文件模板

文件名称：Mali450

文件编号：0001

版 本：A.1

拟 制

审 核

会 签

标准化

批 准

修改记录

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 文件编号 | 版本号 | 拟制人/  修改人 | 拟制日期/  修改日期 | 更改理由 | 主要更改内容  （写要点即可） |
| 0001 | A.1 | xx | 2018-12-18 | 初始版本 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 注：文件第一次归档时，“更改理由”、“主要更改内容”栏写“无”。 | | | | | |

目 录

[1. 模板标题1 3](#_Toc1684283382)

[1.1 模板标题2 3](#_Toc1267391699)

[1.1.1 模板标题3 3](#_Toc397310112)

[1.1.1.1 模板标题4 3](#_Toc476051819)

## 模板标题1

见下文

### 模板标题2

见下文

#### 模板标题3

见下文

##### 模板标题4

见下文

###### 模板标题5

见下文

模板标题6

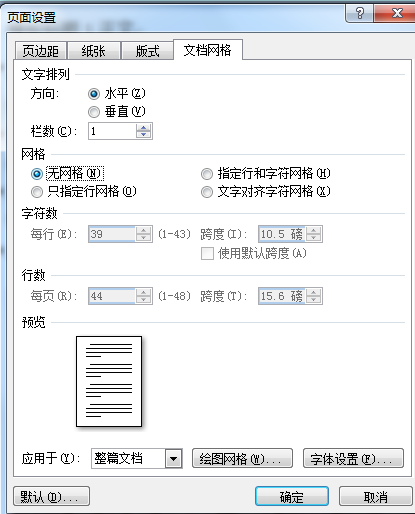
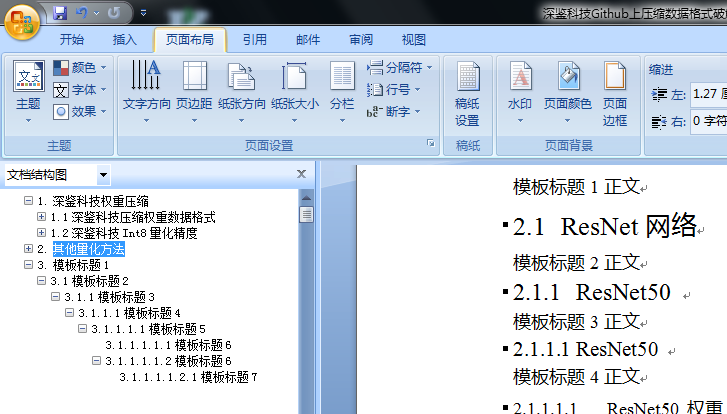
见下文

模板标题6

见下文

模板标题7

见下文



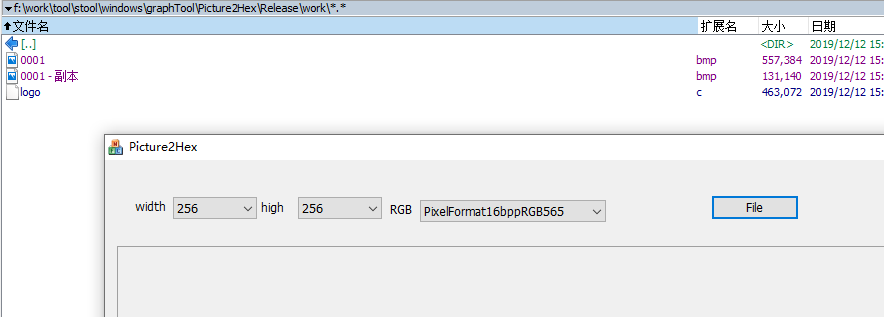
对于WPS来说，可以从”文件”-->”页面设置”-->”文档网络”，然后选择“无网络”

