P10®

Reference Guide Volume I - Overview

DRAFT

3D labs®

P 10®

Reference Guide Volume I - Overview

PROPRIETARY AND CONFIDENTIAL INFORMATION

Issue 1

Proprietary Notice

Email: info@3dlabs.com
Web: http://www.3dlabs.com

3 Dlabs Ltd.
Meadlake Place
Thorpe Lea Road, Egham
Surrey, TW20 8HE
United Kingdom
Tel: +44 (0) 1784 470555
Fax: +44 (0) 1784 470699

3 Dlabs K.K.
Shiroyama JT Mori Bldg 16F
40301 Toranomon
Minato-ku, Tokyo, 105, Japan
Tel: +81-3-5403-4653
Fax: +91-3-5403-4646

3 Dlabs Inc.
480 Potrero Avenue
Sunnyvale, CA 94086,
United States
Tel: +1 (408) 530-4700
Fax: +1 (408) 530-4701

Change History

Document	Issue	Date	Change
174.2.1 01	1	08/06/2001	Creation

User Note

This manual uses hyperlinks in MSWord file distributions to improve ease of access to relevant information for online users. To enable hyperlinks, the complete *Reference Guide* and *Programmer's Guide* file set should be in a single Windows directory or folder.

Table of Contents

1

Introduction

The P10 Graphics Processor is a scalable design using extensive parallelism and programmability to render multiple primitives per clock cycle and support texture-intensive APIs such as Microsoft DX8. Using ***programmable T&L*** and ***programmable pixel shaders*** in conjunction with highly optimised fixed-function units results in a simpler, faster and more flexible design.

Programmable registers also allow dynamic reconfiguration of the number of vertex shaders, the number of texture pipes and the number of rasterizers.

1.1 Design Performance

Performance estimates are based on design simulation rates pending availability of silicon-based test results. Primitive rates assume single tile coverage (reduced to 8x4 for z), Single directional light, Gouraud shaded, Depth buffered and .13 micron manufacturing. The feature set shown is in addition to features normally supported on earlier devices.

P10 Performance Overview		
3Dmark (DX8)		Bench-marks
ProCDRS-03 (Workstation)	133	
Quake III Quincunx FSAA (OpenGL)		
Points, lines	75M lines/Sec.	Primi-tives
Triangles	75M lines/Sec.	
AA Lines	75M lines/Sec.	
Vertex rendering – no depth, texture or lighting	150M vertices/sec.	Trans-form & Lighting
Vertex rendering – with depth, not texture or lighting	132M vertices/sec.	
Vertex rendering – texture and fog, no lighting	106M vertices/sec.	
Scissor (core:memory)	19.2: - G/sec. (64 primitives/ cycle)	Pixel Fill Rates
32bpp Clear (core:memory)	4.8:4.25 G/sec.	
GID rejected (core:memory)	19.2:17 G/sec.	
Trilinear (core:memory, 32bpp, one texel/pixel read)	1.2 : 1.1G/s	
Peak Memory Bandwidth	17 GBytes/s	Bus

P10 Performance Overview		
Max. memory	128Mbytes	
Operating Frequency (0.13 micron/.18 micron)	300MHz/200 MHz	
Up to 8 textures per primitive with any combination of trilinear, 3D, anisotropic filtering, bump mapping or cube mapping.	4	
Programmable texture co-ordinate generation	4	
Programmable shaders (i.e. texture combiners)	4	
Programmable pixel unit	4	
Accumulation buffering and convolution	4	
Precomputed displacement maps and tesselation	4	
T buffer full-scene antialiasing	4	
Integrated geometry and lighting	4	

Table 1.1 P10 Performance Overview

1.2 Changes from Earlier Chips

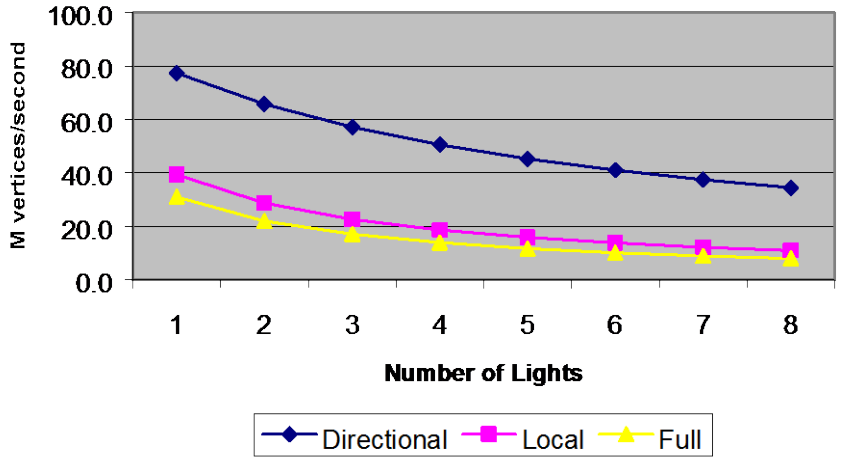


Table 1.3 Lighting Performance

Because of the extent of P10's paradigm shift a complete list of changes is pointless. However the table below illustrates the areas where developers wil find the most extensive innovation.

Previous Rasterizer Chips (P4/R4, MX)	P10
Scanline Framebuffer	Tiled framebuffer
DDA based interpolators	Plane equations
Edge-walking rasterization	Tile-seeking rasterization
Multiple cycles per primitive	Multiple primitives per cycle
Fixed function units	Fixed/Programmable hybrid
FIFO-based memory	Cache-based memory
Asynchronous pipeline	Parallel pipes with pre-emption
Command and control data visits every unit	Command and control independent routing

Table 1.1 Evolutionary Changes

1.2.1 Tile-based working

P10 adopts the tile as its sole unit of internal work. All operations are performed on 8x8 square screen-aligned ***planar byte pixel tiles*** similar to the 64x1 pixel spans used in earlier chips. All data types are stored the same way, so for example anything (e.g. the Depth buffer) can be a texture, and it is possible to render to a texture. Each memory access returns a planar byte tile. Two or more accesses are used for pixel depths greater than 8 bits, which allows unusual formats such as 24, 40 and 48 bpp. All memory accesses are virtual and page faults are handled with a CPU-like page swap.

This uniformity results in tile scalability and substantial performance improvements, particularly in 3D and small 2D primitives (e.g. characters) where the improved scanline coherence and memories efficiencies are most noticeable. Performance is further enhanced by the use of 256-bit DDR memories running at 266MHz (peak bandwidth 17GB/s).

1.2.2 Multitasking

Architecture innovations include the Context unit, which implements ***pre-emptive multitasking*** to support time-critical operations such as render during frame blank. The Context unit caches context data and keeps a copy in local memory. A small cache handles frequently updated values such as mode registers. When a context switch is needed the cache is flushed, the new context record is read from memory and the data converted into a message stream to update downstream units. Because only a small amount of cache data needs to be saved this process can be very fast – typically ¼ scanline.

1.2.3 Command Input

Unlike earlier graphics processors, P10 command and control data (register updates, mode changes etc.) does not generally take the same route as pixel data. This improves flexibility and bandwidth between units.

P10 uses two independent Command Units - one servicing the GP stream (for 3D and general 2D commands), the other servicing the Isochronous stream. Both command units

manage the Circular Buffers and Input DMA. The GP Command unit also manages Vertex Arrays.

1.2.3.1 Circular Buffers

Circular buffers, also new in P10, allow small packets of work to be transferred rapidly without incurring the delays and overhead of setting up a DMA buffer and making an escape call to the O/S. Because DMA transfers take time to initiate they are normally optimized for large bursts of data to improve efficiency. This can result in the graphics system being idle while some work has accumulated in the DMA buffer, but not enough to trigger a burst.

Circular buffers are usually stored in local memory and mapped into the ICD. When commands and data are added to the circular buffers, chip-resident write pointer registers are updated accordingly (without any O/S intervention). When the current circular buffer goes empty the hardware automatically searches the pool of 16 circular buffers for more work and instigates a context switch if necessary.

Circular buffers process the command stream identically to input DMA and can even call DMA buffers.

1.2.3.2 Vertex Arrays and Vertex Caching for Indexed Arrays

Vertex arrays are supported for compactness and flexibility in data layout. An array element can hold up to 16 parameters, which can be stored consecutively in memory or held in arrays. Vertex elements can be accessed in sequence or using array indices. The most recent 16 array indices are cached to allow comparison with the current index to check for vertex meshing, which in turn allows substantial savings in memory reads and Shader processing.

1.2.4 Scalability

The design allows unusual flexibility in adapting performance to specific applications and to market targets as well as future proofing:

- Tile size can be varied
- the number of texture pipes and vertex shaders is configurable
- Changing the number of pipes and shaders does not affect the API
- Memory devices can be picked to suit market conditions (although 256bit DDRs are preferred).
- When a programmable register is idle it can be reprogrammed on the fly as an additional rasterizer to further improve fill and small primitive rates.

1.2.5 Legacy Support

Because of the design paradigm shift it has not been possible to continue support for many legacy items. This has incidentally removed up to 40% of the total code lines, which translates into a substantial reduction in gate count and chip complexity and a smaller, more flexible and faster design.

