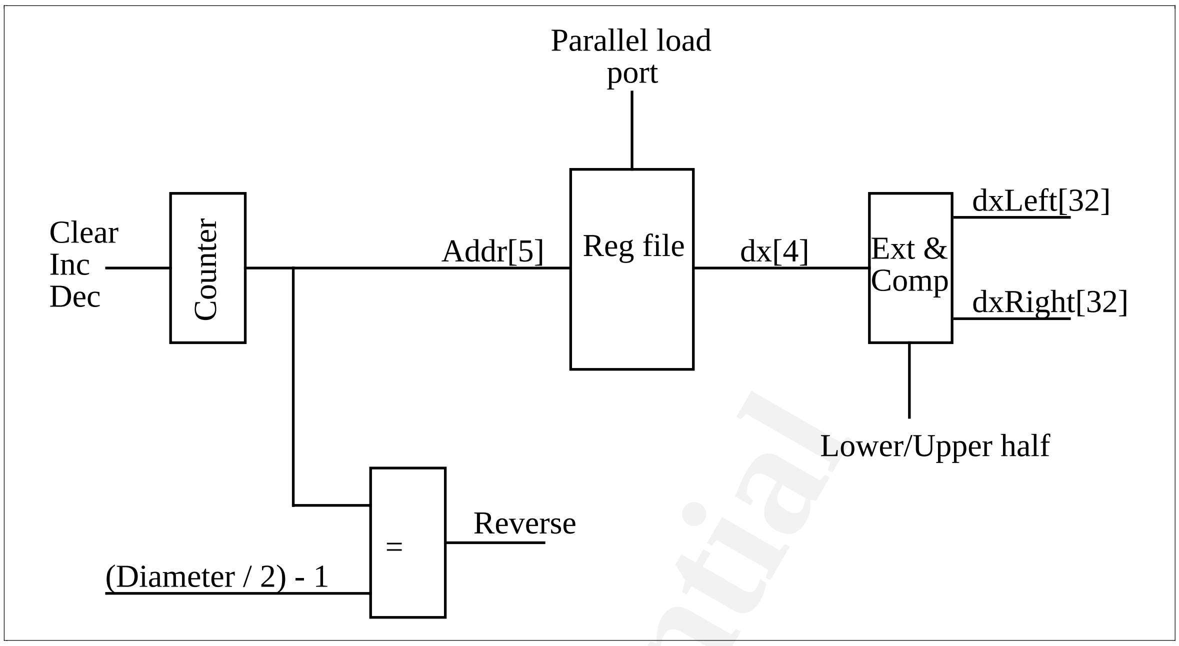
In this example the point table will have three entries. Note in the case of a even diameter the last of the required entries in the table is set to zero.

The Ext & Comp block convert the delta X value from the table to the 32 bit 2's complement values needed by the left and right X DDA units for the appropriate quadrants each DDA unit handles.

The table is quite small - 32 entries deep by 4 bits wide. The delta values in the table are held as one bit integer and 3 bits fraction. From the hosts view the table is organised as four 32 bit words so the overhead of downloading when the point size changes is minimal. Only the parts of the table needed for a particular point size need to be loaded. The format of the points table is as follows:

		24		16		8		0
PointTable0	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0
PointTable1	P_{15}	P_{14}	P_{13}	P_{12}	P_{11}	P_{10}	P_9	P_8
PointTable2	P_{23}	P_{22}	P_{21}	P_{20}	P_{19}	P_{18}	P_{17}	P_{16}
PointTable3	P_{31}	P_{30}	P_{29}	P_{28}	P_{27}	P_{26}	P_{25}	P_{24}

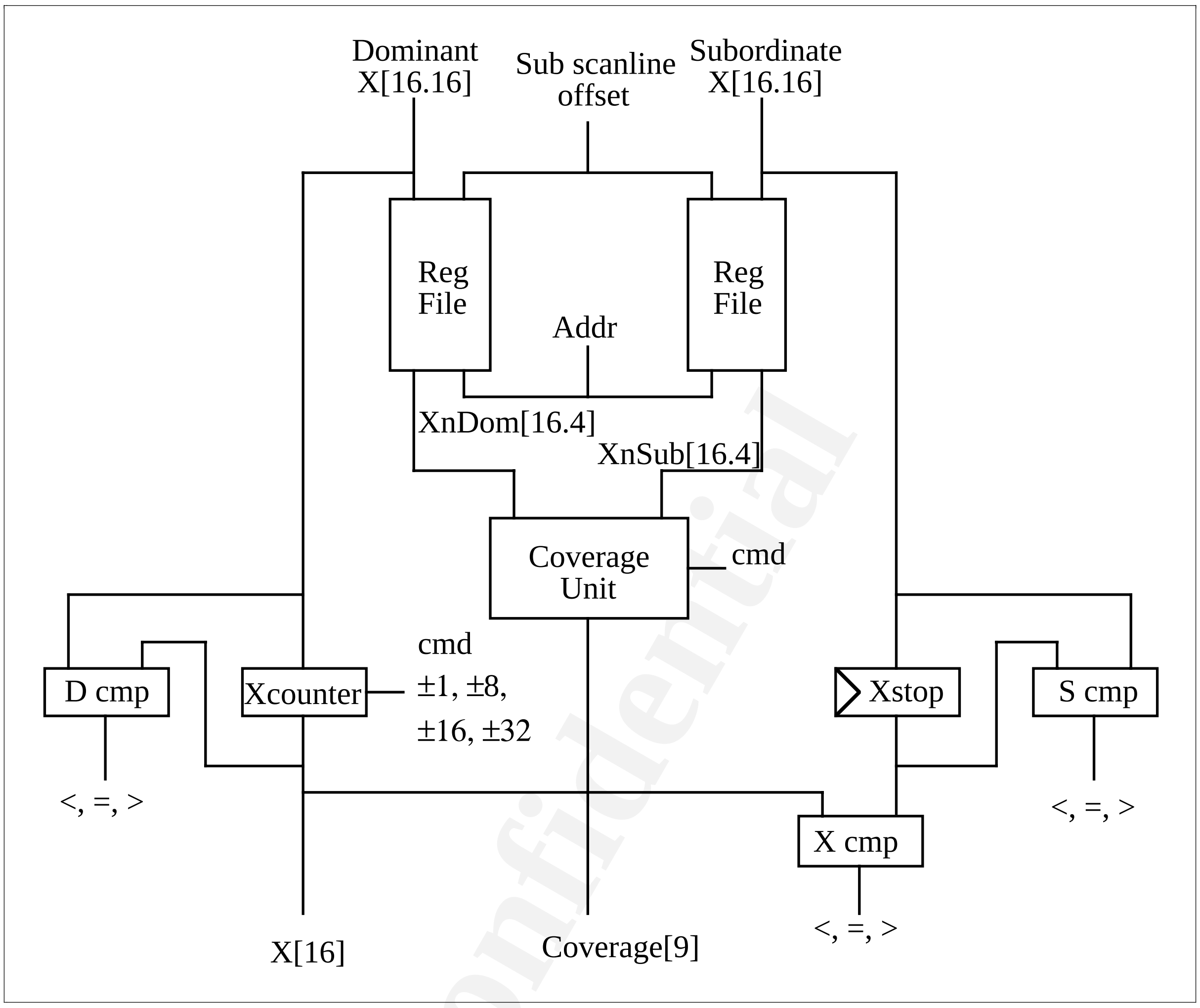
The block diagram of the Point Unit follows:



3.6.4. Span Fill Unit

The Span Fill Unit generates fragments from the dominant X coordinate to the subordinate X coordinates loaded into it.

The block diagram of the unit follows:



The operation of the Span Fill Unit depends on whether the trapezoid is being antialiased or not.

3.6.4.1. Aliased Operation

When a new pair of X value are loaded into the Span Fill Unit the X_{dom} value is routed directly into the counter (Xcounter). The X_{sub} value is loaded into the Xstop register and presented to the comparator (X cmp).

The values of X_{dom} (in Xcounter) is compared with X_{sub} (in Xstop register). The fragments which are generated are determined by the following table:

$X_{dom} > X_{sub}$	Here X_{dom} is to the right so the first fragment is generated as a PassiveXStep one, then all further fragments up to but not including X_{sub} are generated as ActiveXStep. The fragment at Xsub is generated by an ActiveStepYDomEdge which also takes the DDA units to the next scanline.
$X_{dom} < X_{sub}$	Here X_{dom} is to the left so all fragments up to but *not including* X_{sub} are generated by ActiveXStep. The next step is a PassiveStepYDomEdge wich takes the DDA units to the next scanline.
$X_{dom} = X_{sub}$	No fragments are produced. A PassiveStepYDomEdge is done wich takes the DDA units to the next scanline.

3.6.4.2. Antialiased Operation

When antialiasing is enabled the initial stages, are more complicated as we need to determine the direction of scan conversion, and the minimum and maximum X values to fill between.

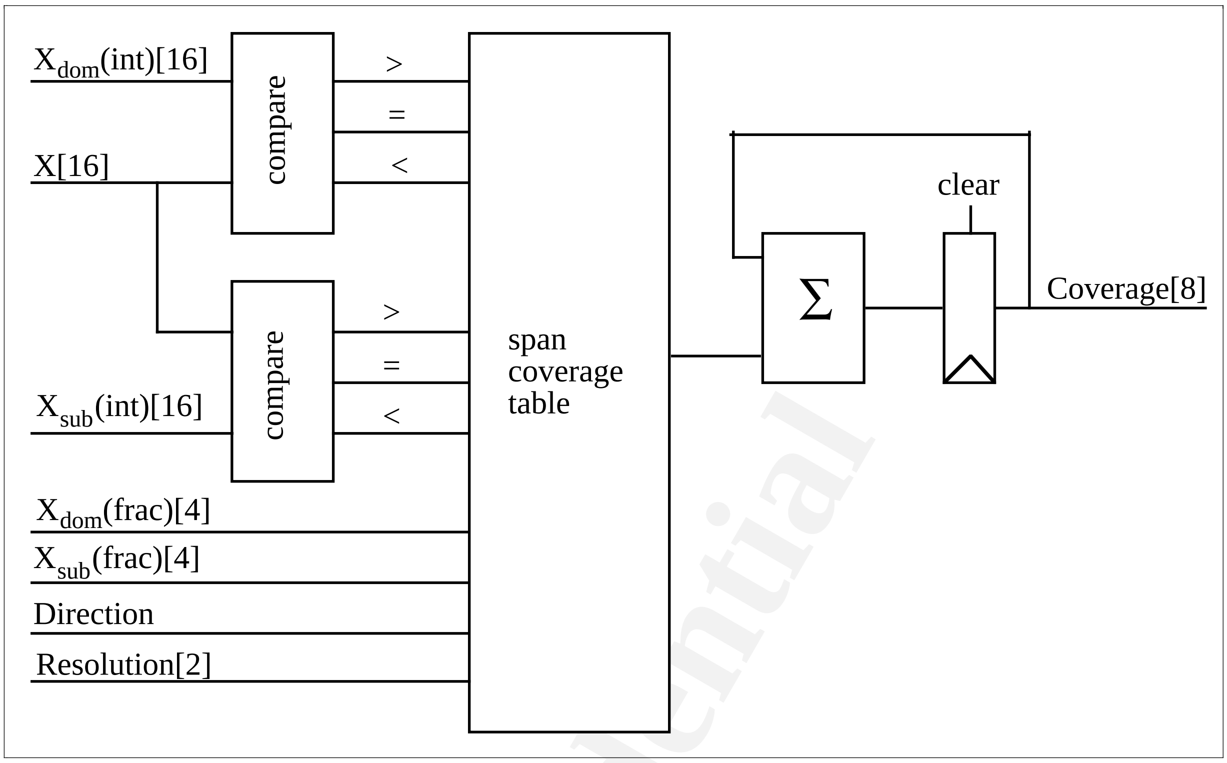
The first pair of X values are loaded into the Xcounter and Xstop register as for the aliased case. If these values are different then the scan direction is obvious so subsequent pairs of X values can be conditionally loaded into the Xcounter and Xstop as shown in the following table:

Direction	X_{dom} ? Xcounter	X_{sub} ? Xstop	Xcounter	Xstop
Left to Right	≤	≤	Load	Hold
	≤	>	Load	Hold
	>	≤	Hold	Load
	>	>	Hold	Load
Right to Left	≤	≤	Hold	Load
	≤	>	Hold	Load
	>	≤	Load	Hold
	>	>	Load	Hold

If the first pair of X values are the same, so the scanning direction is indeterminate, then both the Xcounter and Xstop registers are loaded for each subsequent sub scanline until they become different, and then the remaining sub scanlines (for this scanline) are processed as above.

To fill in the span each pixel between the minimum and maximum values inclusive is examined. For each pixel all the active sub pixel spans are tested against the pixels X coordinate and the coverage value incremented by the appropriate amount. The Coverage Unit calculates the sub span coverages and accumulates them.

The Coverage Unit is shown below:



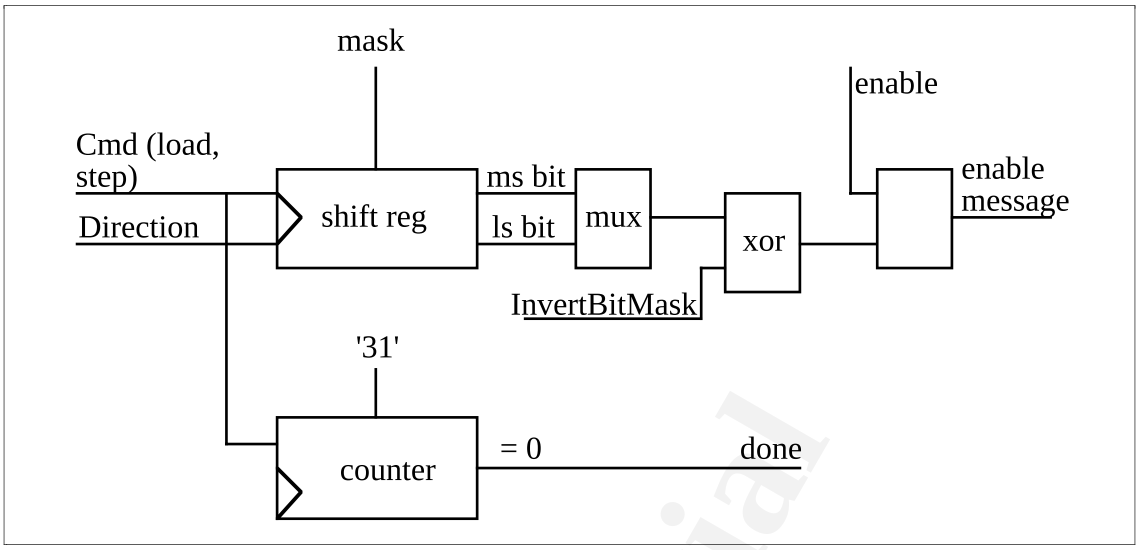
The coverage for to each sub scanline, at position X, is determined using the following table. Note the resolution (i.e. the number of sub pixel sample points in X) is set to 4 in the table. We may want to allow a range of resolutions to give a quality vs. speed trade off, however this should be balanced with how fast we can blend pixels in the Alpha Blend Unit.

Direction	int(X_{dom})	int(X_{sub})	Coverage
$X_{dom} < X_{sub}$ (left to right)	> X	Don't care	0
	Don't care	< X	0
	< X	> X	4 (full, say)
	= X	≠ X	4 - fract (X_{dom})
	≠ X	= X	fract (X_{sub})
	= X	= X	ABS (fract (X_{sub}) - fract (X_{dom}))
$X_{dom} \geq X_{sub}$ (right to left)	< X	Don't care	0
	Don't care	> X	0
	>X	< X	4 (full, say)
	= X	≠ X	fract (X_{dom})
	≠ X	= X	4 - fract (X_{sub})
	=X	= X	ABS (fract (X_{sub}) - fract (X_{dom}))

3.6.5. Bitmask Unit

The Bitmask Unit is used to disable fragment generation (or at least convert the fragment into a passive one) during the rasterisation process. This filtering process will work with all primitive types, however it is mainly intended to support character generation (X and OpenGL), large rectangle stipples (used by X as the 32x32 internal stipple pattern in too small in some cases) and X complex line stipples.

The block diagram of the bitmask unit is shown below:



The action on each fragment is controlled by a bitmask which is rotated by one bit per fragment. If all 32 bits in the mask register have been used before the primitive has been fully rasterised the rasterisation is suspended until a new mask in loaded in. If, while waiting for a new mask, we receive any other message then the bitmasks rasterisation is aborted[8]. This prevents a deadlock occurring when the host doesn't send enough mask data.

Bits in the mask are consumed from the least significant end towards the most significant end (MirrorBitMask = 0) or from the most significant end towards the least significant end (MirrorBitMask = 1). The bits in the mask are used in the order the fragments are generated in so bitmaps work in all four scanning directions. Obviously if the font bitmask has been defined with one scanning direction in mind then that scanning direction should be passed to the renderer (via the start and ±delta values).

This unit is used by OpenGL and X, and because it is so small it is probably worth duplicating the whole unit rather than implement some state save and restore mechanism.

3.6.6. Message Generation Unit

The Message Generation Unit, as the name suggests, generates the different rasteriser messages based on the coordinates it is given, plus other status signals. It also provides the synchronisation with the host when downloading image data and bitmasks.

Any potentially active walk messages only stay as active ones providing the following conditions are met.

1. The X coordinate must be ≥ 0.
2. The Y coordinate must be ≥ 0.
3. If SyncOnBitMask is set then the bit mask test must pass.
4. If AntialiasingEnabled is set and the coverage value > 0%.

If these conditions are not met then the equivalent passive message is generated instead.

[8]The behaviour is slightly different if the host is providing image data as well as the bitmask. This was covered earlier.

4. Scissor and Stipple Unit

4.1. Description

The scissor and stipple unit filters out fragments as they pass through the message stream. Each active fragment is subjected to four tests and must pass all enabled tests to proceed as an active fragment. If any of the tests are failed then the fragment is changed into a passive one. The four tests are:

1. A user scissor test. This compares the XY coordinate of the fragment against the upper and lower bounds of the scissor rectangle. The fragment must be within these limits to pass the test.

2. A screen scissor test. This test converts, if necessary, the fragment's coordinates to be screen relative and compares them against the screen limits. If the fragment lies within the screen then it passes the test, otherwise if fails.

3. A line stipple test. In this test the corresponding bit in a line stipple mask it tested. If the bit is zero then the test fails, otherwise it passes. The line stipple pattern is 16 bits in length and is scaled by a repeat factor, r, (in the range 1 - 256). The stipple mask bit, b, which controls the acceptance or rejection of a fragment is determined using

   b = (floor (s / r)) mod 16

   where s is a counter which is incremented for every fragment (normally along the line). This counter is reset to zero at the start of a polyline, but between segments it continues as if there were no break. Normally when b equals zero the least significant bit of the pattern is used. When the Mirror bit is set in LineStippleMode the stipple pattern is mirrored so that when b equals zero the most significant bit of the pattern is used.

4. An area stipple test. In this test the least significant bits of the fragment's XY coordinates index into a 2D stipple pattern. If the selected bit is set then the fragment passes the test, otherwise it fails. An offset is added to the XY coordinate and the result optionally mirrored before the stipple bit is accessed. The stipple bit can also be inverted before it is used.

4.1.1. OpenGL

In OpenGL the four tests will be configured as follows:

1. The user scissor test holds the intersection of the user defined scissor region (if any) and the window. The server hides the dual use from the user and gives the impression of exclusive use. Qualifying the user boundaries (if any) with those of the window is a useful optimisation to prevent accesses outside of the window from occurring. If they did occur they would be clipped out later by the GID test in the Local Buffer, but at the expense of unnecessary Local Buffer reads.

2. The screen scissor test is always enabled. All OpenGL fragments will be tested against this so if a window is moved partially off the screen the off screen (but in window) fragments are detected and prevented from causing erroneous memory accesses.

3. The line stipple test is used exclusively for aliased lines. Antialiased lines need to have the stipple pattern modify the pattern of rectangles used to draw the line. The line stipple test has two enables (which must both be set to enable the test). The first enable is held in a register within this unit which is loaded via the LineStippleMode message and is effective until changed by a new LineStippleMode message. The second enable is the

LineStippleEnable bit in the PrepareToRender[9] message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable line stippling when a primitive (i.e. polygon) must not have it enabled.

The mirror bit will be clear so the stipple is applied from the least significant bit towards the most significant bit.

Note that a repeat factor value loaded into the hardware must be one less than the actual repeat factor you want.

Wide lines, if drawn as multiple offset single lines, will have the line stipple reset for each single line. This can be achieved using the UpdateLineStippleCounters message or by setting the ResetLineStipple bit in the PrepareToRender message.

Polylines are easily drawn by resetting the line stipple at the start of the first segment and letting it carry on counting for the remaining segments. This will ensure the continuity of the stipple pattern along the length of the polyline.

For wide polylines there are two ways of managing the continuity of the stipple pattern. The first way is to draw multiple polylines offset to give the width, and resetting the line stipple at the start of each polyline. The second way is to draw each segment multiple times to build up the width before moving on to the next segment in the polyline. The first method has the advantage that the hardware will work with no additions, however the server will need to hold the entire polyline to repeat the rendering operation. This give a problem in that there is no limit on the number of segments in the polyline. The second method does not have this problem as each segment, once drawn can be discarded, however it does need the line stipple state to be saved and then restored at the start of each line within a segment to ensure the continuity of the stipple pattern. This is supported in the hardware using the SaveLineStippleState and UpdateLineStippleCounters messages.

4. The area stipple test is only used for polygon rendering and not Bitmaps or Pixel Rectangle operations in OpenGL. These last two cases still use the rasteriser to trace out the fragments so the area stipple mode must be temporarily disabled for these fragments (using the AreaStippleEnable bit in the PrepareToRender message).

   The area stipple test has two enables (which must both be set to enable the test). The first enable is held in a register within this unit which is loaded via the AreaStippleMode message and is effective until changed by a new AreaStippleMode message. The second enable is the AreaStippleEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable area stippling when a primitive must not have it enabled.

   The area stipple pattern is always 32x32 and is window relative. The least significant 5 bits of X and Y index the controlling bit in the stipple pattern.

   Antialiased polygons are stippled the same way as aliased polygons.

   The XY offsets are zero, the X and Y mirror bits are cleared so the least significant bit of the pattern is towards the left of the window, and the invert bit is cleared so a set bit in the pattern allows a fragment through.

[9]The PrepareToRender message is not normally issued by the host, but generated internally by the Rateriser Unit whenever a Render message is sent by the host. The data field associated with the Render message accompanies the PrepareToRender message.

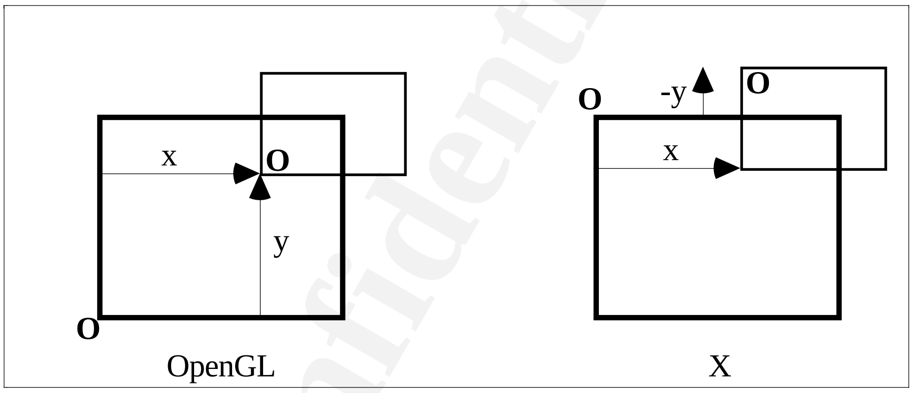
The area and line stipple tests are mutually exclusive in OpenGL, but the hardware does not enforce this.

4.1.2. X

In X the four tests will be used as follows:

1. The user scissor test is used by the X server to do window clipping when the simple clipping strategies fail. Each window is divided up into a number of visible tile rectangles and a primitive is rendered once per tile, with the scissor set up to reflect the tile's position and size.

2. The screen scissor test is always enabled, although all of X's needs are handled by the user scissor test (it doesn't cost anything and is an extra safety measure). Where the origin is important in determining the values of these offsets. This is demonstrated in the following diagram:



   The diagram also shows that the offsets can be negative.

3. The line stipple test is used by X for simple line stipple pattern. The more complex pattern, which can have arbitrary length, are handled in the rasteriser by the bitmask unit. The simple stipple patterns suited to this unit are where the mark space ratio of the pattern is equal and the repeat factor can be used to scale the length.

   The line stipple test has two enables (which must both be set to enable the test). The first enable is held in a register within this unit which is loaded via the LineStippleMode message and is effective until changed by a new LineStippleMode message. The second enable is the LineStippleEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received.

   The mirror bit will be set so the stipple is applied from the most significant bit towards the least significant bit.

   The line stipple can be reset by using the UpdateLineStippleCounters message or by setting the ResetLineStipple bit in the PrepareToRender message.

   Note that a repeat factor value loaded into the hardware must be one less than the actual repeat factor you want.

4. The area stipple test only supports a subset of the stipple patterns X can use. The stipple pattern can be any size, and screen or window relative. Fortunately most of the common stipple pattern are compatible with the area stipple test.

The area stipple test has two enables (which must both be set to enable the test). The first enable is held in a register within this unit which is loaded via the AreaStippleMode message and is effective until changed by a new AreaStippleMode message. The second enable is the AreaStippleEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable area stippling when a primitive must not have it enabled.

The area stipple pattern size can be 1, 2, 4, 8, 16 or 32 and can be different for the X and Y axis. This is selected in the AreaStippleMode register.

If the pattern size is greater than 32 bits then it must be handled in software, perhaps using the bitmask facilities in the Rasteriser Unit.

If the size is less than 32 and not a power of two then it must be handled in software, perhaps using the bitmask facilities in the Rasteriser Unit.

The XY offsets are zero for screen relative patterns and set as appropriate for window relative patterns. The X mirror bit is set so the most significant bit of the pattern is towards the left of the window, and the invert bit is cleared so a set bit in the pattern allows a fragment through.

The stipple pattern is fixed to the coordinate system provided (i.e. screen or widow relative, origin to the top or bottom left). If these coordinates are later interpreted as screen relative then the stipple pattern is screen relative as well. Similarly for window relative coordinates[10]. If the required stipple is relative to a different coordinate system to the one in effect then the XY offset can be set up to compensate for this.

The area and line stipple tests can both be used simultaneously in X.

4.2. Implementation

4.2.1. User Scissor Implementation

The user scissor test is implemented with 4 comparators and registers to test that:

$$\text{Xmin} \le X < \text{Xmax}$$

$$\text{and} \quad \text{Ymin} \le Y < \text{Ymax}$$

If any of these conditions are false and the test is enabled then the active walk message is converted into the corresponding passive walk message. Note that the input X and Y coordinates, and the scissor limits are all guaranteed to be positive.

Each context maintains its own ScissorMinXY and ScissorMaxXY registers, but can share the comparators.

4.2.2. Screen Scissor Implementation

The screen scissor test compares the *screen* relative coordinates of a fragments against the screen's boundaries. The screen boundaries are (0, 0) to (width - 1, height - 1) inclusive. The first step is to convert the window relative coordinate to a screen relative one. If the resulting

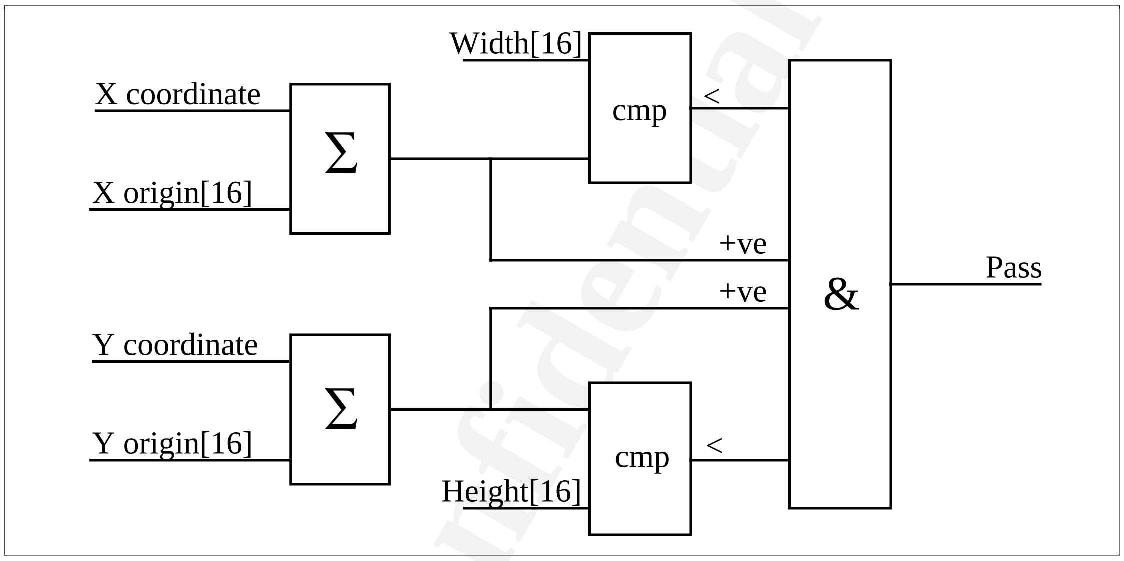
[10]When doing window management activities X will be using screen relative addressing, but it may be advantageous to use window relative addressing when rendering into windows. It is easy for X to switch between the two.

screen coordinate is positive and less than the (width, height) then the fragment is on screen and the test passes, otherwise the test fails and the fragment is converted into a passive one. Calculating the screen coordinate is easily done by adding the window's screen origin to fragment's coordinate. The window's origin (in screen coordinates) will be negative if the origin of the window is off screen.

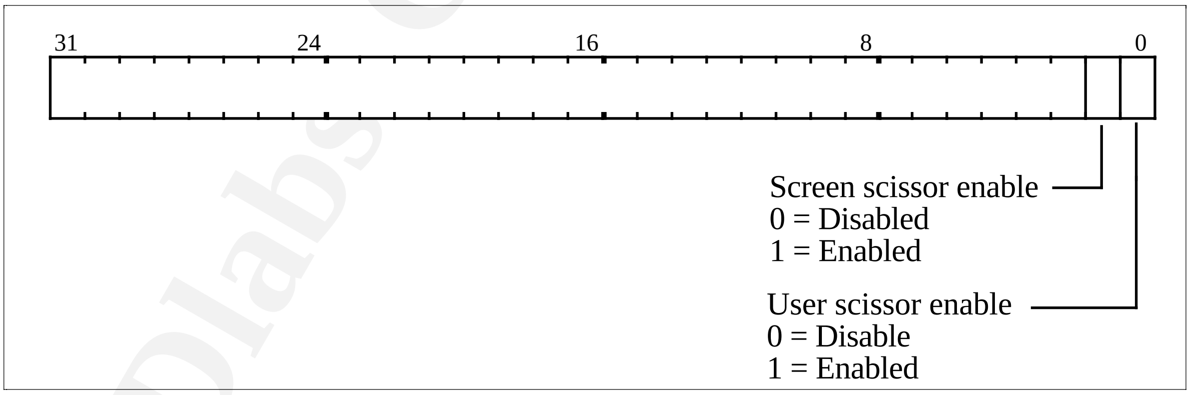
If the coordinates are already in the screen coordinate space then the window's screen origin is set to (0, 0).

The screen width and height are constant and only need to be initialised once.

The block diagram for this operation follows:



The screen and user scissor tests are controlled by the ScissorMode message, the data field of which follows.



4.2.3. Line Stipple Implementation

The line stipple is governed by the equation:

b = (floor (s / r)) mod 16

where b is the bit in the stipple mask to use.

The cost of implementing this equation is prohibitive (because of the modulo operation) however an incremental solution is possible using counters.

There are two counters:

A 4 bit counter (bit_counter) to select which bit in the 16 bit long stipple mask to use. A value of zero selects the least significant bit when the mirror bit is clear and bit 15 when the mirror bit is set. This counter can be incremented, reset to zero, or loaded from the segment register.

A nine bit counter (repeat_counter) to count the current repeat factor level. This counter can be incremented, reset to zero, or loaded from the segment register.

The sequence of events, when line stipple is enabled (by both the LineStippleEnable in the PrepareToRender message and enable in the LineStippleMode message), is as follows:

1. On receiving a walk message the bit in the mask selected by the bit_counter is tested. If the bit is set then the walk message is passed on unchanged, otherwise it is converted into an equivalent passive message.

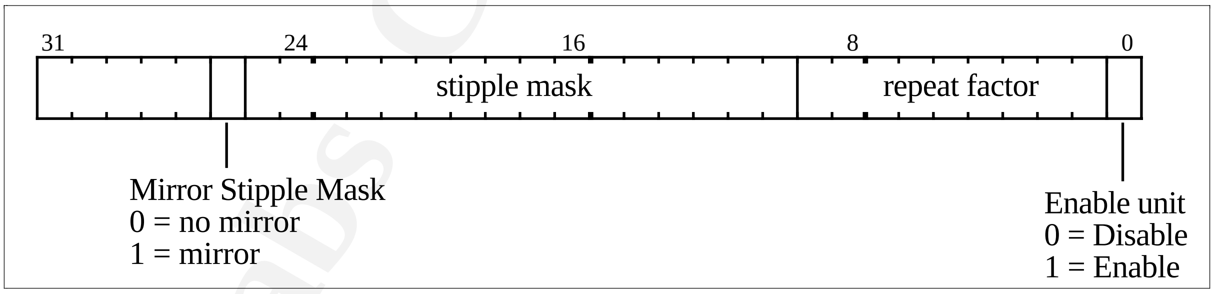
2. The repeat_counter is then incremented and compared against the repeat factor. If they are equal then the repeat_count is reset to zero and the bit_counter is incremented.

3. A UpdateLineStippleCounters message causes either the counters to be reset or loaded from the segment registers.

4. A SaveLineStippleState message causes the segment registers to be loaded with the current repeat_count and bit_count values.

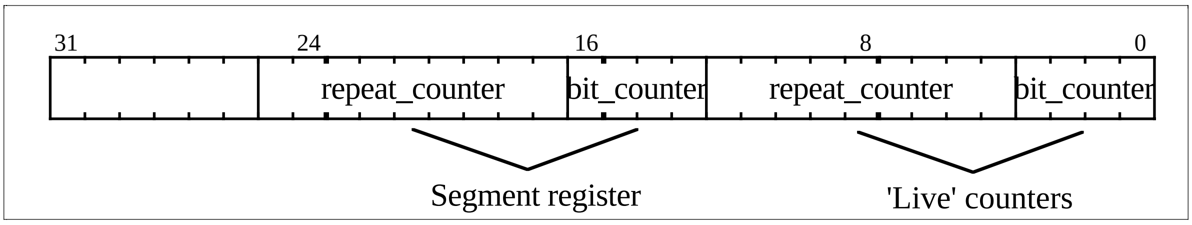
5. A PrepareToRender message with the ResetLineStipple bit set causes the counters to be reset.

The format of the LineStippleMode message is as follows:



Note that the value stored as the repeat factor is one less than the desired repeat factor.

In order to be able to task switch GLiNT it is necessary to be able to read and restore the state of the stipple counters and the segment register. The reading of these values is via the 'side ways' read back path and the restoring is by the LoadStippleCounters message. The data field holds the data to restore and its format is:



4.2.4. Area Stipple Implementation

The address of the bit in the stipple pattern to use in the test is calculated as follows:

- Add the Xoffset and Yoffset to the bottom five bits of X and Y coordinates of the fragment respectively. Ignore any overflow.
- If the corresponding mirror bits are set then invert the X and Y addresses.
- Extract the bottom n,m bits of the resulting X and Y values where n and m are determined by the X Sel and Y Sel fields respectively. The extracted address fields are zero extended to 5 bits where necessary and are now called X' and Y' respectively.

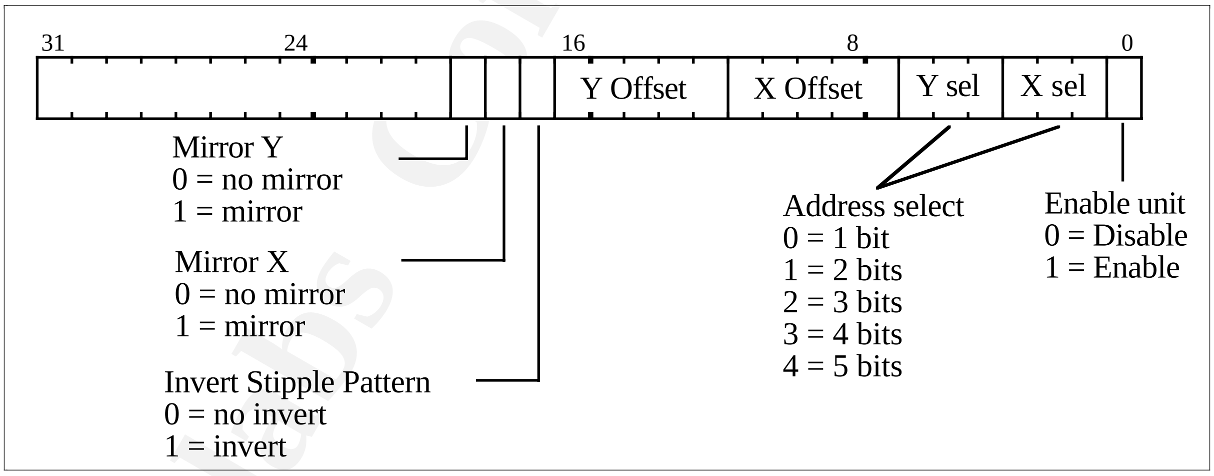
The Y' field selects the row in the pattern RAM (row zero is at AreaStipplePattern[0]) and X' selects the column bit (the least significant column bit is 0).

If area stipple is enabled and the invert pattern bit is 0 then if the selected pattern bit is zero the step message is converted into an equivalent passive message, otherwise the step message is passed unchanged.

If area stipple is enabled and the invert pattern bit is 1 then if the selected pattern bit is one the step message is converted into an equivalent passive message, otherwise the step message is passed unchanged.

Both the AreaStippleEnable in the PrepareToRender message and enable in the AreaStippleMode message must be set to enable the area stipple test.

The format of the AreaStippleMode message is as follows:



4.3. Input Messages

4.3.1. External Messages

Scissor and Stipple Register Update Messages		
Tag Mnemonic	Data Field	Description
ScissorMinXY	Current (X, Y) X in ls 16 bits, Y in next 16 bits	Rectangle corner closest to the screen origin.

ScissorMaxXY	Current (X, Y) X in ls 16 bits, Y in next 16 bits	Rectangle corner farthest from the screen origin.
ScissorMode	see above	
ScreenSize	Width in ls 16 bits. Height in ms 16 bits	Screen dimensions. This register is shared by both the X and OpenGL contexts.
WindowOrigin	X origin in ls 16 bits. Y origin in ms 16 bits.	
AreaStipplePattern[32]	32 bit row pattern	This message is decoded to select which row in the pattern RAM to update from the data field.
AreaStippleMode	see above	General control
LineStippleMode	see above	General control
LoadLineStippleCounters	see above	Used to restore the line stipple counters and segment register after a task switch. The counters are incremented during a line stipple so the value read from them, via the readback path may not match the value loaded in to them using this message.