## Mali450

见下文

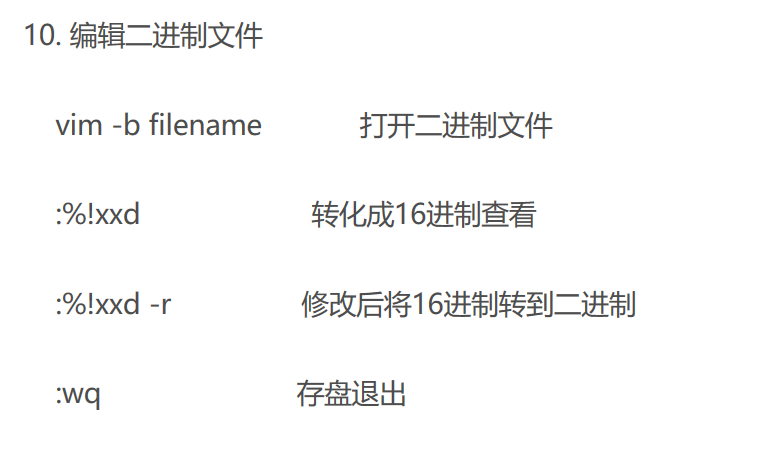
### 图像显示方法

#### 借助工具转换成rgb565图片显示

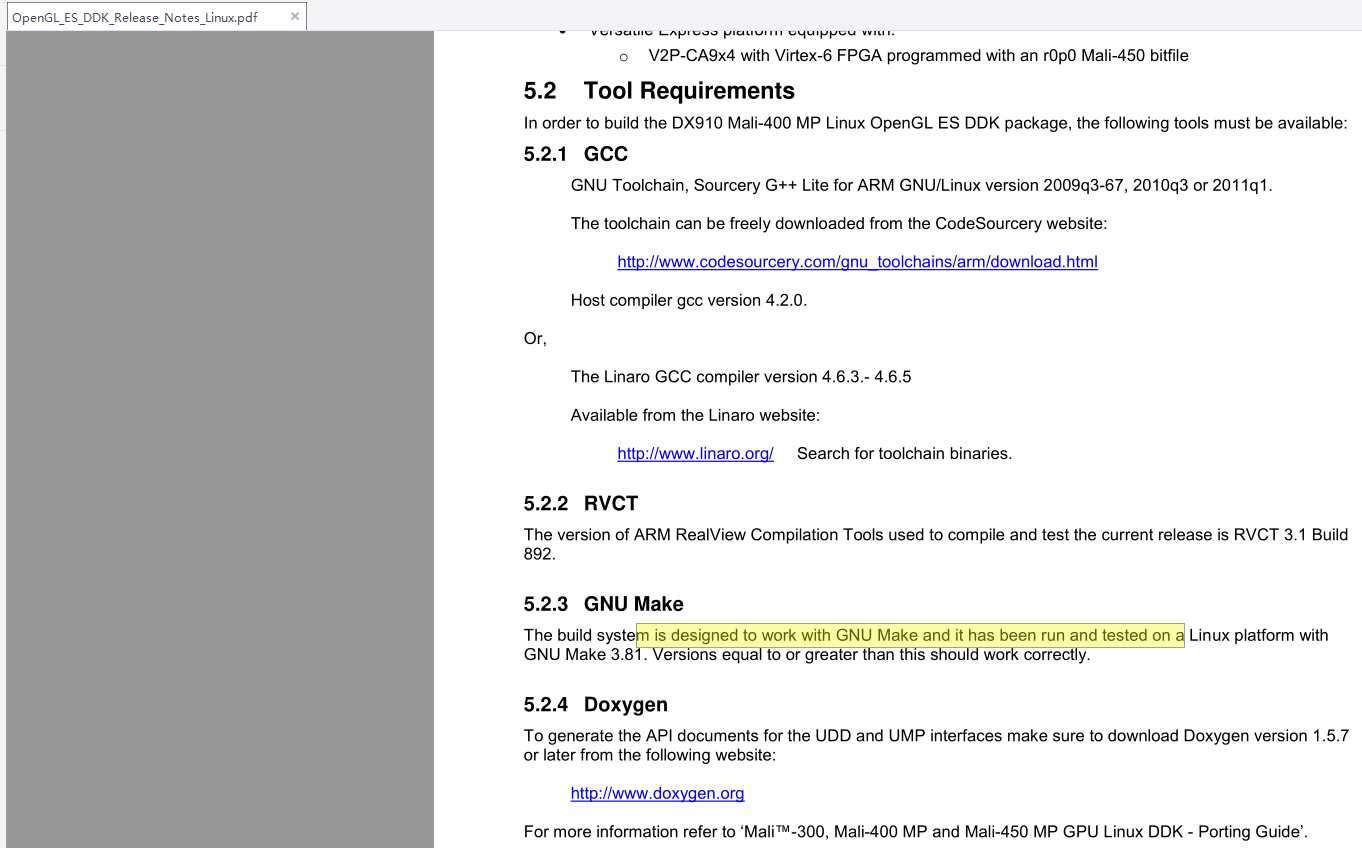


##### gvim直接修改bin文件再保存方法

使用gvim 直接修改bin文件，然后再用命令转换到二进制



### ARM软件版本



### GPU知识补充

[GPU](http://www.eetop.cn/cpu_soc)的全称是 Graphics Processing Unit，也即是图形处理单元的意思。这个概念的最早确定是NVIDIA（英伟达）公司在1999年发布型号为 GeForce （代号NV10）的三维[芯片](http://www.eetop.cn/semi)时首次提出，当时的定义是：