1.3 Block Diagrams

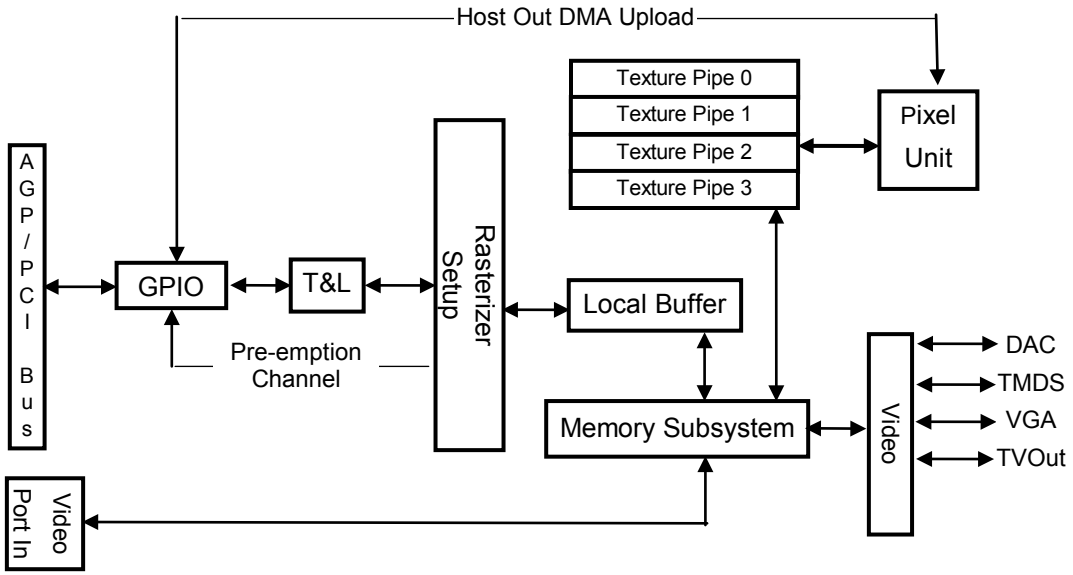


Figure 1.1 Chip-level Block Diagram

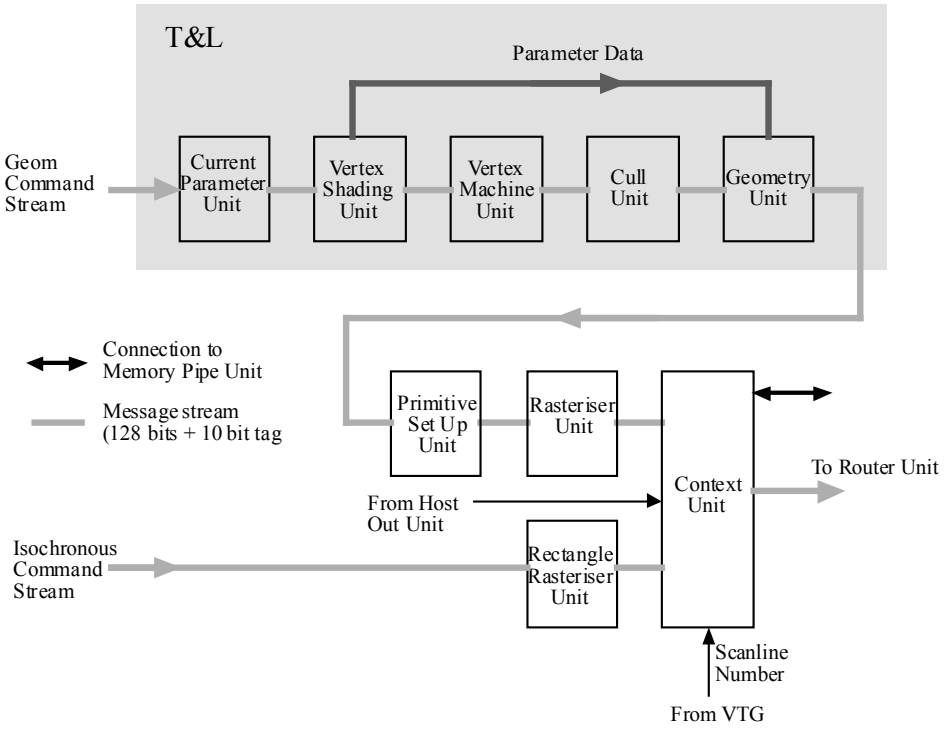


Figure 1.2 Transformation and Lighting Block Diagram

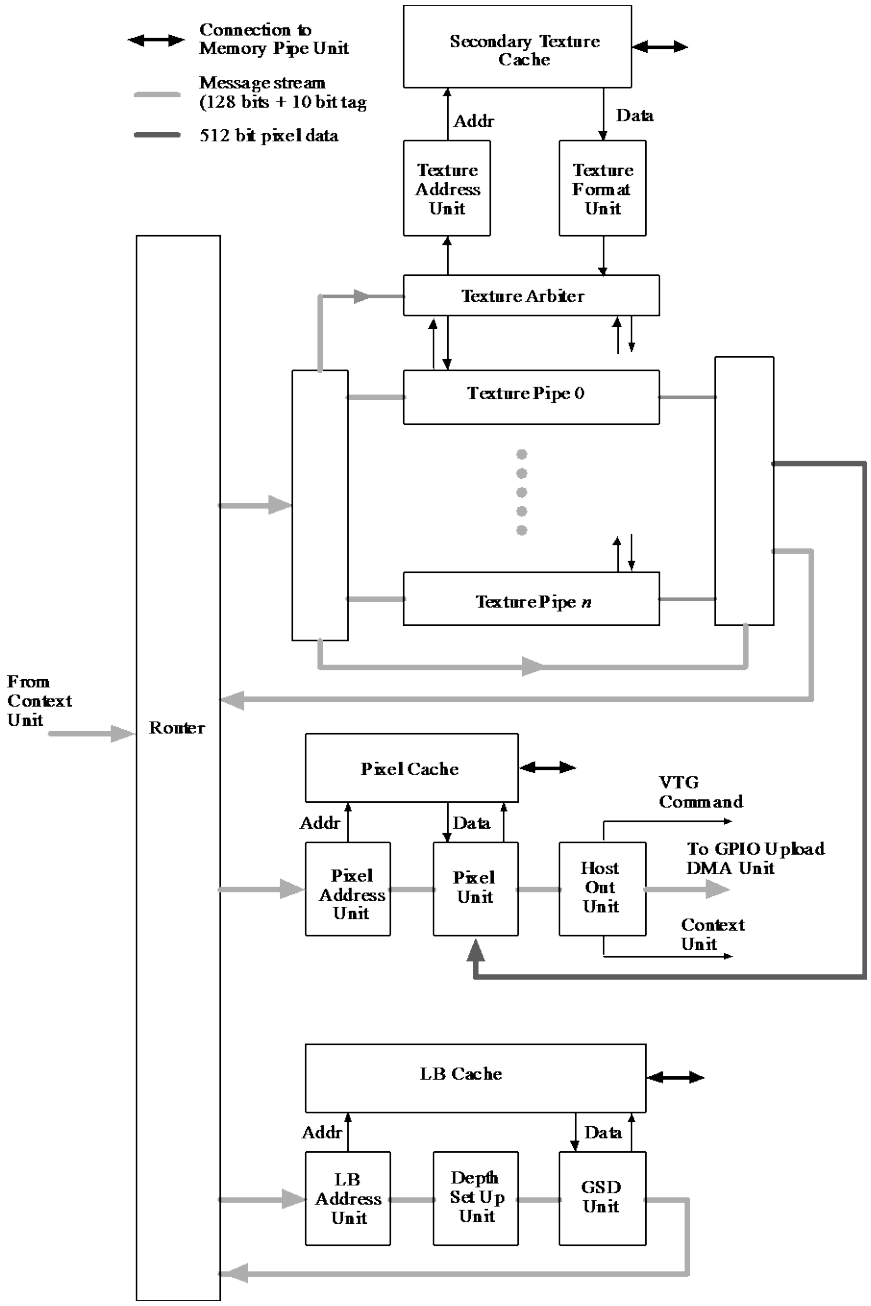


Figure 1.3 Rasterizer Block Diagram

1.3.1 Isochronous Command Stream and Context Switching

Microsoft's 'hot button' for GDI+ establishes a new requirement for real-time processing slaved to the display state to support tasks such as rendering during frame blank or non-tear bliting to a window.

P10 addresses this need by implementing a separate graphics core pre-emption channel which uses fast on-board context-switching (including switching during a primitive).

As context switchable state flows through into the rasterizer it goes through a Context Unit which snoops and caches the context data and keeps a local copy for context switches.

A second command queue handles real-time rendering commands, i.e. those using Video Timing Generator (VTG) and scanline timestamps. If the context switch is to allow isochronous rendering it invokes a small, dedicated isochronous stream rasterizer. A typical partial context switch to and from an isochronous context should take less than 700 cycles (3.5µs at 200MHz or ¼ scanline).

The Isochronous rasterizer only deals with rectangular primitives, which it can render in either direction. It is not a parallel blit engine – it is invoked only for Isochronous service requests using existing processor capacity.

For more information, see the **Timestamp**, **Changeport** and **HoldPort** commands in the *Miranda P10 Reference Guide* volume III.

2

Graphics Pipeline Organization

This chapter describes the Miranda P10 graphics pipeline functional layout.

2.1 Transform and Lighting System

Transform and Lighting (T&L) functionality includes five functional groupings ('Units') which handle vertex setup, transforms, lighting and culling.

2.1.1 Current Parameter Unit

The Current Parameter unit tracks the 16 possible vertex parameters (position, colour(s), texture coordinates(s) and normal) for each vertex and forwards any missing ones to the Vertex Shader Unit.

In the process, it substitutes any missing parameters the most recent version of that parameter. This is particularly relevant for OpenGL where a parameter like colour can be sent once and it is then applied to all subsequent vertices until re-sent. This is frequently found where the Begin/End paradigm is in use.

Note: For vertex arrays or vertex buffers in D3D this functionality is not needed as each parameter is supplied for every vertex.

To avoid passing all 16 parameters for each vertex to the Vertex Shader Unit, this unit counts how many times each parameter has been sent and stops sending when each recipient vertex store now holds it. For example if the Vertex Shader processes *n* vertices in parallel and the vertex store is double-buffered then after each parameter has been sent 2n times each vertex store should contain current values. This causes an initial flurry of transfers during context switches but after 2n vertices the steady state condition prevail and the minimum number of messages will be generated per vertex.