Line Stipple Control Messages

Tag Mnemonic	Data Field	Description
UpdateLineStippleCounters	0 = Reset counters to zero. 1 = Load counters from segment register.	Counter control
SaveLineStippleStateCounters	not used	Copies the current counter values into the segment register.

4.3.2. Internal Messages

This unit reacts to all the messages in the Rasteriser Walk message group. See the rasteriser section for details on these messages.

4.4. Output Messages

No new messages generated, although an active walk message can be changed into an equivalent passive one.

4.5. Behavioural Model

```
Wait for input message
{
  switch (on input message)
  {
    case RegisterUpdateGroup:
          Update the selected register;
          Flush the input message;
          break;
    case PrepareToRender:
          Save LineStippleEnable and AreaStippleEnable bits;
          if (line stipple enabled and ResetLineStipple set)
          {
            Reset the line stipple counters;
          }
          Pass message through to the next unit;
          break;
    case ActiveWalkGroup:
          User Scissor test the given XY coordinate;
          Screen Scissor test the given XY coordinate;
          Line stipple test;
          Area stipple test the given XY coordinate;
          Wait for room in receiving queue;
          if (all four test pass)
          {
            Send walk message through;
          }
          else
          {
            Convert walk message to be passive;
            Send walk message through;
          }
          if (line stipple test enabled)
          {
            Increment the counters;
          }
          break;
    case UpdateLineStippleCounters:
          if (data field == 0)
          {
            Reset the counters;
          }
          else
          {
            Load the counters from the segment register;
          }
          Flush the input message;
          break;
    case SaveLineStippleState:
          Load the segment register from the counters;
          Flush the input message;
          break;
    default:
          Pass the input message to the next block;
          break;
  }
}
```

5. Colour DDA Unit

5.1. Description

The Colour DDA unit is responsible for generating the colour information (red, green, blue and alpha) components under the direction of the Step messages from the Rasteriser Unit. Sub pixel corrections can be applied to correct for an initial start error on a span.

All four components are always generated even though in Colour Index mode only one is needed. The colour index value is assumed to be in the least significant byte of the Colour message by units down stream.

The output of this unit is the Colour message, which is normally generated on an active step. In normal operation this causes two messages per active fragment in the message stream. This will limit the maximum through put of visible fragments to 25M per second, although it may be reduced further by other units. In some 3D[11] rendering operations (e.g. flat shading), where the internal colour is constant, it is advantageous not to send the colour with every active step message to increase the potential fragment through put rate. One way to do this is to disable the Colour DDA Unit and have the host send down the Colour message itself, but there are some caveats:

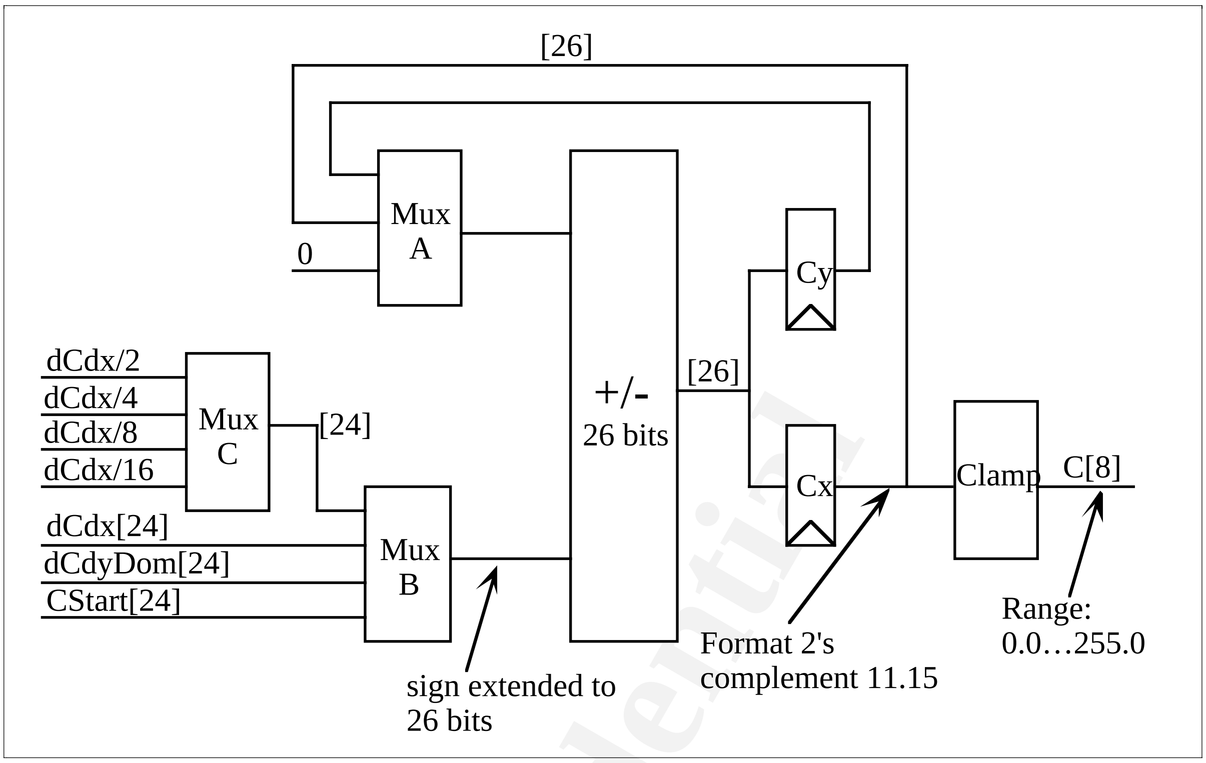
- Any mode changes must be done before the Colour message is sent. In the normal operation this is enforced by Colour messages only starting once raterisation is under way, and mode changes cannot occur while rasterising. Several units hold temporary copies of the Colour message and pre-process the value depending on their mode settings. A good rule is send the Colour message as the last message before the Render message (it can be sent before any message in the rateriser group so the indexed tag mode can be used to group the rasteriser messages with the Render message).

- The Colour message should be viewed a temporary state which cannot be saved and restored. As a Colour message flows down the pipeline the colour value is changed in various units so the temporary colour registers in the Texture/Fog Colour Unit, Alpha Blend Unit and Dither Unit may all hold different colour values.

- Some mode changes will invalidate the colour so it is best to resend it for every primitive. For example if the Colour message is sent when the Texture/Fog Colour Unit is enabled and on the next primitive this unit is disabled the actual colour which will be used is the last one the Texture/Fog Colour Unit calculated and not the one the host sent.

The block diagram of one of the component DDAs (C represents R, G, B or A) is:

[11]For the 2D case there are additional mechanisms to save message bandwidth by not sending a Colour message along with active steps. Here the preferred way is to send an FBWriteData message when no logical ops are used, and just use the Colour message when logical ops are.



Some notes on the diagram:

- When the shading mode is Flat the colour value is taken from the ConstantColour register and not from the DDA unit. This register holds the 32 bit RGBA value (R is in the least significant byte) or the raw 32 bit framebuffer data depending on how the down stream units are set up.

- The adder/subtracter normally works as an adder except when sub pixel compensation is being applied where it can work in either mode depending on the direction of the correction. At first sight an additive correction is all that is necessary because we always correct in the direction that the dx derivative is set up. However the geometry of the primitive has been biased by *nearly a half* (of a pixel) to make the rasterisation rules simpler to implement, while the interpolated parameters are set up with respect to the unbiased geometry.

- The nomenclature in the diagram is as follows:

dCdx	change in component C for unit change in X
dCdyDom	change in component C for unit change in Y along the dominant edge of a trapezoid, or along a line
CStart	start value of component C

- The Cx register holds the colour component as a span is filled. The Cy register holds the value of the colour component as we step up the dominant edge of the primitive and is used to seed the span interpolation.

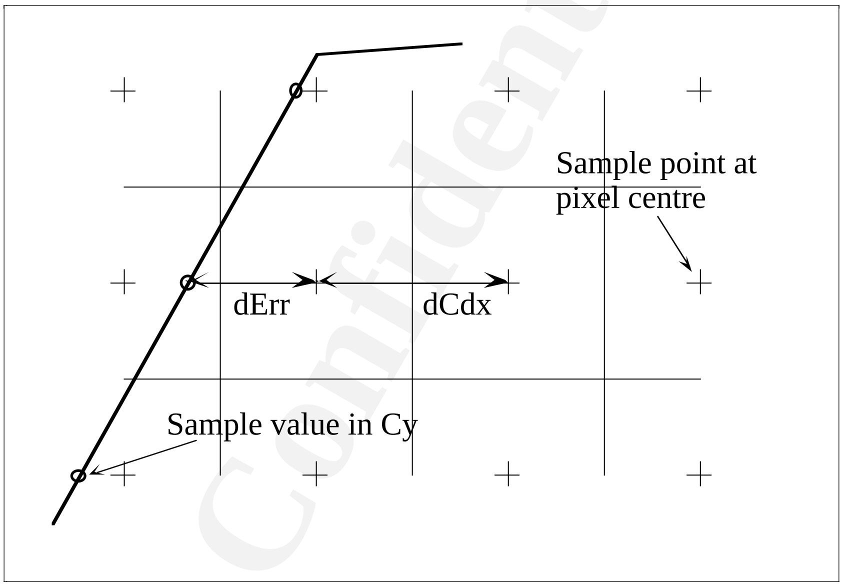
- The start and derivative number format is 2's complement fixed point integer: 9 bits integer and 15 bits fraction. The internal format has an additional 2 bits to extend the range to give more protection against over or underflow. This guard band is only designed to cope with small amounts of overflow (or underflow) when one or two extra deltas are added during rasterisation, usually as a result of the sample point (i.e. pixel centre) being outside of the boundary of the primitive. If the additional range provided by the guard band is exceeded then wrap around will occur and an abrupt change in the colour will be visible (the clamp logic cannot prevent this). This situation will only occur if the delta values are set up incorrectly for the number of iterations the DDA works through, i.e. the gradient is specified as being steeper than it really is.

- The Clamp logic prevents the interpolated value wrapping around (when the colour component is extracted) when overflowing or underflowing occurs during interpolation.

  This should not be necessary if the derivatives, etc. are set up correctly and sub pixel corrections are done at the start of every span. However we are working with finite precision numbers so slight rounding errors will occur.

  The clamp logic is simply a multiplexer with selects either the result, zero or 255.0 based on comparing the result against 255.0 and the sign bit.

- Multiplexer C is used to implement the sub pixel correction. The need for the sub pixel correction is easily shown in the following diagram:



The start value has already been compensated to move it to the centre of a pixel in the Y directions. This only needs to be done once per triangle and is done in software. As the DDA is stepped in Y along the dominant edge the value of the interpolated parameter is calculated at the points marked by circles, whereas the value is required in the pixel's centre.

If dErr is the distance the edge is away from the pixel's centre (must be < 1) and dCdx is the change in C for unit change in x then the correct value at the first sample point is:

$$Cx = Cy + dErr * dCdx$$

The distance dErr is sent by the rasteriser in the SubPixelCorrection message and whenever this message is received the above equation is implemented. The SubPixelCorrection message allows for 4 bits of resolution of dErr. The correction is applied to a resolution of 1/16 of the dx derivative. The multiply operation is too expensive to do using a multiplier so it is implemented by addition of the partial products. For a positive correction this means adding in dCdx/2 when bit 3 is set, dCdx/4 when bit 2 is set, dCdx/8 when bit 1 is set and dCdx/16 when bit 0 is set. For a negative correction the corresponding partial products are subtracted.

The value of dErr is in sign magnitude format with bit 4 holding the sign (0 = positive), and bit 3 representing 2^{-1}, bit 2 representing 2^{-2}, etc..

The division of dCdx is always done by a power of two so is just an arithmetic right shift (to retain the sign of the dCdx).

The correction will add one cycle for every bit set in dErr so worst case will take 4 cycles per scanline, but on average 2 cycles. This time is hidden under the page break time allowed for every scanline so will not impact performance.

5.2. Input Messages

5.2.1. External Messages

Colour Register Update Messages		
Tag Mnemonic	Data Field	Description
RStart	24 bit 2's comp fix pt	Red start value
dRdx	24 bit 2's comp fix pt	Red derivative unit X
dRdyDom	24 bit 2's comp fix pt	Red derivative unit Y, dominant edge
GStart	24 bit 2's comp fix pt	Green start value
dGdx	24 bit 2's comp fix pt	Green derivative unit X
dGdyDom	24 bit 2's comp fix pt	Green derivative unit Y, dominant edge
BStart	24 bit 2's comp fix pt	Blue start value
dBdx	24 bit 2's comp fix pt	Blue derivative unit X
dBdyDom	24 bit 2's comp fix pt	Blue derivative unit Y, dominant edge
AStart	24 bit 2's comp fix pt	Alpha start value
dAdx	24 bit 2's comp fix pt	Alpha derivative unit X
dAdyDom	24 bit 2's comp fix pt	Alpha derivative unit Y, dominant edge
ConstantColour	32 bit value	Holds either an encoded colour (RGBA) or the raw framebuffer data value.
ColourDDAMode	Bit 0: 0 = Disable 1 = Enable Bit 1: 0 = Flat 1 = Gouraud	Misc control fields

These messages are decoded and cause the corresponding register to be updated - they behave just like a synchronous memory write cycle.

None of these messages are passed on to the next unit.

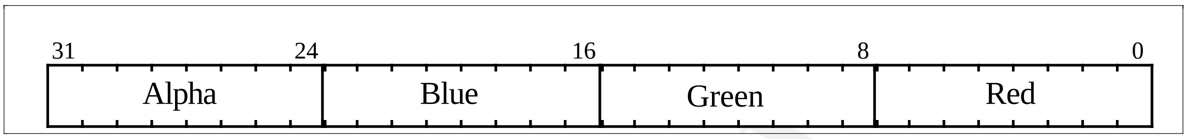
5.2.2. Internal Messages

The active step messages cause the colour message to be produced and then update the colour component DDAs as directed, if the unit is enabled. The SubPixelCorrection message from the rasteriser is also used if the unit is enabled.

The passive step messages just update the colour component DDAs as directed, if the unit is enabled.

5.3. Output Messages

Colour Messages		
Tag Mnemonic	Data Field	Description
Colour	See below	



5.4. Behavioural Model

```
Wait for input message
{
  if (input message is in the RegisterUpdateGroup)
  {
    Update the identified register;
    Flush the input message;
  }
  else
  {
    if (unit is enabled)
    {
      switch (on input message type)
      {
        case SubPixelCorrection:
                if (shading is Gouraud)
                {
                  if (sign of dxErr is positive)
                    For each bit set in dxErr add the
                      corresponding fraction of dCdx to Cx
                      and update Cx.
                  else
                    For each bit set in dxErr subtract the
                      corresponding fraction of dCdx from Cx
                      and update Cx.
                }
                break;
        case ActiveStep:
                Wait for room in receiving message queue;
                Send Colour message to receiving queue;
                if (shading is Gouraud)
                  Update DDA based on Step type;
                break;
        case PassiveStep:
                if (shading is Gouraud)
                  Update DDA based on Step type;
                break;
        case PrepareToRender:
                Update DDA from the *Start registers
                break;
      }
    }
    Wait for room in receiving message queue;
    Copy input message to receiving queue;
  }
}
```

DDA Update Controls					
Message	Shading	Mux A output	Mux B output	Register Cx	Register Cy
PrepareToRender	Gouraud	zero	Cstart	Load	Load
StepX	Gouraud	Cx	dCdx	Load	Hold
StepYDomEdge	Gouraud	Cy	dCdyDom	Load	Load
SubPixelCorrection	Gouraud	Cx	Mux C o/p	Load	Hold

6. Texture/Fog Colour Unit

6.1. General Description

This unit does the colour processing for Texture and Fog operations:

- The texture operations are done in two stages: first the texels (input texture values from the texture maps) are filtered and second the resultant texture value is used to modify the fragment's colour.

  The filtering takes from 1 to 8 texels and uses nearest neighbour and/or linear, bilinear or trilinear interpolation to calculate the final value. The calculations are done on 1 to 4 components in the texels. Different filtering operations can be specified for minification and magnification.

  The filtered value is combined with the colour (from the colour message) using blend, modulate or decal modes to give a new colour value which is passed on for any fog calculations.

  If the texturing is disabled (either from the mode register or from the Texture Enable bit in the PrepareToRender message) then the colour is passed on to the fog calculations unchanged. All the calculations are initiated by an active step message.

- The fog operation takes the colour value, after texture processing if necessary, and interpolates between the colour value and a fog colour (from the FogColour message) using an internally generated interpolation coefficient. The interpolation coefficient is calculated on a per fragment basis using a DDA unit. The DDA unit tracks the active and passive step.

  If the fog is disabled (either from the mode register or from the Fog Enable bit in the PrepareToRender message) then the colour is passed on to the next unit unchanged. All the calculations are initiated by an active step message.

The texture colour and fog operations have been combined in this unit as they both need the same linear interpolation hardware, and naturally follow each other in the OpenGL fragment processing sequence.

6.2. Texture Processing

6.2.1. Filter Operations

The OpenGL filter operations are:

Minification	Nearest
	Linear
	NearestMipMapNearest
	NearestMipMapLinear
	LinearMipMapNearest
	LinearMipMapLinear
Magnification	Nearest
	Linear

The choice of the minification and magnification filtering methods are held constant while rendering a primitive. The selection between using the minification filter or the magnification filter is done on a fragment by fragment basis depending on the projected area of the fragment on to the texture map. The TextureFilter message indicates which filter method to use. Obviously if the filter method is the same for both minification and magnification, or doesn't change between adjacent fragments[12] then this message is not needed for every fragment.

The filtering operations above are further qualified by the type of primitive the fragment originates from and this is factored in to the TextureFilter message.

In the case of lines all the filtering options are valid, however lines are one dimensional entities in mathematics, even though they can have width in OpenGL. When the filter operation includes a linear term two texels are combined in proportions given by the interpolation coefficient, i.e. linear interpolation.

In the case of trapezoids all the filtering options are valid and trapezoids have true area. When the filter operation includes a linear term four texels are combined in proportions given by two interpolation coefficients, i.e. bilinear interpolation.

Mip Mapping is a technique to allow the efficient filtering of texture maps when the projected area of the fragment covers more than one texel. A hierarchy of texture maps is held with each one being half the size (or one quarter the area) of the preceding one and the pair of maps to use is governed by the projected area. In terms of filtering this means that three filter operations are done: one on the first map, one on the second map and one between the maps. The first filter name (Nearest or Linear) in the …MipMap… specifies the filtering to do on the two maps, and the second filter name to filtering to do between maps.

The data to do the filtering is sent to this unit in the Texel[0…7] and Interp[0…4] messages. The messages (or values) used in the different filter operations are shown below (Texels are $T_0 \ldots T_7$ and Interps are $I_0 \ldots I_4$):

Primitive	Filter	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	I_0	I_1	I_2	I_3	I_4
Lines	Nearest	•												
	Linear	•	•							•				
	NearestMMNearest	•												
	NearestMMLinear	•	•							•				
	LinearMMNearest	•	•							•				
	LinearMNLinear	•	•	•	•					•	•	•		
Trapezoid	Nearest	•												
	Linear	•	•	•	•					•	•			
	NearestMMNearest	•												
	NearestMMLinear	•	•							•				
	LinearMMNearest	•	•	•	•					•	•			
	LinearMNLinear	•	•	•	•	•	•	•	•	•	•	•	•	•

The filtering equations are described now. In these equations

- R is the result of the filter operation,

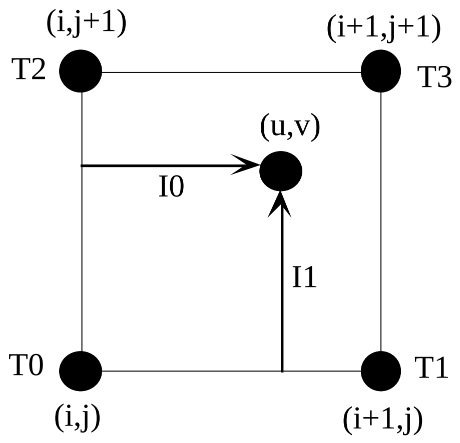
- Lerp(A, B, α) linearly interpolates between A and B using α as the interpolation coefficient. This is defined as:

$$Lerp(A, B, \alpha) = (1 - \alpha)A + \alpha B$$

[12]In general it will not change between fragments because the projected area exhibits a fair degree of coherence between fragments on the same scanline.

Primitive	OpenGL Filter name	Equation	Filter type
	Nearest	$R = T_0$	Nearest
	Linear	$R = Lerp(T_0, T_1, I_0)$	Linear2
Lines	NearestMMNearest	$R = T_0$	Nearest
	NearestMMLinear	$R = Lerp(T_0, T_1, I_0)$	Linear2
	LinearMMNearest	$R = Lerp(T_0, T_1, I_0)$	Linear2
	LinearMNLinear	$R_0 = Lerp(T_0, T_1, I_0) \quad R_1 = Lerp(T_2, T_3, I_1)$ $R = Lerp(R_0, R_1, I_2)$	Trilinear 4
	Nearest	$R = T_0$	Nearest
	Linear	$R_0 = Lerp(T_0, T_1, I_0) \quad R_1 = Lerp(T_2, T_3, I_0)$ $R = Lerp(R_0, R_1, I_1)$	Linear4
Trapezoid	NearestMMNearest	$R = T_0$	Nearest
	NearestMMLinear	$R = Lerp(T_0, T_1, I_0)$	Linear2
	LinearMMNearest	$R_0 = Lerp(T_0, T_1, I_0) \quad R_1 = Lerp(T_2, T_3, I_0)$ $R = Lerp(R_0, R_1, I_1)$	Linear4
	LinearMNLinear	$R_0 = Lerp(T_0, T_1, I_0) \quad R_1 = Lerp(T_2, T_3, I_0)$ $R_2 = Lerp(R_0, R_1, I_1)$ $R_0 = Lerp(T_4, T_5, I_2) \quad R_1 = Lerp(T_6, T_7, I_2)$ $R_3 = Lerp(R_0, R_1, I_3)$ $R = Lerp(R_2, R_3, I_4)$	Trilinear 8

The relationship between the texels and interpolants are show in the following diagram:



These equations are applied once for each component in the texture map. There can be 1, 2, 3 or 4 components in the texture map but as the Lerp hardware is replicated for each component this unit always works on 4 components. The same interpolation coefficient is applied to all components. It is the responsibility of the Texture Read Unit to expand fewer components into the 4 component case. This conversion is done as follows (L is luminance):

Number of comps	Descriptio n	R channel	G channel	B channel	A channel
1	L	L	L	L	1.0
2	LA	L	L	L	A
3	RGB	R	G	B	1.0
4	RGBA	R	G	B	A

6.2.2. Applying the texture value

Once the texture value, $R_,$ has been calculated it is used in one of three ways: Modulate, Decal or Blend. The equations for each of these are:

Modulate

$$C_{rgba} = R_{rgba} * F_{rgba}$$

Decal

$$C_{rgb} = Lerp(F_{rgb}, R_{rgb}, R_a)$$

$$C_a = F_a = Lerp(F_a, R_a, 0.0)$$

Blend

$$C_{rgb} = Lerp(F_{rgb}, K_{rgb}, R_{rgb})$$

$$C_a = F_a * R_a$$

where C is the final colour after texture has been applied, F is the fragment colour (in a Colour message), R is the value calculated from the filtering of texels and K is a constant colour stored in a register locally (loaded by the TextureEnvColour message). The equations are executed on the four colour components in parallel and the suffixes show how the different component values are combined.

6.2.3. Implementation

The general idea is that when an active step message is received the necessary calculations are undertaken. In this respect this unit is no different to the other units (this is the normal mode of operation). Where this unit differs from the rest (excluding the other texture units) is that in the worst case it will take approximately 25 cycles to calculate the result message (a Colour message) and require 16 input messages. While the calculation is in progress the input messages will be blocked (to prevent a value waiting to be used from being over written). Thus it will take an additional 16 cycles to get them in to the registers and the upstream units are likely to be blocked as well. Hence the overall parallelism in the system is reduced leading to less performance. There are a number of ways the message input cycles can be hidden under the calculations and the blocking of upstream units prevented:

- Double buffering. The critical information (the Texel[0…7], Interp[0…4], Colour and TextureFilter) can be double buffered and on an ActiveStep message the buffers swapped.

- Ganged registers. The critical information can be transferred into internal registers in parallel on an ActiveStep message, leaving the normal message registers to be updated. This is easier to control than double buffering.

- Score boarding. Each of the critical registers has an associated bit in a scoreboard register. On an ActiveStep message all the registers needed for the calculations have their score board bit set to mark it busy. Once a register has been finished with the corresponding score board bit is cleared to mark it free. A register can only updated by a message if it is free, otherwise the message is blocked until the destination register becomes free.

The first two methods duplicate the storage elements for the critical data so will add another 304 bits (8 * 32 + 5 * 9 + 3) of storage, or approx 3000 gates. The messages can arrive in any order.

The third method adds 14 bit of storage in the score board register but has slightly more complicated control. One other requirement for this method to work well is that the order of message arrival is matched by the order the registers are freed (assuming this unit is the limiting factor and not the message rate). The Texel, Interp and TextureFilter messages can arrive in an appropriate order from the Texture Address and Texture Read Units[13]. The Colour message is a bit more of a problem. It arrives first normally and is used last so this would block the message stream. The Colour message could be handled differently using the ganged register method or the unit ordering can be changed from:

‡ Colour DDA ‡ Texture Addr ‡ Texture Read ‡ Texture/Fog Colour ‡

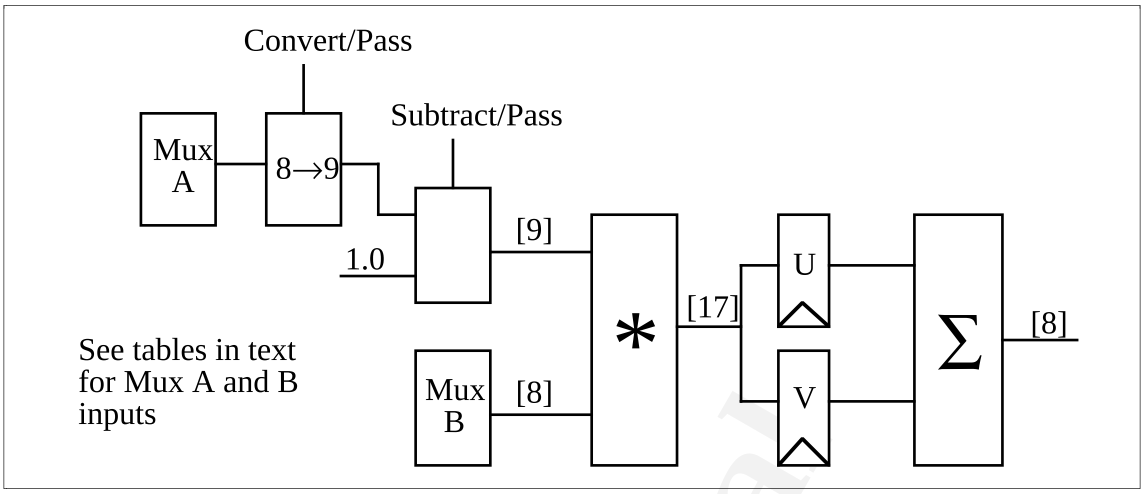
to:

‡ Texture Addr ‡ Texture Read ‡ Colour DDA ‡ Texture/Fog Colour ‡

The change in ordering is more gate efficient. This change in order doesn't need to be exposed in user visible documentation as the logical functioning of the chip hasn't changed.

The following diagram shows the configuration of one of the processing channels used for the interpolation, texture application and fog colour calculations. This is repeated identically for each of the four components and controlled in an identical fashion.

[13]The order from the host in the case of GLiNT1 is unlikely to be important as this unit will be processing the data much faster than the host can calculate it part. If DMA is used then this will still be true at the system level as the host still needs to prepare the next DMA buffer.



The 8→9 block will convert an 8 bit number to a 9 bit one or just pass a 9 bit one through unchanged. This is only needed when the result of the filter operation is used as the interpolation coefficient.

Conversion from the 8 bit input format to the 9 bit output format is done using the equations:

$$output = \frac{256}{255} input$$

which, in the limited precision we have available, translates into:

output = input for $0 \leq$ input < 255
output = 256 for input = 255

The discontinuity in the normally monotonic number set is inevitable unless the number of fractional bits is increased.

The subtract/pass block either passes its input through unchanged or outputs (1.0 - input).

The 9 bit input to the unsigned multiplier is formatted as 1 bit integer and 8 bits fraction and has a legal range of 0.0 … 1.0 inclusive.

Three temporary registers (32 bits wide) are needed to hold the intermediate results. These are called R0…R2. When these are used as the source of data for the for multiplier 'A' (via the A mux) port they will need to be converted to 9 bit numbers.

The following tables shows the input to the A multiplexer for each of the colour channels. The subscripts identify the RGBA components where necessary. Things have been arranged so that the per component multiplexers are switched to the same line and the 8→9 conversion and subtraction are enabled the same for component on a line. Note that 255 is an eight bit quality to match the size of $R0_a$ rather than being defined as 1.0.

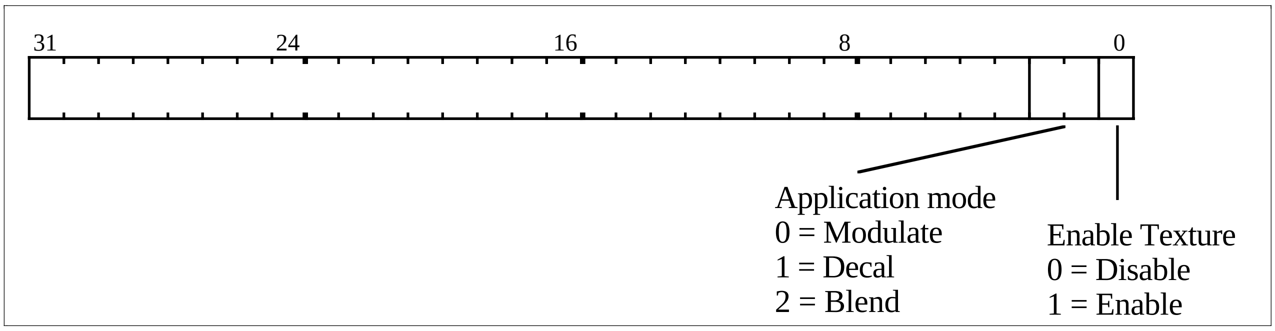
	R	G	B	A
Interpolation	Interp0	Interp0	Interp0	Interp0
	Interp1	Interp1	Interp1	Interp1
	Interp2	Interp2	Interp2	Interp2
	Interp3	Interp3	Interp3	Interp3
	Interp4	Interp4	Interp4	Interp4
Modulation	$R0_r$	$R0_g$	$R0_b$	$R0_a$
Blend	$R0_r$	$R0_g$	$R0_b$	255
Decal	$R0_a$	$R0_a$	$R0_a$	0
Fog (CI)	1.0	1.0	1.0	1.0
Fog (CI)	FogIndex	FogIndex	FogIndex	FogIndex
Fog (RGBA)	FogIndex	FogIndex	FogIndex	1.0

The following table shows the inputs to the B multiplexer for each of the colour channels. The subscripts identify the RGBA components where necessary.

	R	G	B	A
Interpolation	$Texel0_r$	$Texel0_g$	$Texel0_b$	$Texel0_a$
	$Texel1_r$	$Texel1_g$	$Texel1_b$	$Texel1_a$
	$Texel2_r$	$Texel2_g$	$Texel2_b$	$Texel2_a$
	$Texel3_r$	$Texel3_g$	$Texel3_b$	$Texel3_a$
	$Texel4_r$	$Texel4_g$	$Texel4_b$	$Texel4_a$
	$Texel5_r$	$Texel5_g$	$Texel5_b$	$Texel5_a$
	$Texel6_r$	$Texel6_g$	$Texel6_b$	$Texel6_a$
	$Texel7_r$	$Texel7_g$	$Texel7_b$	$Texel7_a$
	$R0_r$	$R0_g$	$R0_b$	$R0_a$
	$R1_r$	$R1_g$	$R1_b$	$R1_a$
	$R2_r$	$R2_g$	$R2_b$	$R2_a$
Modulate, Blend, Decal	$Colour_r$	$Colour_g$	$Colour_b$	$Colour_a$
Blend	$TexEnvColour_r$	$TexEnvColour_g$	$TexEnvColour_b$	$Colour_a$
Fog	$FogColour_r$	$FogColour_r$	$FogColour_b$	$FogColour_a$

Texture processing has two enables which must both be set to enable modification of the Colour message. The first enable is held in a register within this unit which is loaded via the TextureColourMode message and is effective until changed by a new TextureColourMode message. The second enable is the TextureEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable texturing when a primitive (i.e. polygon) must not have it enabled.

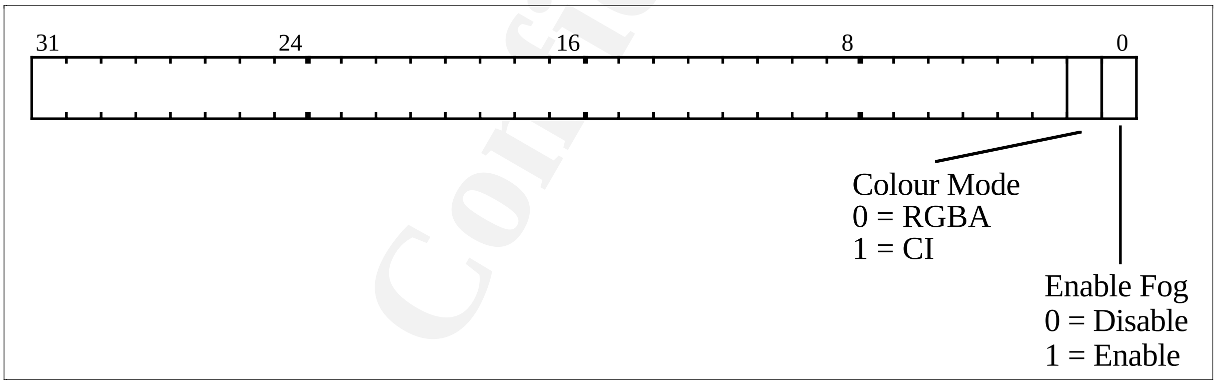
The operation of the texture part of this unit is controlled by the TextureColourMode message and the control bits in the data field are defined as follows:



6.3. Fog Processing

Fog processing is a two stage operation: calculate the fog index for this fragment and then use it to control the interpolation between the current colour (optionally modified by the texture mapping process) and a background or fog colour. OpenGL allows the fog index between vertices to be linearly interpolated even though the fog at a vertex can be calculated from an exponential, exponential squared or linear equation. Fog processing is defined for RGB and CI modes however the equations are the same so this unit doesn't need to differentiate between the two colour modes. The alpha component is not changed by fog.

The operation of the fog unit is controlled by the FogMode message and this has the following format:



Fog processing has two enables which must both be set to enable modification of the Colour message. The first enable is held in a register within this unit which is loaded via the FogMode message and is effective until changed by a new FogMode message. The second enable is the FogEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable fog application when a primitive (i.e. polygon) must not have it enabled.

6.3.1. Fog DDA unit

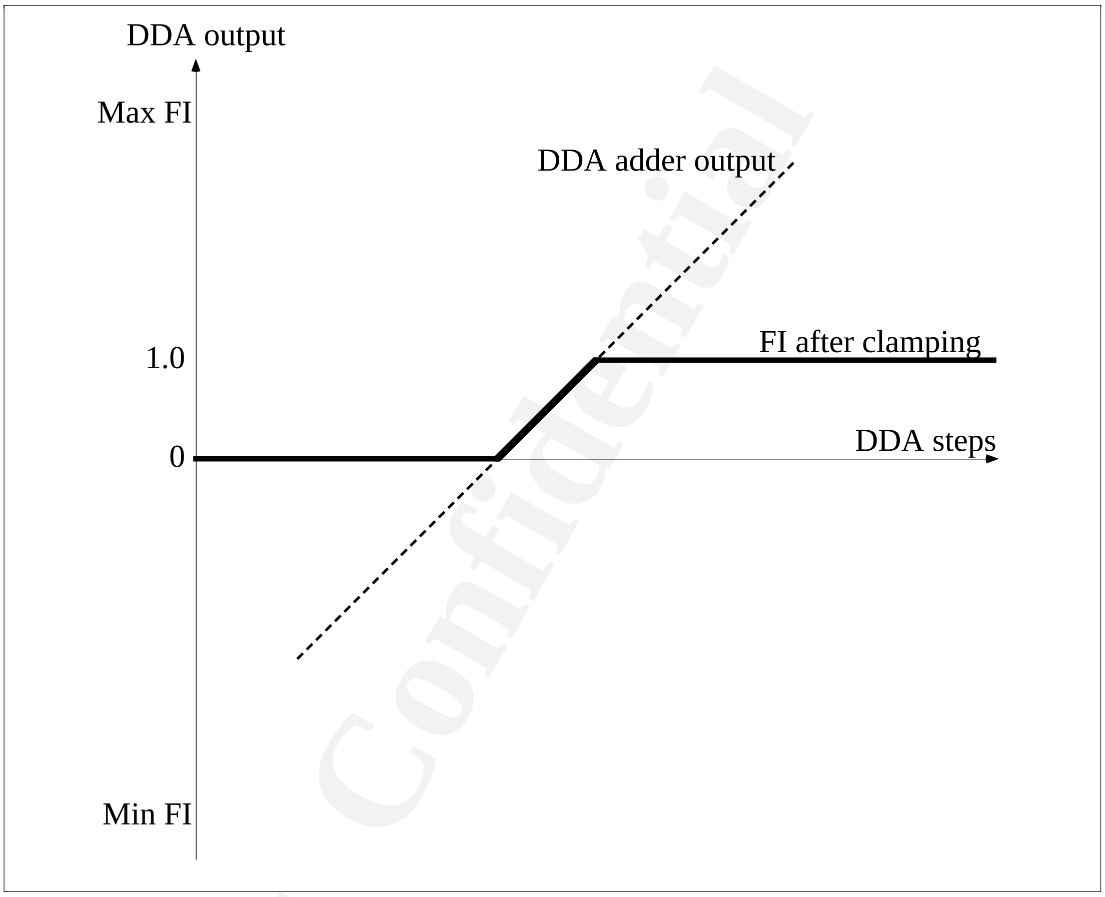
The Fog DDA unit is responsible for generating the fog information under the direction of the step messages from the Rasteriser Unit. Sub pixel corrections can be applied to correct for an initial start error on a span.

The fog DDA is unusual in how it is used. Normally a DDA unit is operated within its natural output range and the clamping just prevents any slight numerical errors from causing wrap around of the numbers and hence an abrupt change in value.

To give the appearance of fog the fragment's colour is effected differently depending in which of three range bands the fragment is in:

- In the near band the fragment's colour is not changed.
- In the middle band the fragment's colour is blended with the fog's colour.
- In the far band the fragment's colour is replaced by the fog's colour.