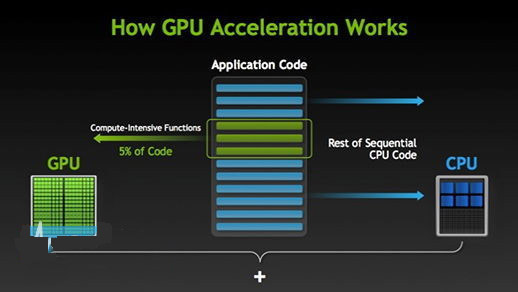
　　三角形变换能力达到每秒一千万个三角形以上的三维[芯片](http://www.eetop.cn/semi)。

[**GPU**](http://www.eetop.cn/cpu_soc)**到底是如何实现三维渲染呢？**

　　要了解[GPU](http://www.eetop.cn/cpu_soc)是如何进行渲染操作其实并不难，宏观角度来可以将其简化为下面的样子：

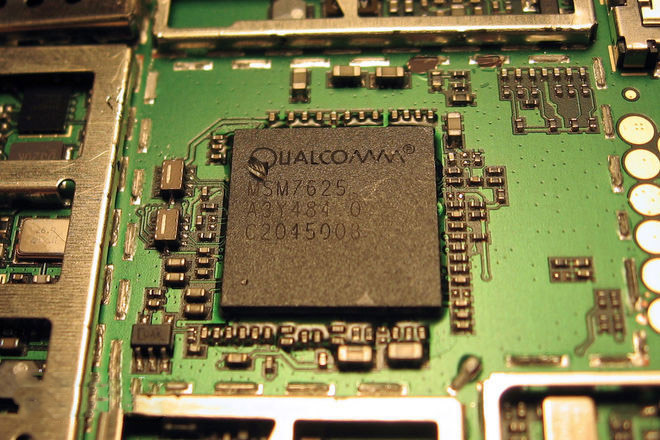
　　应用程序-》几何处理-》光栅处理

　　在图形处理中，应用程序执行的相关操作包括了碰撞侦测、全局加速算法、动画处理、物理模拟等。



几何处理就是对图元进行处理，所谓图元是指点、线、面这类几何体，而光栅化则是将确定了位置、大小和光照的几何体映射到屏幕空间栅格化后的处理，例如像素着色、贴图、混合。