Note: The parameters are typeless – the names (VertexData0…VertexData15) are simply placeholders. The program running in the Vertex Shader Unit assigns meaning to the parameters, although conventional meanings are used in our documentation. This allows the use of the Vertex Shader for much more varied applications.

OpenGL can interrogate the current vertex values at any time. To avoid the performance constraints of tracking this in software, the **GetCurrent** command dumps the current values using the **Upload128** command so that they appear in the Host Out FIFO. From there they can be read or DMAed into memory.

All 16 parameters are written from this unit and the Vertex Machine Unit appends the current edge flag information.

Colour Material support: OpenGL allows the Colour parameter to be used to edit one or more material parameters on a per vertex basis. Updating the material values stored in the Coefficient memory in the Vertex Shading Unit would be very bad for performance as this would prevent the parallel vertex processing (it is done as a SIMD architecture where the Coefficient memory is broadcast to all processing elements). Instead the program is

changed to expect the material parameter to come from the colour parameter in the vertex store rather than the material value in the Coefficient memory. When the Colour Material mode changes the real material parameter(s) must be updated from the current colour. As outlined above the driver software is not tracking this parameter and you certainly don't want to do a Get to find its value (getting state out of the hardware is

2.1.2 Vertex Shading Unit

This performs the bulk of the transformation, lighting and texture generation work. As noted previously the unit is fully programmable. Programs can be up to 256 instructions long, including subroutines and loops. Details of programmable registers and the Vertex Shader instruction set are in "Programmable Registers", Volume III of the *P10 Reference Guide*.

The Vertex Shading Unit is implemented as a 16 element SIMD array, with each element (VP) working on a separate vertex. The floating point ALU in each VP is a scalar multiplier accumulator which also supports multi cycle vector instructions.

2.1.3 Vertex Machine Unit

Co-ordinate results from the Vertex Shader are passed to the Vertex Machine Unit via the message stream. The 16 parameter results go directly to the Geometry Unit via a private bus. (Two output ports allow for a higher vertex throughput.)

The Vertex Machine Unit monitors vertex coordinates (really window coordinates now) as they pass through. When enough vertices for the given primitive type have passed through, the unit issues a draw command for the appropriate primitive. Keeping the orientation of triangles constant, which vertex is a provoking vertex, when to reset the line stipple, etc. are all handled here. The Vertex Machine uses all 16 vertex cache entries (even though for many of the primitives it is not possible to extract any more than the inherent cache locality) as this greatly reduces the chance of stalling while loading a scoreboarded parameter register.

2.1.4 Cull Unit (Primitive Assembly)

The Cull Unit caches the window coordinates for the 16 vertices. When cull and geometry processing for a primitive starts it uses the cached window coordinates to test clip against the viewing frustum and, for triangles, do a back-face test. Any primitives failing these tests (if enabled) are discarded. Any primitives passing these tests are passed on. If the clip test is inconclusive the primitive is further tested against the guard band limits. A pass against these new limits means that it will be left to the rasterizer to clip the primitive while it is being filled - it can do this very efficiency and spends very little time in 'out of view' regions. A fail against the guard band limits or the near, far or user clip plane will cause the primitive to be geometrically clipped in the Geometry Unit.

2.1.5 Geometry Unit

The Geometry Unit holds the full vertex cache for 16 vertices. Each entry holds 16 parameters and a window coordinate. As each primitive is processed the Geometry Unit checks that the necessary vertex data is present. It tracks the progress of the destination circular buffers and the state of the downstream setup units. If vertex data is missing it supplies it. The Geometry Unit can accept vertex data faster than it can be passed on to

the rasterizer and it filters out vertex data for culled primitives. This allows for a faster cull rate than rendering rate.

Note: Primitives which need to be geometrically clipped are clipped in this Unit. [1] *The clip polygon is rendered as a series of triangles.*

2.2 Rasterizer Setup

The Rasterizer's Primitive Setup Subsystem decomposes geometric objects into primitives and converts windows relative coordinates into absolute coordinates.

2.2.1 Primitive Setup Subsystem

This subsystem is made up from:

- Primitive SetUp Unit
- Depth SetUp Unit
- Parameter SetUp Unit(s)

Input to this subsystem is the coordinates, colours, texture coordinates, etc. per vertex and these are stored in local vertex stores. The vertex stores are distributed so each Setup Unit only holds the parameters it is concerned with.

Primitive Setup does any primitive specific processing. This includes calculating the area of triangles, splitting stippled lines (aliased and antialiased) into individual line segments, converting lines into quads for rasterization and converting points into screen aligned squares for rasterization. Window relative coordinates are converted into fixed point screen relative coordinates. Finally it calculates the projected x and y gradients from the floating point coordinates (used when calculating the parameter gradients) for all primitives.

Depth Setup and Parameter Setup are very similar with the differences being limited to the parameter tag values, input clamping requirements and output format conversion. The Depth Setup Unit has a 16-entry direct-mapped vertex store. The common part is a plane equation evaluator which implements 3 equations - one for the gradient in x, one for the gradient in y and one for the start value. These equations are common for all primitive types and are applied once per parameter per primitive. The Setup units are adjacent to their corresponding units which will evaluate the parameter value over the primitive.

2.3 Rasterizer

The Rasterizer subsystem consists of a Rasterizer Unit and a Rectangle Rasterizer Unit.

The Rectangle Rasterizer Unit only rasterizes rectangles and is located in the isochronous stream – see the Isochronous Command Stream section for a discussion. .

The input to the Rasterizer Unit is in fixed point 2's complement 14.4 fixed point coordinates. When a Draw* command is received the unit will then calculate the 3 or 4

[1] This is done by calculating the barycentric coordinates for the vertices in the clip polygon using the Sutherland Hodgman clipping algorithm.

edge functions for the primitive type, identify which edges are inclusive edges (i.e. should return inside if a sample point lies exactly on the edge[2]) and identify the start tile.

Once the edges of the primitive and a start tile is known the rasterizer seeks out tiles which are inside the edges or intersect the edges. This seeking is further qualified by a user defined visible rectangle (VisRect) to prevent the rasterizer visiting tiles outside of the screen/window/viewport. Tiles which pass this stage will be either totally inside or partially inside the primitive. Tiles which are partially inside are further tested to determine which fragments in the tile are inside the primitive and a tile mask built up.

The output of the rasterizer is the **Tile** command which controls the rest of the core. Each **Tile** holds the tile's coordinate and tile mask. The tiles are always screen relative and are aligned to tile (8x8 pixel) boundaries. Before a **Tile** command is sent it is optionally scissored and masked using the area stipple pattern. The rasterizer generates tiles in an order that maximises memory bandwidth by staying within a single memory page as much as possible. Memory is organised in 8x8 tiles[3] and these are stored linearly in memory.

The rasterizer has an input coordinate range of ±8K, but after visible rectangle clipping this is reduced to 0…8K. This can be communicated to the other units in 10 bit fields for x and y by omitting the bottom 3 bits (which are always 0). Destination tiles are always aligned as indicated above, but source tiles can have any alignment. The Pixel Address Unit uses a local 2D offset to generate non aligned tiles, but converts these into 1, 2 or 4 aligned tile reads to memory, merges the results and passes them on to the Pixel Unit for processing.

The triangle, antialiased triangles, lines, antialiased lines, points and 3D rectangles are all rasterized with basically the same algorithm, however antialiased points and 2D rectangles are treated as special cases.

The **DrawRectangle2D** primitive is limited to rasterizing screen aligned rectangles but can rasterize tiles in any of four directions (left to right, right to left, top to bottom, bottom to top) so overlapping blit regions can be implemented. The rasterization of the rectangle is further qualified by an operation field so a rectangle can sync on host data (for image download), or sync on bit masks (for monochrome expansion or glyph handling) in which case the tiles are output in linear scanline order.

Each tile is visited multiple times, but with one row of fragments selected so that the host can present data in scanline order without any regard to the tile structure of the framebuffer. The packed host data is unpacked and aligned and sent to the Pixel Unit before the **Tile** command.

The host bitmask is aligned to the tile and row position and then forwarded to the Pixel Unit as a PixelMask before the **Tile** command, where it can be tested and used. Alternatively the bitmask can be anded with the **Tile** mask. For image upload the tiles can also be visited in scanline order.

---

[2] This needs to vary depending on which is the top or right edge so that butting triangles don't write to a pixel twice.

[3] The aim is to have memory appear as a linear layout and do any patching during the read or write operation, but if this proves impossible without sacrificing performance then a single tiled layout will be used by everything and any changes needed for internal operation (such as for texture caching) will be done on the fly. This will save having any units which read or write to memory from having to understand 4 different layout formats as in earlier chips..

2.3.1 Rasterization Process

The Rasterizer Unit handles arbitrary quad and triangle rasterization, antialias subpixel mask and coverage calculation, scissor operations and area stippling. The rasterization process can be broken down into three parts:

- Calculate the bounding box of the primitive and test this against the VisRect. The VisRect defines the only pixels which are allowed to be touched. In a dual P10 system each P10 is assigned alternating super tiles (64x64 pixels) in a checker board pattern. If the bounding box of the primitive is contained in the other P10's super tile the primitive is discarded at this stage.
- Visiting the tiles which are interior to, or on the edge of a primitive while spending no time visiting tiles outside the primitive or in clipped out regions of the primitive which fall outside of the VisRect. Extra sample points outside of the current tile being processed are used as 'out riggers' to assist in this. One other area where care is needed is on thin slivers of primitives which fall between sample points and give a zero tile mask, thereby giving the impression the edge of a primitive has been found.
- Computing the tile mask to show which fragments in the tile are inside the primitive. This also extends to calculating the coverage mask for antialiasing.