This effect is easily achieved by running the fog DDA over a wide range and clamping within a small range (0.0 to 1.0) and letting the interpolation do its job. This gives the following transfer function:



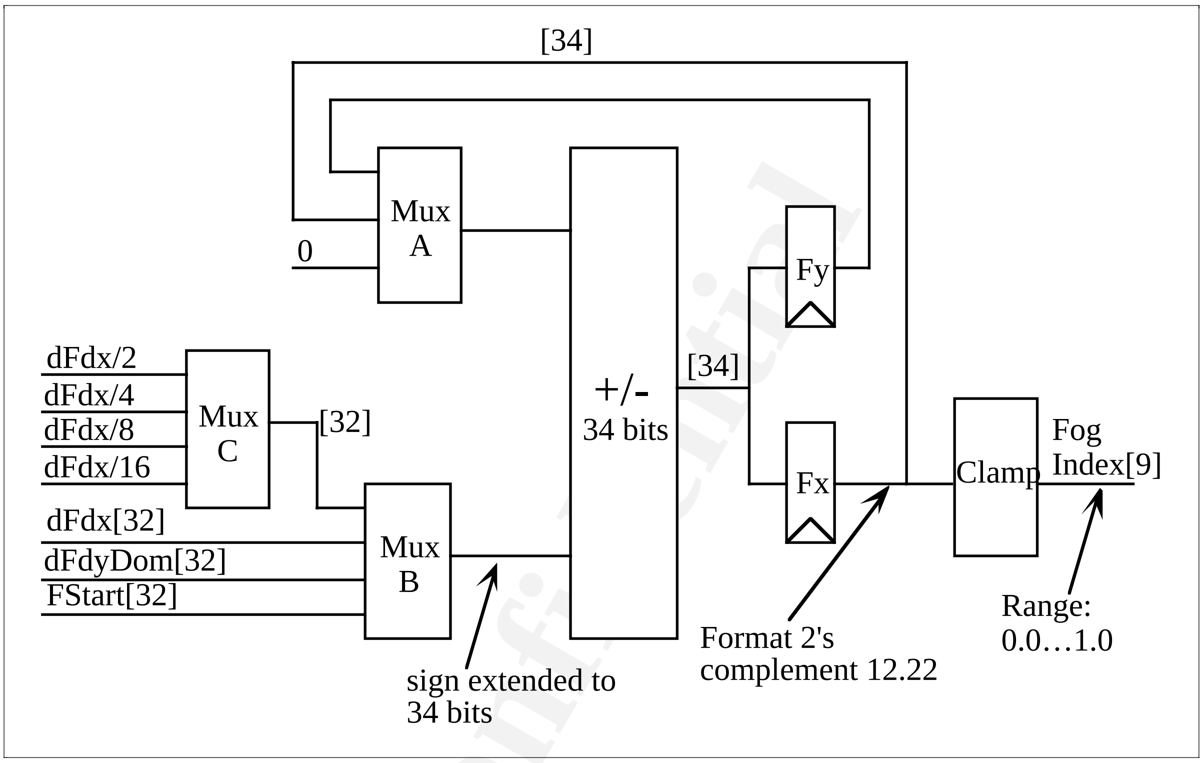
The DDA only has a relatively narrow range[14] (+511 to -512) compared to the range of depths so the software will need to be careful in how the DDA is set up. There are four cases:

- If all the vertices are in the near range then the DDA should be set up to output 1.0 with a delta of 0.
- If all the vertices are in the far range then the DDA should be set up to output 0.0 with a delta of 0.
- If all the vertices are within the DDA's range then the DDA's parameters are set up as normal.
- One or more of the vertices are out of the DDA's range so will need to be clamped before the DDA's parameters are set up. This will only occur on very large polygons which

[14]Internally there are a few more bits to prevent wrap around errors.

extend from near the eye point into the far distance. The result of clamping the input values to the DDA will be to change the effective position and width of the fog band (i.e. middle range), but this is unlikely to be noticeable. If it is noticeable then tessellating the polygon will solve the problem.

The block diagram of the DDA is:



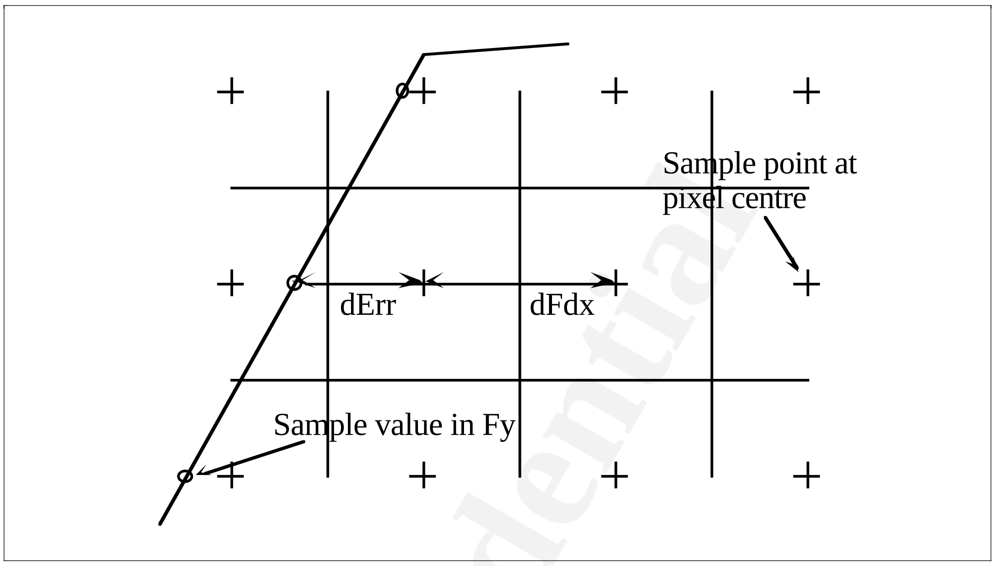
Some notes on the diagram:

- The Fx register holds the fog index as a span is filled. The Fy register holds the value of the fog index as we step up the dominant edge of the primitive and is used to seed the span interpolation.

- The adder/subtracter normally works as an adder except when sub pixel compensation is being applied where it can work in either mode depending on the direction of the correction. At first sight an additive correction is all that is necessary because we always correct in the direction that the dx derivative is set up. However the geometry of the primitive has been biased by *nearly a half* (of a pixel) to make the rasterisation rules simpler to implement, while the interpolated parameters are set up with respect to the unbiased geometry.

- The start and derivative number format is 2's complement fixed point integer: 10 bits integer and 22 bits fraction. The internal format has an additional 2 bits to extend the range to give more protection against over or underflow. This guard band is only designed to cope with small amounts of overflow (or underflow) when one or two extra deltas are added during rasterisation, usually as a result of the sample point (i.e. pixel centre) being outside of the boundary of the primitive. If the additional range provided by the guard band is exceeded then wrap around will occur and an abrupt change in the colour will be visible (the clamp logic cannot prevent this). This situation will only occur if the delta values are set up incorrectly for the number of iterations the DDA works through, i.e. the gradient is specified as being steeper than it really is.

- The Clamp logic limits the fog index to be in the range 0.0 to 1.0 inclusive. The clamp logic is simply a multiplexer with selects either the result, zero or 1.0 based on comparing the result against 1.0 and the sign bit.
- Multiplexer C is used to implement the sub pixel correction. The need for the sub pixel correction is easily shown in the following diagram:



The start value has already been compensated to move it to the centre of a pixel in the Y directions. This only needs to be done once per triangle and is done in software. As the DDA is stepped in Y along the dominant edge the value of the interpolated parameter is calculated at the points marked by circles, whereas the value is required in the pixel's centre.

If dErr is the distance the edge is away from the pixel's centre (must be < 1) and dFdx is the change in F for unit change in x then the correct value at the first sample point is:

$$Fx = Fy + dErr * dFdx$$

The distance dErr is sent by the rasteriser in the SubPixelCorrection message and whenever this message is received the above equation is implemented. The SubPixelCorrection message allows for 4 bits of resolution of dErr. The correction is applied to a resolution of 1/16 of the dx derivative. The multiply operation is too expensive to do using a multiplier so it is implemented by addition of the partial products. For a positive correction this means adding in dFdx/2 when bit 3 is set, dFdx/4 when bit 2 is set, dFdx/8 when bit 1 is set and dFdx/16 when bit 0 is set. For a negative correction the corresponding partial products are subtracted.

The value of dErr is in sign magnitude format with bit 4 holding the sign (0 = positive), and bit 3 representing 2^{-1}, bit 2 representing 2^{-2}, etc..

The division of dFdx is always done by a power of two so is just an arithmetic right shift (to retain the sign of the dFdx).

The correction will add one cycle for every bit set in dErr so worst case will take 4 cycles per scanline, but on average 2 cycles. This time is hidden under the page break time allowed for every scanline so will not impact performance.

- The fog index used later is a positive 9 bit fixed point number with 1 bit integer and 8 bits fraction.

- The fog DDA responds to rasteriser messages as follows:

Message	Mux A output	Mux B output	Register Fx	Register Fy
PrepareToRender	zero	FStart	Load	Load
StepX	Fx	dFdx	Load	Hold
StepYDom	Fy	dFdyDom	Load	Load
SubPixelCorrection	Fx	Mux C o/p	Load	Hold

- If fog is disabled then the DDA is not updated on active step or PrepareToRender messages.

6.3.2. Application of the fog

Once the fog index has been found and then it is used to control the interpolation between the fog colour (FC) and the current colour (C). The application of fog is different for RGBA and CI modes and this is controlled by the ColourMode bit in the FogMode message.

In RGBA mode the the same Lerp function is used as earlier, but note that alpha component is not changed by the Lerp operation:

$$V_{rgb} = Lerp\left(FC_{rgb}, C_{rgb}, FI\right)$$
$$V_a = C_a = Lerp(FC_a, C_a, 1.0)$$

In CI mode the equation is:

$$V_{rgba} = C_{rgba} + (1 - FI)FC_{rgba}$$

Note that the CI value is only in the red channel for units down steam but doing the same equation on all colour channels keeps the control simpler.

6.4. Input Messages

6.4.1. External Messages

Texture/Fog Colour Register Update Messages		
Tag Mnemonic	Data Field	Description
TextureColourMode	See above	
TextureEnvColour	32 bit RGBA format, R in least significant byte	
FogMode	See above	
FogColour	32 bit RGBA format, R in least significant byte	
FStart	32 bit 2's complement fixed point format (10.22).	Fog start value
dFdx	32 bit 2's complement fixed point format (10.22).	Fog derivative unit X
dFdyDom	32bit 2's complement fixed point format (10.22).	Fog derivative unit Y, dominant edge

Scoreboarded Register Update Messages		
Tag Mnemonic	Data Field	Description
Texel0	32 bit RGBA format, R in least significant byte	
Texel1		
Texel2		
Texel3		
Texel4		
Texel5		
Texel6		
Texel7		
Interp0	Positive 9 bit value in fixed point format (1.8).	
Interp1		
Interp2		
Interp3		
Interp4		
Colour	32 bit RGBA format, R in least significant byte	
TextureFilter	Three bit field (bits 0, 1 and 2) 0 = Nearest 1 = Linear2 2 = Trilinear4 3 = Linear4 4 = Trilinear8	Specifies the filter operation to carry out on this fragment

None of these messages are passed on to the next unit.

Note that the Texel[0…7], Interp[0…4] and TextureFilter messages would normally be shown as internal messages as the Texture Read Unit would generate them. In the first version of GLiNT this unit is absent so they will be sent by the host, hence they are documented here for completeness.

6.4.2. Internal Messages

The unit responds to the Colour, PrepareToRender, SubPixelCorrection and the Step messages. See the Rasteriser and Colour DDA Units for details on these messages.

6.5. Output Messages

No new messages are generated, however the data field in the colour message may be modified.

6.6. Behavioural Model

The behavioural model is divided into to stages: an input stage to handle the scoreboarding of some of the registers, and a Colour Processing stage to do the real work.

```
Wait for input message;
{
   switch (on message type)
   {
      case PrepareToRender:
           Wait for any colour processing to finish;
           Set all scoreboard bits to free;
           Pass message onto ColourProcessing;
           break;
      case Scoreboard register group:
           Wait for appropriate scoreboard bit to become free;
           Update appropriate register;
           if (message was Colour)
           {
              if (fog is disabled and texture is disabled)
              {
                 Wait for room in next unit;
                 Forward on the Colour message;
              }
           }
           break;
      case ActiveStep:
           Wait for any colour processing to finish;
           Set all scoreboard bit to busy;
           Pass message onto ColourProcessing;
           break;
      case default:
           Wait for any colour processing to finish;
           Pass message onto ColourProcessing;
           break;
   }
   Flush input message;
}
```

```
ColourProcessing (message)
{
  switch (on message type)
  {
    case Texture and fog data group:
        Update the appropriate register;
        break;
    case PrepareToRender:
        Save state of Texture and Fog enable flags;
        if (fog is enabled)
          Update DDA from FStart register;
        Wait for room in next unit;
        Forward on message to next unit;
        break;
    case SubPixelCorrection:
        if (fog is enabled)
        {
          if (sign of dxErr is positive)
            For each bit set in dxErr add the
              corresponding fraction of dFdx to Fx
              and update Fx.
          else
            For each bit set in dxErr subtract the
              corresponding fraction of dFdx from Fx
              and update Fx.
        }
        Forward on message to next unit;
        break;
    case PassiveStep:
        if (fog enabled)
          Step the fog DDA in the appropriate direction;
        Wait for room in next unit;
        Forward message to next unit;
        break;
    case ActiveStep:
        Wait for room in next unit;
        if (texture is disabled and fog is enabled)
        {
          if (fog colour mode is RGBA)
          {
            new colour = Lerp (FogColour, Colour, DDA
                                          output);
            new alpha = Lerp (FogColour, Colour, 1.0);
          }
          else
          {
            new colour, alpha = Colour * 1.0 +
                          (1.0 - DDA output) * FogColour;
          }
          Send Colour message with new colour and alpha
            results;
        }

        if (texture is enabled and fog disabled)
        {
          DoTextureCalculation;       // result left in R0
          Send Colour message with R0 data;
        }
```

```
          if (texture is enabled and fog enabled)
          {
             DoTextureCalculation;       // result left in R0
             if (fog colour mode is RGBA)
             {
                new colour = Lerp (FogColour, R0, DDA
                                          output);
                new alpha = Lerp (FogColour, R0, 1.0);
             }
             else
             {
                new colour, alpha = R0 * 1.0 +
                               (1.0 - DDA output) * FogColour;
             }
             Send Colour message with new colour and alpha
                results;
          }

          if (fog enabled)
             Step the fog DDA in the appropriate direction;

          Set any remaining scoreboard bits to free;
          Wait for room in next unit;
          Forward active step message to next unit;
          break;
     case default:
          Wait for room in next unit;
          Forward message to next unit;
          break;
   }
}
```

```
DoTextureCalculation
{
   // Note that after a scoreboarded register has been
   // used for the last time the corresponding scoreboard bit is
   // set to free.
   // This is not shown here to keep the code uncluttered.
   switch (filter type)
   {
      case Nearest:
            R0 = Texel0;
            break;
      case Linear2:
            R0 = Lerp (Texel0, Texel1, Interp0);
            break;
      case Linear4:
            R0 = Lerp (Texel0, Texel1, Interp0);
            R1 = Lerp (Texel2, Texel3, Interp0);
            R0 = Lerp (R0, R1, Interp1);
            break;
      case Trilinear4:
            R0 = Lerp (Texel0, Texel1, Interp0);
            R1 = Lerp (Texel2, Texel3, Interp1);
            R0 = Lerp (R0, R1, Interp2);
            break;
      case Trilinear8:
            R0 = Lerp (Texel0, Texel1, Interp0);
            R1 = Lerp (Texel2, Texel3, Interp0);
            R0 = Lerp (R0, R1, Interp1);
            R1 = Lerp (Texel4, Texel5, Interp2);
            R2 = Lerp (Texel6, Texel7, Interp2);
            R1 = Lerp (R1, R2, Interp3);
            R0 = Lerp (R0, R1, Interp4);
            break;
   }
   switch (application method)
   {
      case Modulate:
            R0 = R0 * Colour;
            break;
      case Decal:
            R0[rgb] = Lerp (Colour[rgb], R0[rgb], R0[a]);
            R0[a] = Lerp (Colour[a], R0[a], 0.0);
            break;
      case Blend:
            R0[rgb] = Lerp (Colour[rgb], TextureEnvColour[rgb],
                                  R0[rgb]);
            R0[a] = Colour[a] * 0.0 + Colour[a] * 1.0;
            break;
   }
}
```

```
Lerp (A, B, I)
{
   Select A and route to multiplier 8 bit port;
   Do 1.0 - I and route to multiplier 9 bit port;
   Multiply and store result in U;
   Select B and route to multiplier 8 bit port;
   Select I and route to multiplier 9 bit port;
   Multiply and store result in V
   Add U and V and return bits 8 to 15;
}
```

7. Alpha Test Unit

7.1. Description

The alpha test unit has two functions:

- When antialiased primitives are being rendered the fragment's colour is weighted by the percentage area of the pixel the fragment covers. An approximation to the area covered is calculated in the rasteriser unit and sent to this unit in the CoverageValue message. The coverage is specified in a 9 bit number where 256 represents 100% and 0 represents 0%. If antialiasing is disabled then the colour is passed onto the alpha test stage unchanged.
- The Alpha Test, if enabled, compares the alpha value of a Colour message, after coverage weighting, against a reference value and if the compare passes the fragment is allowed to continue. If the comparison fails the fragment is terminated.

7.1.1. Antialiasing

In colour index (CI) mode the bottom 4 bits of the colour index are replaced by the scaled coverage value. The scale value is 15/256 and the result is rounded to the nearest integer before replacing the bottom 4 bits of the CI value. If an erroneous Coverage value has been sent (i.e. it is greater than 256 or 100%) then this calculation may overflow. In this case the result is clamped to the maximum value of 0xf before replacing the bottom 4 bits of the CI value.

In RGBA mode the alpha component is multiplied by the coverage value, but the RGB components are not changed.

The alpha component in the Colour message is 8 bits in size and this need to be converted to the 9 bit internal format (the other 3 components are not effected by the application of the coverage value). The 9 bit unsigned format is one bit integer and 8 bits fraction so that 1.0 can be easily represented and manipulated. Conversion from the 8 bit input format to the 9 bit output format is done using the equations:

$$output = \frac{256}{255} input$$

which, in the limited precision we have available, translates into:

output = input for $0 \leq$ input < 255
output = 256 for input = 255

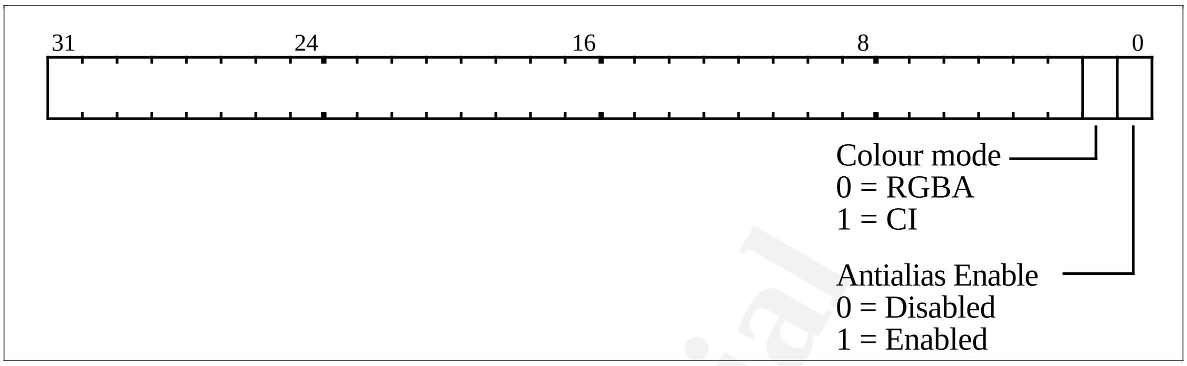
The discontinuity in the normally monotonic number set is inevitable unless the number of fractional bits is increased.

The unsigned multiplication of the 9 bit alpha by the 9 bit coverage value yields an 18 bit result. The two most significant bits are the integer and the lower 16 bits the fraction. If any of the integer bits are set then the 8 bit result is set to 255, otherwise the most significant 8 bits of the fraction are used as the result. This conversion, like the 8 to 9 bit conversion done earlier, is a compromise of the real conversion equation:

$$A_o = \frac{255}{256} A_i$$

The modified Colour message is forwarded on to the next unit and the modified alpha component used in the alpha test, if necessary.

The antialiasing is controlled using the AntialiasMode message. It has the following format:



Antialiasing has two enables which must both be set to enable modification of the alpha value in the Colour message. The first enable is held in a register within this unit which is loaded via the AntialiasMode message and is effective until changed by a new AntialiasMode message. The second enable is the CoverageEnable bit in the PrepareToRender message and this is only effective until the next PrepareToRender message is received. This second enable is used to temporarily disable the coverage application when a primitive (i.e. polygon) must not have it enabled.

7.1.2. Alpha test

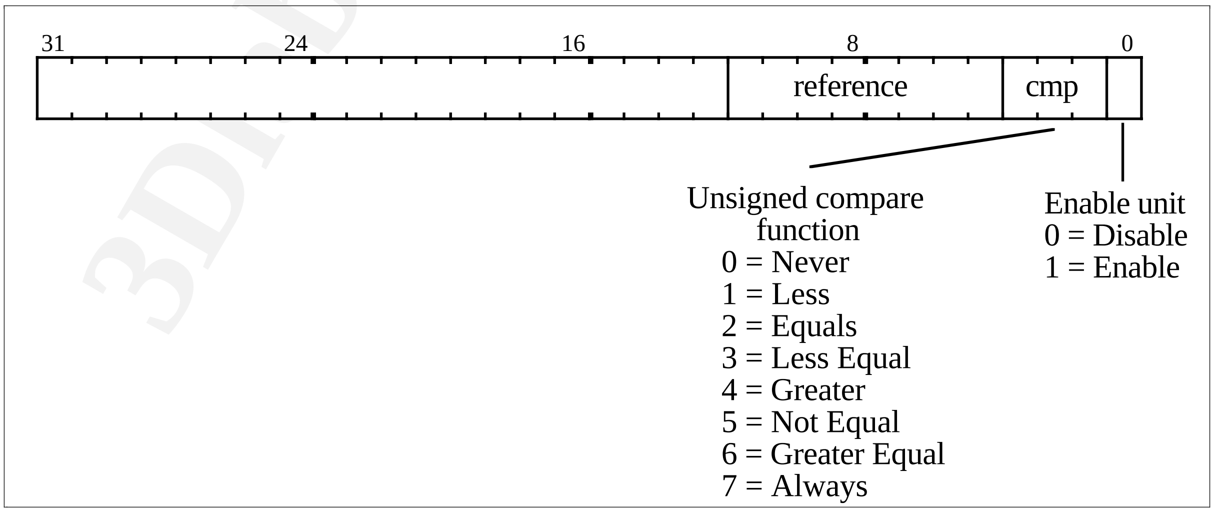
This is implemented in the following steps:

1. When a Colour message is received the alpha component (after any antialiasing) is compared against the reference value using the specified test.

2. If the test passes then the next Step message is passed through unchanged.

3. If the test fails then the next Step message is converted into a passive one and send to the next unit.

The operation of the unit is determined by the AlphaTestMode message. The data field has the following format:



If the alpha test is disabled then it is as if the alpha test always passes.

The compare operation is done unsigned.

The compare operation compares the fragments alpha value (from the colour message) against the reference alpha value. If the compare function is 'Less' and the result is true then the fragment value is less than the reference value.

7.2. Input Messages

7.2.1. External Messages

Alpha Test Register Update Messages		
Tag Mnemonic	Data Field	Description
AntialiasMode	See above	Antialiasing control field
AlphaTestMode	See above	Alpha test control fields

7.2.2. Internal Messages

This unit reacts to all the messages in the Rasteriser Walk group and the CoverageValue message. See the Rasteriser Unit for details on these messages.

This unit reacts to the Colour message. See the Colour DDA Unit for details on this message.

7.3. Output Messages

No new messages are generated, however an active walk message can be converted to a passive one and the Colour message may be modified.

7.4. Behavioural Model

```
Wait for input message
{
  switch (on input message)
  {
    case AlphaTestMode:
    case AntialiasMode:
    case CoverageValue:
            Update the appropriate register;
            break;
    case Colour message:
            Wait for room in next unit;
            if (antialiasing enabled and CoverageEnabled)
            {
              if (CI mode from AntialiasMode)
              {
                Merge scaled coverage value into CI value
                  from Colour message and send to next unit;
              }
              else
              {
                Format the alpha component into 9 bit format;
                Multiply the 9 bit alpha colour component
                  by the coverage value and clamp result
                  back to 8 bits;
                Replace the alpha component in the Colour
                  message with the new one and send to next
                  unit;
              }
            }
            else
            {
              Forward Colour message onto the next unit;
            }
            break;
    case PrepareToRender:
            Save state of CoverageEnable bit;
            Wait for room in receiving queue;
            Send PrepareToRender message to next unit;
            break;
    case ActiveStepX:
    case ActiveStepYDomEdge:
            Wait for room in receiving queue;
            if (alpha test is enabled)
            {
              if (alpha compare test result is pass)
              {
                Send ActiveStep message to next unit;
              }
              else
              {
                Change ActiveStep message to be passive;
                Send PassiveStep message to the next unit;
              }
            }
            else
```

```
                {
                   Send ActiveStep message to next unit;
                }
                break;
       default:
                Wait for room in receiving queue;
                Pass on message;
                break;
    }
    Flush the input message;
}
```

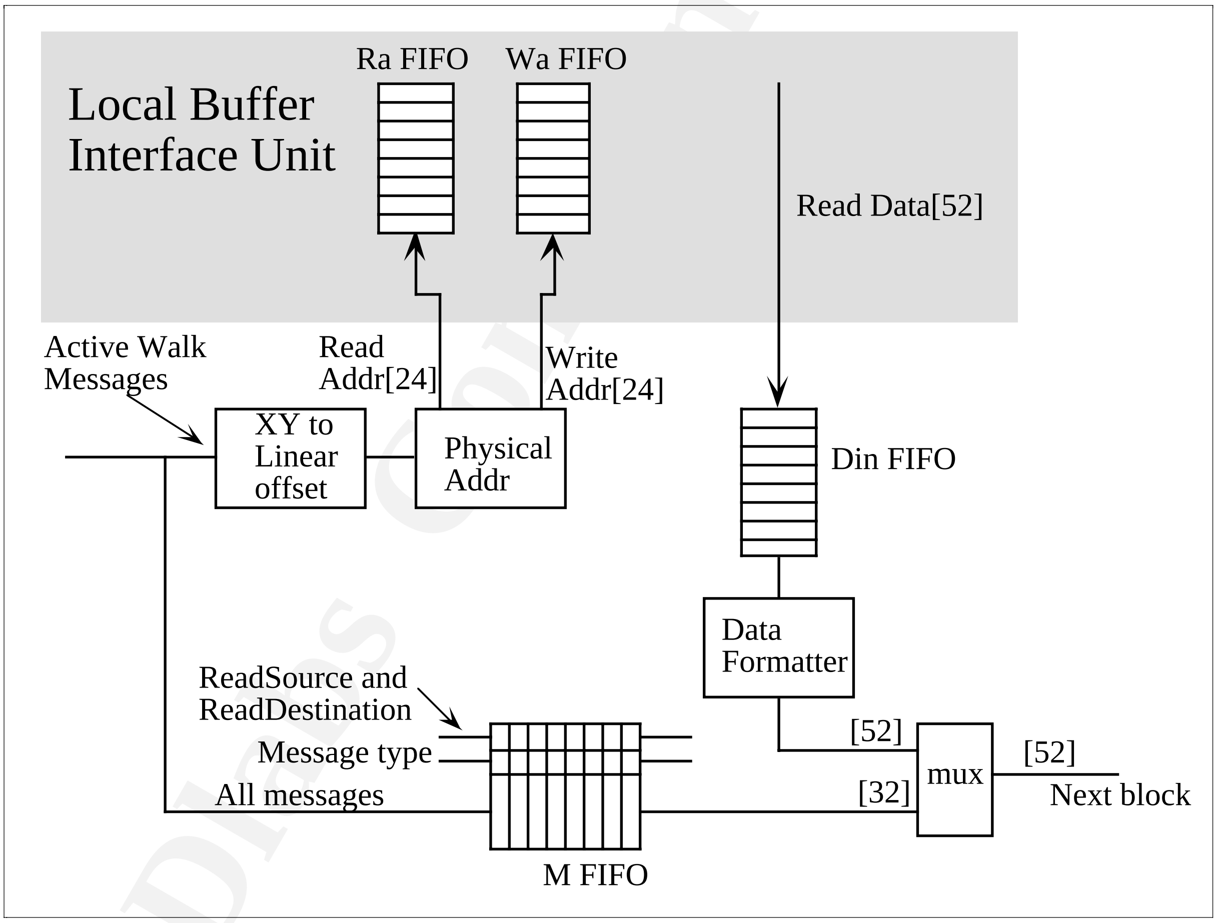
8. Local Buffer Read Unit

8.1. Description

The Local Buffer Read Unit is responsible for:

- calculating the read address(es) of the fragment in the local buffer,
- calculate the write address of the fragment in the local buffer,
- dispatch the addresses to the Local Buffer Interface Unit,
- receive the local buffer data some time later,
- converting from the variable external format into the constant internal format.

A simple diagram of the Local Buffer Read Unit follows:



The actual reading of the memory and request arbitration is done in the Local Buffer Interface Unit.

The Local Buffer Unit, in response to active walk messages will calculate zero, one or two read addresses and one write address[15]. These will be sent to the Local Buffer Interface Unit, freeing the Local Buffer Unit to wait and act on a new active walk message. All messages will be delayed in the M FIFO so the read data can be matched up with the active message which generated the request. Each message stored in the M FIFO has the number of reads associated with it (0, 1 or 2 encoded in the ReadSource and ReadDestination bits) so this amount of data can be read off the Din FIFO and passed onto the next block before the message is sent.

An active walk message generates the following message stream:

first:	LBData message	(if present)
	LBSourceData message	(if present)
last:	Active Walk message	

The order of messages is important as the last message (Active Walk message) triggers the GID, depth and stencil tests to occur.

8.1.1. Local Buffer Interface Unit

This section does not describe the Local Buffer Interface Unit (LBIU) but defines the expected behaviour of it and how it is interfaced to the Local Buffer Read Unit (LBRU) and the Local Buffer Write Unit (LBWU).

The following signals are present between the units:

[15]A write address is always generated. When no write is needed, for example in Image upload, the write is disabled in the Local Buffer Write Unit. A disable write facility has not been included here to prevent a possible deadlock occurring if the write enable control is not consistent between the two units.

Name	Width	Source	Description
LBReadAddr	24	LBRU	The read address of the local buffer data. The whole width (max. 52 bits) is selected by this address. The address is sent to the Ra FIFO.
LBReadEnable	1	LBRU	When active instigate a read from this address.
LBSuspendRead	1	LBRU	When active suspend on this read. This bit is sent to the Ra FIFO.
LBWriteAddr	24	LBRU	The write address of the local buffer data. The whole width (max. 52 bits) is selected by this address. The address is sent to the Wa FIFO
LBWriteAddrEnable	1	LBRU	When active clocks the WriteAddr into the Wa FIFO.
LBRdAddrFull	1	LBIU	When active the Local Buffer Interface Unit cannot accept any more read addresses and the Local Buffer Read Unit must stall.
LBWrAddrFull	1	LBIU	When active the Local Buffer Interface Unit cannot accept any more write addresses and the Local Buffer Read Unit must stall.
LBReadData	52	LBIU	The local buffer data returned from a read operation and the data is sent to the Din FIFO.
LBRdDataFull	1	LBRU	When active the Local Buffer Read Unit cannot accept any more read data and the Local Buffer Interface Unit must stall, or switch to doing writes.
LBReadDataValid	1	LBIU	When active qualifies the read data to be valid.
LBWriteData	52	LBWU	The data to be written to the local buffer and is sent in the Write data FIFO.
LBWriteCmd	2	LBWU	Specifies the required write action: 0 = Write data 1 = Discard write 2 = Resume read These bits are sent to the Write data FIFO.
LBWrDataEnable	1	LBWU	When active the WriteData and WriteCmd is clocked into the write data FIFO.
LBWrDataFull	1	LBIU	When active the Local Buffer Interface Unit cannot accept any more write data and the Local Buffer Write Unit must stall.
LBWrComplete	1	LBIU	Active when the write data FIFO is empty and all outstanding writes completed.

Some general rules and observations:

1. Multiple read and write requests can be outstanding to give the opportunity to group reads and writes together to make better use of the page structure in the memory.

2. At the interface level the read data must be returned in the same order the read addresses were presented in. The actual memory reads can be executed in any order provided the ordering at the interface is maintained.

3. The write data will be given to the Local Buffer Interface Unit an arbitrary number of cycles after the write address, and in fact there will have been other write addresses in the mean time. The write data will be sent in the same order the write addresses were. The write may be identified to be discarded if there is no need to change the data in the memory. The actual memory writes can be executed in any order once the address and data have both been received.

4. There is no requirement for the reads and writes to occur in the order they were issued in, however a later read operation to an outstanding write address must return the new write data rather than the pre-write data. Catching the occurrence of this in hardware is very expensive, however the special nature of the rasterisation process[16] can be used to easily flag any potential situations where this might occur. Each read address is qualified by a Suspend Reads flag, which is normally reset. A PrepareToRender message will cause a write to the Ra FIFO with this flag set and an undefined address. On detecting this flag set the Local Buffer Interface Unit will not start this read, or any subsequent ones received until it detects the Resume Reads signal being set. The Resume Reads signal is issued by the Local Buffer Write Unit once all the writes for the old primitive have been issued and only allows the reads to resume once all the writes have been completed. It is the PrepareToRender message which activates these flags.

5. Read will always have priority over writes, except in the case where a read would break page for which there are one or more outstanding writes queued up.

6. There needs to be sufficient (and identical) buffering in both the interface unit and core unit to amortise the cost of page breaks when the read and write addresses are in different pages as a result of a copy operation. This buffering will also help with (5). How deep should the buffers be? Ideally the deeper the better, but some practical limit, such as 8 is a good starting point. The cost associated with this figure needs to looked at and this is some what helped by elimination of the write pending logic using the method presented in (4).

8.1.2. XY to linear offset conversion

Calculating the Y offset in a window from the XY pixel offset is done by the equation:

Y offset = Y * W W = screen width

If any screen width is allowed then a true multiplier is needed with the attendant gate cost. Very few screen widths[17] need to be supported so the multiplier can be reduced to some shifters and adders. A binary multiplier with inputs Y and W works as follows: A partial product (Y << n) is generated for every bit *n* in W which is set. The sum of the partial products gives the product of Y and W.

By limiting the number of partial products, and by implication the allowable screen widths, reduces the size of the multiplier significantly. The choice of how many partial products and which combinations to support is important to give flexibility without a large gate count. If the exact screen width required is not supported then it is always possible to use the next highest one which is, albeit at the cost of an unused strip in memory.

[16]The rasterisation of a primitive guarantees that a pixel is only visited once so the only time when stale data could be read is when the rasterisation of the next primitive starts, and writes from the previous primitive are outstanding.

[17]The assumption here is that the local buffer matches the resolution of the screen (i.e. each screen pixel has a corresponding local buffer pixel for depth, stencil, etc.). In fact the hardware will also support the buffers to be allocated on a per window basis so pixels in overlapping windows do not share depth, stencil and GID data. What makes this paradigm less attractive is managing memory allocation and fragmentation, especially when windows are made bigger. Also the amount of memory used could grow to 16 times the full screen size.

Three partial products can be specified and they can be selected from the group:

Partial Product Code	Add into product	Shift amount
0	0	0
1	Y * 32	Y << 5
2	Y * 64	Y << 6
3	Y * 128	Y << 7
4	Y * 256	Y << 8
5	Y * 512	Y << 9
6	Y * 1024	Y << 10
7	Y * 2048	Y << 11

Some common screen width and the corresponding partial product codes are:

Screen width	Partial Product 0	Partial Product 1	Partial Product 2	Product
640	5	3	0	Y * (512 + 128 + 0)
1024	6	0	0	Y * (1024 + 0 + 0)
1152	6	3	0	Y * (1024 + 128 + 0)
1280	6	4	0	Y * (1024 + 256 + 0)
1600	6	5	2	Y * (1024 + 512 + 64)

The screen widths supported are:

0	32	64	96	128
160	192	224	256	288
320	352	384	416	448
512	544	576	608	640
672	704	768	800	832
896	1024	1056	1088	1120
1152	1184	1216	1280	1312
1344	1408	1536	1568	1600
1664	1792	2048	2080	2112
2144	2176	2208	2240	2304
2336	2368	2432	2560	2592
2624	2688	2816	3072	3104
3136	3200	3328	3584	4096
4128	4160	4224	4352	4608
5120	6144			

8.1.3. Y offset to destination address conversion

All XY coordinates are relative to some origin and once the Y offset has been found need to be converted to the physical address of the pixel. OpenGL has it's origin in the bottom left corner of a window and X has it's origin in the top left corner. This requires a slightly different equation to calculate the physical address for a pixel when the physical address of the window's origin is known:

OpenGL:	destination addr = LBWindowBase - Y offset + X
X:	destination addr = LBWindowBase + Y offset + X

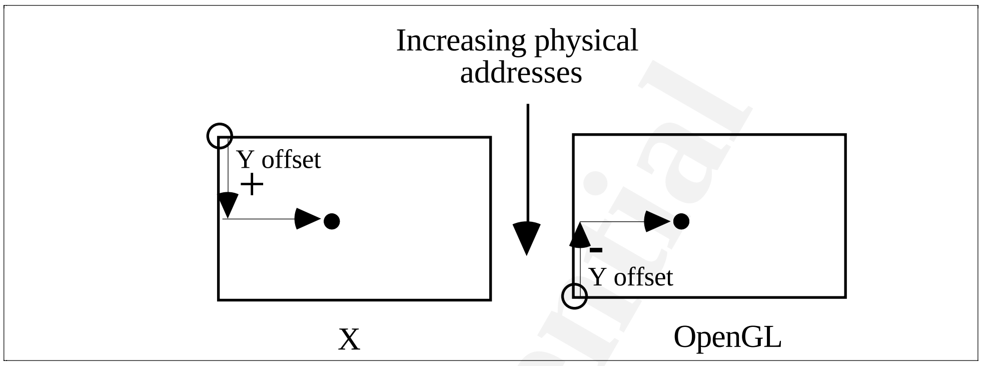
The origin to use is specified by a bit in the in the LBReadMode message.