在没有图形[芯片](http://www.eetop.cn/semi)（显卡）之前，几何处理、光栅处理都是由 [CPU](http://www.eetop.cn/cpu_soc) 或者 FPU（浮点单元）、SIMD （单指令多数据）单元协助来完成，随着[芯片](http://www.eetop.cn/semi)技术的进步，其中的几何处理、光栅处理开始逐步放到专门的[芯片](http://www.eetop.cn/semi)上执行，之后这些专用[芯片](http://www.eetop.cn/semi)又被集成到一块，逐渐形 成了现在的 [GPU](http://www.eetop.cn/cpu_soc)。移动[处理器](http://www.eetop.cn/cpu_soc)同样经历PC这一过程，比如说高通最早的[处理器](http://www.eetop.cn/cpu_soc)MSM7225/7625就没有集成[GPU](http://www.eetop.cn/cpu_soc) ，甚至是2D处理都是交由[CPU](http://www.eetop.cn/cpu_soc)完成。



　　（智能手机与PC一样，经历过无[GPU](http://www.eetop.cn/cpu_soc)时代

应用程序把需要进行三维渲染或者计算的数据和指令递交给 [GPU](http://www.eetop.cn/cpu_soc)，由[GPU](http://www.eetop.cn/cpu_soc)来执行几何处理以及光栅处理，这样的处理方式被称作流水线（pipeline）。

　　采用流水线的方式可以将工作拆分为若干个处理环节，也就是所谓的工位（stage）或者功能阶段，这些工位本身也可以继续拆分成若干部分，也可以实现（部分的）并行化。

　　几何处理阶段需要做些什么啥呢？

　　几何处理阶段执行的是顶点、多边形级别的处理。这一步可以拆解为 5 个工位或者说 5 个步骤：

　　•对模型及视图进行变换（transform）

　　•顶点着色

　　•投影

　　•裁剪

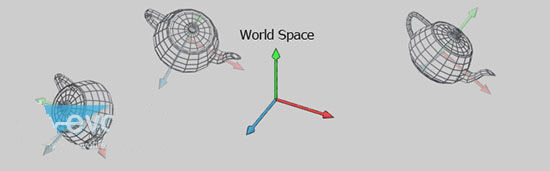
　　•屏幕映射

**模型及视图的变换**

　　模型变换

　　由于每个模型都有自己的坐标，因此在成为屏幕上的画面对象之前，模型需要变换为多个空间或者坐标系。

　　作为对象的模型空间（Model Space，或者叫模型自空间）的坐标被称作模型坐标，在经过坐标变换后，模型就会处于世界坐标或者世界空间（World Space）里，也就是确定了该模型在场景中的方向、位置。

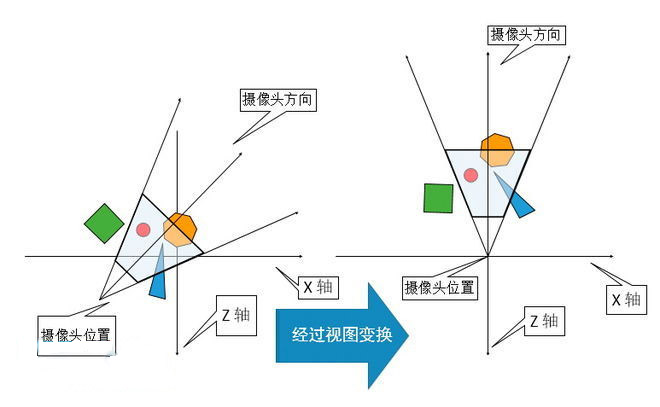




　　我们允许在场景中存在多个模型的拷贝（被称作引用），这些大小一样的引用可以在同一个场景中有不同的位置、方向。

**视图（Viewport，或者视口）变换**

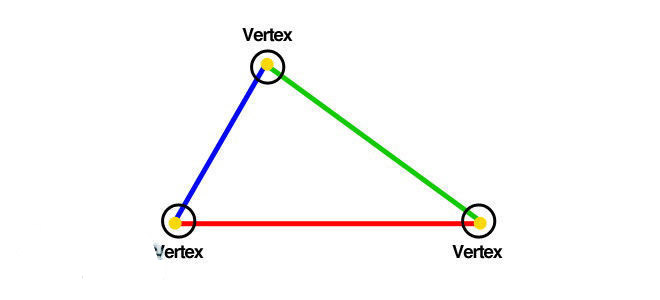
　　现在的实时渲染场景中包含的对象（模型）可以有很多个，但是只有被摄像机（或者说观察者，也即是设定的视角覆盖）的区域才会被渲染。这个摄像机在世界空间里有一个用来摆放的位置和面向的方向。