There are 4 edge function generators so that arbitrary quads can be supported, although these will normally be screen aligned parallelograms or non screen aligned rectangles for aliased lines or antialiased lines respectively. Screen aligned rectangles are used for 2D and 3D points. Triangles only need to use 3 edge function generators.

The edge functions will test which side of an edge the 64 sample positions in a tile lay and return an inside mask. ANDing together the 3 or 4 inside masks will give a tile mask with the inside fragments of the primitive for this tile set. Sample points which lie exactly on an edge need to be handled carefully so shared edges only touch a sample point once.

The sample points are normally positioned at the centre of the pixels[4], but when antialiasing up to 16 sample points are configured to lie within a pixel. The 16 subpixel sample points are irregularly positioned (via a user programmable table) on a regular 8x8 grid within the pixel so that any edge moving across a pixel will cover (or uncover) the sample points gradually and not 4 at a time. This emulates stochastic (or jittered) sampling and gives better antialiasing results as, in general, more intensity levels are used.

2.3.2 Antialiasing

Antialiasing is done by jittering the tile's position and generating a new tile mask. The jittered tile masks are then accumulated to calculate a coverage value or coverage mask for each fragment position. The number of times a tile is jittered can be varied to trade off antialiasing quality against speed. Tiles which are totally inside the primitive are automatically marked with 100% coverage so these are processed at non antialising speeds. This information is also passed to the Pixel Unit so it can implement a faster processing path for fully covered pixels.

The UserScissor rectangle will optionally modify the tile mask if the tile intersects the scissor rectangle or delete a Tile message if it is outside of the scissor rectangle. This, unlike the VisRect, does not influence which tiles are visited.

[4] D3D expects the sample point to be at the origin of the pixel and this is allowed for when the appropriate mode bit is set.

Finally the tile mask is optionally ANDed with the 8x8 area stipple mask extracted from the stipple mask table. The stipple mask held in the table is always 32x32 and screen aligned[5].

The rasterizer computes the tile mask in a single cycle and this may seem excessively fast (and hence expensive) when the remainder of the core is usually taking, say 4…8 cycles per tile. The reasons for this apparent mismatch are:

- To allow guard band clipping and scissoring to occur faster.
- Searching for interior tiles when the start tile is outside the primitive (maybe due to guard band clipping) is wasted processing time and should be minimised.
- To allow for some inefficiencies in tracking the primitive boundary where empty tiles outside the primitive are visited.
- The antialiasing hardware uses the same 64 point sampler to calculate the subsamples values so could take up to 16 cycles to calculate each fragment's coverage.
- It allows some simple operations to run much faster. Examples of this are clearing a buffer, GID testing and early exit depth testing.

Antialised points are processed in a different way as it is not possible to use the edge function generators without making them very expensive or converting the point to an polygon. The method used it to calculate the distance from each subpixel sample point in the point's bounding box to the point's centre and compare this to the point's radius. Subpixel sample points with a distance greater than the radius do not contribute to a pixel's coverage. The cost of this is kept low by only allowing small radius points hence the distance calculation[6] is a small multiply and by taking a cycle per subpixel sample per pixel within the bounding box. This will limit the performance on this primitive, however this is not a performance critical operation but does need to be supported as the software has no way to substitute alternative rendering commands due to polymode behaviour.

2.4 Texture

The texture subsystem is the largest and most complicated subsystem and will be further split up for this description.

The main components of the texture subsystem are:

- Texture Switch Unit
- One or more Texture Pipes
- Texture Arbiter Unit
- Texture Address Unit
- Texture Format Unit
- Secondary Texture Cache
- Texture Mux Unit

[5] This is much simpler than in earlier chips where different size stipple masks could be held and these masks could be aligned to window coordinates, screen coordinates and be mirrored and inverted. Now it is software's responsibility to replicate the mask to 32x32 and to realign if the window moves (if necessary).

[6] Really distance squared to avoid the square root.

The Texture Switch Unit provides the interface for all the texture unit (except the Parameter Unit and the Shading Unit) to the message stream. It will decode tags and, where necessary, cause the state in each the texture pipe to be updated.

A texture pipe does all the colour and texture processing necessary for a single tile so the Texture Switch Unit distributes the Tile messages in round robin fashion to the active texture pipes. Distributing the work in this fashion (as opposed to the alternative described in the footnote) probably takes more gates[7], but does have the following advantages:

- It allows the design to be more scalable and not limited to a power of two number of pipes.
- The performance is not quantised as much when the number of textures is not an exact multiple or fraction of the number of pipes. For example 3 textures would leave one pipe unused with the alternative scheme, whereas with this approach all pipes are kept at maximum throughput.
- The number of texture pipes is transparent to the software and the Texture Switch Unit can avoid using texture pipes with manufacturing defects. Obviously this will reduce performance but it does allow a device which would have otherwise been scrapped to be recovered and sold into a market where the drop in texture performance is acceptable. This will improve the effective manufacturing yield.
- The Texture Switch Unit is much simpler than would have been true with texture pipes working together with feedback from one pipe to the next.
- Small primitive performance is improved because each pipe only sets up and processes the tiles (i.e. small primitives) given to it.

Each texture pipe works autonomously and computes the filtered texture values for the valid fragments in the tile it has been given. It will do this for up to eight sets of textures and pass the results to the Shader Unit in the pipe, and potentially back to the Texture Coordinate Unit for bump mapping. Processing within the texture pipe is done as a mixture of SIMD units (Texture Coordinate Unit and Shading Unit) or one fragment at a time (Primary Texture Cache Unit and Texture Filter Unit) depending on how hard to parallelise the required operations.

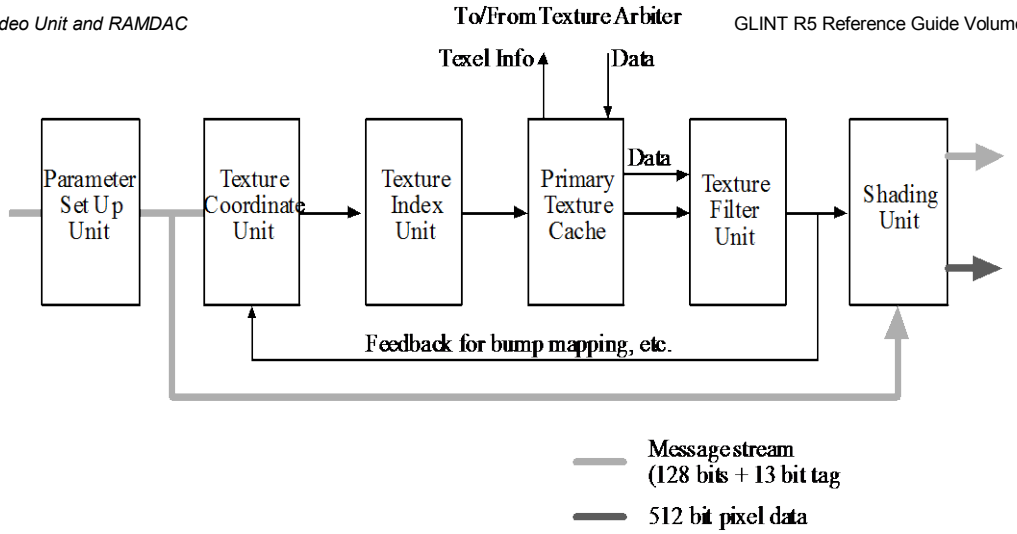
Each texture in a pipe can be trilinear filtered with per pixel LOD, cube mapped, bump mapped, anisotropic filtered and access 1D, 2D, or 3D maps. The texture pipe will issue read requests to the Texture Arbiter when cache misses occur. The texture pipe will be expanded on later.

The Texture Arbiter collects texture read requests from the texture pipes, serialises them and forwards them onto the Texture Address Unit. When the texture data is returned, after any necessary formatting, this unit will then route it to the requesting pipe. Each pipe has pair of ports in each direction so that requests from different mip map levels can be grouped together[8]. The arbitration between the texture pipes is done on a round robin basis.

---

[7] The 64 plane equations (8 texture coordinates and 8 colours) are duplicated as is the Parameter Setup Unit.

[8] This is probably not necessary now because all texture reads will come from a secondary level of cache so grouping requests from the same map together is probably not necessary (to get good memory efficiency) and reducing the number of ports may help layout etc..



The Texture Address Unit calculates the address in memory where the texel data resides. This operation is shared by all texture pipes (to saves gates by not duplicating it), and in any case it only needs to calculate addresses as fast as the memory/secondary cache can service them. The texture map to read is identified by a 3 bit texture ID, its coordinate (i, j, k), a map level and a cube face. This together with local registers allow a memory address to be calculated. This unit only works in logical addresses and the translation to physical addresses and handling any page faulting is done in the Memory Controller. The layout of texture data in cube maps and mip map chains is now fully specified algorithmically so just the base address needs to be provided. The maximum texture map size is 8Kx8K and they do not have to be square or a power of two in size[9].

Once the logical address has been calculated it is passed on to the Secondary Texture Cache Unit. This unit will check if the texture tile is in the cache and if so will send the data to the Texture Format Unit. If the texture tile is not present then it will issue a request to the Memory Pipe Unit and when the data arrives update the cache and then forward the data on. The cache lines hold a 256 byte block of data and this would normally represent an 8x8 by 32bpp tile, but could be some other format (8 or 16 bpp, YUV or compressed). The cache is 4 way set associative and holds 128 lines (i.e. for a total cache size of 32Kbytes), although this may change once some simulations have been done. Cache coherence with the memory is not maintained and it is up to the programmer to invalidate the cache whenever textures in memory are edited. The Secondary Texture Cache capitalises on the coherency between tiles or sub tiles when more than one texture is being accessed.