The source address (if needed) is calculated:

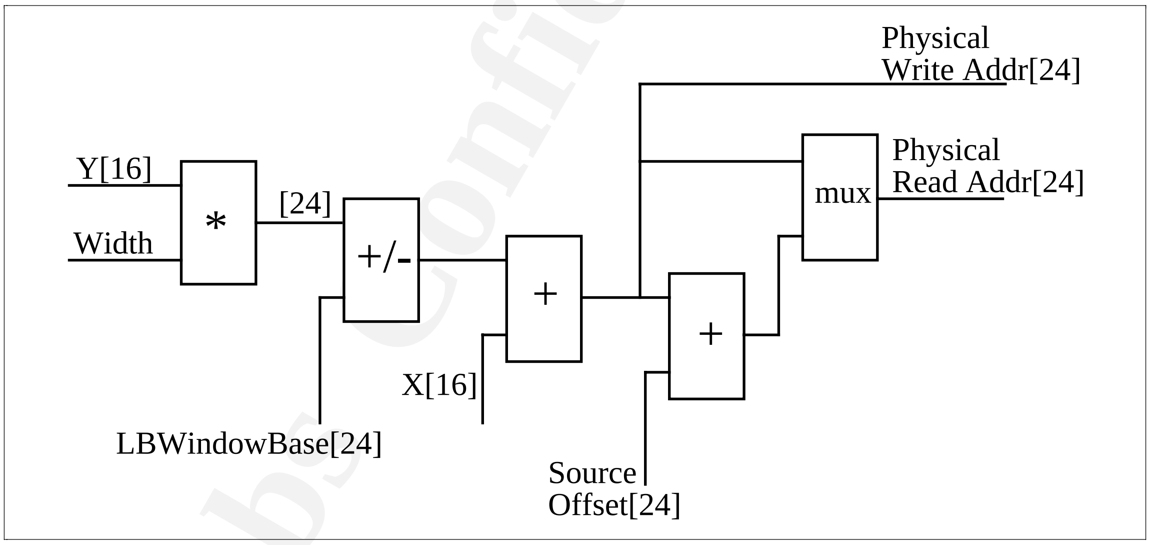
source address = destination address + SourceOffset

Note that the origin's physical address can be negative[18] if the window is part off the screen. A negative physical address is not normally generated[19] as these fragments will already have been turned into passive ones by the screen scissor test.

The following diagram demonstrates this:



The full address calculation chain is shown below:



All address calculations are done to 24 bit and any overflow just wraps around. This wrap around allows ±maximum offset to be applied with a 24 bit number. The X and Y coordinates are sign extended up to the required width.

[18]Viewing the address as a negative number is a mathematical convenience as obviously an address is really unsigned. The wrap around nature of the address calculation gives the address the duality such that a 'negative' address is treated by the adders as just a very large positive address. For example if an address of -3 is specified and an offset of 5 is added to it then the expected result is 2, or in 24 bit unsigned format: 0xfffffd + 5 = 2 if overflow is ignored and wrap around occurs.

[19]An exception to this is during an OpenGL image upload operation where some negative addresses may be generated and read from, but in this case invalid data is acceptable.

Once the destination address has been calculated for the original XY coordinate this address is passed to Wa FIFO in the Local Buffer Interface Unit (the write may later be cancelled though). The address(es) sent to the Ra FIFO depends on the ReadSource and ReadDestination enable bits in the LBReadMode message. This is tabulated below:

ReadSource Enable	ReadDestination Enable	Address(es) sent to the Ra FIFO
0	0	No addresses sent. This would be used during rendering in the X context where it has no interest in the local buffer.
0	1	The destination address is sent to the FIFO. This is the normal mode where a read-modify-write is necessary when, for example, depth buffering is enabled. The read data is sent in a LBData message.
1	0	The source address is sent to the FIFO. This is used when all the fields in a pixel are to be copied in the local buffer and no pixel ownership tests (i.e. GID tests) are needed. The read data is sent in a LBSourceData message.
1	1	The destination address is sent to the FIFO first followed by the source address. This is used when depth and/or stencil field needs to be copied to and pixel ownership testing of the destination is needed. The destination read data is sent in the LBData message and the source data in a LBSourceData message.

A read of the physical address(es) is done by the Local Buffer Interface Unit and the raw data returned.

When only one read (per Active walk message) is done the data is inserted into the data field of the Default message type, LBDepth or LBStencil message depending on the Message Type specified in the LBReadMode register. In the case of the Default the actual message sent depends on the whether the read was from the source or destination address. In the case of the LBDepth or LBStencil message the appropriate field is right justified to be in the least significant bits of the data field so it can pass through all the subsequent blocks. Note that the width of the data field has been expanded to 52 bits to accommodate the maximum width of the Local Buffer.

When two reads (per Active walk message) are done the first read data is sent in a LBData message and the second read data is sent in a LBSourceData message.

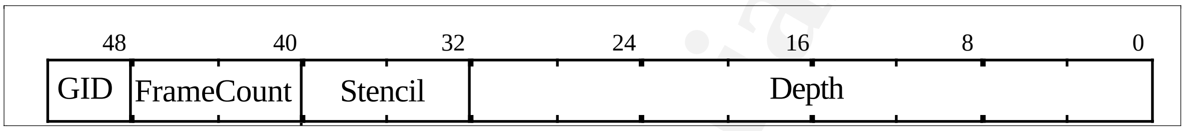
8.1.4. Data Format

Different data formats are supported in the local buffer to allow the width of the memory system to be varied to suite the cost of the system and what an application needs. The local buffer memory can be from 16 bits (assuming a depth buffer is always needed) to 52 bits wide in steps of 4 bits. The four fields supported in the local buffer, their allowed lengths and positions are shown in the following table:

Field	Lengths	Start positions
Depth	16, 24, 32	0
Stencil	0, 4, 8	16, 20, 24, 28, 32
FrameCount	0, 4, 8	16, 20, 24, 28, 32, 36, 40
GID	0, 4	16, 20, 24, 28, 32, 36, 40, 44, 48

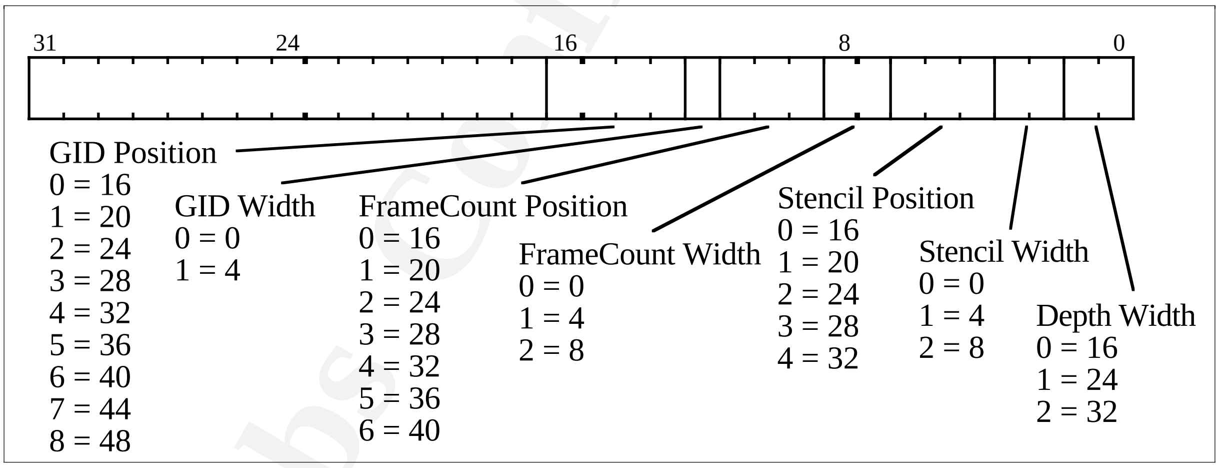
The anticipated order of the fields is as shown with the depth field at the least significant end and GID field at the most significant end. The GID is at the most significant end so that various combinations of the Stencil and FrameCount field widths can be used on a per window basis without the position of the GID fields moving. If the GID field is in a different positions in different windows then the ownership tests become impossible to do.

The local buffer data is always formatted into a consistent internal format which is:



The GID, FrameCount, Stencil and Depth fields in the local buffer are right justified if they are less than their internal widths so the unused bits are the most significant bits and they are set to 0.

The format of the local buffer is specified in the LBReadFormat message and the data field is defined as follows:



It is still possible to part populate the local buffer so other combinations of the field widths are possible (i.e. depth field width of 0), however this may give problems if texture maps are to be stored in the local buffer as well.

The behavioural model described later allows for the LBReadFormat to be changed while there are outstanding reads. The new format takes effect immediately resulting in an undesirable race condition. The correct fix is to pass the LBReadFormat message through the M FIFO and load the register in the 'output stage', however this is too big a change so a software work-around is used instead.

Before the LBReadDataFormat is changed all outstanding reads must have been completed. This can be achieved in several ways:

- Send a Sync message and wait for to appear at the bottom of the message stream.

- Send 8 messages, the Sync message will do if other useful messages are not available. No wait for the Sync at the output is necessary.

Changing the LBReadFormat usually only occurs on a context change so either of these two work-around will present a good solution.

8.1.5. Fast Clear Operation

The Fast Clear mechanism provided a method where the cost of clearing the depth and stencil buffers can be amortised over a number of clear operations issued by the application. This works as follows:

The window is divided up into *n* regions, where *n* is the range of the frame counter (16 or 256 in our case). Every time the application issues a clear command the reference frame counter is incremented (and allowed to roll over if it exceeds its maximum value) and the n^{th} region is cleared only. The clear updates the depth and/or stencil buffers to the new values and the frame count buffer with the reference value. This region is much smaller than the full region the application thinks it is clearing so takes less time and hence give the speed up.

When the local buffer is subsequently read and the frame count is found to be the same as the reference frame count (held in the Window message) the local buffer data is used directly.

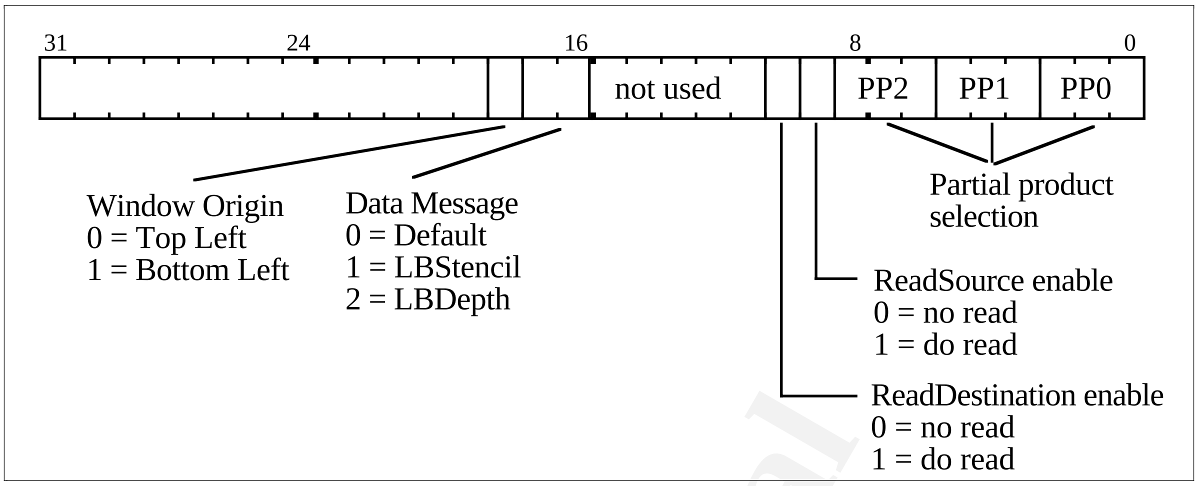
However, if the frame count is found to be different from the reference frame count (held in the Window message) the data which would have been written, if the local buffer had been cleared properly, is substituted for the stale data returned from the read. Any new writes to the local buffer will set the frame count to the reference value so the next read on this pixel works normally without the substitution. The data substitution is done in the GID, Stencil and Depth Unit.

The fast clear mechanism does not present a total solution as the user can elect to clear just the stencil planes or just the depth planes, or both. The situation where the stencil planes only are 'cleared' using the fast clear method, then some rendering is done and then the depth planes are 'cleared' using the fast clear will leave ambiguous pixels in the local buffer. The server will need to catch this and fall back to using a per pixel write to do the second clear. Which field(s) the frame count plane refers to is recorded in the GID, Stencil and Depth Unit.

When clear data is substituted for real memory data (during normal rendering operations) the write enables for selected fields (stencil and/or depth) are disabled to mimic the OpenGL operation when a buffer is cleared.

8.1.6. General control

The general mode selection for this unit is done with the LBReadMode message and the data field specifies the required modes. The data field has the following format:



The Data message field specifies the message type the data read back from the local buffer is encapsulated in. If the type is Default then message type used is either LBData or LBSourceData depending where the data was read from. The selected fields in the local buffer are aligned to the least significant end of the message data field so they don't get truncated when the message width returns to 32 bits. This mode is only used when the contents of the local buffer are being uploaded to the host. All other fields have been described already.

The ReadSource, ReadDestination and Data Message fields are sent down the M FIFO to the output stage where they are needed. This ensures they are not changed asynchronously to any outstanding reads.

8.2. Operating Modes

This section looks at how this unit is set up to do some common operations in OpenGL and X. The partial product selection is set up for the size of screen and never changed. Fast clear is also disabled for all the example operations.

8.2.1. OpenGL Rendering Operation

The normal rendering operation (line, triangle, clears, etc.) involve reading the local buffer, doing some tests (at least the GID test) and writing back to the same pixel. The unit is set up as follows:

ReadDestination enable	1
ReadSource enable	0
LBSourceOffset	Don't care
Data message	Default

8.2.2. X Rendering Operations

Normal X doesn't use the Local Buffer while rendering to the framebuffer. In addition to this unit being 'disabled' the Local Buffer Write Unit is disabled to prevent any writes occurring. The unit is set up as follows:

ReadDestination enable	0
ReadSource enable	0
LBSourceOffset	Don't care
Data message	Don't care

8.2.3. Window initialisation

Window initialisation occurs when a window is first mapped to the display, or when part of an obscured window is made visible. Initialising the Local Buffer is just a straight write to all the pixels in the window. No pixel level GID testing is done so no read on the local buffer is needed.

The GID, Depth and Stencil Unit is set up to provide the constant GID, stencil and depth values to write and unconditional writing is enabled. The rasteriser is set up to scan convert the rectangular regions which make up the visible parts of the window. The unit is set up as follows:

ReadDestination enable	0
ReadSource enable	0
LBSourceOffset	0
Data message	Don't care

8.2.4. OpenGL Copy Operation

OpenGL has a function which allows the user to specify a source rectangular region (in the window) to copy to a destination region. This function allows the source data to be scaled or remapped in very general ways during the copy and for pixel replicated zooming. To effect these operations the source image must be uploaded into the OpenGL server, modified and then downloaded. If source pixels lie in parts of the window which are obscured then the resulting pixels are undefined. The destination pixels must under go the normal fragment processing, including pixel ownership tests (i.e. GID test) when written.

If the image is a straight copy with no data formatting or zooming then GLiNT can handle this directly. Either or both the stencil and depth fields can be moved. The GID field is *never* moved by OpenGL.

If no fast clear operation is enabled for this window then the copy proceeds simply: The source pixel is read and the value inserted into the LBSourceData message. The destination pixel is read (the address is the source address plus the write offset value) and this is inserted into the LBData message. The GID field is then tested before the destination is updated.

The following description is only included here for completeness as the functionality is not implemented in this unit but in the GID, Stencil and Depth Unit.

If a fast clear operation is enabled then the copy is complicated by state of the frame count in both the source and destination pixels as well as the fast clear mode (i.e. whether is applies to the stencil and/or depth fields). Two simple rules control the behaviour:

- If the source pixel's frame count is not equal to the reference value then the field(s) which the fast clear apply are replaced by the clear data for those field(s), otherwise the actual local buffer source data is used.

  If the source pixel's frame count is equal to the reference value then then the local buffer source data is used.

- If the destination pixel's frame count is not equal to the reference value then the field(s) which the fast clear apply are updated irrespective of the current write mask state for those field(s). The remaining fields (if any) are updated using the normal OpenGL rules.

The unit is set up as follows:

ReadDestination enable	1
ReadSource enable	1
LBSourceOffset	As appropriate
Data message	Default

Note that the destination primitive is rasterised.

8.2.5. X Copy Operation

The X copy operation only occurs when an OpenGL window has been moved and X is repairing the local buffer on behalf of OpenGL. The update of the destination pixels are done unconditionally and all the source data fields are copied. No pixel level clipping is needed. The unit is set up as follows:

ReadDestination enable	0
ReadSource enable	1
LBSourceOffset	As appropriate
Data message	Default

Note that the destination primitive is rasterised

8.2.6. Image Download Operation

This assumes the download is done by OpenGL as X has no need to download to the local buffer.

The GID Stencil and Depth Unit will do the necessary merging of the download data if the GID test passes. The unit is set up as follows:

ReadDestination enable	1
ReadSource enable	0
LBSourceOffset	Don't care
Data message	Default

8.2.7. Image Upload Operation

This assumes the upload is done by OpenGL as X has no need to upload from the local buffer.

The data in the local buffer is read and packed into either the LBStencil or LBDepth message as appropriate. These messages fall through all subsequent units to the Host Interface (Out) Unit. The OpenGL specification states that the data is undefined if you read from a pixel not owned by your window.

Care needs to be taken to ensure enough data is returned to the host as it expects so any unit which can convert an Active walk message into a passive one needs to be disabled. The screen scissor test also falls into this category in case a window is part off the screen. Note the user scissor test may need to be disabled as well to ensure this. The unit is set up as follows:

ReadDestination enable	1
ReadSource enable	0
LBSourceOffset	Don't care
Data message	LBStencil or LBDepth depending on which field is wanted.

8.3. Input Messages

8.3.1. External Messages

Local Buffer Register message group		
Tag Mnemonic	Data Field	Description
LBReadMode	See above	General mode of operation of the unit
LBReadFormat	See above	Defines the field widths and positions in the local buffer.
LBSourceOffset	2's Complement offset as a 24 bit number	Difference between destination and source addresses
LBWindowBase	The base address as a 24 bit number	

8.3.2. Internal Messages

This unit reacts to all the messages in the Rasteriser Walk group. See the rasteriser section for details on these messages.

8.4. Output Messages

Local Buffer Data message group		
Tag Mnemonic	Data Field	Description
LBData	See earlier for format.	Holds the data associated with a destination fragment.
LBSourceData	See earlier for format	Holds the data for the source fragment during a copy operation.
LBStencil	Stencil (bits 7-0) FrameCount (bits 15-8) GID (bits 19-16)	
LBDepth	32 bit, zero extended depth	

8.5. Behavioural Model

The behavioural model is best described in two blocks: The input behaviour in calculating addresses and passing them onto the Local Buffer Interface Unit, and the output block which collects the data returned by the Local Buffer Interface Unit and merges it with the normal message stream.

The input block:

```
Wait for input message
{
   switch (on message type)
   {
     case RegisterUpdateGroup:
             Update the identified register;
             break;
     case PrepareToRender:
             Wait for room in the Ra FIFO;
             Notify the Local Buffer Interface Unit to suspend
                reads by setting the SuspendRead signal and
                clocking it into the Ra FIFO with an undefined
                address;
             Wait for room in the M FIFO;
             Copy input message to M FIFO and reset the
                ReadDestination and ReadSource fields in the
                FIFO;
             break;
     case ActiveStep:
             Calculate destination address;
             Calculate source address;
             switch (Read enables from the LBReadMode message)
             {
                case ReadDestination = 0 and ReadSource = 0:
                       break;
                case ReadDestination = 0 and ReadSource = 1:
                       Wait for room in the Ra FIFO;
                       Send source address to Ra FIFO;
                       break;
                case ReadDestination = 1 and ReadSource = 0:
                       Wait for room in the Ra FIFO;
                       Send destination address to Ra FIFO;
                       break;
                case ReadDestination = 1 and ReadSource = 1:
                       Wait for room in the Ra FIFO;
                       Send destination address to the Ra FIFO;
                       Wait for room in the Ra FIFO;
                       Send the source address to the Ra FIFO;
                       break;
             }
             Wait for room in the Wa FIFO;
             Send destination address to the Wa FIFO;
             Wait for room in the M FIFO;
             Copy input message to M FIFO and set up the
                ReadDestination, ReadSource and message type
                fields in the FIFO;
             break;
     default:
             Wait for room in the M FIFO;
             Copy input message to M FIFO and reset the
                ReadDestination and ReadSource fields in the
                FIFO;
             break;
   }
   Flush the input message;
}
```

The output block:

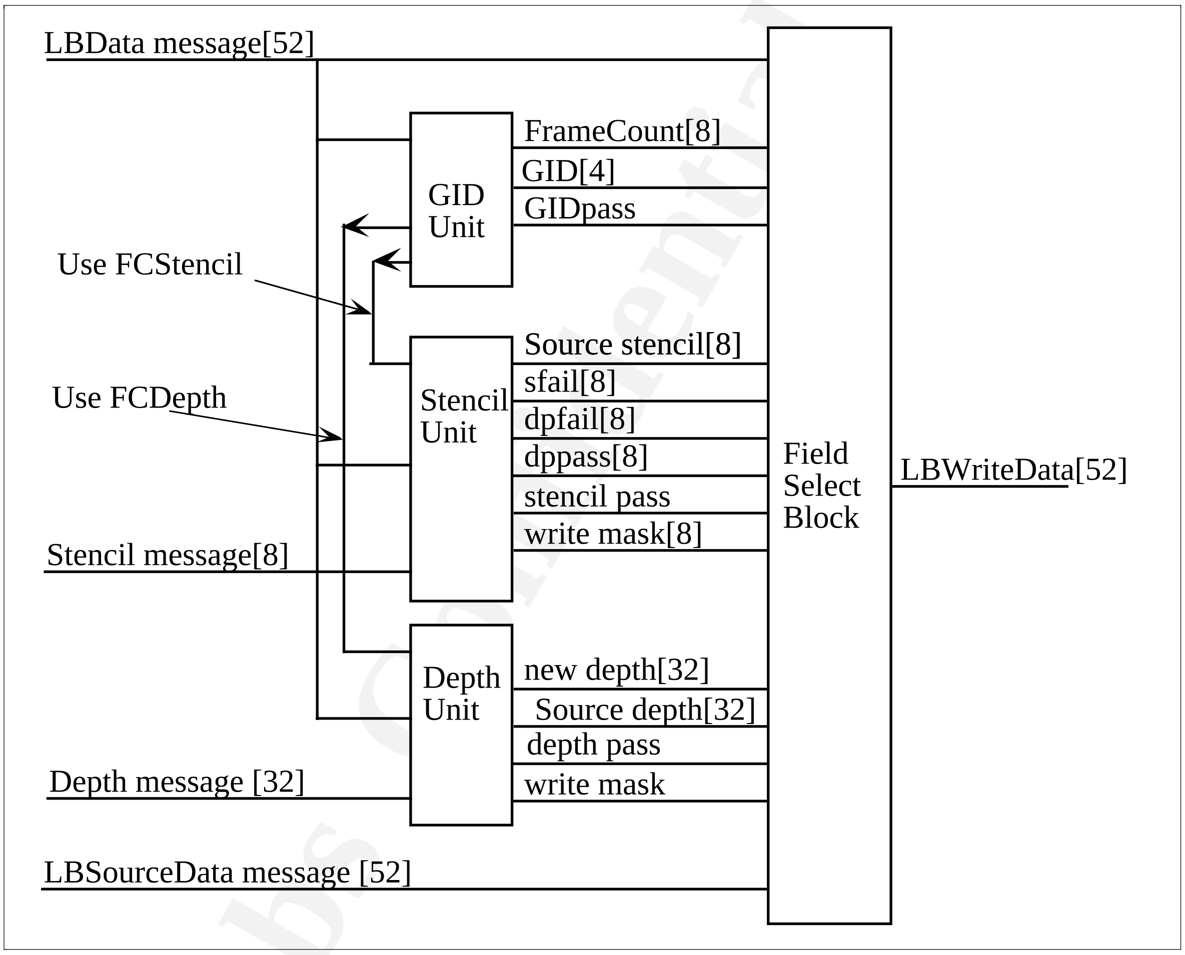
```
Wait for message in the M FIFO
{
   switch (Read enables from the M FIFO)
   {
      case ReadDestination = 0 and ReadSource = 0:
               break;
      case ReadDestination = 0 and ReadSource = 1:
               Wait for room in next unit;
               Wait for data in the Din FIFO;
               switch (on Data Message type in LBReadMode)
               {
                  case Default:
                        Send LBSourceData message to next unit with
                           formatted Din FIFO data in the data
                           field;
                        break;
                  case LBStencil:
                        Send LBStencil message to next unit with
                           the stencil, FrameCount and GID fields
                           from formatted Din FIFO packed into ls 32
                           bits of FIFO data in the data field;
                        break;
                  case LBDepth:
                        Send LBDepth message to next unit with the
                           depth field from formatted Din FIFO data
                           in the ls 32 bits of the data field;
                        break;
               }
               Flush one word from Din FIFO;
               break;
      case ReadDestination = 1 and ReadSource = 0:
               Wait for room in next unit;
               Wait for data in the Din FIFO;
               switch (on Data Message type in LBReadMode)
               {
                  case Default:
                        Send LBData message to next unit with
                           formatted Din FIFO data in the data
                           field;
                        break;
                  case LBStencil:
                        Send LBStencil message to next unit with
                           the stencil, FrameCount and GID fields
                           from formatted Din FIFO packed into ls
                           32 bits of FIFO data in the data field;
                        break;
                  case LBDepth:
                        Send LBDepth message to next unit with the
                           depth field from formatted Din FIFO data
                           in the ls 32 bits of the data field;
                        break;
               }
               Flush one word from Din FIFO;
               break;
      case ReadDestination = 1 and ReadSource = 1:
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            Send LBData message to next unit with formatted Din
               FIFO data in data field;
            Flush one word from Din FIFO;
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            Send LBSourceData message to next unit with
               formatted Din FIFO data in data field;
            Flush one word from Din FIFO;
            break;
   }
   Wait for room in next unit.
   Send message in M FIFO to next unit.
   Flush message from M FIFO;
}
```

9. Graphic ID, Stencil and Depth Unit

9.1. Description

This unit is the amalgamation of the three units: Graphic ID, Stencil and Depth described in the introductory sections of this document. The main reason for combining these units is because the data they all require is read in parallel from the Local Buffer, and the interactions between the stencil and depth tests.

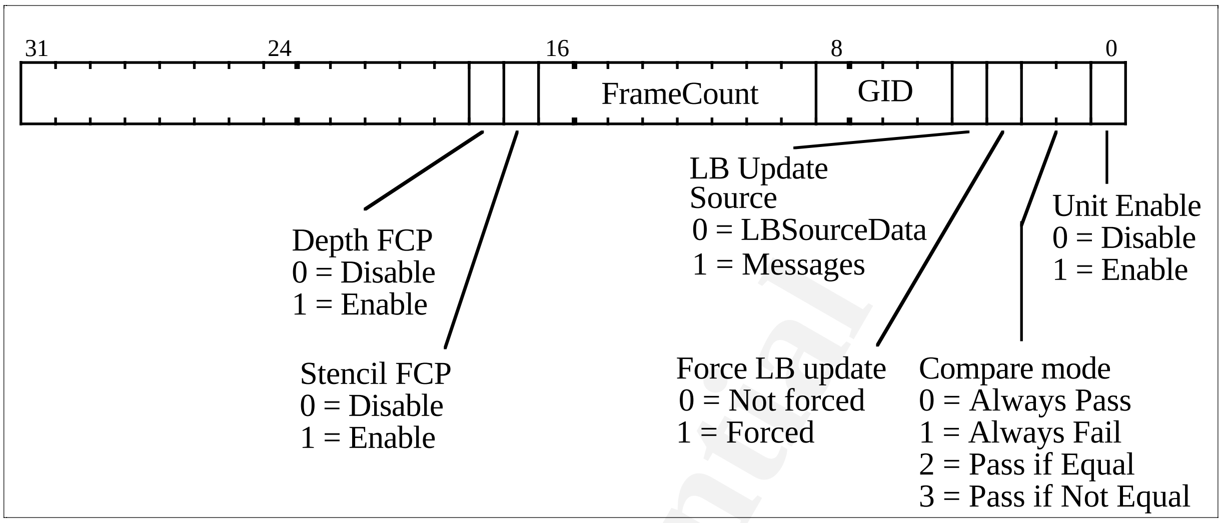
The block diagram of the unit is shown below:



This diagram shows the three units working in parallel, however there are dependencies between the units. These dependencies are handled by the field select block and this selects either the original input values or the values generated by the individual blocks. This is described later.

9.1.1. Graphics ID Unit

The Graphics ID (GID) Unit is controlled by the following fields in the Window message:



This unit simply compares the incoming Graphics ID value (GID) in the LBData Message with the GID and compare mode defined for this context.

If the unit is disabled then it is as if the window test always passes.

The Force LB Update bit, when set overrides all the tests done in the GID, Stencil and Depth units and the per unit enables to force the local buffer to be updated. When this bit is clear any update is conditional on the outcome of the GID, stencil and depth tests. The main use of this bit is during window initialisation or copy. It may also be useful for hardware diagnostics. The data used for the update depends on the setting of the LB Update Source bit.

The FrameCount is an eight bit field which is compared with the FrameCount read from the local buffer. If these are not equal then the fast clear mechanism can be used, however how this is used (if at all) is determined by the Depth FCP and Stencil FCP bits. If these bit(s) are set then the fast clear function is enabled for the corresponding field(s) and this determines where the data 'read' from the local buffer comes from. The following table shows this (FCDepth and FCStencil are values stored in registers in the depth and stencil units respectively):

Depth FCP	FrameCount equal?	Depth Source
0	X	LBData
1	No	FCDepth
1	Yes	LBData

Stencil FCP	FrameCount equal?	Stencil Source
0	X	LBData
1	No	FCStencil
1	Yes	LBData

When the local buffer is updated and either of the Depth FCP or Stencil FCP bits are set the FrameCount field is updated from the FrameCount field in the Window message so subsequent reads use the real data and not the fast clear data.

There is an interaction between the fast clear mechanism and the write masks for each field during a normal update because the write masks are not used during the clear operation (in the OpenGL specification). When the local buffer is going to be updated, and the depth or stencil field is going to be updated with the 'clear value' the write masks must be ignored. The tables describing this are in the Field Select Block section.

The write mask function is done by read-modify-write to the local buffer rather than controlling the write enables to the memories.

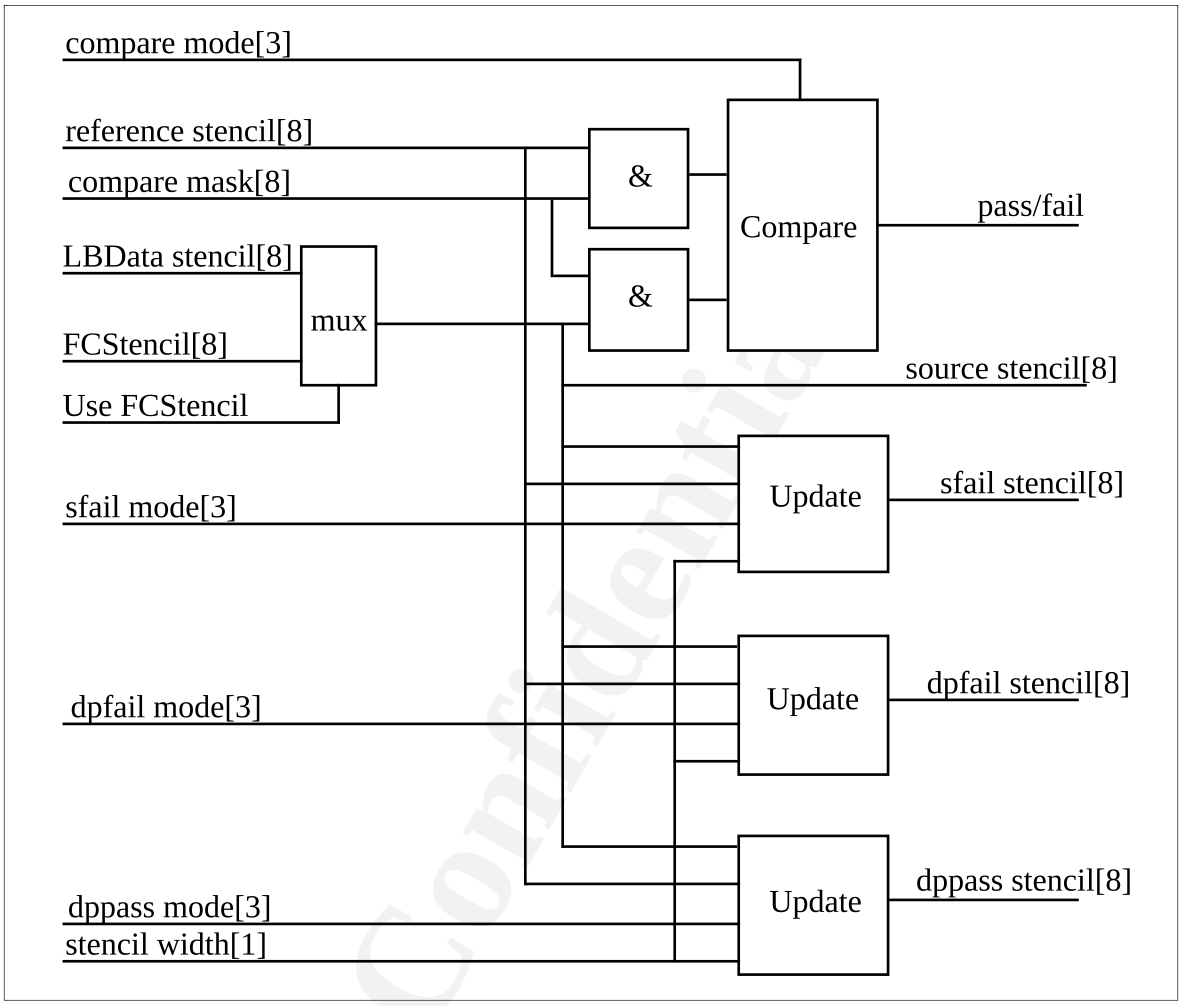
A more detailed explanation of the fast clear planes and their use can be found in the local buffer read unit.

The outputs of this unit to the Field Select Block (excluding the single bit control signals in the Window message) are:

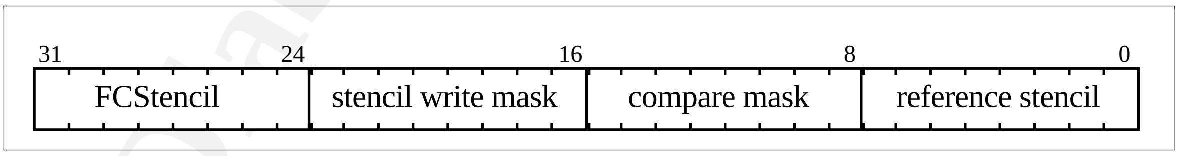
FrameCount[8]	This is the frame count value to write to the local buffer. When the Force LB Update bit is clear: If either the Depth FCP or Stencil FCP bit is set then the FrameCount from the Window message is used otherwise the value in the LBData message for this field is used. When the Force LB Update bit is set: If the LB Update Source is 'Message' then the FrameCount from the Window message is used otherwise the LBSource Data value is used.
GID[4]	When the Force LB Update bit is clear: The GID field in the LBData message is used. When the Force LB Update bit is set: If the LB Update Source is 'Message' then all the GID bits are taken from the GID field in the Window message, otherwise the LBSourceData value is used.
GIDpass	If the GID unit is disabled, or the unit is enabled and the GID compare passes then this signal is asserted, otherwise it is de-asserted.

9.1.2. Stencil Unit

The Stencil Unit is shown below:



The LB stencil value is provided by the LBData message or the Fast Clear stencil value under control of the Fast Clear logic and is subsequently called the Source Stencil. All other stencil values originate in the StencilData message and are held in the data field in the following format:



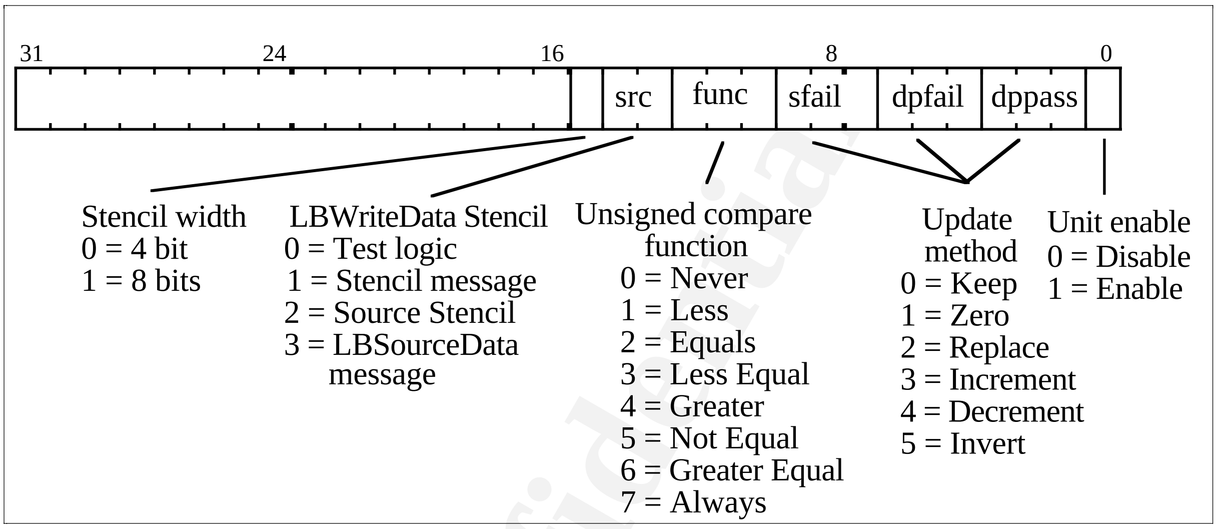
The FCStencil field is the stencil value which the stencil buffer would have been cleared with if the fast clear facility had not been used. This value is now substituted for the local buffer stencil value when the selected frame count is equal and Stencil FCP is enabled.

The write mask is used to write protect bits during an update to the stencil field in the local buffer. When a bit in the mask is set the corresponding bit in the stencil field can be updated. This is assumed to be consistent with the stencil width.

The compare mask is used to eliminate bits from the compare operation. When a bit in the mask is set the corresponding bit in the stencil field and reference value take part in the compare operation. This is assumed to be consistent with the stencil width.

If the stencil field in the local buffer is less than 8 bits the field is right justified and the unused bits (in the most significant bit positions) are set to zero.

The operation of the stencil unit is controlled by the data fields in the StencilMode message. The format of these fields is as follows:



If the stencil unit is disabled then it is as if the stencil test always passes and no modification to the stencil value in the local buffer occurs. The stencil buffer *will* be updated in this situation if fast clear is enabled and the frame count is not equal.

The compare operation compares the stencil reference value against the source stencil value. If the compare function is 'Less' and the result is true then the reference value is less than the source value.

If the stencil unit is enabled and the test fails then the fragment is discarded, or in our terms the active message is turned into a passive one. In this case the stencil buffer is updated[20] with the stencil value calculated by the sfail method, and the depth buffer is not updated.

If the stencil unit is enabled and the test passes then the fragment is kept and depth tested. The value the stencil buffer is updated with depends on the outcome of the depth test. If the depth test passes the dppass method is used otherwise the dpfail method is used. Note that the stencil buffer is updated irrespective of the outcome of the depth test.

[20]If the new stencil value is the same as what is in the local buffer already then the write to memory may not occur, depending on what changes are needed by the other fields.

The update methods are defined:

Method	Result
Keep	Source stencil
Zero	0
Replace	Reference stencil
Increment	Clamp (Source stencil + 1) to $2^{\text{stencil width}} - 1$
Decrement	Clamp (Source stencil -1) to 0
Invert	~Source stencil

All input data to the stencil update operations is masked to the width of the stencil field before the update calculations are done.

All of the above methods are further masked with ($2^{\text{stencil width}} - 1$) to ensure the compare of the LBData and LBWriteData data fields does not show a write is necessary when the only differences are in the unused bits of the stencil field.