　　为了实现接下来的投影、裁剪处理，摄像机和模型都需要进行视图变换这个操作，目的是将摄像机放置在坐标原点上，使其正对的方向为 Z 轴（负向），Y 轴指向上（上图是从摄像机正上方俯视，所以没法给出 Y 轴），X 轴指向右。

**顶点着色**

　　所谓着色就是指确定光照在物料上所呈现效果的操作，这类操作既可能运行于几何阶段的模型顶点上，也可能运行于光栅阶段的各个像素上，也就是所谓的顶点着色和像素着色。



　　在顶点着色的时候，各个顶点需要存放若干个相关的物料数据，例如顶点位置、法线、色彩以及其他任何进行着色处理计算相关的数字信息。

　　顶点着色的计算结果（可以是色彩、向量、纹理坐标以及任何其他着色数据）会被发送到光栅化阶段进行插值处理。

**投影**

　 　在完成了着色处理后，渲染系统会把可视体转换为一个位于（-1， -1， -1）到（1， 1， 1）的单元立方体（unit-cube）中，这个立方体被称作正则观察体（canonical view volume），使用到的投影方式一般有两种：平行投影和透视投影。

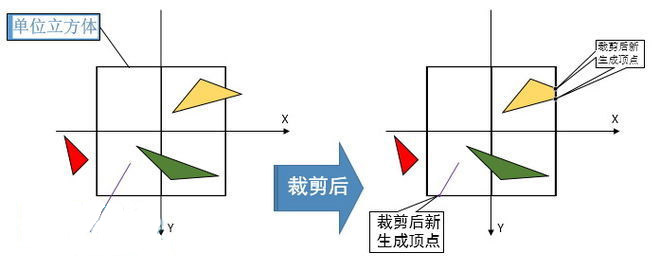
　　前一种主要在 CAD 等软件中使用，后一种因为模拟了我们人类的视觉体验，所以在游戏或者虚拟现实中经常使用：



　　上图分别是一部 iPhone 6s Plus 以平行投影和透视投影的方式呈现在屏幕上的效果

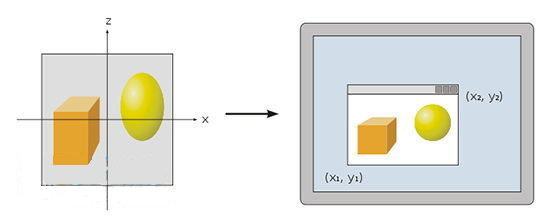
**三角形裁剪**

　　只有在可视体内的图元会被传送到在屏幕上绘制这些图元的光栅阶段，在进行裁剪动作的时候，如果图元有顶点落在可视体之外，裁剪的时候就会将可视体之外的这部分剪切掉并且在可视体与图元相交的位置生成新的顶点，而位于立方体外部的旧图元就会被抛弃掉。



**屏幕映射**

　　经过上一步裁剪后的位于可视体内的图元会被传递到屏幕映射阶段，此时的坐标信息依然是三维的。图元的 X、Y 坐标被变换到屏幕坐标系，屏幕坐标再加上 Z 轴坐标就被称作窗口坐标。



　　我们假定场景要渲染到一个最小角落坐标为（x1， y1）和最大角落坐标为 （x2， y2）的窗口中，也就是 x1 《 x2，y1 《 y2。此时，屏幕映射执行的是一个缩放处理后的转换操作。Z 轴坐标并不受此操作的影响。

　　现在，新的 x 轴、 y 轴坐标就是屏幕坐标，有对应的屏幕像素位置，不再是之前投影处理后的那个立方体所采用的映像坐标系统。

**光栅处理阶段要干些什么呢？**

　　在获得了经过变换和投影处理的顶点及其相关联的着色信息后，光栅化处理阶段的目的就是计算并设置好被对象覆盖区域的像素颜色。这个处理被称作光栅化或者扫描转换，也就是把二维坐标上包含深度（Z 轴）信息和各种相关着色信息的顶点到屏幕上像素的转换。