The primary texture cache in the texture pipes always holds the texture data as 32bpp 4x4 tiles so when the Texture Format Unit receives the raw texture data from the Texture Secondary Cache Unit it needs to convert it into this format before passing it on to the Texture Arbiter Unit. As well as handling the normal 1, 2, 3 or 4 component textures held as 8, 16 or 32 bits it also does any YUV 422 conversions (to YUV 444) and expands the DX compressed texture formats. Indexed textures are not handled directly but are converted to one of the texture formats when they are downloaded. Border colours are

[9] Mip mapping requires a map size which is a power of two or has a border. Cube maps are always square and each face always has the same size map (if present).

converted to a memory access as the border colour for a texture map is held in the memory location after the texture map.

The Texture Mux Unit collects the fragment data for each tile from the various texture pipes and the message stream and multiplexes them to restore temporal ordering before passing them onto the Pixel Unit or Router respectively.

2.4.1 Texture Pipe Components

A Texture Pipe comprises six units:

- Parameter Setup Unit
- Texture Coordinate Unit
- Texture Index Unit
- Primary Texture Cache Unit
- Texture Filter Unit
- Shading Unit

2.4.1.1 Parameter Setup

The Parameter Setup Unit sets up the plane equations for the texture coordinates and colour values used in the Texture Coordinate Unit and Shading Unit respectively. See earlier for details.

2.4.1.2 Texture Coordinate

The Texture Coordinate Unit computes one or more perspectively correct texture coordinates for each fragment and the appropriate level of detail (lod) when mip mapping. In addition the texture coordinates can be perturbed by an earlier texture access (bump mapping) or treated a the index into a cube (cube mapping). Higher qualities of filtering are supported by way of anisotropic mip mapping and high order filters (bicubic for example). Texture coordinates can have 1, 2 or 3 components to support 1D, 2D or 3D texture maps. A fragment can have multiple texture maps applied to it and any combination of the above are allowed by the APIs. There is support for 8 simultaneous texture maps.

The Texture Coordinate Unit is a programmable SIMD array 4x4 in size which runs a program once per sub-tile for those sub tiles with valid fragments. All the texture calculations for a sub-tile are done before moving on to the next sub sub-tile.

A SIMD architecture is used so these steps are carried out sequentially, but on multiple fragments at a time. If this 'program' takes *n* cycles to implement and the desired performance is to generate one set of texture coordinates per cycle then the SIMD array needs to hold *n* fragment processors. The value of *n* is constrained to be a power of two for ease of implementation and as will be seen later (in the section on programming) the above program takes about 14 cycles to run, hence *n* is ideally 16 processors.

Plane equation evaluation, cube mapping coordinate selection, bump mapping transformation and coordinate perturbation, 3D texture generation, perspective division and level of detail calculation are all done by the program. Anisotropic filtering loops through the program depending on the amount of filtering needed. The integration of the different filter samples in the Shading Unit is controlled from here.

2.4.1.3 Texture Index Unit

The final conversion to fixed point u, v, w coordinate includes an out-of-range test so the wrapping is all done in the Texture Index Unit.

The Texture Index Unit takes the u, v, w, lod and cube face information from the Texture Coordinate Unit and converts it in to the texture indices (i, j, k) and interpolation coefficients depending on the filter and wrapping modes in operation. Filtering across the edge of a cube map is handled by surrounding each face map with a border of texels taken from the butting face. Texture indices are adjusted if a border is present. The output of this unit is a record which identifies the 8 potential texels needed for the filtering, the associated interpolation coefficients, map levels and face number.

2.4.1.4 Texture Cache

The Primary Texture Cache Unit uses the output record from the Texture Index Unit to look up in the cache directory if the required texels are already in the cache and if so where. Texels which are not in the cache are passed to the Texture Arbiter so they can be read from memory (or the secondary cache) and formatted. The read texture data passes through this unit on the way to the Texture Filter Unit (where the data part of the cache is held) so the expedited loading can be monitored and the fragment delayed if the texels it requires are not present in the cache. Expedited loading of the cache and FIFO buffering (between the cache lookup and dispatch operations) allows for the latency for a round trip to the secondary cache without any degradation in performance, however secondary cache misses cause stalls[10].

The primary cache is divided into two banks and each bank has 16 cache lines, each holding 16 texels in a 4x4 patch. The search is fully associative and 8 queries per cycle (4 in each bank) can be made. The replacement policy is LRU, but only on the set of cache lines not referenced by the current fragment or fragments in the latency FIFO. The banks are assigned so even mip map levels or 3D slices are in one bank while odd ones are in the other. The search key is based on the texel's index and texture ID not address in memory (saves having to compute 8 addresses). The cache coherency is only intended to work within a sub tile or maybe a tile and never between tiles.[11]

2.4.1.5 Texture Filter Unit

The Texture Filter Unit holds the data part of the primary texture cache in two banks and implements a trilinear lerp between the 8 texels simultaneously read from the cache. The texel data is always in 32 bit colour format and there is no conversion or processing between the cache output and lerp tree. The lerp tree is configured between the different filter types (nearest, linear, 1D, 2D and 3D) by forcing the 5 interpolation coefficients to be 0.0, 1.0 or take their real value. The filtered results are passed on to the Shading Unit and include the filtered texel colour, the fragment position (within the tile), a destination register and some handshaking flags. The filtered texel colour can be feedback to the Texture Coordinate Unit for bump mapping or any other purpose.

---

[10] It is very likely that some texture access patterns (bilinear minification, for example) or simultaneous misses in all texture pipes will also cause some stalls. The impact of these could be reduced by making the latency FIFO deeper.

[11] Recall that the tiles are distributed between pipes so it is very unlikely adjacent tiles will end up in the same texture pipe and hence Primary Texture Cache Unit.

2.4.1.6 Shader Unit

The Shading Unit is a programmable SIMD machine operating on a logical 8x8 array of bytes (i.e. one per fragment position within a tile). The physical implementation uses a 4x4 array to save gate cost. The Shading Unit is passed up to 8 tiles worth of texture data, has storage for 32 plane equations (an RGBA colour takes 4 plane equations) and 32 byte constant values. These values are combined under program control and passed to the Pixel Unit, via the Texture Mux Unit, for alpha blending, dithering, logical ops, etc. Fragments within a tile can be deleted so chroma keying or alpha testing is also possible. All synchronisation (i.e. with the texture data) is done automatically in hardware so the program doesn't need to worry where the texture data will come from or when it will turn up.

Typically the Shading Unit program will do some combination of Gouraud shading, texture compositing and application, specular colour processing, alpha test, YUV conversion and fogging12.

The Shading Unit's processing element is 8 bits wide so takes multiple cycles to process a full colour. The ALU has add, subtract, multiply, lerp and a range of logical operations. It does not have divide or inverse square root operations. Saturation arithmetic is also supported and multi byte arithmetic can be done. Programs are limited to 128 instructions and conditionals jumps and subroutines are supported. The arrival of a Tile message initiates the execution of a program and a watchdog timer prevents lockups due to an erroneous program.

In order to support some of the more complex operations such as high order filtering, convolution and go beyond 8 textures per fragment several programs can be run on the same sub tile, with different input data before the final fragment colour is produced. This multi pass operation is controlled by the Texture Coordinate Unit. This works in a very similar way as the multi pass operation of the Pixel Unit.

2.5 Framebuffer

The Framebuffer subsystem is responsible for combining the colour calculated in the Shading Unit with the colour information read from the framebuffer and writing the result back to the framebuffer. Its simplest level of processing is therefore antialiasing coverage, alpha blending, dithering, chroma keying and logical operations, but the same hardware can also be used for doing accumulation buffer operations, multi buffer operations, convolution and T buffer antialiasing. This is also the main focus for 2D operations with most of the other units (except the rasterizer) being quiescent, except perhaps for some of the esoteric 2D operations such as anisotropically filtered perspective text.

2.6 Local Buffer

See Localbuffer in the *Miranda P10 Programmer's Guide* volume I.

12 Table based fog is implemented as a 1D texture as it is too expensive to allow each fragment access to an internal look up table.

2.7 Memory Pipe

P10 memory is cache-based and all data types are stored as 8bpp planar tiles. All memory access is logical/virtual and page faults cause CPU-like page swaps.

Memory is preferably 256 bit wide DDR devices running at 266MHz. From 32MB to 256MB of x32 devices are supported, or alternatively up to 512MB of x16 devices.[13] SDR devices are not supported.

There are two independent 128bit controllers which hold alternating groups of 8 tiles. Memory is divided into 4 regions corresponding to the 4 internal banks of a DDR device:

Bank	Controller 0	Controller 1
0	0-7	8-15
1	16-23	24-31
2	32-39	40-47
3	48-55	56-63
0	64-71	72-79

Local memory is used to store color, depth, stencil, and texture data. These are largely interchangeable depending on the microcode application context. For more information on data typing and usage refer to the *Miranda P10 Programmers Guide*.

For more information on Memory devices and layouts see “Memory Systems” in the *Miranda P10 Reference Guide*.

2.8 AGP/PCI Interface

2.8.1 PCI Interface

2.8.1.1 PCI Target features

- PCI Config Space transactions
- PCI Memory Space transactions
- PCI Fast Writes (2X and 4X)
- PCI I/O Space transactions
- VGA palette write snooping
- 32-bit and 64-bit addressing (dual address cycles)
- PCI multi-function operation

2.8.1.2 PCI Master features

- PCI Memory Space transactions
- 32-bit read and write data transfers
- 32-bit and 64-bit addressing (dual address cycles)

[13] The additional address lines will somewhat constrain performance with x16 memories.

2.8.2 AGPBus

AGP 4X is Intel's high performance, component level interconnect targeted at 3D display applications, which uses a 66MHz PCI specification as an operation baseline and provides significant performance extensions to the PCI specification.