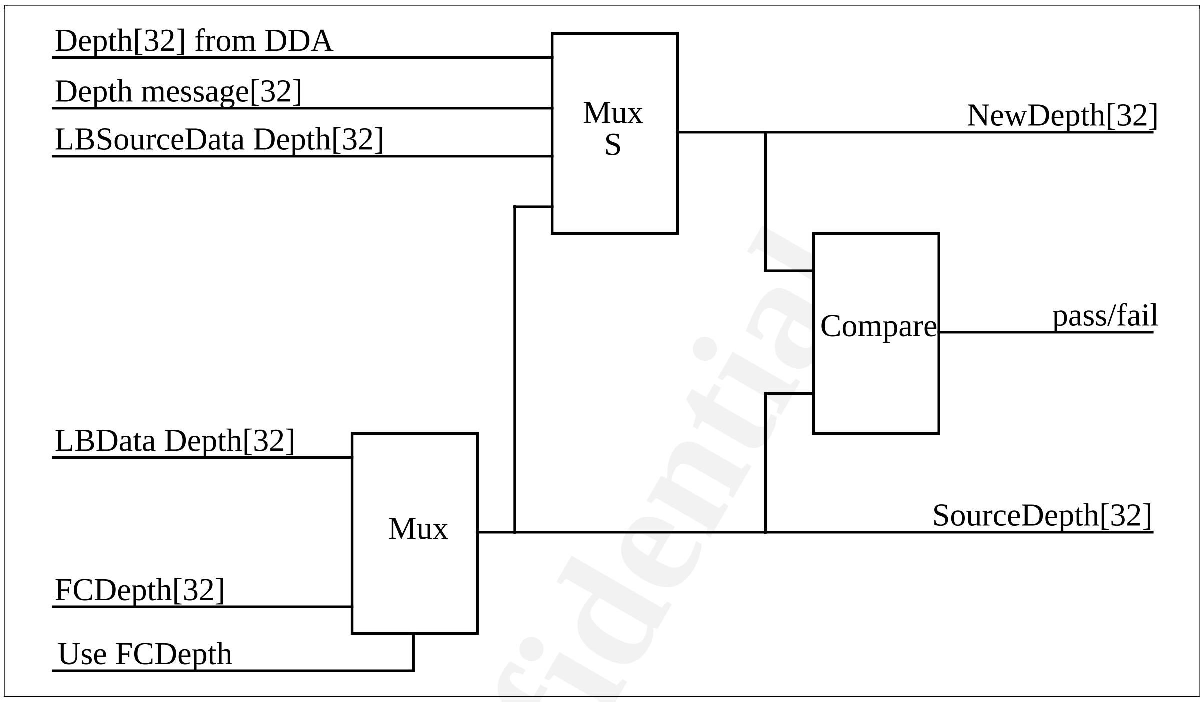
The stencil value used to update the local buffer can be sourced from several places (the multiplexer to do this is in the Field Select Block):

LBWriteData Stencil	Use
Test logic	This is used normally.
Stencil message	This is used in the OpenGL DrawPixels function where the host supplies the stencil values. This is used when a constant stencil value is needed, for example, when clearing the stencil buffer when fast clear planes are not available.
LBSourceData message	This is used in the OpenGL CopyPixels function when the stencil planes are to be copied to the destination.
Source Stencil	This is used by X during the a window copy operation where all the fields in the pixel are moved. This is used in the OpenGL CopyPixels function when the stencil planes in the destination are *not* updated. The stencil data will come either from the LBData message or the FCStencil register depending the state of the Fast Clear modes in operation.

Note the update has been shown as three blocks but could easily be implemented in one block and the update method to this block be selected according to the stencil and depth test results. Doing all three update operations and selecting the appropriate one makes things cleaner and faster (probably a moot point) but at the expense of more gates.

9.1.3. Depth Unit

The data routing and comparison for the depth unit is shown below. The DDA part of the depth unit is covered later:



Some points to note are:

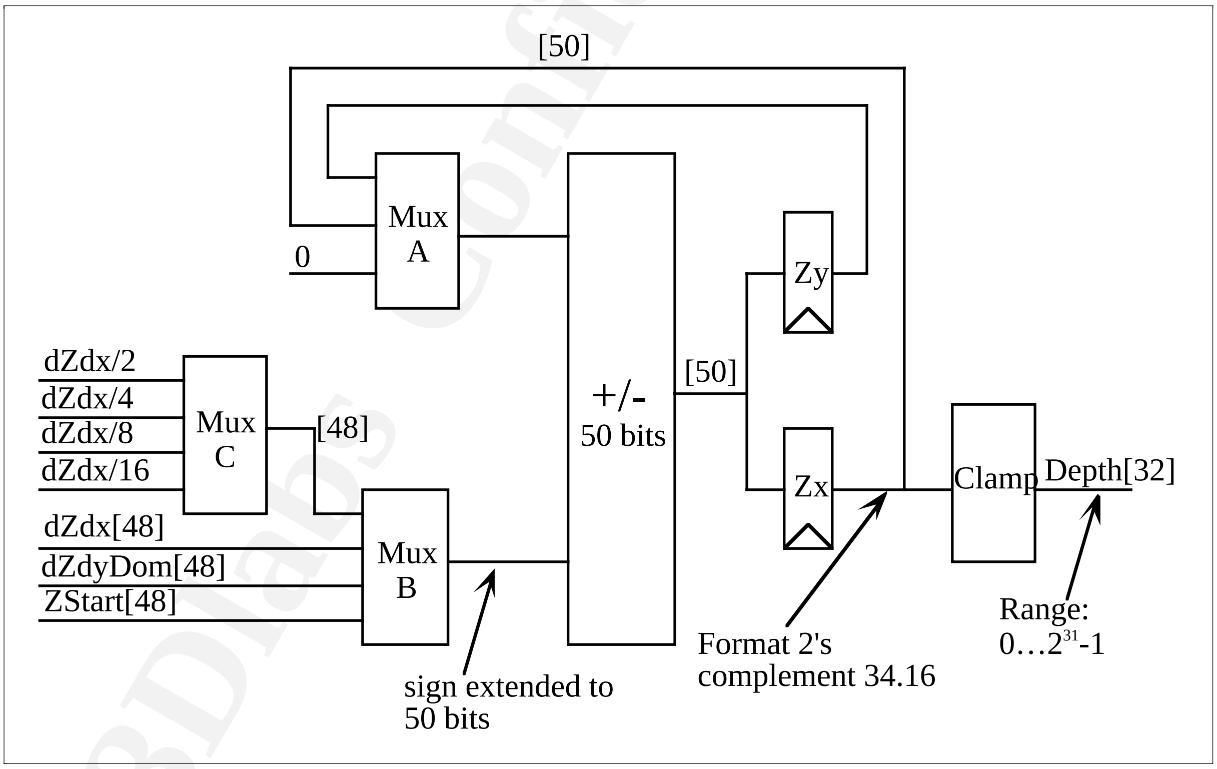
- The compare operation is done unsigned.
- The compare operation compares the new depth value against the source depth value. If the compare function is 'Less' and the result is true then the new value is less than the source value.

- Mux S allows the depth value used for the comparison and potential update to be sourced several places:

Source	Use
DDA	This is used for normal Z buffered 3D rendering.
Depth message	This is used in the OpenGL DrawPixels function where the host supplies the depth values. This is used when a constant depth value is needed, for example, when clearing the depth buffer (when fast clear planes are not available) or 2D rendering where the depth is held constant.
LBSourceData message	This is used in the OpenGL CopyPixels function when the depth planes are to be copied to the destination.
Source Depth	This is used by X during the a window copy operation where all the fields in the pixel are moved. This is used in the OpenGL CopyPixels function when the depth planes in the destination are *not* updated. The depth data will come either from the LBData message of the FCDepth register depending the state of the Fast Clear modes in operation.

The depth DDA unit is responsible for generating the depth information under the direction of the step messages from the Rasteriser Unit. Sub pixel corrections can be applied to correct for an initial start error on a span.

The block diagram of the DDA is:

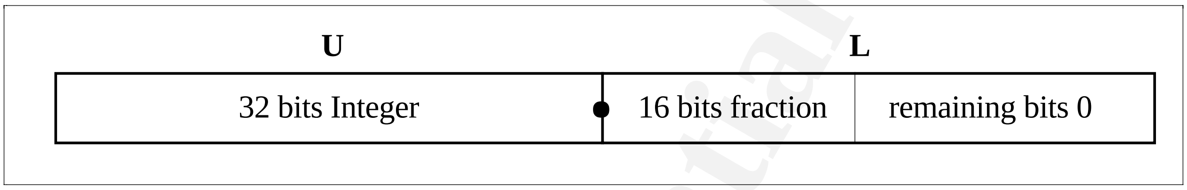


Some notes on the diagram:

- The Zx register holds the depth value as a span is filled. The Zy register holds the value of the depth as we step up the dominant edge of the primitive and is used to seed the span interpolation.

- The adder/subtracter normally works as an adder except when sub pixel compensation is being applied where it can work in either mode depending on the direction of the correction. At first sight an additive correction is all that is necessary because we always correct in the direction that the dx derivative is set up. However the geometry of the primitive has been biased by *nearly a half* (of a pixel) to make the rasterisation rules simpler to implement, while the interpolated parameters are set up with respect to the unbiased geometry.

- The start and derivative number format is 2's complement fixed point integer: 32 bits integer and 16 bits fraction. The integer part is stored in the U register and the fraction part in the L register. The fraction is left justified, with the unused low order bits set to zero. This is shown diagramatically below:



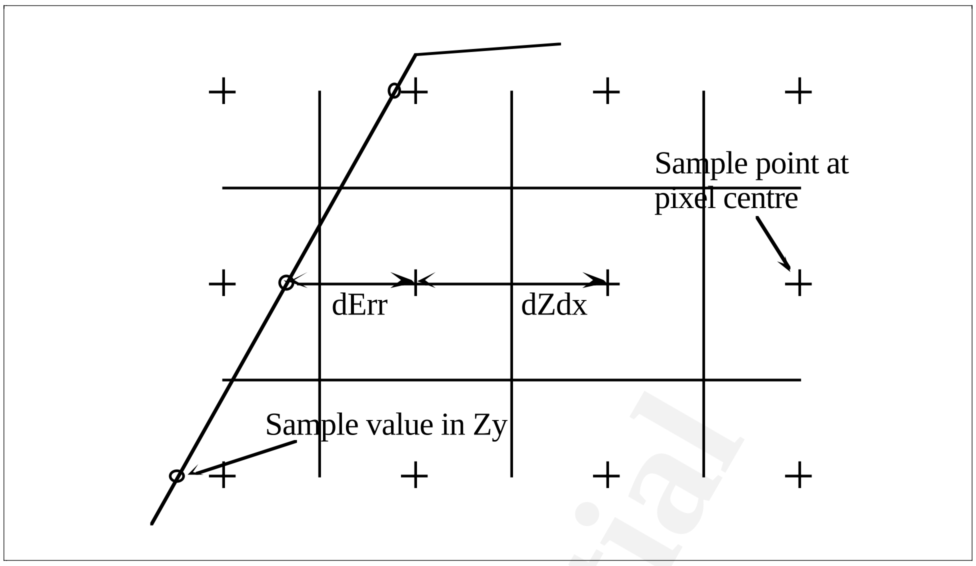
  The internal format has an additional 2 bits to extend the range to give more protection against over or underflow. This guard band is only designed to cope with small amounts of overflow (or underflow) when one or two extra deltas are added during rasterisation, usually as a result of the sample point (i.e. pixel centre) being outside of the boundary of the primitive. If the additional range provided by the guard band is exceeded then wrap around will occur and an abrupt change in the colour will be visible (the clamp logic cannot prevent this). This situation will only occur if the delta values are set up incorrectly for the number of iterations the DDA works through, i.e. the gradient is specified as being steeper than it really is.

- The Clamp logic prevents the interpolated value wrapping around (when the depth value is extracted) when overflowing or underflowing occurs during interpolation.

  This should not be necessary if the derivatives, etc. are set up correctly and sub pixel corrections are done at the start of every span. However we are working with finite precision numbers so slight rounding errors will occur.

  The clamp logic is simply a multiplexer with selects either the result, zero or 2^{31}-1 based on comparing the result against 2^{31}-1 and the sign bit.

- Multiplexer C is used to implement the sub pixel correction. The need for the sub pixel correction is easily shown in the following diagram:



The start value has already been compensated to move it to the centre of a pixel in the Y directions. This only needs to be done once per triangle and is done in software. As the DDA is stepped in Y along the dominant edge the value of the interpolated parameter is calculated at the points marked by circles, whereas the value is required in the pixel's centre.

If dErr is the distance the edge is away from the pixel's centre (must be < 1) and dZdx is the change in Z for unit change in x then the correct value at the first sample point is:

$$Zx = Zy + dErr * dZdx$$

The distance dErr is sent by the rasteriser in the SubPixelCorrection message and whenever this message is received the above equation is implemented. The SubPixelCorrection message allows for 4 bits of resolution of dErr. The correction is applied to a resolution of 1/16 of the dx derivative. The multiply operation is too expensive to do using a multiplier so it is implemented by addition of the partial products. For a positive correction this means adding in dZdx/2 when bit 3 is set, dZdx/4 when bit 2 is set, dZdx/8 when bit 1 is set and dZdx/16 when bit 0 is set. For a negative correction the corresponding partial products are subtracted.

The value of dErr is in sign magnitude format with bit 4 holding the sign (0 = positive), and bit 3 representing 2^{-1}, bit 2 representing 2^{-2}, etc..

The division of dZdx is always done by a power of two so is just an arithmetic right shift (to retain the sign of the dZdx).

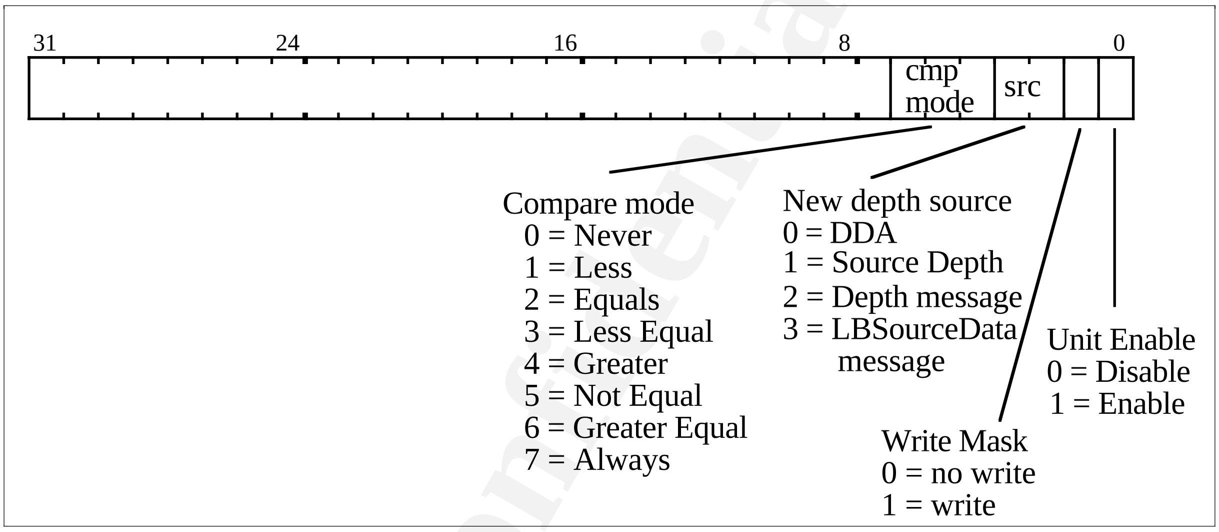
The correction will add one cycle for every bit set in dErr so worst case will take 4 cycles per scanline, but on average 2 cycles. This time is hidden under the page break time allowed for every scanline so will not impact performance.

- The depth DDA responds to rasteriser messages as follows:

Message	Mux A output	Mux B output	Register Zx	Register Zy
PrepareToRender	zero	ZStart	Load	Load
StepX	Zx	dZdx	Load	Hold
StepYDomEdge	Zy	dZdyDom	Load	Load
SubPixelCorrection	Zx	Mux C o/p	Load	Hold

- If depth unit is disabled then the DDA is not updated on active step or PrepareToRender messages.

The control of the depth unit is governed by the DepthMode message. The format of the data field for this message is:



The single bit write mask enables or disables the updating of all the depth bits in the local buffer. The write masking is done by read-modify-write operations to the local buffer and not by controlling the write enables to the memories.

If the unit is disabled then this is as if the depth test passed, but no update of the depth field occurs. The depth buffer *will* be updated in this situation if fast clear is enabled and the frame count is not equal.

If the unit is enabled and the stencil test has passed then the depth compare is done and if the result is pass then depth buffer is updated, otherwise it is not.

9.1.4. Field Select Block

This block is responsible for:

- Building up the value to write back to the local buffer from the test results, fast clear status, write enables, etc.
- Using the test results and unit enables to determine if an active step message should be converted into a passive one.
- Comparing the LBData with the LBWriteData . If they are different and the test results and enables are suitably set then allow a LBWriteData message to be sent, otherwise the LBCancelWrite message is sent.

The Local Buffer Write Unit uses the LBWriteData message to provide the data to the Local Buffer Interface Unit (it already has the address) to issue a write request.

The LBCancelWrite message is used cancel the write address already queued up in the Local Buffer Interface Unit.

The following Boolean expressions shows whether an active walk message is converted to a passive one (i.e. the fragment is to be discarded) based on what is enabled and the test results:

```
if (GID enable & GID result = Fail)
   GID pass = False;
else
   GID pass = True;

if (Stencil enable & Stencil result = Fail)
   Stencil pass = False;
else
   Stencil pass = True;

if (Depth enable & Depth result = Fail)
   Depth pass = False;
else
   Depth pass = True;

if (Force LB Update | (GID pass & Stencil pass & Depth pass))
   Keep ActiveStep as ActiveStep
else
   Change ActiveStep to PassiveStep
```

The following Boolean expressions shows whether a LBWriteData or LBCancelWrite message is sent based on what is enabled, the test results and if the write would actually change the data in memory.

```
if (GID enable & GID result = Fail)
   GID pass = False;
else
   GID pass = True;

if (Depth enable & Depth result = Fail)
   Depth pass = False;
else
   Depth pass = True;

if (Force LB Update
   | (GID pass & Stencil Enable & (LBWriteData != LBData))
   | (GID pass & Depth pass & (LBWriteData != LBData)))
   Send LBWriteData
else
   Send LBCancelWrite
```

This table shows what value of depth should be inserted into the depth field in the LBWriteData message:

Force LB Update	Depth Unit Enable	Stencil test result	Depth FCP	Frame Count equal?	Depth Write Mask	Depth test result	Depth value
1	X	X	X	X	X	X	Depth message or LBSourceData depending on the LB Update Source bit.
0	0	X	X	X	X	X	Source Depth
0	1	Fail	X	X	X	X	Source Depth
0	1	Pass	X	X	1	Pass	New Depth
0	1	Pass	0	X	0	Pass	Source Depth (= LBData)
0	1	Pass	0	X	X	Fail	Source Depth (= LBData)
0	1	Pass	1	No	X	Fail	Source Depth (= FCDepth)
0	1	Pass	1	No	0	Pass	Source Depth (= FCDepth)
0	1	Pass	1	No	1	Pass	New Depth
0	1	Pass	1	Yes	X	Fail	Source Depth (= LBData)
0	1	Pass	1	Yes	0	Pass	Source Depth (= LBData)
0	1	Pass	1	Yes	1	Pass	New Depth

This table shows what value of stencil should be inserted into the stencil field in the LBWriteData message:

Force LB Update	Stencil Unit Enable	Stencil FCP	Frame Count equal?	Stencil Write Mask[8]	Output stencil (applied on a per bit basis when the write mask is not don't care)
1	X	X	X	X	Stencil message or LBSourceData depending on the LB Update Source bit.
0	0	X	X	X	Source Stencil
0	1	0	X	0	Source Stencil (= LBData)
0	1	0	X	1	Depends on the setting of the src field in StencilMode message. If set to test logic then also depends on stencil and depth test resells and the update methods. The Keep, Increment, Decrement and Invert methods will use the Source stencil (= LBData) in this case.
0	1	1	No	0	Source Stencil (= FCStencil)
0	1	1	No	1	Depends on the setting of the src field in StencilMode message. If set to test logic then also depends on stencil and depth test resells and the update methods. The Keep, Increment, Decrement and Invert methods will use the Source stencil (= FCStencil) in this case.
0	1	1	Yes	0	Source Stencil (= LBData)
0	1	1	Yes	1	Depends on the setting of the src field in StencilMode message. If set to test logic then also depends on stencil and depth test resells and the update methods. The Keep, Increment, Decrement and Invert methods will use the Source stencil (= LBData) in this case.

This table shows what stencil value is used as a function of the stencil and depth test results:

Stencil test	Depth test	Test stencil value
Fail	X	sfail output
Pass	Fail	dpfail output
Pass	Pass	dppass output

9.2. Operating Modes

This section looks at how to do common operation in OpenGL and X.

9.2.1. OpenGL Rendering Operations

The Stencil width, compare and update are set up as appropriate to the rendering operation. The Depth compare and write mask are also assumed set up as appropriate to the rendering operation.

The Local Buffer Read Unit is set up to do one read. The write is conditional on the outcome of the tests and writes are enabled in the Local Buffer Write Unit.

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	As appropriate
	LBWriteData Stencil	Test Logic
DepthMode	Unit Enable	As appropriate
	Depth Source	DDA for variable Z, Depth message for constant Z

9.2.2. X Rendering Operations

The local buffer is not used in X rendering so the number of reads set up in the Local Buffer Read Unit are set to 0, the writes to the local buffer are disabled in the Local Buffer Write Unit.

Window	Unit Enable	Don't care
	GID	Don't care
	Force LB Update	No force
	Compare mode	Don't care
StencilMode	Unit Enable	Don't care
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	Don't care
	Depth Source	Don't care

9.2.3. Window Initialisation

Window initialisation is only concerned with setting the GID field in the local buffer. The contents of the Depth and Stencil fields are largely irrelevant as OpenGL will initialise these later on and in general X will just update them with some default value (derived from the Stencil and Depth messages). Keeping the original Depth and Stencil values is possible but requires the local buffer to be read as well as written.

Window	Unit Enable	Disabled
	GID	Value for the new window
	Force LB Update	Force update
	LB Update Source	Message data
	Compare mode	Don't care
StencilMode	Unit Enable	Disabled
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	Disabled
	Depth Source	Don't care

9.2.4. OpenGL Copy Operation

All OpenGL copy operations must do two reads: the first read is the source data and the second read is the destination data. The source and destination are merged together as shown below. The destination read is always necessary to establish pixel ownership.

To copy just the stencil field:

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	Enabled
	LBWriteData Stencil	LBSourceData
DepthMode	Unit Enable	Disabled
	Depth Source	Don't care

To copy just the depth field:

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	Disabled
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	Enabled
	Depth Compare	Always Pass
	Depth Source	LBSourceData

To copy both the depth and stencil fields:

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	Enabled
	LBWriteData Stencil	LBSourceData
DepthMode	Unit Enable	Disabled
	Depth Compare	Always Pass
	Depth Source	LBSourceData

9.2.5. X Copy Operation

The X copy operation only needs to do one read and one write and is used to move the local buffer region associated with the window on the screen which is being moved.

Window	Unit Enable	Disabled
	GID	Don't care
	Force LB Update	Force update
	LB Update Source	LBSourceData
	Compare mode	Don't care
StencilMode	Unit Enable	Disabled
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	Disabled
	Depth Source	Don't care

9.2.6. Image Download

This assumes the download is done by OpenGL and X has no need to download to the local buffer.

To download to the stencil field:

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	As appropriate
	LBWriteData Stencil	Test Logic
DepthMode	Unit Enabled	Disabled
	Depth Source	Don't care

To download to the depth field

Window	Unit Enable	Enabled
	GID	As appropriate for this window
	Force LB Update	No force
	Compare mode	Pass if Equal
StencilMode	Unit Enable	Disabled
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	As appropriate
	Depth Compare	Always Pass
	Depth Source	Depth message

9.2.7. Image Upload

For image upload this unit is totally disabled.

Window	Unit Enable	Disabled
	GID	Don't care
	Force LB Update	No force
	Compare mode	Don't care
StencilMode	Unit Enable	Disabled
	LBWriteData Stencil	Don't care
DepthMode	Unit Enable	Disabled
	Depth Source	Don't care

9.3. Input Messages

9.3.1. External Messages

GID/Stencil/Depth Register Update Messages

Tag Mnemonic	Data Field	Description
Window	see above	Window info for context
StencilData	see above	Stencil data for context
Stencil Mode	see above	Stencil mode for context
Stencil	ls 8 bits hold stencil	Externally sourced data
DepthMode	see above	Depth info for context
Depth	32 bits	Externally sourced data, left justified and unused bits set to zero.
ZStartU	Integer part	2's complement Depth start value
ZStartL	Fractional part	
dZdxU	Integer part	2's complement Depth derivative unit X
dZdxL	Fractional part	
dZdyDomU	Integer part	2's complement Depth derivative unit Y, dominant edge, or along a line.

dZdyDomL	Fractional part	
FastClearDepth	32 bits	FCDepth value, left justified and unused bits set to zero.

9.3.2. Internal Messages

This unit responds to the LBData and LBSourceData message. See the Local Buffer Read Unit for details.

This unit reacts to all the messages in the Rasteriser Walk message group. See the rasteriser section for more details on these messages.

9.4. Output Messages

Local Buffer Data Modified message group		
Tag Mnemonic	Data Field	Description
LBCancelWrite	Not used	Cancels the outstanding write for this fragment.
LBWriteData	Same format as the LBData message	Holds the new data associated with a fragment. This data is to be written into the local buffer.

Also Active Walk messages can be converted into Passive ones.

9.5. Behavioural Model

```
Wait for input message
{
  switch (on message type)
  {
    case RegisterUpdateGroup:
    case LBData:
    case LBSourceData:
            Update the identified register;
            break;
    case PrepareToRender message:
            if (depth is enabled)
              Update DDA from the ZStart registers;
            Forward the input message on to the next block;
            break;
    case SubPixelCorrection:
          if (depth is enabled)
          {
            if (sign of dxErr is positive)
              For each bit set in dxErr add the
                corresponding fraction of dZdx to Zx
                and update Zx.
            else
              For each bit set in dxErr subtract the
                corresponding fraction of dZdx from Zx
                and update Zx.
          }
          Forward the input message on to the next block;
          break;
    case PassiveStep:
            if (depth is enabled)
              Update DDA based on Step type;
            Forward the input message on to the next block;
            break;
    case ActiveStep:
            Do the GID, Stencil and Depth tests and set up the
              LBWriteData data field;
            Wait for room in receiving message queue;
            if (local buffer need to be updated)
            {
              // see text for what constitutes an update
              Send LBWriteData message to next block;
            }
            else
            {
              Send LBCancelWrite message to next block;
            }
            Wait for room in receiving queue;
            if (tests result in fragment being discarded)
            {
              // see text for what constitutes a discard
              Send passive version of Step message to next
                block;
            }
            else
            {
               Send the active step message to next block;
            }
            if (depth is enabled)
               Update the DDA based on the Step type;
            break;
     default:
            Forward the input message to the next block;
            break;
   }
   Flush the input message;
}
```

10. Local Buffer Write Unit

10.1. Description

The Local Buffer Write Unit issues write requests to the Local Buffer Interface Unit. More details regarding the interface to the Local Buffer Interface Unit can be found in the Local Buffer Read Unit.

This unit responds to very few messages:

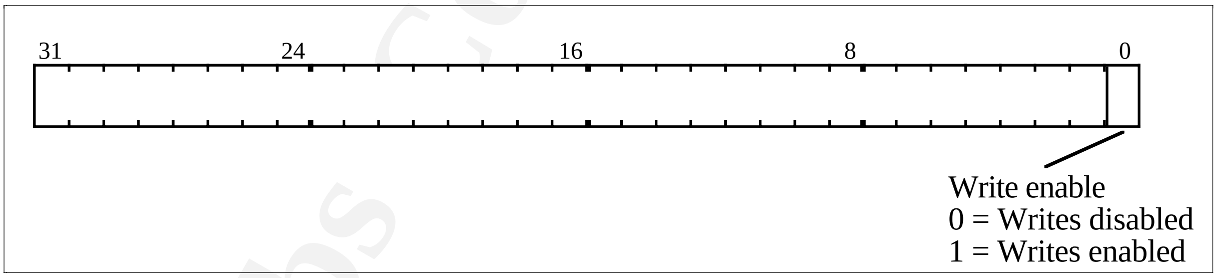
- On receiving a LBWriteData message the write enable bit is checked. If it is set then the data is formatted and passed on to the Local Buffer Interface Unit and a write request is issued. The address for this write is already queued up in the Local Buffer Interface Unit. If writes are disabled then the pending write operation is cancelled so the write address stored in the Local Buffer Interface Unit is discarded.

- On receiving a LBCancelWrite message the outstanding write in the Local Buffer Interface Unit is cancelled.

- On receiving a PrepareToRender message the Local Buffer Interface Unit is notified that it can, after all queued writes are complete, resume doing reads. See the Local Buffer Read Unit for detail on what this is used for.

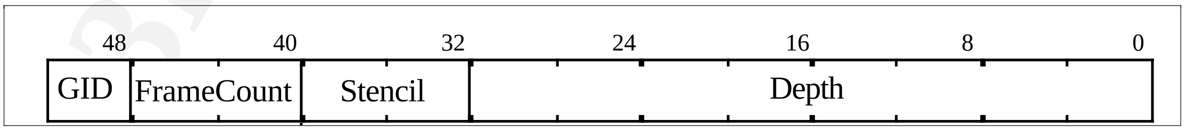
- On receiving a Sync message this unit waits for all outstanding data in the Local Buffer Interface Unit to have been 'written', as shown by the LBWrComplete signal being asserted. Once this has occurred then the Sync message is forwarded onto the next unit. This ensures that all the work associated with a primitive has been completed before the host is informed.

The LBWriteMode message has the following data field format:



10.1.1. Data Format

Different data formats are supported in the local buffer to allow the width of the memory system to be varied to suite the cost of the system and what an application needs. The LBWriteData format is:



and this needs to be converted in to the local buffer format.

The local buffer memory can be from 16 bits (assuming a depth buffer is always needed) to 52 bits wide in steps of 4 bits. The four fields supported in the local buffer, their allowed lengths and positions are shown in the following table:

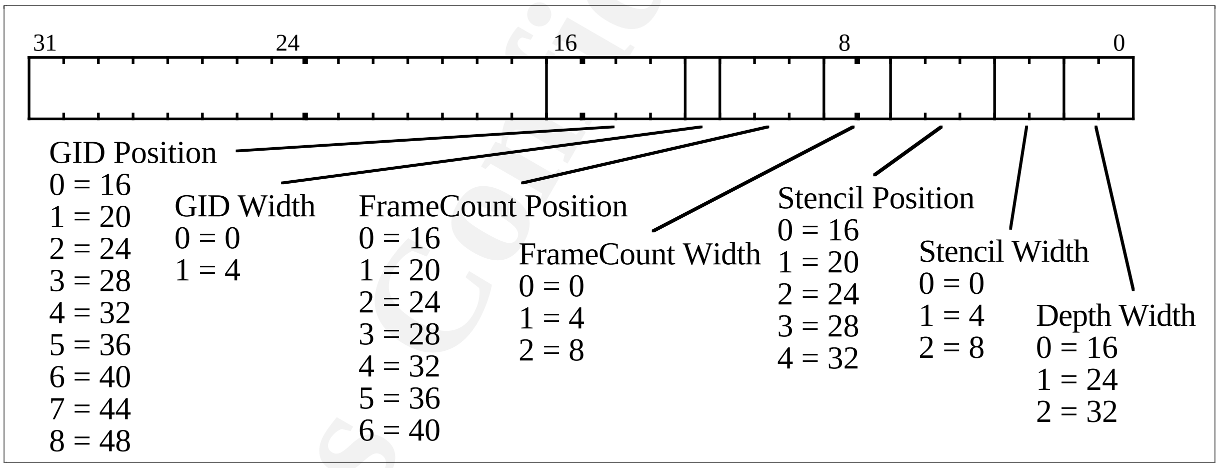
Field	Lengths	Start positions
Depth	16, 24, 32	0
Stencil	0, 4, 8	16, 20, 24, 28, 32
FrameCount	0, 4, 8	16, 20, 24, 28, 32, 36, 40
GID	0, 4	16, 20, 24, 28, 32, 36, 40, 44, 48

The anticipated order of the fields is as shown with the depth field at the least significant end and GID field at the most significant end.

For the GID, FrameCount and Stencil fields when the width in the LBWriteData message are greater than the width in the local buffer the least significant bits of the fields in the LBWriteData message are used.

The Depth field value in the LBWriteData message is clamped to 2^n-1, where *n* is the width of the depth field (16, 24 or 32) in the local buffer. The resulting value is used to update the depth value in the local buffer.

The format of the local buffer is specified in the LBWriteFormat message and the data field is defined as follows:



If the LBWriteFormat is erroneously set up such that two or more fields select the same bits in the local buffer then the hightest priority field determins the value of the local buffer bits. The priority of the fields is (from heighest to lowest) depth, stencil, frame count and then GID.

All unspecified bits in the remainder of the 52 bits of local buffer width are set to zero.

10.2. Input Messages

10.2.1. External Messages

Local Buffer Data Write message group		
Tag Mnemonic	Data Field	Description
LBWriteFormat	See above	Defines the size and position of the fields in the local buffer.
LBWriteMode	See above	Controls the operation of the local buffer write unit.

10.2.2. Internal Messages

This unit only responds to the PrepareToRender message from the rasteriser unit, the LBWriteData and LBCancelWrite messages from the GID, Stencil and Depth Units, and the Sync message from the host. See these sections for more details.

10.3. Output Messages

No new messages are generated.

10.4. Behavioural Model

```
Wait for input message
{
   switch (message type)
   {
      case LBWriteMode:
      case LBWriteFormat:
                  Update the specified register;
                  break;
      case LBCancelWrite:
                  Wait for room in the Write data FIFO in the
                     Local Buffer Interface Unit;
                     Post a Discard Write request and undefined
                        data to the Local Buffer Interface Unit
                        Write data FIFO;
                  break;
      case LBWriteData:
                  Wait for room in the Write data FIFO in the
                     Local Buffer Interface Unit;
                  if (writes are enabled)
                  {
                     Post a Write Data request and formatted data
                        to the Local Buffer Interface Unit FIFO;
                  }
                  else
                  {
                     Post a Discard Write request and undefined
                        data to the Local Buffer Interface Unit
                        Write data FIFO;
                  }
                  break;
      case PrepareToRender:
                  Wait for room in the Write data FIFO in the
                     Local Buffer Interface Unit;
                  Post a Resume Read request and undefined
                     data to the Local Buffer Interface Unit
                     Write data FIFO;
                  Wait for room in receiving message queue;
                  Copy input message to receiving queue;
                  break;
      case Sync:
                  Wait for the WrComplete signal in the
                     Local Buffer Interface Unit to be asserted;
                  Wait for room in receiving message queue;
                  Copy input message to receiving queue;
         default:
                  Wait for room in receiving message queue;
                  Copy input message to receiving queue;
                  break;
   }
   Flush the input message;
}
```

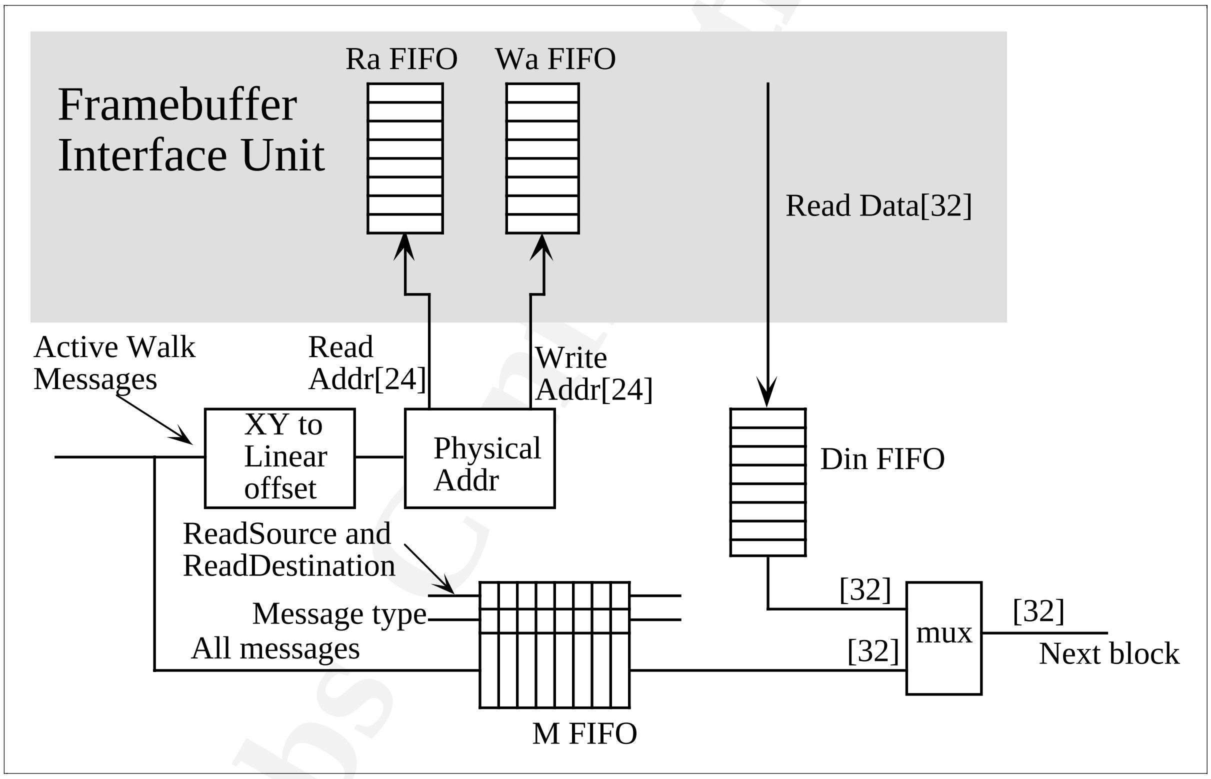
11. Framebuffer Read Unit

11.1. Description

The Framebuffer Read Unit is responsible for:

- calculating the read address(es) of the fragment in the framebuffer,
- calculate the write address of the fragment in the framebuffer,
- dispatch the addresses to the Framebuffer Interface Unit,
- receive the framebuffer data some time later.

A simple diagram of the Framebuffer Read Unit follows:



The actual reading of the memory and request arbitration is done in the Framebuffer Interface Unit.

The Framebuffer Unit, in response to active walk messages will calculate zero, one or two read addresses and one write address[21]. These will be sent to the Framebuffer Interface Unit, freeing the Framebuffer Unit to wait and act on a new active walk message. All messages will be delayed in the M FIFO so the read data can be matched up with the active message which generated the request. Each message stored in the M FIFO has the number of reads

[21] A write address is always generated. When no write is needed, for example in Image upload, the write is disabled in the Framebuffer Write Unit. A disable write facility has not been included here to prevent a possible deadlock occuring if the write enable control is not consistent between the two units.

associated with it (0, 1 or 2) so this amount of data can be read off the Din FIFO and passed onto the next block before the message is sent.

When a FastBlockWrite message is detected the write address is calculated as normal and entered in the Wa FIFO together with the write mode being set to 'block'. No read addresses are entered into the Ra FIFO.