　　这个处理阶段一般可以拆分为 4 个工位：

　　1、三角形设定

　　2、三角形遍历

　　3、像素着色

　　4、输出合并

**三角形设定**

这一步会进行三角形表面的微分以及其他关于三角形表面数据的计算，计算出来的数据会被用于扫描转换以及几何阶段所产生的各种着色数据的插值处理。在[GPU](http://www.eetop.cn/cpu_soc)上这一步会采用固定硬件功能单元来实现。

**三角形遍历**

　　这一步用作确定像素的中心是否被三角形覆盖 ，如果该像素被三角形覆盖的话，就会生成对应的片元（fragment）。

　　查找哪些样本或者像素是否位于三角形内通常被称作三角形遍历或者扫描转换。

　　每个三角形对应片元的属性都是由该三角形的三个顶点数据插值而成，例如片元的深度值以及来自几何阶段的着色数据。

**像素着色**

　　所有的逐像素（per-pixel）着色计算都在这一步执行，使用的输入数据是之前插值的着色数据。像素着色发送到下一个工位的计算的结果可能是一个色彩值也可能是多个色彩值。

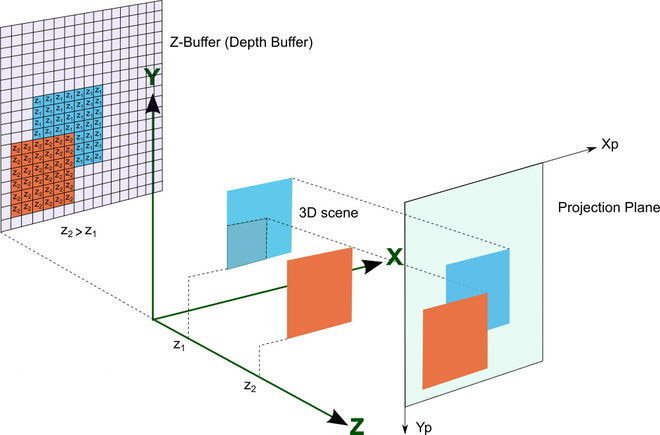
　　和三角形设定以及三角形遍历不同采用固定硬件功能单元不同的是，现在的像素着色都是由可编程的[GPU](http://www.eetop.cn/cpu_soc)内核执行。

　　在像素着色所依赖的众多技术中最为重要的就是贴图（texturing），所谓贴图就是把一张或者多张图片“贴”到对象上。

**输出合并**

　　在这一步执行的操作主要是将之前步骤生成的色彩信息进行合并形成最终输出的像素色彩。

　　用于存放像素色彩信息的缓存被称作色彩缓存，一般情况下是以红、绿、蓝三个色元的方式存放，此外还有一个用于存放像素对应深度信息值的深度缓存（一般采用 Z-Buffer）。



　　在[GPU](http://www.eetop.cn/cpu_soc)中实现这一步的功能单元有几种叫法，例如 ROP、Output Merger 或者 Back-End。

　　在这个阶段，Output Merger 会根据深度缓存（depth buffer 或者 Z-buffer）存放的深度信息判断是否更新色彩缓存中的色彩值。

例如当前像素计算出来的深度值（例如是 0.1）比深度缓存中对应像素的值小（例如是 0.2），则表示当前像素的三角形比色彩缓存存放的像素所对应的三角形更靠近“摄像头”，于是[GPU](http://www.eetop.cn/cpu_soc)会对该图元的色彩进行计算并把新计算出来的色彩值和深度值更新到色彩缓存和深度缓存中，否则的话就不会更新当前像素的缓存。