Implementing these features enables P10 to achieve better than 1 GByte per second bandwidth from the host for instructions, textures, video data (limited by the host system throughput).

The add-in slot defined for AGP uses a connector body which is not compatible with the PCI connector. Boards designed for use in an AGP slot are not mechanically interchangeable with PCI boards. P10 supports AGP2x, AGP4x and PCI at signal voltages from 1.5vdc to 3.3vdc only. Legacy 5vdc PCI logic may severely damage the chip.

2.8.2.1 AGP Master features

- AGP low-priority Read transactions
- AGP low-priority Write transactions
- AGP Fence and Flush transactions
- Operation at 1X, 2X, and 4X data rates
- Sideband and pipe operation
- 48-bit addressing using sideband
- 64-bit addressing using pipe and dual address cycles

2.8.3 SVGA

The on-chip SVGA unit is register level compatible with standard VGA devices and requires no software emulation. It natively supports all standard VGA modes and certain VESA VBE extended modes.

The following standard VESA VBE extended video modes are supported - those not supportable by the SVGA unit may be supported using the Graphics Processor:

Table 1- 2 VESA VBE Graphics Modes

Mode (hex)	Pixels	Colors	Window-ed	Lin-ear	Support-able in SVGA	Support-able in GP
0x100	640x400	256	3	3	3	3
0x101	640x480	256	3	3	3	3
0x103	800x600	256	3	3	5	3
0x105	1024x768	256	3	3	5	3
0x107	1280x1024	256	3	3	5	3
0x109	320x200	32K (5:5:5:1)	3	3	5	3
0x10D	320x200	64K (5:6:5)	3	3	5	3
0x10F	320x200	16.8M (8:8:8)	3	3	5	3
0x110	640x480	32K (5:5:5:1)	3	3	5	3
0x111	640x480	64K (5:6:5)	3	3	5	3
0x112	640x480	16.8M (8:8:8)	3	3	5	3
0x113	800x600	32K (5:5:5:1)	3	3	5	3
0x114	800x600	64K (5:6:5)	3	3	5	3
0x115	800x600	16.8M (8:8:8)	3	3	5	3
0x116	1024x768	32K (5:5:5:1)	3	3	5	3
0x117	1024x768	64K (5:6:5)	3	3	5	3
0x118	1024x768	16.8M (8:8:8)	3	3	5	3
0x119	1280x1024	32K (5:5:5:1)	3	3	5	3
0x11A	1280x1024	64K (5:6:5)	3	3	5	3
0x11B	1280x1024	16.8M (8:8:8)	3	3	5	3

The following VESA VBE text modes are supportable in the SVGA:

Table 1- 3 VESA VBE Text Modes

Mode (hex)	Characters (col/row)
0x108	80x60
0x109	132x25
0x10A	132x43
0x10B	132x50
0x10C	132x60

P10 allows VESA bankswitching to be done through the bypass to enable additional VESA mode support. ModeX is RAMDAC. P10 incorporates high performance 350MHz RAMDAC. Typical screen resolutions up to 1600x1200 are supported with refresh rates of 96Hz or 1920x1080 with refresh rates of 90Hz, or 2048x1536 at 60Hz. It supports packed pixel formats, with color depths of 8, 16, 24, 32 and 40 bits per pixel. It has 4 dot-clock

phase locked loops (PLLs) and triple 10-bit D/A converters. The RAMDAC contains a 64x64x2 bit cursor array to support a 2, 4, or 16 color hardware cursor with cursor shapes cache.

2.8.4 Video Overlay

The video overlay is used to display incoming video data on screen. The overlay selection is based on a transparent color, the overlay key, which can be any RGB color or alpha value. Optionally, the overlay can be blended with the main image by using a 2-bit blend factor. A filter process supports zooming and shrinking at any rate. It combines four pixels into one by using bilinear filtering to achieve best results. Furthermore the filtered output is optionally converted from YUV to RGB color space format.

2.9 DMA

P10 supports a comprehensive set of DMA engines and uses Circular buffer input stream handling to reduce Command DMA setup overhead and latencies. Input streams can be from host or on-card memory with two levels of nesting. Output DMA returns data to host or local memory, performs image uploads and state return.

2.9.1 Graphics Core to Graphics I/O – Upload Controller

The GPIO Upload DMA Unit – GPIOUD – uploads message data from the graphics pipeline to the PCI and AGP bus masters.

The unit is controlled by PCI slave register writes and reads, which are resynchronised from P clock to K clock and back through the PCI slave write (PciGpWr) and PCI slave read (GpPciRd) FIFOs respectively.

The GP input half of the unit maintains 2 input message ports and 16+1 circular buffers. These generate outgoing message streams on the API and Isochronous output message FIFOs.

The GP output half of the unit maintains an output message port and a Sync interrupt signal. These are driven from the incoming message stream on the input message FIFO.

- Autonomous - set-up/fetch parallelism
- No wait state - maximum transfer rate
- Programmable block size - large DMA buffers
- Separate DMA controllers for upload and download can run concurrently

2.9.2 Graphics I/O to Geometry and Rasterizer – GPIO Command DMA

The GPIO Command DMA Unit issues DMA requests and processes the return data for GP command packets. These are inserted into the message stream. DMA packets are usually submitted via circular buffers which manage the GP core command interface.

2.9.3 Circular Buffers

Apart from the input message port, the circular buffer provides the only command interface to the GP core. They replace the GP Input FIFO and command DMA schemes of earlier chips.

The intention is that 16 user contexts (Api) and the GDI+ driver (Iso) each have their own private circular buffer backed by a DMA engine.[14] Wraparound is handled automatically by the GPIO Bus Interface.

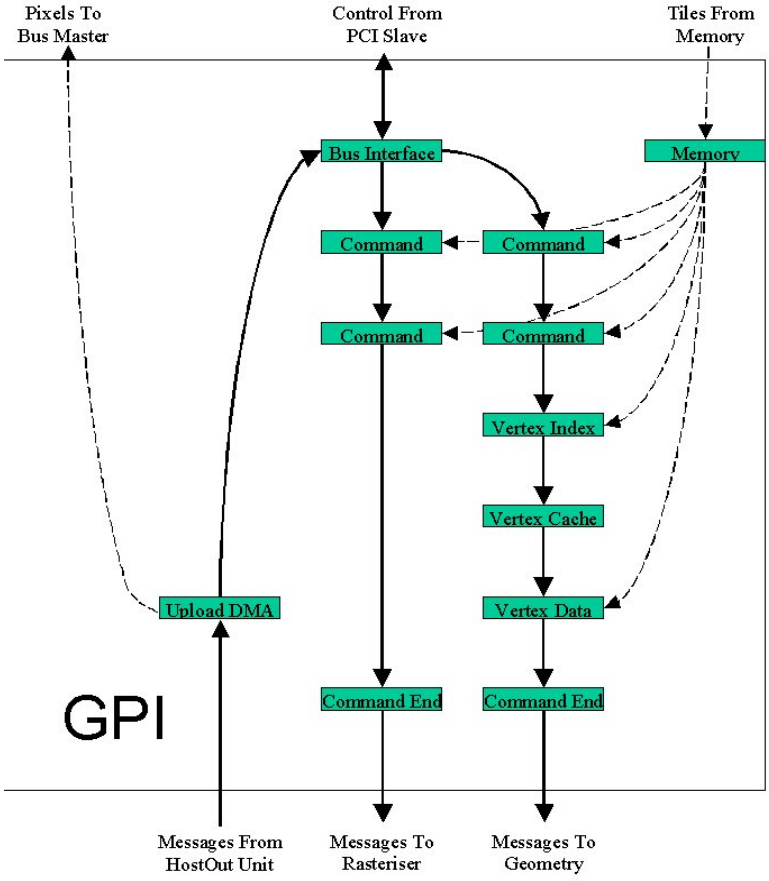


Figure 2.5 Graphics Processor I/O

[14] A "user context" here is considered to be the display driver, an OpenGL ICD process, or anything else wanting to make use of the GP core for 2D or 3D rendering.

2.9.4 Interrupt Controller

- End-of-DMA - allows DMA chaining
- VSYNC - efficient double buffering
- Scanline - special effects
- Texture invalid
- Bypass DMA interrupt
- I2C start condition - alert host to start of I2C transfer
- Sync - indicates graphics core is idle
- Error - e.g. writing to a full FIFO

2.9.5 Video Streaming

P10 supports digital video output. The 24-bit streamed output is designed to work with common PAL/NTSC encoders and flat panel controllers.

2.9.6 ROM support

P10 supports a Flash ROM. The ROM stores code needed for device-specific initialization and the SVGA BIOS. For more information see the *P10 Reference Guide*, volume 4, "Reset".

3

Address Maps and Regions

3.1 PCI Configuration Region

The PCI Configuration Space is intended to provide an appropriate set of configuration 'hooks' which satisfy the needs of current and anticipated system configuration mechanisms. The registers in this 256-byte space are accessed and modified by the use of PCI Configuration Read and Write commands, and are normally initialised by BIOS or similar low-level code at system power-up and reset. The configuration registers are described in detail in *P10 Reference Guide* volume II.

When configured for multi-function operation the bus interface provides a unique 256-byte configuration space for each PCI function, but will map accesses to other regions to the same underlying hardware regardless of the function being addressed.

3.1.1 Control Registers

Region Zero is a 256 KByte region containing control registers, and ports to and from the graphics processor. The control space is mapped twice within the 256 KByte region. In the second 128K the registers are mapped to be byte swappable for big endian hosts. See Section 3 of this document for further details of Region Zero.

3.1.2 Memory Apertures

Two separate apertures are provided to allow access to local memory. Each has a programmable size, and can be disabled if required.

As well as being used to access local memory, these two apertures can also be programmed to allow reading and writing of the Expansion ROM. This ensures that the "ROM" is visible beyond system boot time, allowing an EEPROM device to be reprogrammed in the field. Finally, either aperture can be programmed to forward memory accesses to the VGA memory controller.