An active walk message generates the following message stream:

first:	FBData message	(if present)
	FBSourceData message	(if present)
last:	Active Walk message	

The order of messages is important as the last message (Active Walk message) prompts the unit farther downstream into action so all the data they need must be present.

Both the Graphics context and X context are identical.

11.1.1. Framebuffer Interface Unit

This section does not describe the Framebuffer Interface Unit (FBIU) but defines the expected behaviour of it and how it is interfaced to the Framebuffer Read Unit (FBRU) and the Framebuffer Write Unit (FBWU).

The following signals are present between the units:

Name	Width	Source	Description
FBReadAddr	24	FBRU	The read address of the Framebuffer data. The whole width (max. 32 bits) is selected by this address. The address is sent to the Ra FIFO.
FBReadEnable	1	FBRU	When active instigate a read from this address.
FBSuspendRead	1	FBRU	When active suspend on this read. This bit is sent in the Ra FIFO.
FBWriteAddr	24	FBRU	The write address of the Framebuffer data. The whole width (max. 32 bits) is selected by this address. This address is sent to the Wa FIFO.
FBWriteAddrEnable	1	FBRU	When active clocks the WriteAddr and WriteMode into the Wa FIFO.
FBRdAddrFull	1	FBIU	When active the Framebuffer Interface Unit cannot accept any more read addresses and the Framebuffer Read Unit must stall.
FBWrAddrFull	1	FBIU	When active the Framebuffer Interface Unit cannot accept any more write addresses and the Framebuffer Read Unit must stall.
FBReadData	32	FBIU	The Framebuffer data returned from a read operation.
FBRdDataFull	1	FBRU	When active the Framebuffer Read Unit cannot accept any more read data and the Framebuffer Interface Unit must stall, or switch to doing writes.
FBReadDataValid	1	FBIU	When active qualifies the read data to be valid.
FBWriteData	32	FBWU	The data to be written to the Framebuffer.
FBWriteCmd	3	FBWU	An encoded bit field which specifies the required write action: Bits 0,1, 2 0 = Write data 1 = Discard write 2 = Load Colour register (in VRAM) 3 = Load Write Mask register (in VRAM) 4 = Resume reading once write requests have been satisfied. 5 = Block write
FBWriteDataEnable	1	FBWU	When active the WriteData and WriteCmd is clocked into the FIFO.
FBWrDataFull	1	FBIU	When active the Framebuffer Interface Unit cannot accept any more write data and the Framebuffer Write Unit must stall.
FBWrComplete	1	FBIU	Active when the write data FIFO is empty and all outstanding writes completed.

Some general rules and observations:

1. Multiple read and write requests can be outstanding to give the opportunity to group reads and writes together to make better use of the page structure in the memory.

2. At the interface level the read data must be returned in the same order the read addresses were presented in. The actual memory reads can be executed in any order provided the ordering at the interface is maintained.

3. The write data will be given to the Framebuffer Interface Unit an arbitrary number of cycles after the write address, and in fact there will have been other write addresses in

the mean time. The write data will be sent in the same order the write addresses were. The write may be identified to be discarded if there is no need to change the data in the memory. The actual memory writes can be executed in any order once the address and data have both been received.

4. There is no requirement for the reads and writes to occur in the order they were issued in, however a later read operation to an outstanding write address must return the new write data rather than the pre-write data. Catching the occurrence of this in hardware is very expensive, however the special nature of the rasterisation process[22] can be used to easily flag any potential situations where this might occur. Each read address is qualified by a Suspend Reads flag, which is nomally reset. A PrepareToRender message will cause a write to the Ra FIFO with this flag set and an undefined address. On detecting this flag set the Framebuffer Interface Unit will not start this read, or any subsequent ones received until it detects the Resume Reads signal being set. The Resume Reads signal is issued by the Framebuffer Write Unit once all the writes for the old primitive have been issued and only allows the reads to resume once all the writes have been completed. It is the PrepareToRender message which activates these flags.

5. Read will always have priority over writes, except in the case where a read would break page for which there are one or more outstanding writes queued up.

6. There needs to be sufficient (and identical) buffering in both the interface unit and core unit to amortise the cost of page breaks when the read and write addresses are in different pages as a result of a copy operation. This buffering will also help with (5). How deep should the buffers be? Ideally the deeper the better, but some practical limit, such as 8 is a good starting point. The cost associated with this figure needs to looked at and this is some what helped by elimination of the write pending logic using the method presented in (4).

11.1.2. XY to linear offset conversion

Calculating the Y offset in a window from the XY pixel offset is done by the equation:

offset = Y * W   W = screen width

If any screen width is allowed then a true multiplier is needed with the attendant gate cost. Very few screen widths need to be supported so the multiplier can be reduced to some shifters and adders. A binary multiplier with inputs Y and W works as follows: A partial product (Y << n) is generated for every bit *n* in W which is set. The sum of the partial products gives the product of Y and W.

By limiting the number of partial products, and by implication the allowable screen widths, reduces the size of the multiplier significantly. The choice of how many partial products and which combinations to support is important to give flexibility without a large gate count. If the exact screen width required is not supported then it is always possible to use the next highest one which is, albeit at the cost of an unused strip in memory.

[22]The rasterisation of a primitive guarentees that a pixel is only visited once so the only time when stale data could be read is when the rasterisation of the next primitive starts, and writes from the previous primitive are outstanding.

Three partial products can be specified and they can be selected from the group:

Partial Product Code	Add into product	Shift amount
0	0	0
1	Y * 32	Y << 5
2	Y * 64	Y << 6
3	Y * 128	Y << 7
4	Y * 256	Y << 8
5	Y * 512	Y << 9
6	Y * 1024	Y << 10
7	Y * 2048	Y << 11

Some common screen width and the corresponding partial product codes are:

Screen width	Partial Product 0	Partial Product 1	Partial Product 2	Product
640	5	3	0	Y * (512 + 128 + 0)
1024	6	0	0	Y * (1024 + 0 + 0)
1152	6	3	0	Y * (1024 + 128 + 0)
1280	6	4	0	Y * (1024 + 256 + 0)
1600	6	5	2	Y * (1024 + 512 + 64)

The screen widths supported are:

0	32	64	96	128
160	192	224	256	288
320	352	384	416	448
512	544	576	608	640
672	704	768	800	832
896	1024	1056	1088	1120
1152	1184	1216	1280	1312
1344	1408	1536	1568	1600
1664	1792	2048	2080	2112
2144	2176	2208	2240	2304
2336	2368	2432	2560	2592
2624	2688	2816	3072	3104
3136	3200	3328	3584	4096
4128	4160	4224	4352	4608
5120	6144			

11.1.3. Y offset to destination address conversion

All XY coordinates are relative to some origin and once the Y offset has been found need to be converted to the physical address of the pixel. OpenGL has it's origin in the bottom left corner of a window and X has it's origin in the top left corner. This requires a slightly different equation to calculate the physical address for a pixel when the physical address of the window's origin is known:

OpenGL:	destination addr = FBWindowBase - Y offset + X + FBPixelOffset
X:	destination addr = FBWindowBase + Y offset + X + FBPixelOffset

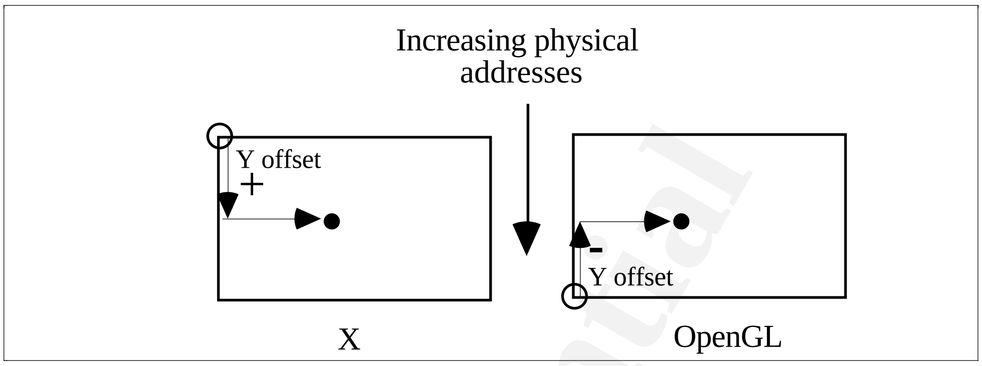
The origin to use is specified by a bit in the in the FBReadMode message. The FBPixelOffset is used for multi-buffer updates, but for normal use will be zero. This is covered later.

The source address (if needed) is calculated:

source address = destination address + FBSourceOffset

Note that the origin's physical address can be negative[23] if the window is part off the screen. A negative physical address is not normally generated[24] as these fragments will already have been turned into passive ones by the screen scissor test.

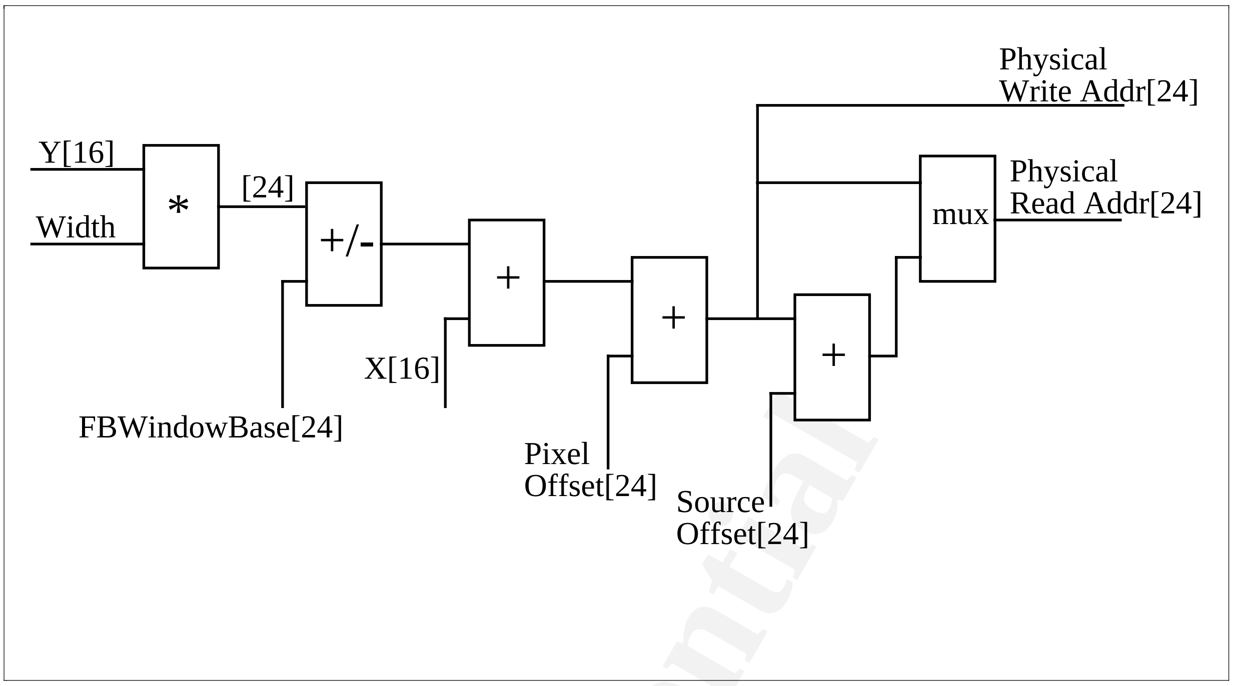
The following diagram demonstrates this:



The full address calculation chain is shown below:

[23]Viewing the address as a negative number is a mathematical convenience as obviously an address is really unsigned. The wrap around nature of the address calculation gives the address the duality such that a 'negative' address is treated by the adders as just a very large positive address. For example if an address of -3 is specified and an offset of 5 is added to it then the expected result is 2, or in 24 bit unsigned format: 0xfffffd + 5 = 2 if overflow is ignored and wrap around occurs.

[24]An exception to this is during an OpenGL image upload operation where some negative addresses may be generated and read from, but in this case invalid data is acceptable.



All address calculations are done to 24 bit and any overflow just wraps around. This wrap around allows ±maximum offset to be applied with a 24 bit number. The X and Y coordinates are sign extended up to the required width.

Once the destination address has been calculated for the original XY coordinate this address is passed to Wa FIFO in the Framebuffer Interface Unit (the write may latter be cancelled though). The address(es) sent to the Ra FIFO depends on the ReadSource and ReadDestination enable bits in the FBReadMode message. This is tabulated below:

ReadSource Enable	ReadDestination Enable	Address(es) sent to the Ra FIFO
0	0	No addresses sent. This would be used during rendering when blending, logical ops and software write masks are all disabled.
0	1	The destination address is sent to the FIFO. This is the normal mode where a read-modify-write is necessary when, for example, alpha blending is enabled. The read data is sent in a FBData message.
1	0	The source address is sent to the FIFO. This is used when all the fields in a pixel are to be copied. The read data is sent in a LBSourceData message.
1	1	The destination address is sent to the FIFO first followed by the source address. This is used when some of the field needs to be copied so the source and destination are merged together before the destination is written. The destination read data is sent in the FBData message and the source data in a FBSourceData message.

The FBPixelOffset is used when multibuffer updates are required and the pixel to be updated is really in another buffer to the one selected by the FBWindowBase message. Remember that OpenGL doesn't control this table so needs another way to select a different buffer.

A read of the physical address(es) is done by the Framebuffer Interface Unit and the raw data returned.

When only one read (per Active walk message) is done the data is inserted into the data field of the FBData, FBColour message depending on the Message Type specified in the FBReadMode register.

When two reads (per Active walk message) are done the first read data is sent in a FBData message and the second read data is sent in a FBSourceData message.

The actual reading of the memory and request arbitration is done in the Framebuffer Interface Unit.

11.1.4. Framebuffer configuration

OpenGL does not impose any restrictions on main planes, overlays, underlays, etc. and the chip should not restrict potential users to the configurations we have used in the past (i.e. the overlay being in the top byte).

If an alpha channel is available it is considered bound to the corresponding RGBA planes for copying pixels around and multi-buffer updates.

To accommodate both these factors the framebuffer interface supports framebuffers of up to 32 bits wide and any multi-buffers (front, back, left, right), overlays and/or underlays are considered as separate framebuffers in the address map.

11.1.5. Data Formats

Different data formats are supported in the framebuffer to allow the width of the memory system to be varied to suite the cost of the system and what an application needs. The possible formats are described here as it is the logical place to hold the description, however the actual conversion to the internal 8 bit per component format is only necessary for alpha blending. The write masking and logical operations are more easily expressed in terms of the physical bit plane organisation. The dither unit is responsible for converting the interanl 8 bit format into the format of the target framebuffer.

The format, or how to interpret the bits, is specified on a per window basis so framebuffers can be used with different visual capabilities:

Mode		Name	Total bits	R	G	B	A
0	RGB	8:8:8:8	32	8	8	8	8
1		5:5:5:5	20	5	5	5	5
2		4:4:4:4	16	4	4	4	4
3		4:4:4:4 Front	32 16 used	4	4	4	4
4		4:4:4:4 Back	32 16 used	4	4	4	4
5		3:3:2	8	3	3	2	
6		1:2:1 Front	8 4 used	1	2	1	
7		1:2:1 Back	8 4 used	1	2	1	
8	CI	CI8	8	8			
9		CI4	4	4			

Notes:

1. The alpha channel is always optional - it can be disabled internally.

2. The memory configuration can be changed on a window basis if the external video controller can cope.

3. The 4:4:4:4 Front and Back modes are special ones to support 12 bit double buffering in a 24 bit system.

4. The 1:2:1 Front and Back modes are special ones to support 4 bit double buffering in an 8 bit system.

5. Double buffering is achieved in memory depth rather than in width.

6. The alpha channel must be with the colour channels as it can be double buffered as well.

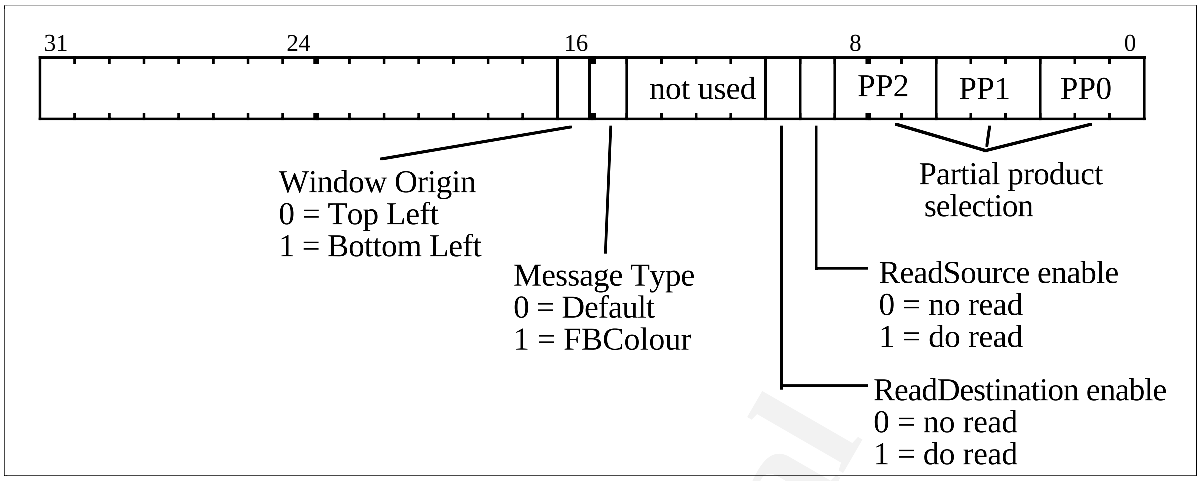
11.1.6. Multi-buffer updates

OpenGL allows for multiple (or no) framebuffers to be written to 'simultaneously'. In general the buffers will be limited to FRONT, BACK, LEFT and RIGHT, although the spec allows for optional auxiliary buffers to be supported. When more than one buffer is selected the alpha blend and logical ops are applied to each buffer independently.

To do multibuffer updated the primitive is rendered multiple times, once for each enabled buffer. In many cases the parameters for the rasterisation, colour DDA, etc. can be reused without reloading. Care does need to be taken to ensure the local buffer is not updated by the first primitive as this may result in the second and subsequent repeats being discarded by the depth test, for example. This is easily arranged by using the write masks or disabling writes in the Local Buffer Write Unit. The FBPixelOffset message is used to access the target pixel in other buffers than the one selected in the FBWindowBase message.

11.1.7. General control

The general mode selection for this unit is done with the FBReadMode message and the data field specifies the required modes. The data field has the following format:



The Message Type field determins the message type the read data is encapsulated in. Normally this is Default so depending on the state of the ReadSource and ReadDestination bits the data will be sent in the FBSourceData or FBData messages respectively. For image uploads it would be FBColour.

11.2. Operating Modes

This section looks at how this unit is set up to do some common operations in OpenGL and X. The partial product selection is set up for the size of screen and never changed.

11.2.1. Rendering Operation (no alpha or logical ops)

The normal rendering operations for X and OpenGL with no alpha blending or logical ops only involve writing. The unit is set up as follows:

ReadSource	0
ReadDestination	0
FBPixelOffset	0
FBSourceOffset	Don't care
Data message	Don't care

11.2.2. Rendering Operation (with alpha or logical ops)

The normal rendering operations for X and OpenGl with alpha blending or logical ops involves reading and writing the framebuffer. The unit is set up as follows:

ReadSource	0
ReadDestination	1
FBPixelOffset	0
FBSourceOffset	Don't care
Data message	Default

11.2.3. Copy Operation

OpenGL has a function which allows the user to specify a source rectangular region (in the window) to copy to a destination region. This function allows the source data to be scaled or remapped in very general ways during the copy and for pixel replicated zooming. To effect these operations the source image must be uploaded into the OpenGL server, modified and then downloaded. If source pixels lie in parts of the window which are obscured then the resulting pixels are undefined. The destination pixels must under go the normal fragment processing, including pixel ownership tests (i.e. GID test) when written.

If the image is a straight copy with no data formatting or zooming then GLiNT can handle this directly. X copies are handled directly. The rasteriser is set up to scan convert the destination rectangle.

If hardware write masks are available, or all the framebuffer planes are being copied then the operations is as for normal rendering with the addition that the write offset is set up as appropriate:

ReadSource	1
ReadDestination	0
FBPixelOffset	0
FBSourceOffset	As appropriate
Data message	Default

If hardware write masks are not available then the source and destination pixels need to be read and merged (in the logical ops unit). The unit is set up as follows: The destination pixel is read and this is inserted into the FBData message. The source pixel is read and the value inserted into the FBSourceData message.

ReadSource	1
ReadDestination	1
FBPixelOffset	0
FBSourceOffset	As appropriate
Data message	Default

11.2.4. Image Upload Operation

The data in the framebuffer is read and stored in a FBColour message in its raw format (i.e. no colour conversion is done). This message falls through all subsequent units to the Host Interface (Out) Unit. The OpenGL specification states that the data is undefined if you read from a pixel not owned by your window.

Care needs to be taken to ensure enough data is returned to the host as it expects so any unit which can convert an Active walk message into a passive one needs to be disabled. The screen scissor test also falls into this category in case a window is part off the screen. Note the user scissor test may need to be disabled as well to ensure this. The unit is set up as follows:

ReadSource	0
ReadDestiantion	1
FBPixelOffset	0
SourceOffset	Don't care
Data message	FBColour

11.3. Input Messages

11.3.1. External Messages

Framebuffer Read Register message group		
Tag Mnemonic	Data Field	Description
FBReadMode	See above	General mode of operation of the unit
FBSourceOffset	24 bit 2's Complement offset	Difference between destination and source addresses for copy operations
FBPixelOffset	24 bit 2's Complement offset	Offset between buffers in multibuffer operation
FBWindowBase	24 bit unsigned base address	

11.3.2. Internal Messages

This unit reacts to all the messages in the Rasteriser Walk group. See the rasteriser section for details on these messages.

11.4. Output Messages

Framebuffer General message group		
Tag Mnemonic	Data Field	Description
FBColour	framebuffer data	Used for Image upload.
FBData	framebuffer data	Used for normal framebuffer reads
FBSourceData	framebuffer data	Used for copy operations.

11.5. Behavioural Model

The behavioural model is best described in two blocks: The input behaviour in calculating addresses and passing them onto the Framebuffer Interface Unit, and the output block which collects the data returned by the Framebuffer Interface Unit and merges it with the normal message stream.

The input block:

```
Wait for input message
{
  switch (on message type)
  {
    case RegisterUpdateGroup:
            Update the identified register;
            break;
    case PrepareToRender:
            Wait for room in the Ra FIFO;
            Notify the Framebuffer Interface Unit to suspend
            reads by setting the SuspendRead signal and
            clocking it into the Ra FIFO with an undefined
            address..
            Wait for room in the M FIFO;
            Copy input message to M FIFO and reset the
              ReadDestination and ReadSource fields in the
              FIFO;
            break;
    case FastBlockFill:
            Calculate destination address;
            Wait for room in the Wa FIFO;
            Send destination address to the Wa FIFO;
            Wait for room in the M FIFO;
            Copy input message to M FIFO and set up the
              ReadSource and ReadDestination bits to 0.
            break;
    case ActiveStep:
            Calculate destination address;
            Calculate source address;
            switch (Read enables from the FBReadMode message)
            {
              case ReadDestination = 0 and ReadSource = 0:
                      break;
              case ReadDestination = 0 and ReadSource = 1:
                      Wait for room in the Ra FIFO;
                      Send source address to Ra FIFO;
                      break;
              case ReadDestination = 1 and ReadSource = 0:
                      Wait for room in the Ra FIFO;
                      Send destination address to Ra FIFO;
                      break;
              case ReadDestination = 1 and ReadSource = 1:
                      Wait for room in the Ra FIFO;
                      Send destination address to the Ra FIFO;
                      Wait for room in the Ra FIFO;
                      Send the source address to the Ra FIFO;
                      break;
            }
            Wait for room in the Wa FIFO;
            Send write mode and destination address to the
              Wa FIFO;
            Wait for room in the M FIFO;
            Copy input message to M FIFO and set up the
              ReadDestination, ReadSource and message type
              fields in the FIFO;
            break;
    default:
            Wait for room in the M FIFO;
            Copy input message to M FIFO and reset the
              ReadDestination and ReadSource fields in the
```

```
                 FIFO;
              break;
   }
   Flush the input message;
}
```

The output block:

```
Wait for message in the M FIFO
{
  switch (Read enables from the M FIFO)
  {
    case ReadDestination = 0 and ReadSource = 0:
            break;
    case ReadDestination = 0 and ReadSource = 1:
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            switch (on Data Message type in FBReadMode)
            {
              case Default:
                    Send FBSourceData message to next unit with
                      Din FIFO data in the data field;
                    break;
              case FBColour:
                    Send FBColour message to next unit with
                      the Din FIFO data in the data field;
                    break;
            }
            Flush one word from Din FIFO;
            break;
    case ReadDestination = 1 and ReadSource = 0:
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            switch (on Data Message type in LBReadMode)
            {
              case Default:
                    Send FBData message to next unit with
                      Din FIFO data in the data field;
                    break;
              case FBColour:
                    Send FBColour message to next unit with
                      the Din FIFO data in the data field;
                    break;
            }
            Flush one word from Din FIFO;
            break;
    case ReadDestination = 1 and ReadSource = 1:
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            Send FBData message to next unit with Din
              FIFO data in data field;
            Flush one word from Din FIFO;
            Wait for room in next unit;
            Wait for data in the Din FIFO;
            Send FBSourceData message to next unit with Din
              FIFO data in data field;
            Flush one word from Din FIFO;
            break;
  }
  Wait for room in next unit.
  Send message in M FIFO to next unit.
  Flush message from M FIFO;
}
```

12. Alpha Blend Unit

12.1. Description

The function of the alpha blend unit is to combine the fragments colour with the colour stored in the framebuffer using the alpha blend equation below. Alpha blending only works for pixels stored in the RGBA format. After blending is done the new colour passed onto the next unit in the Colour message. If alpha blending is disabled then the Colour is passed on unchanged.

The alpha blend equation is

$$C_o = C_s S + C_d D$$

where C_o is the output colour (sent in the colour message to the next unit), C_s is the source colour in the colour message, and C_d is the destination colour in the FBData message, after formatting. C_{sa} is the alpha component of the source colour, C_{dr} is the red component of the destination colour, etc.. The source blending function, S, and the destination blending function, D, are defined in the following tables.

Source blending functions:

Mode	Value	R	G	B	A
0	ZERO	0	0	0	0
1	ONE	1	1	1	1
2	DST_COLOUR	C_{dr}	C_{dg}	C_{db}	C_{da}
3	ONE_MINUS_DST_COLOUR	$1 - C_{dr}$	$1 - C_{dg}$	$1 - C_{db}$	$1 - C_{da}$
4	SRC_ALPHA	C_{sa}	C_{sa}	C_{sa}	C_{sa}
5	ONE_MINUS_SRC_ALPHA	$1 - C_{sa}$	$1 - C_{sa}$	$1 - C_{sa}$	$1 - C_{sa}$
6	DST_ALPHA	C_{da}	C_{da}	C_{da}	C_{da}
7	ONE_MINUS_DST_ALPHA	$1 - C_{da}$	$1 - C_{da}$	$1 - C_{da}$	$1 - C_{da}$
8	SRC_ALPHA_SATURATE	min of $(C_{sa}, 1 - C_{da})$	min of $(C_{sa}, 1 - C_{da})$	min of $(C_{sa}, 1 - C_{da})$	1

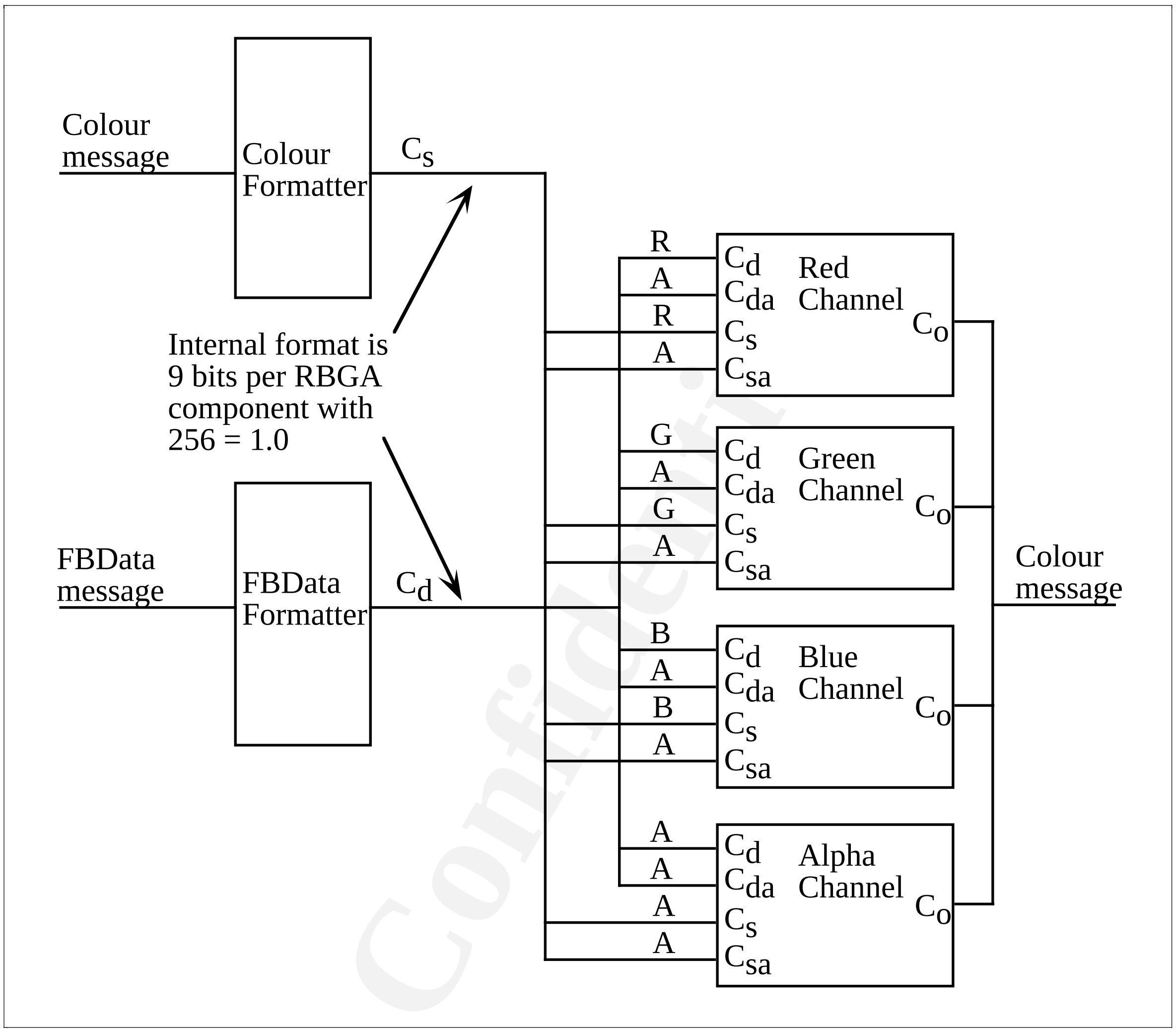
Destination blending functions:

Mode	Value	R	G	B	A
0	ZERO	0	0	0	0
1	ONE	1	1	1	1
2	SRC_COLOUR	C_{sr}	C_{sg}	C_{sb}	C_{sa}
3	ONE_MINUS_SRC_COLOUR	$1 - C_{sr}$	$1 - C_{sg}$	$1 - C_{sb}$	$1 - C_{sa}$
4	SRC_ALPHA	C_{sa}	C_{sa}	C_{sa}	C_{sa}
5	ONE_MINUS_SRC_ALPHA	$1 - C_{sa}$	$1 - C_{sa}$	$1 - C_{sa}$	$1 - C_{sa}$
6	DST_ALPHA	C_{da}	C_{da}	C_{da}	C_{da}
7	ONE_MINUS_DST_ALPHA	$1 - C_{da}$	$1 - C_{da}$	$1 - C_{da}$	$1 - C_{da}$

These equations are defined in terms of floating point numbers, where all the colour components (source and destination values) are assumed to be in the range 0 to 1.0 inclusive.

12.1.1. Implementation

A block diagram of the main data paths of the alpha blend unit is shown below.


Colour message
Colour Formatter
C_s
Internal format is 9 bits per RBGA component with 256 = 1.0
FBData message
FBData Formatter
C_d
R A R A
C_d C_{da} C_s C_{sa} Red Channel C_o
G A G A
C_d C_{da} C_s C_{sa} Green Channel C_o
B A B A
C_d C_{da} C_s C_{sa} Blue Channel C_o
A A A A
C_d C_{da} C_s C_{sa} Alpha Channel C_o
Colour message

This shows the four component in a colour being processed in parallel.

The colour components in the Colour message are all 8 bits in size and these need to be converted to the 9 bit internal format. The colour formatter block does the conversion. The 9 bit unsigned format is one bit integer and 8 bits fraction so that 1.0 can be easily represented and manipulated. Conversion from the 8 bit input format to the 9 bit output format is done using the equations:

$$output = \frac{256}{255} input$$

which, in the limited precision we have available, translates into:

output = input for $0 \le$ input < 255
output = 256 for input = 255

The discontinuity in the normally monotonic number set is inevitable unless the number of fractional bits is increased.

The colour components from the framebuffer (in the FBData message) are still in the framebuffer specific format. They need to be isolated and converted into the 9 bit format suitable for use in the blending equations defined earlier. The FBData formatter block does this conversion. An alpha value of 1.0 is substituted when there is no alpha buffer present as indicated either by the colour format or the NoAlphaBuffer bit being set.

The first task it to isolate the individual colour components from the FBData message. The following table shows the different colour modes supported. In the R, G, B and A columns the nomenclature *n@m* means this component is *n* bits wide and starts at bit position *m* in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode.

In the case of the RGB formats where no Alpha is shown then the alpha field is set to 255. Where the RGB format has an alpha component it may still not exist if those memory planes are not populated. In this case the NoAlphaBuffer bit in the AlphaBlendMode message should be set which causes the alpha component to be set to 255.

Two colour ordering formats are supported, namely ABGR and ARGB, with the right most letter representing the colour in the least significant part of the word. This is controlled by the Colour Order bit in the AlphaBlendMode message, and is easily implemented by just swapping the R and B components *after* conversion into the internal format. The only exception to this are the 3:3:2 formats where the actual bit fields extracted from the framebuffer data need to be modified as well because the R and B components are differing widths. CI processing is not effected by this swap and the result is always on internal R channel.

				Internal Colour Channels			
	Format	Colour Order	Name	R	G	B	A
True Colour	0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
	1	BGR	5:5:5:5	5@0	5@5	5@10	5@15
	2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
	3	BGR	4:4:4:4 Front	4@0	4@8	4@16	4@24
	4	BGR	4:4:4:4 Back	4@4	4@12	4@20	4@28
	5	BGR	3:3:2Front	3@0	3@3	2@6	-
	6	BGR	3:3:2Back	3@8	3@11	2@14	-
	7	BGR	1:2:1Front	1@0	2@1	1@3	-
	8	BGR	1:2:1Back	1@4	2@5	1@7	-
	0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
	1	RGB	5:5:5:5	5@10	5@5	5@0	5@15
	2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
	3	RGB	4:4:4:4 Front	4@16	4@8	4@0	4@24
	4	RGB	4:4:4:4 Back	4@20	4@12	4@4	4@28
	5	RGB	3:3:2Front	3@5	3@2	2@0	-
	6	RGB	3:3:2Back	3@13	3@10	2@8	-
	7	RGB	1:2:1Front	1@3	2@1	1@0	-
	8	RGB	1:2:1Back	1@7	2@5	1@4	-
CI	14	X	CI8	8@0	0	0	0
	15	X	CI4	4@0	0	0	0

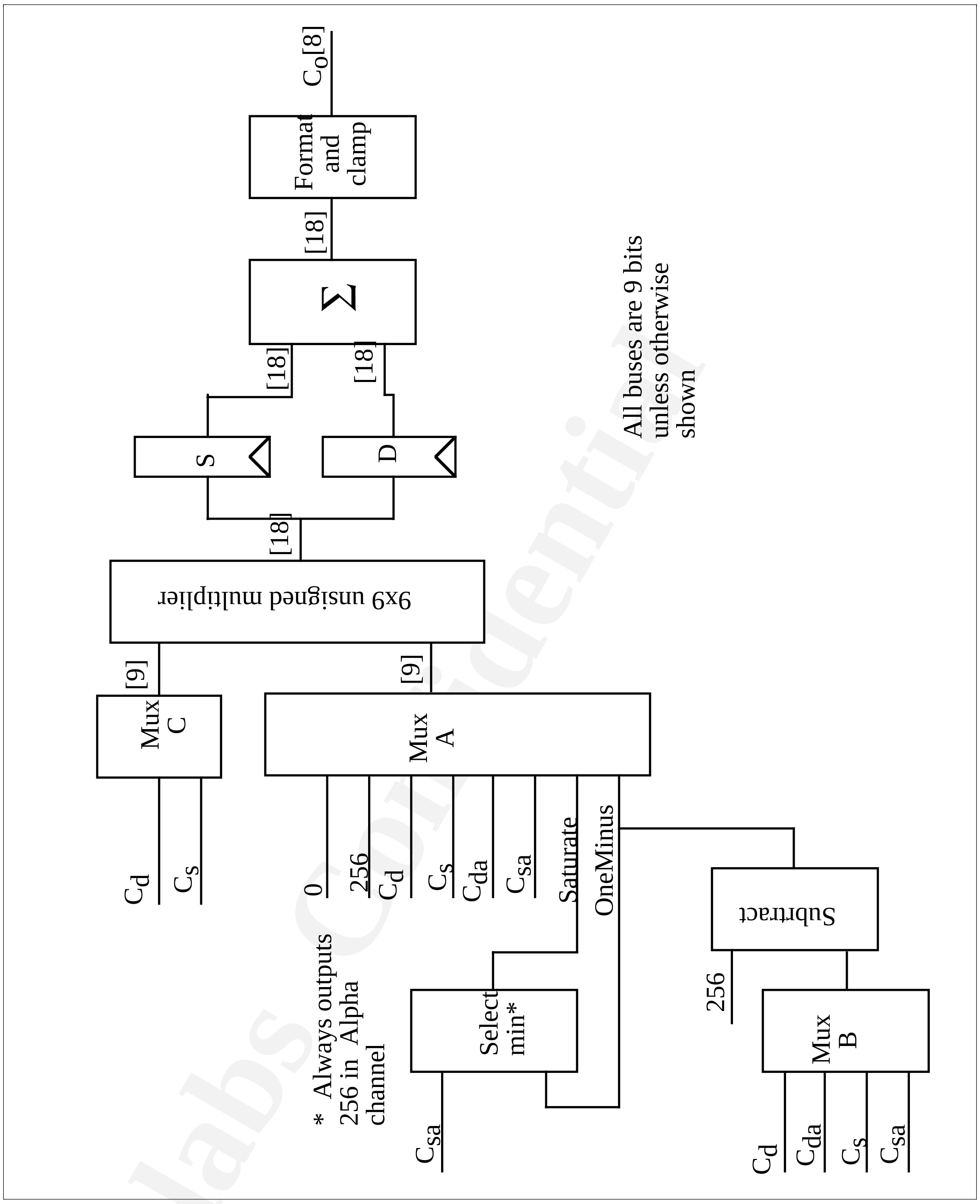
The format to use is held in the AlphaBlendMode message. Note that in OpenGL the alpha blending is not defined for CI mode[25].