3.1.3 Expansion ROM

In earlier 3Dlabs bus interface designs a number of parameters for the bus interface were initialised at reset time using pull-up or pull-down resistors connected to *configuration pins*. These pins were normally tri-state at reset, and their state was sampled on the trailing edge of reset. These configuration signals were then loaded into the *ChipConfig* register, which was used to control the initialisation and operation of the device.

This approach becomes less practical as external signal speeds increase, and so the current design loads all but the most critical initialisation information from the external Expansion ROM. Loading from the ROM is enabled using a single "RomConfig" configuration pin, and default initialisation values are used for registers when loading is disabled.

Once the Configuration Table pointer has been read a sequence of 32-bit words are loaded from the table into configuration space registers in the PCI Config unit and control registers in the ROM Controller unit:

Table Offset	Table Field	Destination Unit	Destination Register
00h	BusConfig	PCI Config	CFGBusConfig
04h	FunConfig	PCI Config	CFGFunConfig
08h	Subsystem	PCI Config	CFGSubsystemID *and* CFGSubsystemVendorID
0Ch	DevConfig	PCI Config	CFGDevConfig
10h	DevConfigMask	PCI Config	CFGDevConfigMask
14h	RomTiming	ROM Controller	ROMTiming

The **CFGBusConfig, CFGFunConfig, CFGDevConfig**, and **CFGDevConfigMask** registers are described below. Each of these four user-defined registers is shared between all functions in a multi-function device, and accesses through any function are mapped to the same underlying register hardware by the bus interface.

3.1.4 VGA Addresses

The bus interface can be configured to respond to standard VGA-compatible Memory and I/O Space addresses (memory addresses 0xA0000 through 0xBFFFF, and various I/O addresses in the ranges 0x3B0 through 0x3BB and 0x3C0 through 0x3DF). Further details are given in Section 4 of this document (“Video and RAMDAC”).

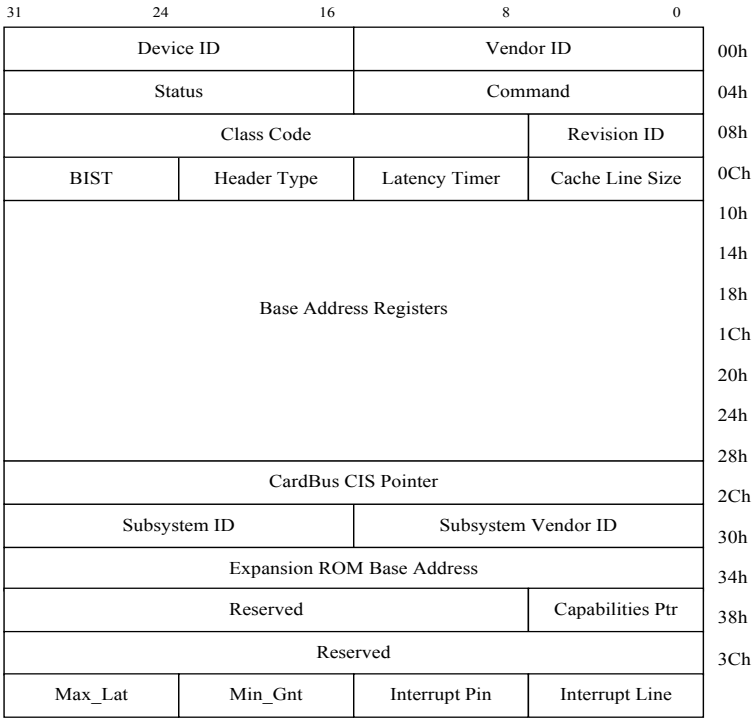


Table 3.1 Predefined and Base Address Registers (offsets 00h to 3Fh)

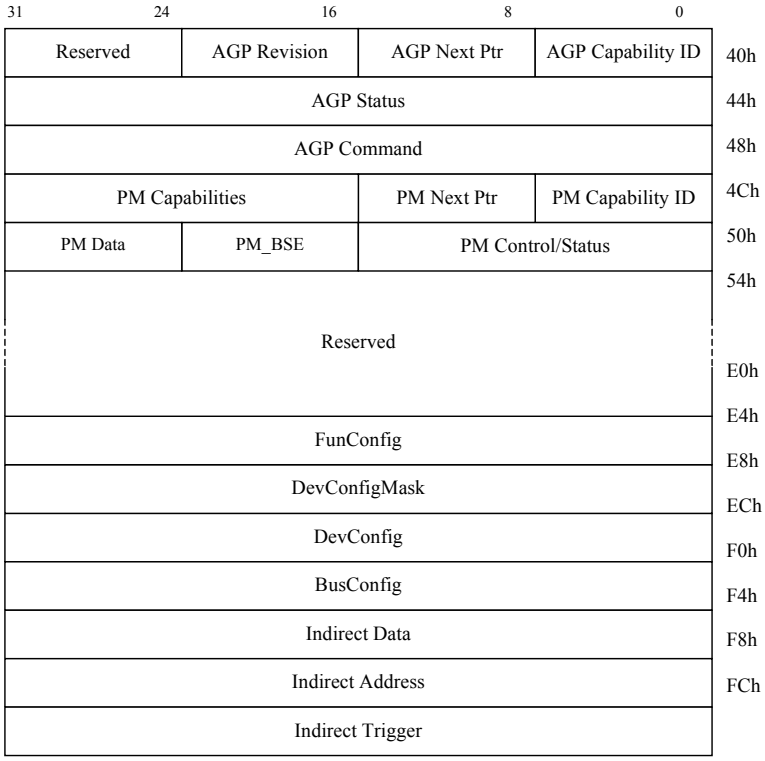


Table 3.2 AGP, Power Management, and User-Defined Registers (offsets 40h to FFh)

3.2 Region Zero Address Map

Region Zero is a 256 KByte region containing control registers and ports to and from the graphics processor. The control space is mapped twice within the 256 KByte region. In the second 128K the registers are mapped as byte swapped for big endian hosts.

Region Zero Address Map			
Address Range	**Size (bytes)**	Sub-Region Select	**Byte Swap**
0x00000 – 0x00FFF	4 K	Bus Interface CSR	No
0x01000 – 0x01FFF	4 K	Interrupt Control	No
0x02000 – 0x02FFF	4 K	Video Head 0 Control	No
0x03000 – 0x03FFF	4 K	Memory Control	No
0x04000 – 0x04FFF	4 K	VGA Control	No
0x05000 – 0x05FFF	4 K	ROM Control	No
0x06000 – 0x06FFF	4 K	Bypass Control	No
0x07000 – 0x07FFF	4K	Video Port Control	No
0x08000 – 0x08FFF	4K	Video Head 1 Control	No
0x09000 – 0x0EFFF	24 K	*reserved*	n/a
0x0F000 – 0x0FFFF	4 K	GPIO Driver Registers	No
0x10000 – 0x1FFFF	64 K	GPIO “User” Registers	No
0x20000 – 0x20FFF	4 K	Bus Interface CSR	Yes
0x21000 – 0x21FFF	4 K	Interrupt Control	Yes
0x22000 – 0x22FFF	4 K	Video Head 0 Control	Yes
0x23000 – 0x23FFF	4 K	Memory Control	Yes
0x24000 – 0x24FFF	4 K	VGA Control	Yes
0x25000 – 0x25FFF	4 K	ROM Control	Yes
0x26000 – 0x26FFF	4 K	Bypass Control	Yes
0x27000 – 0x27FFF	4K	Video Port Control	Yes
0x28000 – 0x28FFF	4K	Video Head 1 Control	Yes
0x29000 – 0x2EFFF	24 K	*reserved*	n/a
0x2F000 – 0x2FFFF	4 K	GPIO Driver Registers	Yes
0x30000 – 0x3FFFF	64 K	GPIO “User” Registers	Yes

Table 3.3 Region Zero Address Map

3.2.1 Reserved Registers

All accesses to *reserved* sub-regions in the table above are intercepted and handled by the bus interface: writes are discarded, and reads return zero. Accesses to non-reserved sections of the address map are forwarded to the appropriate target unit. The bus interface has no information about the internal register map of individual target units, so where target units have a sparse register map they themselves are responsible for

handling accesses to reserved registers. By convention, they too should absorb writes and read back zero from reserved addresses.

3.2.2 PCI Address Regions

The PCI Slave interface implements six PCI Address Regions, shown in the table below. The standard VGA compatible Memory and I/O Space addresses are decoded when the device has been suitably configured. These addresses do not form a single contiguous region but are mentioned in the table for completeness:

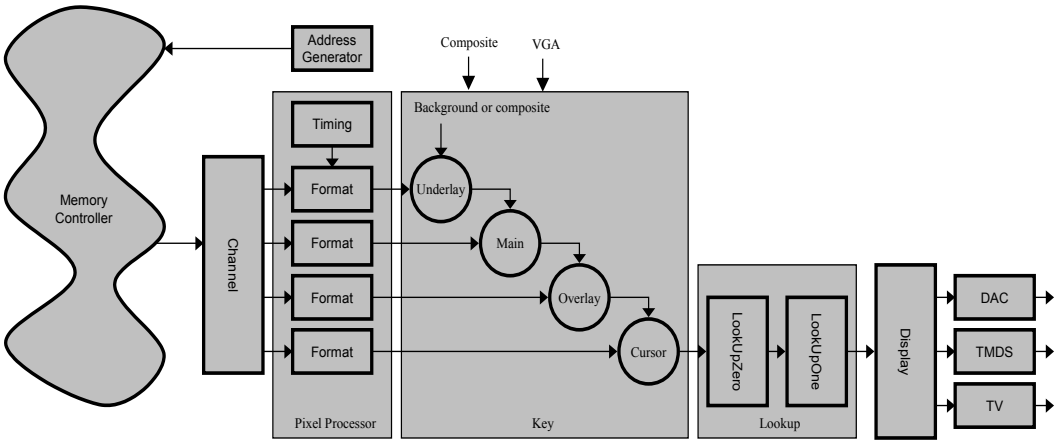
PCI Address Regions				
Region	**Address Space**	**Size (bytes)**	**Description**	**Comments**
Config	Configuration	256	PCI Configuration	PCI Special
Zero	Memory	256 K	Control Registers	relocatable
One	Memory	configured	Memory Aperture One	relocatable
Two	Memory	configured	Memory Aperture Two	relocatable
ROM	Memory	64 K	Expansion ROM	relocatable
VGA	Memory & I/O	-	VGA Address	optional & fixed

Table 3.4 PCI Address Regions

4

Video and RAMDAC

P10 Video displays data held in memory. It generates the video timing, requests data from memory, formats the data returned,and prepares it for display. There are four indpendent channels that can each request data; they are normally used for the underlay, the main image, the overlay, and the cursor. The four channels are combined to form a single channel by applying color key and blend operations. The result is passed through two lookup tables one after the other before being sent to a display device such as DAC or a TV encoder.