When converting a Colour Index value to the internal format any unused bits are set to zero.

The next stage is to convert each component to the 9 bit width. The correct way of doing the conversion is to multiply a component (with *n* significant bits) by $\frac{256}{2^n - 1}$. Where *n* is 1, 2, 3, 4, 5 or 8. For all the cases except where *n* equals eight the increase in resolution will give a more linear conversion process. Note that for some formats the components have different widths, for example 8 bit RBG colours are encoded as 3:3:2.

The four colour components are processed in an identical way (alpha component has a slight simplification). This processing is shown in the following block diagram:

[25]The above table does define how the chip will blend in this case (although the results may not be very useful).



When a Colour or FBData message arrives the data is stored in local registers until needed. These registers are not corrupted by the calculations and the contents are valid for subsequent fragments so, for example, a host computed coverage value can be applied to multiple fragments.

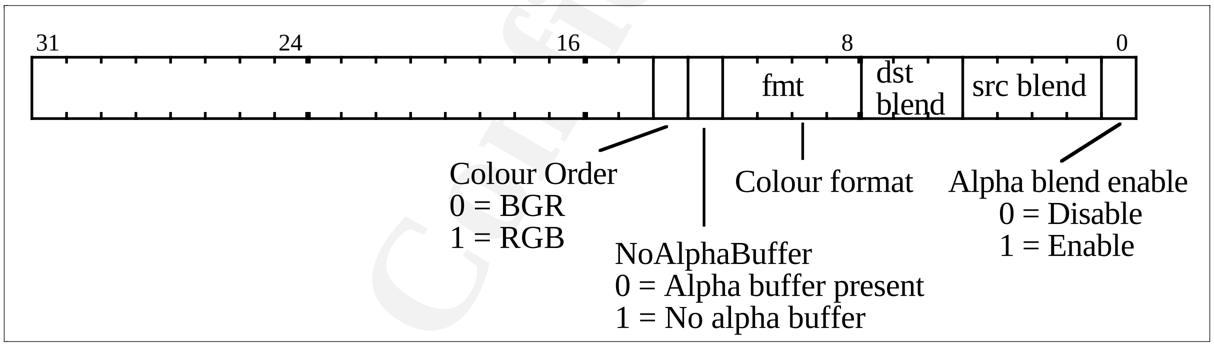
When an active step message arrives the blending calculations[26] are done as specified by the enable and blending function. The multiplexers A, B and C are controlled to select the correct multiplier and multiplicand and the result clocked into either the S or D registers. Once the blending equations have been done the results (in the S and D registers) are summed, formatted and clamped and sent to the next unit in a Colour message.

The output formatting and clamping ensures that the sum of S and D never exceeds 1.0 and is converted to the normal 8 bit format for colour components. The output format of the summation is unsigned 18 bits where the two most significant bits are the integer and the lower 16 bits the fraction. If a carry has been generated or any of the integer bits are set then the 8 bit result is set to 255, otherwise the most significant 8 bits of the fraction are used as the result. This conversion, like the 8 to 9 bit conversion done earlier, is a compromise of the real conversion equation:

$$C_o = \frac{255}{256}(S + D)$$

In the case where the destination blending function is set to ZERO an obvious optimisation is to not read the framebuffer and hence generate an FBData messages. The control of the A and B multiplexers is easily inferred from the source and destination blending tables given earlier.

The alpha blending is controlled using the AlphaBlendMode message. It has the following format:



If the alpha blend is disabled then the colour message is passed through unchanged.

[26]The VHDL implementation has deviated from the description given here in that the Colour and FBData messages are partially processed before being stored in a register. This processing is a function of the contents of the AlphaBlendMode message so changing the AlphaBlendMode after sending a Colour message will not have any effect until the next Colour message is sent. Similarly for an FBData message.

During the normal operation of GLiNT this is not an issue as Colour messages are only generated as a result of a Render message. Any changes to AlphaBlendMode will have been done before the Render message, or have to wait until the Render operation has finished.

If Colour messages are being sent down by the host to gain some performance benefits (see Colour DDA spec for details) then care must be taken to ensure the desired AlphaBlendMode is set up first.

12.2. Input Messages

12.2.1. External Messages

Alpha blend message group		
Tag Mnemonic	Data Field	Description
AlphaBlendMode	See above	General mode of operation of the alpha blend function.

12.3. Internal Messages

The unit responds to the Colour, FBData, the ActiveStep messages. See the Rasteriser, Colour DDA and Framebuffer Read Units for details on these messages.

12.4. Output Messages

No new messages are generated, however the data field in the colour message may be modified.

12.5. Behavioural Model

```
Wait for input message
{
  switch (on message type)
  {
    case AlphaBlendMode:
            Update appropriate register with data;
            break;
    case Colour:
            Update the colour register with the formatted data;
            if (alpha blend disabled)
            {
              Wait for room in next unit;
              Forward colour message on to the next unit;
            }
            break;
    case FBData:
            Update the FBData register with the formatted data;
            Wait for room in next unit;
            Forward FBData message on to next unit;
            break;
    case ActiveStep:
            if (alpha blending enabled)
            {
              Update the S register using the value in the
                Colour register and the source blend function.
              Update the D register using the FBData register
                and destination blend function.
              Add together the S and D colour values and
                convert to the normal 8 bit format;
              Wait for room in next unit;
              Send Colour message with new colour value;
            }
            Wait for room in next unit;
            Forward on the ActiveStep message;
            break;
    case default:
            Wait for room in next unit;
            Forward input message;
            break;
  }
  Flush message from input buffer;
}
```

Note that the order of the FBData and Colour messages get reversed when alpha blending is enabled..

13. Dither Unit

13.1. Description

The dither unit's job is to convert the internal colour format into the format the colour information is stored in the framebuffer. This process is governed by:

- The width of the colour components in the framebuffer.
- The position of the colour components in the framebuffer.
- The colour mode (RGBA or CI).
- Whether dithering is to be used in the conversion process.

The colour value in the data field of the Colour message is modified as required by these factors and then sent on to the next unit.

The internal format is fixed point integer. The number of fraction bits depends on the number of bits left over after the integer width of the colour component has been satisfied. For an 8 bit colour there are no fraction bits, a 5 bit colour has 3 fraction bits, etc. This variation in fixed point format is not desirable but is forced on us as we cannot afford the extra width on the colour components through the preceding stages.

The main implication of this is that the type of dithering varies depending on the colour format:

Component width	Type of dithering
8	None for RGBA or CI colour mode
5	2x2 ordered dither
4	4x4 ordered dither
3	4x4 ordered dither
2	4x4 order dither
1	4x4 ordered dither

The OpenGL specification does allow dithering on 8 bit components but it is doubtful if its absence will cause any visual discrepancy. We still need to verify if this is true with the conformance tests.

When dithering is disabled the components (R, G, B, A) are truncated and a CI value is rounded to the nearest integer value. The result is clamped to the maximum value a component can have.

When dithering is enabled the least significant bits of the X and Y coordinate (window relative) index into a dither table. This dither value is added to the component (suitably aligned to the same fixed point format) and the result clamped and truncated to the width of the component.

In both the RGB and CI modes any 8 bit components are passed through unchanged.

The following table shows the different colour modes supported by the dither unit. In the R, G, B and A columns the nomenclature *n@m* means this component is *n* bits wide and starts at bit position *m* in the framebuffer. The least significant bit position is 0 and a dash in a column indicates that this component does not exist for this mode. When two entries are shown the colour value is replicated into both fields.

Two colour ordering formats are supported, namely ABGR and ARGB, with the right most letter representing the colour in the least significant part of the word. This is controlled by the Colour Order bit in the DitherMode message, and is easily implemented by just swapping the R and B components *befoer* conversion into the framebuffer format. The only exception to this are the 3:3:2 formats where the actual bit fields sent to the framebuffer data need to be modified as well because the R and B components are differing widths. CI processing is not effected by this swap and the pixel data is always on internal R channel.

				Internal Colour Channels			
	Format	Colour Order	Name	R	G	B	A
True Colour	0	BGR	8:8:8:8	8@0	8@8	8@16	8@24
	1	BGR	5:5:5:5	5@0	5@5	5@10	5@15
	2	BGR	4:4:4:4	4@0	4@4	4@8	4@12
	3	BGR	4:4:4:4 Front	**4@0** **4@4**	**4@8** **4@12**	**4@16** **4@20**	**4@24** **4@28**
	4	BGR	4:4:4:4 Back	**4@0** **4@4**	**4@8** **4@12**	**4@16** **4@20**	**4@24** **4@28**
	5	BGR	3:3:2Front	**3@0** **3@8**	**3@3** **3@11**	**2@6** **2@14**	**-** **-**
	6	BGR	3:3:2Back	**3@0** **3@8**	**3@3** **3@11**	**2@6** **2@14**	**-** **-**
	7	BGR	1:2:1Front	**1@0** **1@4**	**2@1** **2@5**	**1@3** **1@7**	**-** **-**
	8	BGR	1:2:1Back	**1@0** **1@4**	**2@1** **2@5**	**1@3** **1@7**	**-** **-**
	0	RGB	8:8:8:8	8@16	8@8	8@0	8@24
	1	RGB	5:5:5:5	5@10	5@5	5@0	5@15
	2	RGB	4:4:4:4	4@8	4@4	4@0	4@12
	3	RGB	4:4:4:4 Front	**4@16** **4@20**	**4@8** **4@12**	**4@0** **4@4**	**4@24** **4@28**
	4	RGB	4:4:4:4 Back	**4@16** **4@20**	**4@8** **4@12**	**4@0** **4@4**	**4@24** **4@28**
	5	RGB	3:3:2Front	**3@5** **3@13**	**3@2** **3@10**	**2@0** **2@8**	**-** **-**
	6	RGB	3:3:2Back	**3@5** **3@13**	**3@2** **3@10**	**2@0** **2@8**	**-** **-**
	7	RGB	1:2:1Front	**1@3** **1@7**	**2@1** **2@5**	**1@0** **1@4**	**-** **-**
	8	RGB	1:2:1Back	**1@3** **1@7**	**2@1** **2@5**	**1@0** **1@4**	**-** **-**
CI	14	X	CI8	8@0	0	0	0
	15	X	CI4	4@0	0	0	0

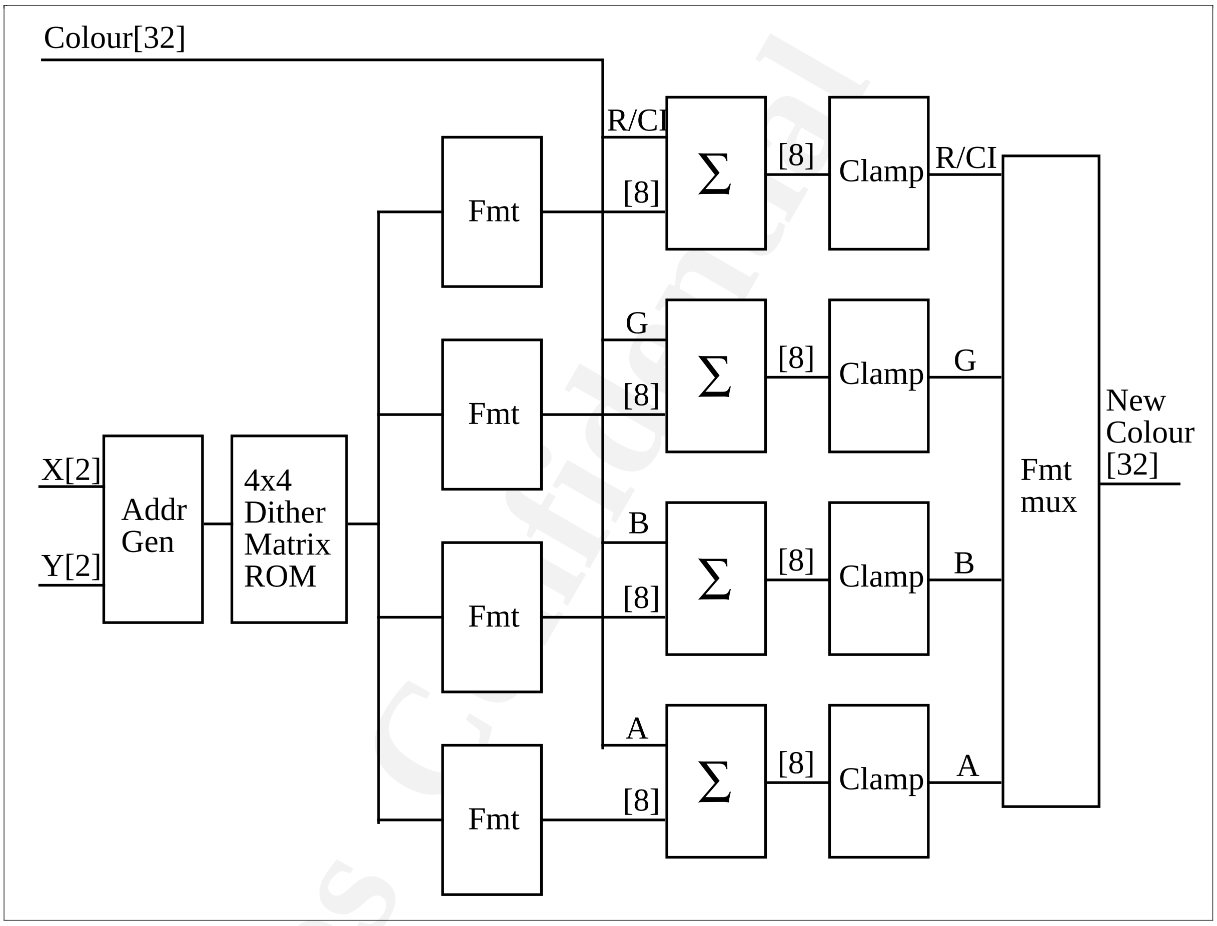
The format to use is held in the AlphaBlendMode message. Note that in OpenGL the alpha Note that modes 3,4 and 6,7 have their components ordered differently in the framebuffer. The orders are $A_bA_fB_bB_fG_bG_fR_bR_f$ and $A_bB_bG_bR_bA_fB_fG_fR_f$ respectively.

Double buffering can be done by reducing the colour resolution and using half of a pixel for the front buffer and half for the back buffer. These modes are designated by Front and Back post-fixes and in this table have identical entries. Separate designations are required when alpha blending to select which buffer to blend with so for consistence they are kept here as well. The colour replication allows single buffered windows to easily exist (at reduced colour resolution) when the whole screen is in double buffered mode. This mode does not, in general, support the OpenGL multi-buffer update operations as their semantics require alpha blending so the front and back colours may be different.

In CI mode the lower byte (CI8) or nibble (CI4) replicated up to the full 32 bit width as an aid to double buffering when the alternative buffers are stored in different bit planes in the same 32 bit word. The replication is done after dithering.

13.1.1. Dither Operation

The dither unit block diagram is as follows:



The dither matrix is held in a 4x4 array, indexed by the two bit values of X' and Y' (output from the Addr Gen block):

		X'			
		11	10	01	00
Y'	00	10	2	8	0
	01	6	14	4	12
	10	9	1	11	3
	11	5	13	7	15

The Addr Gen block takes the least significant two bits of the fragments XY coordinates (from an active step message), and the dither matrix size and calculates the table index from them:

For 2x2 dithering:

X' = (X + XOffset) & 1 << 1
Y' = (Y + YOffset) & 1 << 1

and for 4x4 dithering:

X' = (X + XOffset) & 3
Y' = (Y + YOffset) & 3

The 2x2 dithering equations select the correct 4 values out of the 4x4 matrix so a separate 2x2 matrix is not needed. The 2x2 dither operation is only used for mode 1.

The addition of the XOffset and YOffset values allow window relative dithering to be done when the XY coordinates are screen relative (or device coordinates).

The dither value from the table is formatted in the Fmt blocks to match up with the integer width of a component. It also takes into account if dithering is enabled, and whether the components are RGBA or CI. Each channel has been shown with a different format block because in the 3:3:2 and 1:2:1 formats different formats are needed for the different channels. The following table shows how each format block is set up for each of the supported formats. The value, D, in the table is the value from the dither matrix.

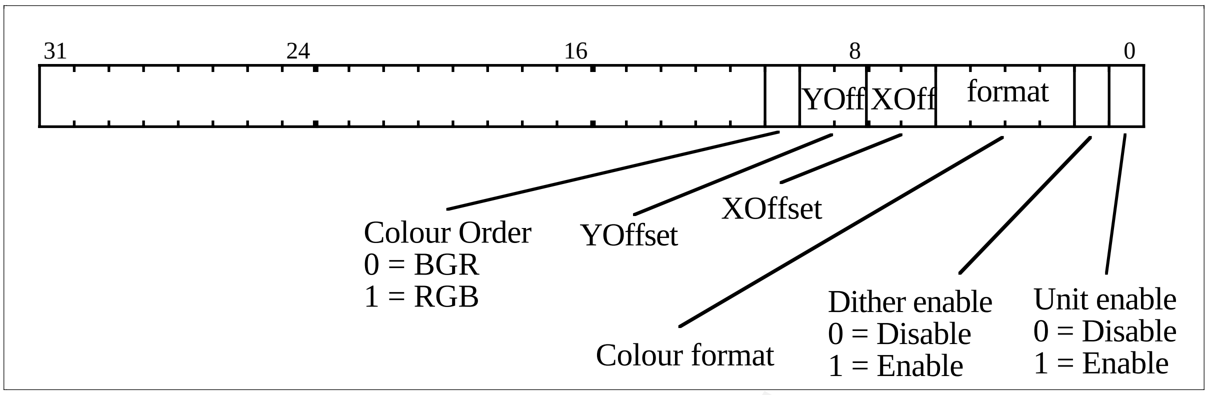
Format	R		G		B		A	
	Dither enabled	Dither disabled	Dither enabled	Dither disabled	Dither enabled	Dither disabled	Dither enabled	Dither disabled
8:8:8:8	0	0	0	0	0	0	0	0
5:5:5:5	D<<1	0	D<<1	0	D<<1	0	D<<1	0
4:4:4:4	D<<0	0	D<<0	0	D<<0	0	D<<0	0
4:4:4:4F	D<<0	0	D<<0	0	D<<0	0	D<<0	0
4:4:4:4B	D<<0	0	D<<0	0	D<<0	0	D<<0	0
3:3:2F	D<<1	0	D<<1	0	D<<2	0	0	0
3:3:2B	D<<1	0	D<<1	0	D<<2	0	0	0
1:2:1F	D<<3	0	D<<2	0	D<<3	0	0	0
1:2:1B	D<<3	0	D<<2	0	D<<3	0	0	0
CI8	0	0	0	0	0	0	0	0
CI4	D<<0	0x8	0	0	0	0	0	0

The clamp unit on each channel monitors the carry flag and when set forces the component value to be the maximum value (i.e. 0xFF). Note that this is independent of the component width, *n*, as only the *n* most significant bits of the output value are used.

Note the 0x8 in the above table for the CI4 mode rounds the input colour.

The Fmt Mux takes the modified colour components and uses them to build up the new colour value as indicated by the mode. Any unspecified bits are set to zero. The widths and positions for each component in the new colour were defined in an earlier table. Where the required component width is less than internal eight bits then the most significant bits are used.

The operation of the unit is controlled by the DitherMode message. The data field of this message has the following format:



If the unit is disabled then the colour message just flows through unchanged.

13.2. Input Messages

13.2.1. External Messages

Register update message group		
Tag Mnemonic	Data Field	Description
DitherMode	See above	Controls mainly how colour data is formatted prior.

13.2.2. Internal Messages

Only the Colour message from the Colour DDA Unit and the Active step messages from the Rasteriser Unit are used.

13.3. Output Messages

No new messages are generated, however the Colour message can have its data field modified.

13.4. Behavioural Model

```
Wait for input message
{
   switch (message type)
   {
      case DitherMode:
               Update the mode register;
               break;
      case Colour:
               Update the colour register;
               if (unit is disabled)
               {
                  Wait for room in receiving message queue;
                  Copy input message to receiving queue;
               }
               break;
      case Active Step:
               if (unit is enabled)
               {
                  Process the colour into the new colour;
                  Wait for room in receiving message queue;
                  Copy the Colour message to receiving queue;
               }
               Wait for room in receiving message queue;
               Forward the Active Step message to receiving
                  queue;
               break;
         default:
               Wait for room in receiving message queue;
               Copy input message to receiving queue;
               break;
   }
   Flush the input message;
}
```

14. Logical Ops and Write mask Unit

14.1. Description

This unit combines the data fields of Colour and FBData messages under control of the write mask and logical operation mode. The FBWriteData message is sent with the result in the data field. The FBData and Colour messages are consumed by this unit.

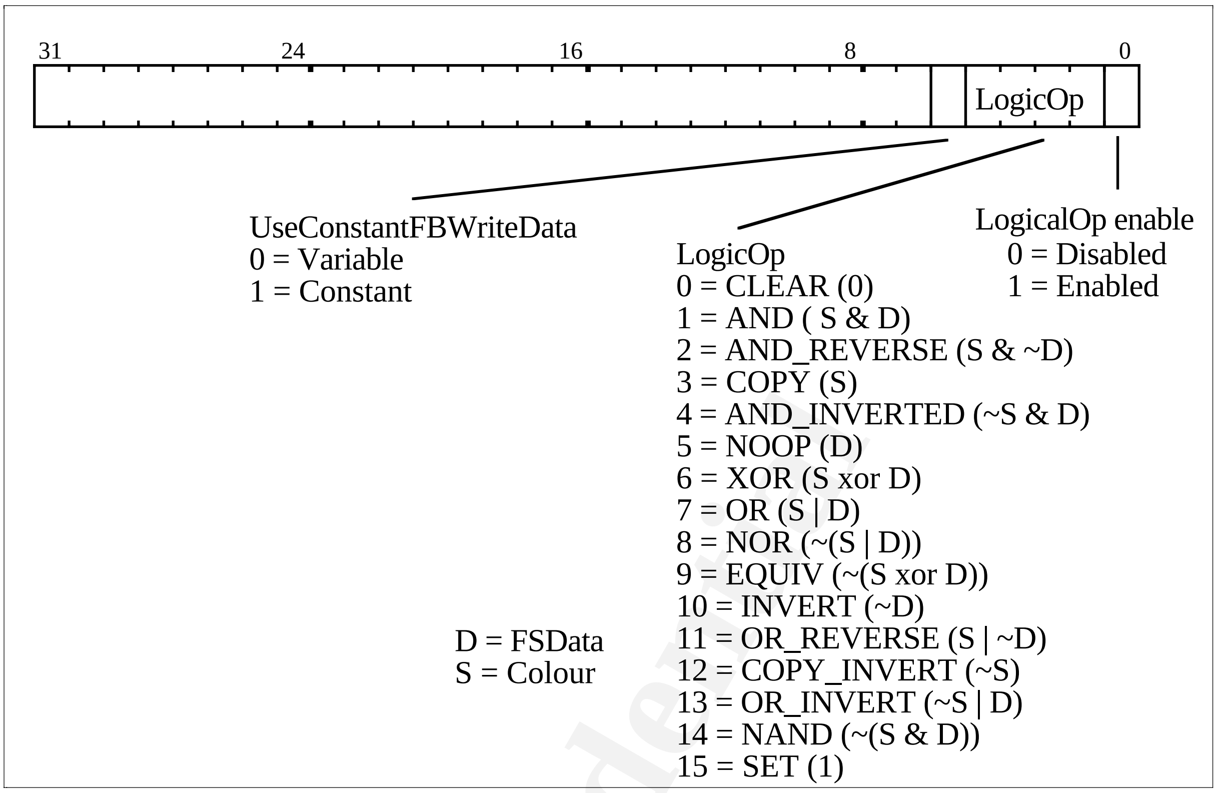
The arrival of an active step message prompts the above actions so the FBData (if present) and Colour messages for this fragment have already been received.

In RGBA mode the OpenGL specification allows the colour components to be masked individually, but doesn't allow the bits within a component to be masked. Also logical operations are not defined for this mode. In CI mode the specification allows for bit wise masking and logical operations. This unit does not differentiate between CI and RGBA modes and bit wise masking and logical operations are available in both cases. It is up to the software to configure the control register in the appropriate way for each mode and the framebuffer data format.

The write masking is controlled with the FBSoftwareWriteMask message. The data field has one bit per framebuffer bit and when set allows that framebuffer bit to be updated. If the framebuffer has hardware write masks then the write masking in this unit can be disabled (by setting all the bits), which requires less framebuffer reads when no logical ops or alpha blending is needed.

The result of any write mask and logical operation is communicated to the next unit in the FBWriteData message. This message can be prevented from being sent by setting the UseConstantFBWriteData bit in the mode message. The reason why this is useful is that when the data to write into the framebuffer is constant across a primitive it doesn't need to be sent for every fragment. If it is not sent then the number of messages per fragment falls from two to one, hence the performance of the core in this mode doubles, and consequently now matches the peak framebuffer bandwidth. As a prelude to running in this mode the host will have sent an FBWriteData message with the required data to write into the framebuffer.

The logical operation is controlled by the LogicalOpMode message and the format of the data field is:



If the LogicalOp is disabled then no logical operation is performed between the Colour message data and the FBData message data.

Write masking is always done between the Colour message data (or the result of the logical op if it is enabled) and the FBData message data, however it can be effectively disabled by setting all 32 bits in the FBSoftwareWriteMask. Obviously for this to work in practice the framebuffer should be read to provide a new FBData message per pixel.

14.2. Input Messages

14.2.1. External Messages

Register update message group		
Tag Mnemonic	Data Field	Description
FBSoftwareWriteMask	Each bit controls the corresponding framebuffer bit.	
LogicalOpMode	See above	

14.2.2. Internal Messages

This unit reacts to the active walk messages from the Rasteriser Unit, the FBData and FBSourceData messages from the Framebuffer Read Unit, and the Colour message from the Colour DDA Unit. See these sections for more details.

Note that the Colour message and FBSourceData message update the same internal register (called the Source register in the behavioural model). The FBSourceData message should only occur when pixels are being copied in the framebuffer so Colour messages will not be present. If they do occur because of a set up mistake then they will be ignored because they arrive before the FBSourceData message and hence get overwritten.

14.3. Output Messages

Logical Ops Output messages		
Tag Mnemonic	Data Field	Description
FBWriteData	Framebuffer colour data	

14.4. Behavioural Model

```
Wait for input message
{
   switch (message type)
   {
      case FBSoftwareWriteMask:
      case LogicalOpMode:
      case FBData:
                Save the data in the appropriate register;
                break;
      case Colour:
      case FBSourceData:
                Save the data in the Source register;
                break;
      case Active Step:
                Wait for room in receiving message queue;
                if (LogicalOp is enabled)
                {
                   Do the required write mask and logical op
                      combination of the Source register and
                      FBData register;
                   if (UseConstantFBWriteData is 0)
                      Send the result to the next unit in the
                         FBWriteData message;
                else
                {
                   Do the required write mask of the Source
                      register and FBData register;
                   if (UseConstantFBWriteData is 0)
                      Send the result to the next unit in the
                         FBWriteData message;
                }
                Wait for room in receiving message queue;
                Copy the Active walk message to receiving queue;
                break;
         default:
                Wait for room in receiving message queue;
                Copy input message to receiving queue;
                break;
   }
   Flush the input message;
}
```

15. Framebuffer Write Unit

15.1. Description

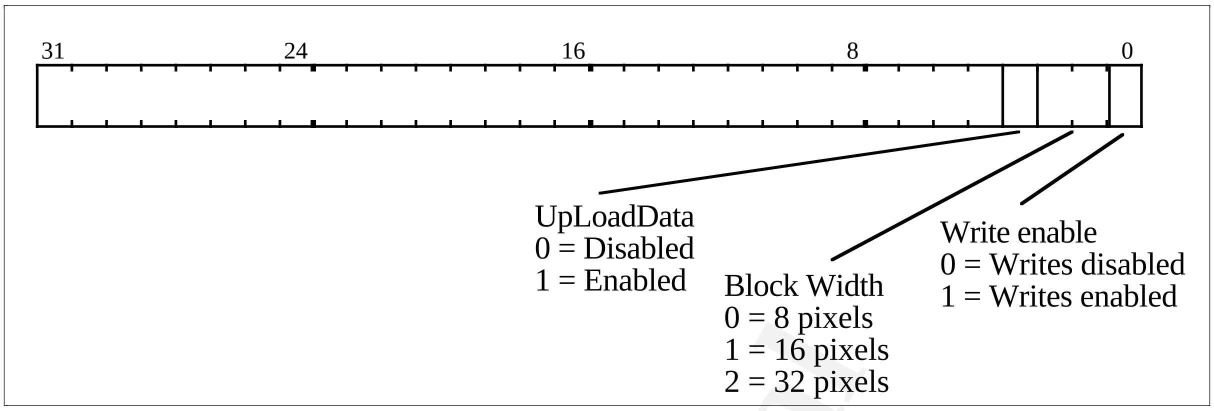
The Framebuffer Write Unit issues write requests to the Framebuffer Interface Unit. More details regarding the interface to the Framebuffer Interface Unit can be found in the Framebuffer Read Unit.

This unit responds to very few messages:

- On receiving a FBWriteData message the data is stored locally until needed.
- On receiving an Active Step message the write enable bit is checked. If it is set then the FBWriteData is passed to Write data FIFO in the Framebuffer Interface Unit and a write request is issued. The address for this write is already queued up in the Framebuffer Interface Unit. If writes are disabled then the pending write operation is cancelled so the write address stored in the Framebuffer Interface Unit is discarded.

  This behaviour is different to that in the Local Buffer Write unit where the LBWriteData and LBCancelWrite control all the updating (i.e. ActiveSteps are not used at all). In this unit all writing is controlled by the ActiveStep messages. When writing constant colour data to the framebuffer (i.e. no dither or logical ops) the colour information can be sent by the host in an FBWriteData message and this will lodge in this unit. Each subsequent ActiveStep message will initiate a write with an overall cost of one message per write, which matches the framebuffers peak bandwidth. If the Local Buffer method were used then two messages per write would be needed, hence the performance would drop.
- On receiving a FBCancelWrite message a flag is set so that on the next ActiveStep message the outstanding write in the Framebuffer Interface Unit is cancelled rather than executed.
- On receiving a PrepareToRender message the Framebuffer Interface Unit is notified that it can, after all queued writes are complete, resume doing reads. See the Framebuffer Read Unit for detail on what this is used for.
- On receiving a Sync message this unit waits for all outstanding data in the Framebuffer Interface Unit to have been 'written', as shown by the FBWrComplete signal being asserted. Once this has occurred then the Sync message is forwarded onto the next unit. This ensures that all the work associated with a primitive has been completed before the host is informed.
- On receiving a FastBlockFill message the pixel write mask is calculated for the given X coordinate and knowledge of the limits of the span to fill (sent in a previous message). This pixel write mask is then sent to the Framebuffer Interface Unit as data. This address in the Framebuffer Interface Unit has been tagged as being a block fill so the data is interpreted as a pixel write mask and not real data.
- The FBBlockColour and FBHardwareWriteMask messages both forward their data field to the write data FIFO in the Framebuffer Write Unit with the appropriate write mode. The data is also held locally for readback. Note that this data in the write data FIFO have no corresponding addresses in the Wa FIFO.

The FBWriteMode message has the following data field format:



The UpLoadData bit, when set, causes the FBWriteData associated with each step message to be forwarded onto the next unit in a FBColour message. This feature allows data read from the framebuffer to be formatted (using the formatting logic in the Alpha Blend and Dither units) into the desired output format before being uploaded.

15.1.1. Pixel Write Mask Generation

Many framestores will provide a mechanism where multiple pixels can be written to simultaneously within a block. A block may be 8, 16 or 32 pixels in size and will be aligned to an 8, 16 or 32 pixel boundary in memory. Each pixel is updated with the same data using the same write mask. This feature is useful for filling spans with constant colour.

Every block write is controlled by a pixel write mask which selects which pixels in the block to update. In the centre of a long span (>> block size) all the pixels will be updated so the pixel write mask is all set. For blocks totally outside the span no pixels are updated so the write mask is all zero. At the edge of a span there will a block where only some of the pixels are to be updated so the pixel write mask will have some bits set and others clear. The pixel write mask generation logic calculates the appropriate write mask in each case. All the X coordinates used to calculate the pixel write mask are positive.

The least significant bit in the pixel write mask corresponds to the left most pixel on the screen (in that block). The unused bits in the mask for the block sizes of 8 and 16 are not used so can take any value.

The inputs which control the pixel write mask generation are the FastBlockLimits message, the FastBlockFill message and the block width in the FBWriteMode message.

The FastBlockLimits register specifies the left and right ends of the span to fill. The left hand pixel is included in the span but the right hand one isn't to be consistent with the point sampling rules the rasteriser follows.

The FastBlockFill message has previously (in the Framebuffer Read Unit) caused the block address to be entered into the Wa FIFO in the Framebuffer Interface Unit. In this block it prompts the generation of the pixel write mask and the write request. The data field of this message holds the X coordinate of a pixel within the block to write to.

We need to define some terms which will be used in the algorithm to describe how the pixel write mask is to be generated:

Left index	The pixel within the left most block the fill starts from. This is the bottom bits of the Left X value from the FastBlockLimits message. The number of bits depends on the size of the block.
Right index	The pixel within the right most block the fill ends before. This is the bottom bits of the Right X value from the FastBlockLimits message. The number of bits depends on the size of the block.
Left block	The left most block which overlaps the span. This is the most significant bits of the Left X value from the FastBlockLimits message. The number of bits depends on the size of the block.
Right block	The right most block which overlaps the span. This is the most significant bits of the Right X value from the FastBlockLimits message. The number of bits depends on the size of the block. Note that Right index is zero then no pixels in this block are updated.
Fragment block	The block the FastBlockWrite message represents. This is the most significant bits of the X value from this message and the number of bits depends on the size of the block.
Left mask	This is only used for the left most block which overlaps with the span. It is calculated by: 0xffffffff << left index with the new ls bits being set to zero.
Right mask	This is only used for the right most block which overlaps with the span. It is calculated by: 0x00000000 << right index with the new ls bits being set to one.

```
Set up the left and right masks;
switch (fragment block)
{
   case fragment block < left block:
   case fragment block > right block:
           pixel write mask = 0;
           break;
   case left block = right block = fragment block:
           pixel write mask = left mask & right mask;
           break;
   case left block = fragment block
           pixel write mask = left mask;
           break;
   case right block = fragment block
           pixel write mask = right mask;
           break;
   default:
           pixel write mask = all bits set;
           break;
}
```

15.2. Input Messages

15.2.1. External Messages

Framebuffer Data Write message group		
Tag Mnemonic	Data Field	Description
FBHardwareWriteMask	Data format corresponds with the raw framebuffer format	Updates the local hardware write mask register and informs the FB Interface Unit..
FBBlockColour	Data format corresponds with the raw framebuffer format	Updates the local hardware colour register and informs the FB Interface Unit. Used when doing block writes to supply the data to write into the framebuffer.
FBWriteMode	See above	Controls the operation of the Framebuffer write unit.

15.2.2. Internal Messages

This unit only responds to the ActiveStep. PrepareToRender, FastBlockFill and FastBlockLimits message from the rasteriser unit, the FBWriteData and FBCancelWrite messages from the Logical Ops Unit, and the Sync message from the host. See these sections for more details.

15.3. Output Messages

No new messages are generated.

15.4. Behavioural Model

```
Wait for input message
{
  switch (message type)
  {
    case FBWriteMode:
    case FastBlockLimits:
    case FBWriteData:
                Update the appropriate register;
                break;
    case FBCancelWrite:
                Set the cancel write flag;
                break;
    case ActiveStep:
                Wait for room in the Write data FIFO in the
                  Framebuffer Interface Unit;
                if (writes are enabled)
                {
                  if (cancel write flag set)
                  {
                    Post a Discard Write request and undefined
                      data to the Framebuffer Interface Unit
                      Write data FIFO;
                  }
                  else
                  {
                    Post a Write Data request and data (from
                      FBWriteData) to the Framebuffer Interface
                      Unit;
                  }
                }
                else
                {
                  Post a Discard Write request and undefined
                    data to the Framebuffer Interface Unit
                    Write data FIFO;
                }
                Clear the cancel write flag;
                if (UpLoadData bit is set)
                {
                  Wait for room in receiving message queue;
                  Send data in FBWriteData register to the next
                    unit in a FBColour message;
                }
                Wait for room in receiving message queue;
                Copy input message to receiving queue;
                break;
    case PrepareToRender:
                Clear the cancel write flag;
                Wait for room in the Write data FIFO in the
                  Framebuffer Interface Unit;
                Post a Resume Read request and undefined
                  data to the Framebuffer Interface Unit
                  Write data FIFO;
                Wait for room in receiving message queue;
                Copy input message to receiving queue;
                break;
        case FastBlockFill:
                Calculate the pixel write mask;
                Wait for room in the Write data FIFO in the
                    Framebuffer Interface Unit;
                Post a Block Write request and data (pixel write
                    mask) to the Framebuffer Interface Unit Write
                    data FIFO;
                break;
        case FBBlockColour:
                Update the local register;
                Wait for room in the Write data FIFO in the
                    Framebuffer Interface Unit;
                Post a Load Colour request and data (from this
                    message) to the Framebuffer Interface Unit
                    Write data FIFO;
                break;
        case FBHardwareWriteMask:
                Update the local register;
                Wait for room in the Write data FIFO in the
                    Framebuffer Interface Unit;
                Post a Load WriteMask request and data (from
                    this message) to the Framebuffer Interface
                    Unit Write data FIFO;
                break;
        case Sync:
                Wait for the WrComplete signal in the Local
                    Buffer Interface Unit to be asserted;
                Wait for room in receiving message queue;
                Copy input message to receiving queue;
        default:
                Wait for room in receiving message queue;
                Copy input message to receiving queue;
                break;
    }
    Flush the input message;
}
```

16. Host Out Unit

16.1. Description

The host out unit has three functions:

- Message filtering. This unit is the last unit in the core so any message not consumed by a preceding unit will end up here. These messages will fall in to three classifications: Rasteriser messages which are never consumed by the earlier units, messages associated with image uploads, and finally programmer mistakes where an invalid message was written to one of the input FIFOs. Syncronization messages are a special category and are dealt with later. Any messages not filtered out are passed on the output FIFO.

- Statistic Collection. Here the active step messages are used to record the extent of the rectangular region where rasterisation has been occurring, or if rasterisation has occurred inside a specific rectangular region. These facilities are useful for picking and debug activities.

- Synchronization. It is often useful for the controlling software to find out when some rendering activity has finished, to allow the timely swapping or sharing of buffers, reading back of state, etc.. To achieve this the software would send a Sync message and when this reached this unit any preceding messages or their actions are guaranteed to have finished. On receiving the Sync message it is entered into the FIFO and optionally generates an interrupt.

16.1.1. Message Filtering

The messages which can reach this unit have been placed into the following categories, and the filtering level for each category can be specified independently.

Message Category	Tag Control Bit	Data Control Bit	Description
Active	0	1	These are PrepareToRender, ActiveStepX and ActiveStepYDomEdge.
Passive	2	3	These are PassiveStepX and PassiveStepYDomEdge.
Depth	4	5	This is the message from image upload of the Depth buffer.
Stencil	6	7	This is the message from image upload of the Stencil buffer.
Colour	8	9	This is the message from image upload of the Framebuffer.
Synchronization	10	11	
Statistics	12	13	These are the messages used to read back the results of the statistic measurements: PickResult, MaxHitRegion, MinHitRegion.
Remainder	14	15	All other message tags which don't fall into the above categories

Each message category has two bits reserved for it in the data field of the FilterMode message to select the filtering level. The Tag Control Bit specifies whether the tag should be included in the FIFO and the Data Control bit specifies if the data should be included in the FIFO. The selection between the tag and/or the data is necessary because the FIFO is only 32 bits wide so if both are required then 2 FIFO words are used. If both tag and data are specified then the tag is always the first word in the FIFO. When a control bit is set the corresponding tag/data value is written to the FIFO.

The Synchronization and Statistics categories have only been added to provide some control over what is returned when the software prompts for the data.

16.1.2. Statistic Collection

The statistic collecting has two modes of operation:

Picking	In this mode the active and/or passive steps have the associated XY coordinate compared against the coordinates specified in the MinRegion and MaxRegion registers (loaded by the messages of the same name). If the result is true then the PickResult flag is set otherwise it holds it previous state. The compare function can be either Inside or Outside. Before picking can start the ResetPickResult message must be sent to clear the PickResult flag.
Extent	In this mode the active and/or passive steps have the associated XY coordinates compared to the MinRegion and MaxRegion registers and if found to be outside the defined rectangular region the appropriate register is updated with the new coordinate(s) to extend the region. The Inside/Outside bit has no effect in this mode.

In both these modes the MinRegion and MaxRegion registers are loaded (from the messages of the same name) to select the region of interest (picking) or the maximum value (MinRegion) and minimum value (MaxRegion).

A coordinate is inside the region if

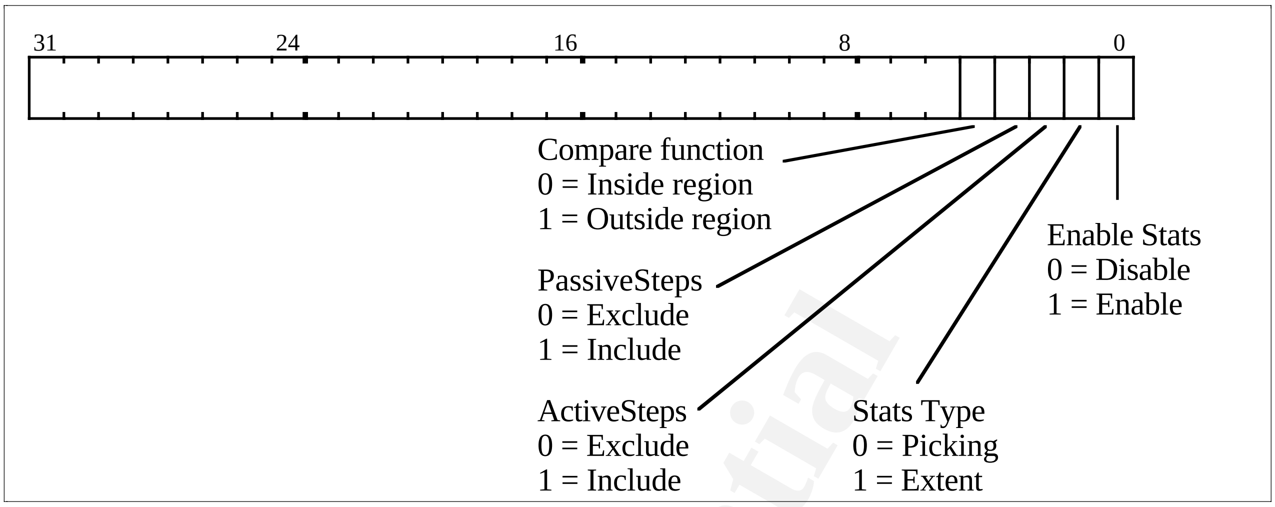
$$X_{min} \le X < X_{max}$$
$$Y_{min} \le Y < Y_{max}$$

where X and Y are from step message and the min/max values are from MinRegion and MaxRegion registers. This comparison is identical to the one used in the scissor tests.

The compares are done signed and the coordinates are in the same format as the coordinates in the step messages, i.e. 16 bit 2's complement with X in the least significant half of the 32 bit word and Y in the most significant half.

Once all the necessary primitives have been rendered the results can be found using the MinHitRegion and MaxHitRegion messages to have the MinRegion and MaxRegion registers respectively written into the FIFO as the data to these messages (under control of the Filter message). The picking result is found using the PickResult message is a similar way.

The Statistic collection is controlled using the StatisticMode message. It has the following format:



16.1.3. Synchronization

The Sync message if filtered and written to the FIFO in a similar fashion to the other messages. If an interrupt is required to be generated then the most significant bit of the data field is set *and* the filtering must be set up to write something into the FIFO. If nothing is written to the FIFO (because of the filter mode) then no interrupt will be generated. The actual interrupt will not be generated until the Sync data or tag has passed through and is on the output of the FIFO, to allow low level resynchronisation between the core and PCI clock domains. The FIFO has an extra bit in width to accommodate the interrupt signal. When both the data and tag are written into the FIFO only the first entry in the FIFO will cause the interrupt (assuming an interrupt was requested).

The remaining bits in the data field are free and can be used by the host to identify the reason for the sync message, for example.

16.2. Input Messages

16.2.1. External Messages

Host Out Messages		
Tag Mnemonic	Data Field	Description
FilterMode	See above	Filter control field
StatisticMode	See above	Statistics control field
MinRegion	ls 16 bits: min X ms 16 bits: min Y	Note this value is not retained if Extent testing is enabled.
MaxRegion	ls 16 bits: max X ms 16 bits: max Y	Note this value is not retained if Extent testing is enabled.
ResetPickResult	Not used.	
MinHitRegion	Not used on input	The data field is replaced with: ls 16 bits: min X ms 16 bits: min Y and passed to output filtering.
MaxHitRegion	Not used on input	The data field is replaced with: ls 16 bits: max X ms 16 bits: max Y and passed to output filtering.
PickResult	Not used on input	The data field is replaced with 0 for false and 1 for true and passed to output filtering.
Sync	See above	Optionally generates an interrupt and passed to output filtering.

16.2.2. Internal Messages

This unit reacts to all the messages in the Rasteriser Walk group, LBDepth, LBStencil and FBColour messages. See the Rasteriser Unit, LBRead Unit and the FBRead Unit for details on these messages.

16.3. Output Messages

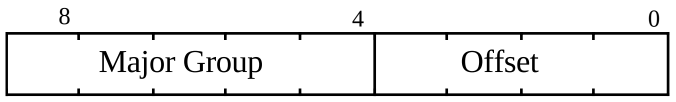
No new messages are generated, however various messages will be passed on the Host Out Interface FIFO and the data fields of the identified Host Out messages are filled in.

16.4. Behavioural Model

```
Wait for input message
{
  switch (on input message)
  {
    case FilterMode:
    case StatisticMode:
    case MinRegion:
    case MaxRegion:
            Update appropriate register;
            break;
    case ResetPickResult:
            Reset the pick result flag;
            break;
    case MinHitRegion:
    case MaxHitRegion:
    case PickResult:
            if (FilterMode.Statistics )
            {
              Wait for room in output FIFO and write
                corresponding tag and/or internal data as
                appropriate;
            }
            break;
    case PrepareToRender:
            if (FilterMode.Active)
            {
              Wait for room in output FIFO and write tag
                and/or data as appropriate;
            }
            break;
    case ActiveStepX:
    case ActiveStepYDomEdge:
            if (Statistics are enabled and include Active)
            {
              if (picking)
              {
                Compare XY coordinates against MinRegion and
                  MaxRegion and set picking flag accordingly;
              }
              else
              {
                Compare XY coordinates against MinRegion and
                  MaxRegion and update these registers
                  accordingly;
              }
            }
            if (FilterMode.Active)
            {
              Wait for room in output FIFO and write tag
                and/or data as appropriate;
            }
            break;
    case PassiveStepX:
    case PassiveStepYDomEdge:
            if (Statistics are enabled and include Passive)
            {
               if (picking)
               {
                  Compare XY coordinates against MinRegion and
                     MaxRegion and set picking flag accordingly;
               }
               else
               {
                  Compare XY coordinates against MinRegion and
                     MaxRegion and update these registers
                     accordingly;
               }
            }
            if (FilterMode.Passive)
            {
               Wait for room in output FIFO and write tag
                  and/or data as appropriate;
            }
            break;
      case LBDepth:
            if (FilterMode.Depth)
            {
               Wait for room in output FIFO and write tag
                  and/or data as appropriate;
            }
            break;
      case LBStencil:
            if (FilterMode.Stencil)
            {
               Wait for room in output FIFO and write tag
                  and/or data as appropriate;
            }
            break;
      case FBColour:
            if (FilterMode.FBColour)
            {
               Wait for room in output FIFO and write tag
                  and/or data as appropriate;
            }
            break;
      case Sync:
            if (FilterMode.Synchronization)
            {
               if (ms bit of data set)
               {
                  // Generate interrupt
                  Wait for room in output FIFO and write tag
                     and/or data, as appropriate, with the
                     interrupt bit set on the first entry
                     written;
               }
                  Wait for room in output FIFO and write tag
                     and/or data as appropriate;
            }
            break;
      default:
            if (FilterMode.Remainder)
            {
                Wait for room in output FIFO and write tag
                   and/or data as appropriate;
             }
             break;
   }
   Flush the input message;
}
```

17. Message Tags

The following table defines all the known message tags and their internal values in GLiNT. The tags have been grouped to make decoding them in hardware as gate efficient as possible. The basic structure of a tag field is:

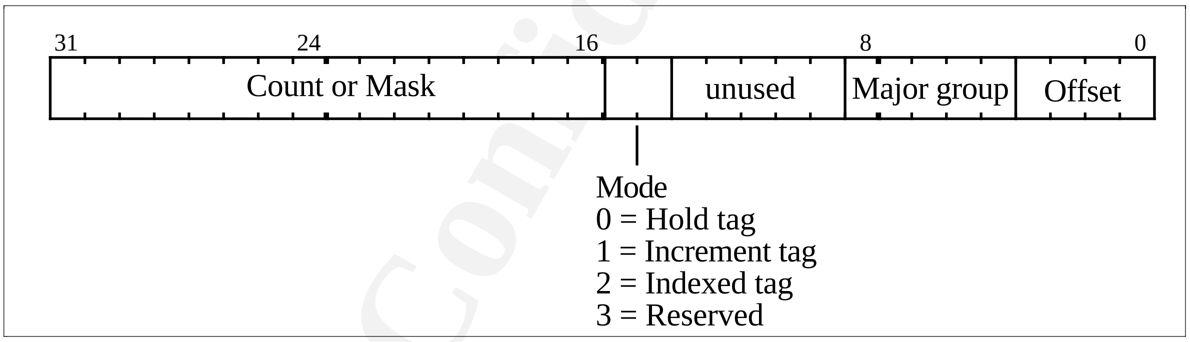


The external tag is only 9 bits (the context bit is internally set based on the FIFO which sourced the tag and data).

17.1. DMA assist

The DMA controller in GLiNT can be used to transfer tag/data pairs from the host memory into the GUI or 3D FIFOs. There are several mechanisms to reduce the burden of having to provide a tag with every data word and this is encoded into the tag word in the other 21 bits (the tag only takes 9 bits out of the 32).

The format of the tag word is as follows:



The Major group and Offset correspond to the tag table later on. The Count or Mask is a dual purpose field which is a count for Hold tag and Increment tag modes, but a mask for Indexed tag mode.

The count value is always specified as one less than the number of data values to load so, for example to load 1 item a count of zero is specified. This count value is only used for tag generation and shouldn't be confused with the count associated with the length of the DMA transfer. The DMA controller always works in 'multi tag' mode but the fields have been arranged so that if they are all set to zero except for the major group and offset then the basic tag/data pair structure is implemented.

The modes of operation are as follows:

Hold tag	In this mode the number of data words given by (count - 1) are read and inserted into the FIFO with the same tag value as specified in the Major group and Offset fields. This is typically used for image download.

Increment tag	In this mode the number of data words given by (count - 1) are read and inserted into the FIFO with the tag incremented between each data word. The start value is taken from the Group and Offset fields.
Indexed tag	In this mode the mask has a bit set for each tag to be generated. Within a Major group there are 16 possible offsets so each offset has a corresponding bit in the mask. Bit 0 in the mask corresponds to offset 0, bit 1 to offset 1, etc.. The bits in the index are tested from the least significant end so to load the StartXDom, dXDom and ContinueNewDom messages, for example, the 4 words in memory will be set up like this:

```
0000 0010 0000 0011 1000 0000 0000 XXXX
StartXDom data
dXDom data
ContinueNewDom data
```

The Offset field is not used so can be any value. This mode is available for any Major group, however it cannot extend across major group boundaries.

17.2. Read back

Many of the messages just cause the associated data in the data field to be stored in a register for use when fragments are being processed by the unit. To read back the data for a message the address which the message was written to is just read.

The read back is asynchronous to the message stream so enough time must be allowed to ensure the register being read back contains the latest data written. How much time to leave is impossible to say because it depends on what, if anything, the rasteriser is doing so it is better to use a Sync message to ensure the message stream is in a quiescent, or empty, before doing the read back.

Reading back a non-existent message, or one which does not support read back just returns all zeros. Reserved or undefined bits in a message are also returned as zero.

Unit	Message	Major Group (hex)	Offset (hex)	Can be read back	Public	Notes
Rasteriser	StartXDom	00	0	•	•	
	dXDom	00	1	•	•	
	StartXSub	00	2	•	•	
	dXSub	00	3	•	•	
	StartY	00	4	•	•	
	dY	00	5	•	•	
	Count	00	6	•	•	
	Render	00	7		•	
	ContinueNewLine	00	8		•	
	ContinueNewDom	00	9		•	
	ContinueNewSub	00	A		•	
	Continue	00	B		•	
	FlushSpan	00	C		•	
	BitMaskPattern	00	D		•	
	PointTable[0…3]	01	0…3	•	•	
	RasteriserMode	01	4	•	•	
	CoverageValue	02	0			
	PrepareToRender	02	1			
	ActiveStepX	02	2			
	PassiveStepX	02	3			
	ActiveStepYDomEdge	02	4			
	PassiveStepYDomEdge	02	5			
	FastBlockLimits	02	6			
	FastBlockFill	02	7			
	SubPixelCorrection	02	8			
Scissor Stipple	ScissorMode	03	0	•	•	
	ScissorMinXY	03	1	•	•	
	ScissorMaxXY	03	2	•	•	
	ScreenSize	03	3	•	•	
	AreaStippleMode	03	4	•	•	
	LineStippleMode	03	5	•	•	
	LoadLineStippleCounters	03	6	•	•	
	UpdateLineStippleCounters	03	7		•	
	SaveLineStippleCounters	03	8		•	
	WindowOrigin	03	9	•	•	
	AreaStipplePattern[0…31]	04 05	0…F	•	•	0…F in major group 04, 10…1F in major group 05
Texture Address		07				
		08				
Texture Read		09				
		0A				
		0B				
	Texel0	0C	0	•	•	
	Texel1	0C	1	•	•	
	Texel2	0C	2	•	•	
	Texel3	0C	3	•	•	
	Texel4	0C	4	•	•	
	Texel5	0C	5	•	•	
	Texel6	0C	6	•	•	
	Texel7	0C	7	•	•	
	Interp0	0C	8	•	•	
	Interp1	0C	9	•	•	
	Interp2	0C	A	•	•	
	Interp3	0C	B	•	•	
	Interp4	0C	C	•	•	
	TextureFilter	0C	D	•	•	

Texture/Fog Colour	TextureColourMode	0D	0	•	•	
	TextureEnvColour	0D	1	•	•	
	FogMode	0D	2	•	•	
	FogColour	0D	3	•	•	
	FStart	0D	4	•	•	
	dFdx	0D	5	•	•	
	dFdyDom	0D	6	•	•	
Colour DDA	RStart	0F	0	•	•	
	dRdx	0F	1	•	•	
	dRdyDom	0F	2	•	•	
	GStart	0F	3	•	•	
	dGdx	0F	4	•	•	
	dGdyDom	0F	5	•	•	
	BStart	0F	6	•	•	
	dBdx	0F	7	•	•	
	dBdyDom	0F	8	•	•	
	AStart	0F	9	•	•	
	dAdx	0F	A	•	•	
	dAdyDom	0F	B	•	•	
	ColourDDAMode	0F	C	•	•	
	ConstantColour	0F	D	•	•	
	Colour	0F	E	•	•	The readback is done from the Texture/Fog Colour Unit.
Alpha Test	AlphaTestMode	10	0	•	•	
	AntialiasMode	10	1	•	•	
LBRead	LBReadMode	11	0	•	•	
	LBReadFormat	11	1	•	•	
	LBSourceOffset	11	2	•	•	
	LBData	11	3			
	LBSourceData	11	4			
	LBStencil	11	5		•	
	LBDepth	11	6		•	
	LBWindowBase	11	7	•	•	Sequence carries on in LBWrite
GID Stencil Depth	Window	13	0	•	•	
	StencilMode	13	1	•	•	
	StencilData	13	2	•	•	
	Stencil	13	3	•	•	
	DepthMode	13	4	•	•	
	Depth	13	5	•	•	
	ZStartU	13	6	•	•	
	ZStartL	13	7	•	•	
	dZdxU	13	8	•	•	
	dZdxL	13	9	•	•	
	dZdyDomU	13	A	•	•	
	dZdyDomL	13	B	•	•	
	FastClearDepth	13	C	•	•	
	LBCancelWrite	13	D			
	LBWriteData	13	E			
LB Write	LBWriteMode	11	8	•	•	
	LBWriteFormat	11	9	•	•	
FB Read	FBReadMode	15	0	•	•	
	FBSourceOffset	15	1	•	•	
	FBPixelOffset	15	2	•	•	
	FBColour	15	3		•	
	FBData	15	4			
	FBSourceData	15	5			
	FBWindowBase	15	6	•	•	Sequence carries on in FBWrite.
Alpha Blend	AlphaBlendMode	10	2	•	•	
Dither	DitherMode	10	3	•	•	
Logical Ops	FBSoftwareWriteMask	10	4	•	•	

	LogicalOpMode	10	5	•	•	
	FBWriteData	10	6	•	•	The readback is done from the FB Write Unit.
	FBCancelWrite	10	7			
FB Write	FBWriteMode	15	7	•	•	
	FBHardwareWriteMask	15	8	•	•	
	FBBlockColour	15	9	•	•	
Host Out	FilterMode	18	0	•	•	
	StatisticMode	18	1	•	•	
	MinRegion	18	2	•	•	
	MaxRegion	18	3	•	•	
	ResetPickResult	18	4		•	
	MinHitRegion	18	5		•	
	MaxHitRegion	18	6		•	
	PickResult	18	7		•	
	Sync	18	8		•	

2. GLiNT1 Structure

Change History

Issue	Date	Change
1.0	5/3/94	First Issue
1.1	18/7/94	Updated to reflect the post simulation results on inter-FIFO depths. Added flow through control.

3. Rasteriser Unit

Change History

Issue	Date	Change
0.9	24/1/94	A new message Continue has been added to allow the current dda values to be carries over for both edges when rasterising a polygon. A new message RateriserMode has been added to hold some long term mode information. See the other changes for details on this. If this register contains zero then the functionality of the rasteriser is as documented in previous versions of this chapter. On the ContinueNewLine message the current DDA values can be used as is, have the fraction bits set to zero, or set to half. These messages do not now update the Count register (the number of scanlines is still sent as a parameter). The behaviour of the ContinueNewSub and ContinueNewDom messages has been clarified. The X and Y start values can be optionally biased by 0.5 when first loaded into the DDAs by the Render, ContinueNewDom and ContinueNewSub messages. The BitMaskPattern can now be consumed from the most significant end as well as the least significant end. The BitMaskPattern can now be optionally inverted before it is used. The StepXY message in ProcessLinePrimitive in the behavioural model has been replaced by the StepYDomEdge message as this was overlooked in a previous change. Some typos.
1.0	28/1/94	First issue under change control.
1.1	31/1/94	Section 6.6.4.2: The length of the subpixel span is calculated incorrectly when Xdom = Xsub = X at the pixel level but Xdom > Xsub within the pixel. The entry in the table for this case is fixed by taking the abs value of the expression (fract (Xsub) - fract (Xdom)). In the behavioural model for FillSpan if (Xcounter == XStop) a Generate (PassiveStepYDomEdge) is needed so that on slithers the downstream DDA units are kept in step when no fragments are generated in X direction. In the behavioural model for ProcessAntialiaseTrapezoidPrimitive the second LoadSubSpanData is a bug and has been deleted as it causes the coverage calculations to include the first span twice. The bitmask is now rotated rather than shifted to give a more definitive behaviour if it is consumed during an image download with bitmask (i.e. SyncOnBitMask and SyncOnHostData). Now the bitmask is reused.

1.2	22/2/94	Removed all reference to the second context. The FastFillLimits message has been changed to FastBlockLimits to be consistent with the rest of the core architecture. Additional bit CoverageEnable added to the Render message data field. The RasteriserMode message now allows the FractionalAdjust and BiasCoordinates to include *nearly half*, i.e. 0x7fff. Defined the format of the points table. The FastBlockLimits messages in the FillSpan behaviour should take their data from XStop and XCounter rather than XSub and XDom as these have already been stepped on. In the ProcessAntialiaseTrapezoidPrimitive more description has been added for the case when the PointTable is being used. Rasteriser action when aborted clarified. Missing flush message added to the MGSyncOnHostDataAndSyncOnBitMask behavioural model. In LoadSubSpanData the Increment subspan counter statement has been moved from the end and into the first if and second else blocks so that it isn't incremented when we have moved onto another scanline. The X and Y values used in GenerateMessage (or really MG* routines) are explicitly set up now as up to now they have not been defined. This also fixes the problem when antialiased trapezoids are rendered as the act of moving off the current scanline starts the subspan filling but now the scanline used in GenerateMessage is out by one. The scan direction set up in antialiasing (LoadSubSpanData routine) uses the full precision Xdom and Xsub rather than the integerised XCounter and XStop values so it can be set correctly for a slither which is within a pixel. Also the test at the end has been moved to just before the scanning direction is tested prior to updating the Xcounter and XStop values. The sub spanline coverage table has been expanded to include the case where $X_{dom} > X_{sub}$ and int (X_{dom}) = int (X_{sub}) = X. The heading have also been made more descriptive.
1.3	23/2/94	Added sub pixel corrections for aliased trapeziod rasterisation. This is controlled by a bit in the Render message. Extended the Render Data field tables in the example section to include the enables (Texture, Fog, Coverage and SubPixelCorrection) in the Render message.
1.4	14/3/94	Bit 14 specified twice in the Render message data field table. In ProcessTrapezoidPrimitive the Step X DDAs… statement needs to be moved to after the FillSpan line as it corrupts the sub pixel correction amount.

1.5	23/3/94	When running in BlockFill mode the last block is not written sometimes. An extra FastBlockFill message is sent when the while loop has finished to catch this case in the SpanFill routine.
1.6	13/4/94	Defined a Render message with an illegal primitive type to just consume the message and do nothing else (in ProcessRenderMessage behavioural code). Previously the rasteriser would scan convert a trapezoid. This change is in response to bug 40.
1.7	9/5/94	In MGSyncOnHostDataAndSyncOnBitMask behavioural model the current step is lost when the BitMaskPattern message is received. On receiving the BitMaskPattern message we need to carry on waiting for the image data. Ref. bug 101
1.8	23/5/94	Added FBData to the list of valid messages the rasteriser will wait for when SyncOnHostData is active. This addition extends the image download functionality to support format conversion from the input format to the framebuffer format.
1.9	28/6/94	Corrected the conditions under which the Message Generation Unit (section 6.6.6) outputs active messages to include the changes to the SyncOnBitMask functions (mask invert and take from ms end). Changed the scanning direction to be right to left when $X_{dom} = X_{sub}$. This was changed to bring the specs in line with the VHDL as it was considered to late to change the VHDL to conform to the spec! The last table in section 6.6.4.1 was changed.
1.10	30/6/94	Expanded on what happens when the scanning direction is indeterminate because $X_{dom} = X_{sub}$ in the behavioural model.

4. Scissor and Stipple Unit

Change History

Issue	Date	Change
0.9	11/1/94	Line stipple: Extra control bit added to allow the line stipple to be generated from the ms end as well as the original ls end. Area stipple: Extra control bit added to invert the pattern before it is tested. Area stipple: Extra control bit sadded to mirror the pattern in X and/or Y. Area stipple: X and Y offsets added so window relative patterns can be implemented using device coordinates. Xleft, Xright, Ytop, Ybottom, ScissorBL and ScissorTR changed to Xmin, Xmax, Ymin, Ymax and ScissorMinXY, ScissorMaxXY in the text as these had been overlooked when this change was originally done. LoadLineStippleCounters message moved to the Update Register table.
1.0	28/1/94	First issue under change control.
1.1	1/2/94	In the Area Stipple Implementation the mirror of the X and Y coordinates should be done before the extract and mask stage and not afterwards as shown.
1.2	16/2/94	Updated to remove all references to the second context.

5. Colour DDA Unit

Change History

Issue	Date	Change
0.9	13/1/94	Clamping description improved (but functionality not changed). DDA Update control table had Mux A and Mux B headings swapped.
1.0	28/1/94	First issue under change control.
1.1	14/2/94	Changed to introduce sub pixel corrections of the DDA values to improve image quality and eliminate 'bright edge' artifacts.
1.2	16/2/94	Removed all reference to the second context.
1.3	22/2/94	The Mux A entry for SubPixelCorrection in the DDA Update Controls table should be Cx and not Cy as shown.
1.4	18/7/94	Added clarification regarding using the Colour message from the host.

6. Texture/Fog Colour Unit

Change History

Issue	Date	Change
1.0	14/2/94	First issue.
1.1	16/2/94	Removed all reference to the second context.
1.2	18/2/94	Renames the dFog… messages to dF… to be consistent with other DDA names. Some omissions and "cut and paste" errors in the behavioural model fixed.
1.3	23/2/94	dFdyDomEdge message renames dFdyDom to make it consistant with other DDA names. Colour message was missed from the Scoreboard group. The behaviour when this message is received has also been changed. Some minor fixes to the behavioural model.
1.4	25/2/94	A CI mode for fog application has been added as the OpenGL equations are slightly different between CI and RGB modes. The fog DDA has been extended by 8 bits to cover a larger range. The blend equation in the behavioural model has been modified to fit in with the multiplexer A and B options in the respective tables.
1.5	14/3/94	Clarified when the registers R0…R3 are converted into 9 bit quantities. Corrected the description of when the Fog DDA should be set to output 1.0 and 0.0 with respect to the near and far ranges (top of page 12).
1.6	12/5/94	Clarification that the TextureFilter message just contains a single 3 bit wide field.

7. Alpha Test Unit

Change History

Issue	Date	Change
0.9	24/1/94	Added in the antialiase application function as its previous position in the pipeline (in the alpha blend unit) was wrong.
1.0	28/1/94	First issue under change control.
1.1	3/2/94	The coverage application is now qualified by the CoverageEnable bit (bit 15) in the PrepareToRender message so antialiasing can be temporarily disabled for a primitive.
1.2	16/2/94	Removed all reference to the second context.
1.3	25/5/94	Added clarification to what happens if the antialiasing calculation overflows in CI mode.
1.4	29/6/94	Removed the use of a result flag from the behavioural model (alpha test part) so the actual alpha test is done on an active step message and not on a colour message. This change brings this unit's behaviour more in line with other units and actually reflects what the VHDL does now. Previously the VHDL used the result flag method to save gates and the spec followed this so there was a documented mechanism to clear the result flag at the start of a primitive.

8. Local Buffer Read Unit

Change History

Issue	Date	Change
0.9	17/1/94	Added the LBDataFormat message to the external message table as it was omitted from the previous issue. This does not change the functionality of the unit.
1.0	28/1/94	First issue under change control.
1.1	16/2/94	Removed all references to the second context. The LBWindowBase table has therefore been replaced by a single register of the same name. LBDataFormat message omitted from the message tables.
1.2	17/2/94	More of the second context description removed as it was hiding too well yesterday!
1.3	28/2/94	Interface to the Local Buffer Interface Unit has been updated to tie in with what has been implemented (no functional change). Clarification in the behavioural model (no functional change).
1.4	15/3/94	Clarified the implementatin of the address adder chain with regard to overflow and sign extension.
1.5	16/3/94	Added extra signal to the interface table with the Local Buffer Interface Unit to indicate when the Write Data FIFO is empty. This is needed for the Sync message.
1.6	29/3/94	The LBWrDataEmpty interface signal with the Local Buffer Interface Unit has changed to LBWrComplete (no functional changes).
1.7	13/6/94	The Depth field is now right justified so the 'binary point' is always in a fixed place. This ensures the compare operations are more reliable. Ref. bug 137.
1.8	27/6/94	Highlighting the problem of changing the LBReadFormat while there are outstanding reads.

9. Graphic ID, Stencil and Depth Unit

Change History

Issue	Date	Change
0.9	14/1/94	The clamping of the depth value has been clarified (no functional change). Fast clear mechanism has changed so the FCS field in the Window message has changed name and got larger. Very minor changes to the commentary describing how fast clears interact with this unit. The allowed stencil widths has been reduced to fit in with the new layout of the local buffer (see Local Buffer Read and Write units for the changes here). Internally this makes no difference (except for the fewer stencil widths) as input/output to the unit is still the same 52 bit format. The GID field in the LBData field has been restored to its original position in the most significant nibble of the 52 bit data field. This is to present the same field ordering as in the local buffer itself. This is documented in the Local Buffer Read Unit spec.
1.0	28/1/94	First issue under change control.
1.1	1/2/94	Added table to describe DDA action under rasteriser actions. The tables describing when an active step is made passive and when a Write or Cancel message have been replaced by boolean equations to present a more compact view of the behaviour and also fix some undefined states in the tables. The LB Update Source field in the Window message should have LBSourceData as one of the options and not LBData otherwise it cannot be used to copy windows. In the stencil mode message the Stencil source field has been renamed to be LBWriteData Stencil to be more descriptive. The LBData message option in this field has been changed to Source Stencil to be less ambiguous and the documentation concerning this field has been updated. The table describing the options and use for the different Depth sources in the DepthMode message has had extra detail added to it. The format of the depth information has been clarified (no functional change).
1.2	1/4/94	The change to LBSourceData in the LB Update Source field (see 1.1 changes) was not made in some of the descriptive text. The Depth pass = True on page 11 should have been Stencil pass = True.
1.3	16/2/94	All reference to the second context has been removed. Some clarification on unit disable with fast clear operation added.

1.4	17/2/94	Sub pixel correction added to depth DDA. More second context stuff removed which escaped the knife on the last issue. The first line in both the depth and stencil tables to say what gets inserted into the LBWriteData message should take data from the LBSourceData message and not the LBData message as shown. This follows on from the 1.2 changes but were omitted at the time. The Z DDA unit is now only updated by rasteriser messages when the depth unit is enabled to be compatible with the other DDA units.
1.5	21/2/94	All update methods for the stencil operation are now masked to set all bits greater than the specified width to 0 to provide a more robust test of equality with the incoming LBData values. Bit values clarified in the stencil write and compare masks. The second line in both the depth and stencil tables to say what gets inserted into the LBWriteData message should be Source Depth and Source Stencil respectively. The fast clear mode ultimately selects the LBData or FC* registers as the true source.
1.6	14/3/94	Specified that all input data to a stencil update operation is masked to the width of the stencil buffer before the update calculation is done.
1.7	14/6/94	The Depth format has changed to place the binary point between the U and L registers. i.e. the format is 32 bit integer and 16 bit fraction.
1.8	27/6/94	The clamping of the depth value should have changed as a result of change 1.7, but this was omitted. The clamping has now changed.

10. Local Buffer Write Unit

Change History

Issue	Date	Change
0.9	18/1/94	Added formatting block. New message type added: LBWriteFormat.
1.0	28/1/94	First issue under change control.
1.1	16/2/94	All references to the second context removed. Specified the state undefined bits take after formatting.
1.2	17/2/94	LBWriteFormat message omitted from the behavioural model.
1.3	28/2/94	Clarification of the behavioural model (no functional changes).
1.4	16/3/94	Unit now responds to a Sync message so that a Sync message is only passed on when all outstanding writes have occured.
1.5	29/3/94	The LBWrDataEmpty signal does not guarantee when the writes have all be done because of pipelining in the Local Buffer Interface Unit so a new signal LBWrComplete is now used.
1.6	13/6/94	The right most bits in the Depth field are written to the local buffer so the 'binary point' is always in a fixed place. This ensures the compare operations are more reliable. Ref. bug 137.

11. Framebuffer Read Unit

Change History

Issue	Date	Change
0.9	13/1/94	In the Framebuffer Read Unit to Framebuffer Interface Unit interface the context bit in the WriteCmd was set to 4: is should be 3. The signal names between this unit and the Framebuffer Interface Unit have been prefixed with FB.
1.0	28/1/94	First issue under change control.
1.1	17/2/94	Removed all reference to a second context.
1.2	25/2/94	FBWriteMode removed from the Framebuffer Interface and the block mode information is now encoded into the FBWriteCmd. Some of the behavioural description has been improved but this hasn't changed the functionality.
1.3	15/3/94	Clarified the implementatin of the address adder chain with regard to overflow and sign extension.
1.4	16/3/94	Added extra signal to the interface table with the Framebuffer Interface Unit to indicate when the Write Data FIFO is empty. This is needed for the Sync message.
1.5	29/3/94	The FBWrDataEmpty interface signal with the Framebuffer Interface Unit has changed to FBWrComplete (no functional changes).

12. Alpha Blend Unit

Change History

Issue	Date	Change
0.9	24/1/94	The antialiasing coverage calculations have been removed from this unit because they need to occur before the alpha test unit to meet the OpenGL Spec.
1.0	28/1/94	First issue under change control.
1.1	16/2/94	Removed all reference to the second context.
1.2	11/3/94	Added a new RGB format (8 bit double buffered). The colour format numbers have also been re-assigned to allow for additional RGBA formats in the future.
1.3	4/5/94	Clarified the conversion of CI values into the internal format. (ref bug 60).
1.4	23/5/94	Clarified the order dependency between the Colour and AlphaBlendMode messages.
1.5	27/5/94	Added support for RGB in addition to the already supported BGR format.

13. Dither Unit

Change History

Issue	Date	Change
0.9	13/1/94	Error shift table for the 5:5:5:5 entry. The internal colour register is now updated even when the unit is disabled to be consistent with all the other units.
1.0	28/1/94	First issue under change control.
1.1	7/2/94	Added byte and nibble replication in CI8 and CI4 mode respectively to allow double buffering in CI mode within the same 32 bit framebuffer word.
1.2	16/2/94	All reference to the second context removed. Rounding mode in CI with no dithering highlighted.
1.3	11/3/94	Added a new RGB format (8 bit double buffered). The colour format numbers have also been re-assigned to allow for additional RGBA formats in the future.
1.4	27/5/94	Added support for RGB in addition to the already supported BGR format.
1.5	15/6/94	The colour values are duplicated in the front and back pixels when double buffering, rather than setting the other buffer to zero. This help single buffered windows on a double buffered screen.

14. Logical Ops and Write mask Unit

Change History

Issue	Date	Change
0.9	15/12/93	Missed references from previous issue to FBCancelWrite deleted. No change in functionality.
1.0	28/1/94	First issue under change control.
1.1	16/2/94	Removed all reference to the second context.
1.2	25/3/94	Added 'UseConstantFBWriteData' bit to the LogicalOpMode message.
1.3	4/5/94	Removed reference in the text to comparing the new framebuffer value to what is in the framebuffer to cancel the write. (ref bug 69).

15. Framebuffer Write Unit

Change History

Issue	Date	Change
0.9	14/12/93	All writes to the FIFO in the FB Interface unit are now qualified by a context bit. The FBHardwareWriteMask and FBBlockColour message values are held locally so they can be read back. The FBWriteData message just stored the data in a register. An active step message causes the actual write operation to proceed.
1.0	28/1/94	First issue under change control.
1.1	16/2/94	Removed all reference to the second context.
1.2	28/2/94	Clarification of the behavioural model (no functional changes).
1.3	14/3/94	Clarify the range of X values during pixel write mask generation. Changed the behavioural model to fix a bug where the FBCancelWrite would cause the write to be cancelled twice, once from the FBCancelWrite message and once from the ActiveStep message.
1.4	16/3/94	Unit now responds to a Sync message so that a Sync message is only passed on when all outstanding writes have occured
1.5	29/3/94	The FBWrDataEmpty signal does not guarantee when the writes have all be done because of pipelining in the Framebuffer Interface Unit so a new signal FBWrComplete is now used.
1.6	6/4/94	Active step messages should be passed on to the next unit. The behavioural model has been changed to do this.
1.7	24/5/94	Added UpLoadData to the mode register to allow the FBWriteData to be forwarded onto the Host Out unit. This is useful for diagnostics and for formatting of data during image upload.
1.8	25/5/94	The previous change should generate an FBColour message and not a Colour message as shown.

16. Host Out Unit

Change History

Issue	Date	Change
1.0	4/2/94	First issue.
1.1	16/2/94	Removed all reference to the second context. The bit numbering in the message filter data field should have started from zero and not one.
1.2	21/2/94	Changed the way interrupts and the Sync message work to simplify the low level interaction with the PCI interface.
1.3	3/5/94	Clarification of unused bits when the pick result is returned (ref bug 77).

17. Message Tags

Change History

Issue	Date	Change
1.0	28/1/94	First issue under change control.
1.1	3/2/94	Regrouped the Alpha Test, Alpha Blend, Dither and Logical Ops into one group to be more useful in Indexed Tag mode in DMA operation. The group and offset numbers for these units have all changed. The LBWrite group has been merged into the LBRead group to make Indexed Tags more useful. The FBWrite group has been merged into the FBRead group to make Indexed Tags more useful. Note the sort order remains the same even though the grouping has changed. BitmaskPattern tag was missed out of th'In GUI' column. FastFillLimits message was renamed to FastBlockLimits a while ago so has been changed here.
1.2	4/2/94	Added the message tags for the Host Out Unit.
1.3	17/2/94	Removed all reference to a second context. Moved the WindowOrigin register (formerly a table) into the main scissor/stipple group. Similaly for the LBWindowBase and FBWindowBase registers into their associated groups. Added the new messages introduced for the Texture and Fog Colour Unit.
1.4	18/2/94	Added in SubPixelCorrection message. Changed dFog… to dF… to be more consistent with DDA names. Area stipple table move to be on a 32 bit boundary. LB write and FB write message shifted down to make room for the *BWindowBase message (there was a clash).
1.5	23/2/94	Added in information to show which messages can have their data read back using the 'Read back' mechanism. dFdyDomEdge message changed to dFdyDom to bring it into align with the other DDA names. FBCancelWrite added as it was missing from the Logical Ops unit.
1.6	11/4/94	FBWriteData and Colour messages have been added to the readback list so when they are used by the host as optimisations they can be read back for context switches. The ResetPickRegion message name should have been ResetPickResult.

1.7	4/5/94	Added column to show which are public messages and which are private messages. All the Texture Read messages are now marked as being read backable. (ref bug 55). Interp1 was missed out from the Texture Read group and Interp5 does not exist. This has now been corrected.
1.8	12/5/94	The following messages were eroneously marked as readbackable: ResetPickResult, MinHitRegion, MaxHitRegion. Ref bug...104