All channels are identical except that they have a fixed stacking order (cursor over overlay over main over underlay) and the main and overlay channels can be run in stereo mode.

4.1 LUTs

There are two lookup tables to remap the pixel color. Typical uses include:

- using one table to dereference index data while another gamma corrects RGB data
- supporting two different gammas (perhaps one for video and another for 3D).

Each channel selects which LUTs to use from a bitfield in a register; each channel can use one or both LUTs. Alternatively the LUT may be selected from the upper 2 bits of the alpha component, again as a bit field indicating one or both LUTs.

If a LUT is in index mode, as opposed to RGB mode, it uses one channel to index all 3 components; the channel to use can be selected.

4.2 Display Resolutions

TBD

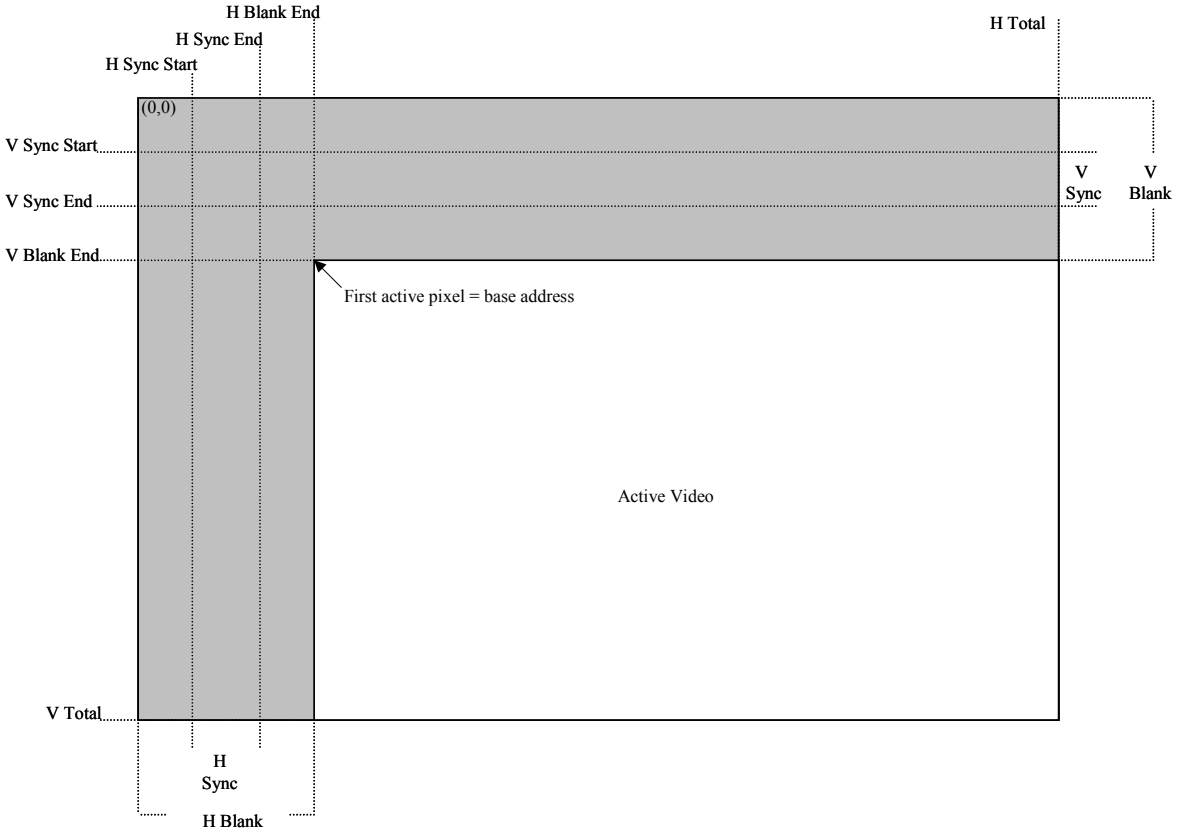
4.3 Display Data Channels

P10 supports both analogue and digital I/O. Digital input channels include an I2C bus

4.4 Analogue Display Timing Parameters

All timing values are relative to the first clock of vertical blank (i.e. the first pixel in vertical blank is at zero horizontal and 0 vertical). The horizontal values are defined in pixels and the vertical value in lines. The base address of the frame buffer is the first pixel to be displayed. Addresses must be aligned to the nearest tile and offsets with the tile specified through the pan regsiter. The screen stride may be different to the screen width.

The values loaded into registers represent the pixel (or line) on which the event takes place, so the horizontal blank end register holds the last pixel of blank; as the count is from zero, the number in the register is length of the horizontal blank.



There is an implementation requriement that the horizontal total value be no less than 4 and that there must be at least one line of vertical blank.

4.4.1 Synchronization

There are two lock bits which may be used to synchronize different channels within a head, or different heads. The lock registers hold a mask of which channels take part in the lock; there are two lock registers per head.

The locking operation uses an open-drain pin which is pulled high by a resistor. When a channel is not ready to synchronize it pulls the pin low, so when the pin is sampled it returns 'not locked.' When a channel is ready to synchronize it tri-states the pin, and when all channels have tri-stated the pin is pulled high and returns 'locked.'

All heads have access to all lock pins so they can be used to synchronize two heads in the same chip; the pins can also be shared by separate chips.

Note: Synchronization, PLL setup and Genlocking are described in more detail in the P10 Programmer's Guide Chapter 6 – Synchronization – and section 5.6.3, Dual Head Video Oputput. .Sync on Green is not supported on P10

4.4.2 Multi-Head

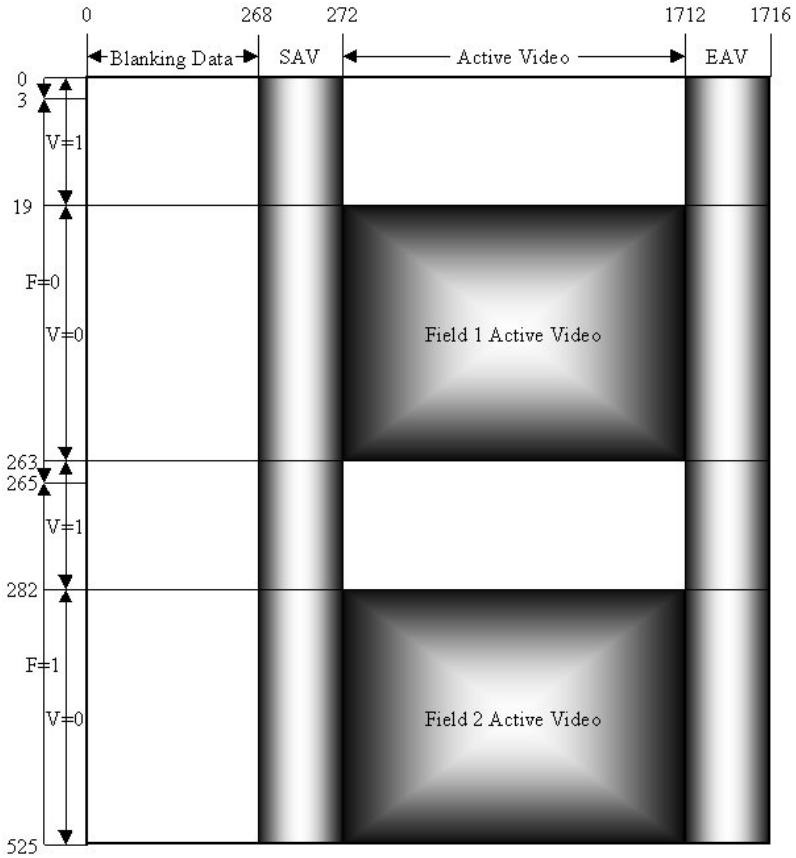
There is one PClk process that holds all registers for all heads; most registers are unique to the head they control, some are shared by all heads. The DClk processes are repeated for each head in the system.

4.5 Digital Display Timing Parameters

VIP2 DTV display formats are specified by the position of the SAV and EAV. These in turn identify the task, field, and blanking intervals.

In the VPU, horizontal samples are counted from 0 at the start of the horizontal blanking interval, and the first line of the frame is the first line of the vertical blanking interval. In interlaced video, with two vertical blanking intervals, the start field is also specified.

For example, this is how the 525-line System is formatted for the VPU:



Refer to the *P10 Programmer's Guide* for more information.

4.6 Multi-rasterizer and Genlock

Refer to the *Miranda P10 Programmer's Guide* for details of multi-rasterizer and Genlock implementation and use.

4

INDEX