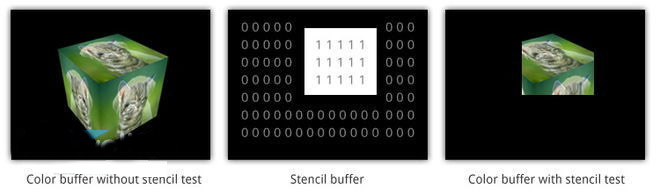
　　在整个场景完成渲染后，色彩缓存中存放的都是从摄像机视角位置看到的可视图元色彩值。

　　这样处理的好处是三角形可以使用任意次序来渲染，但是如果图元或者说三角形是部分透明的话，则必须依照从远到近的三角形层次进行渲染。这是 Z-Buffer 的主要缺点之一。

　　像素除了色彩缓存和深度缓存外，还有其他利用通道或者缓存的技术来用于过滤和捕捉片元（fragmet）信息。

通常和色彩一起存放于色彩缓存的阿尔法通道（Alpha Channel）包含了每个像素的相对不透明值，开发人员可以在进行深度[测试](http://www.eetop.cn/measurement)之前对到来的片元先执行名为阿尔法[测试](http://www.eetop.cn/measurement)（Alpha Test）的操作。如果片元的 alpha 值[测试](http://www.eetop.cn/measurement)（一般是等于、大于等简单的操作）为“假”，那么这个像素的后续处理操作就会被省略掉。这个操作通常用于确保完全透明的片元不会对 Z-buffer 构成影响。

　　此外，还可能会涉及到名为蜡板缓存（Stencil Buffer）的技术。Stencil Buffer 作为一个离屏缓存一般用于存放已渲染图元的位置，它通常用于进行一些特殊效果的处理，例如将一个“实心”圆存放到蜡板缓存中，之后配合其他操作，就可以将 被覆盖图元的色彩值控制为只有在位于这个实心圆中的时候才被呈现，相当于一个遮罩的作用。



　 　以上这些缓存都被统称为帧缓存，但是在一般情况下，帧缓存特指色彩缓存和深度缓存。由于画面渲染是需要时间的，为了保证出到显示器或者显示屏的时候图元 都是已经完成渲染的，人们引入了双缓存技术，渲染中的被称为后台缓存（back buffer），完成渲染的称作前台缓存（front buffer），在后台缓存完成渲染后，马上变成前台缓存，而前台缓存就切换为后台缓存，以此类推。

**片元（fragmet）和像素（pixel）的区别？**

　 　上文中我们提到了片元和像素，像素是相对容易理解的，严格来说，像素就是对应屏幕上的一个点，它有表示屏幕位置的 x、y 坐标以及颜色的红绿蓝（RGB）值（像素是没有 Alpha 通道值的），图形流水线所作的所有事情都是为了给输入的图元计算在屏幕上像素的颜色。

　　那么，片元又是怎么一回事呢？

　　在三角形遍历和输出合并之间计算的栅格数据就是片元，它们是像素的前身，在遍历的时候由顶点内插而成。片元除了具备像素的 x、y 坐标外，还有表示深度的 z 坐标以及顶点的属性信息（颜色、片元法线、纹理坐标）。



　　正如前面所说的，深度坐标记录的是图元的相对距离，在深度[测试](http://www.eetop.cn/measurement)的时候被遮蔽（或者阿尔法[测试](http://www.eetop.cn/measurement)中为透明）的图元会在输出合并阶段被抛弃掉，

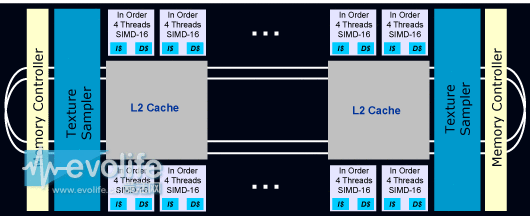
片元这个说法是 OpenGL 或者大多数实时图形渲染文献中的概念，而 D3D 则没有这么严格的区分，片元和像素都统称为像素。

　　我们这里讨论的都是以实时三角形渲染为例，除此以外还有其他三维渲染流水线形式，例如 micropolygon（微型多边形）、Voxel（体素） 渲染。

　　现在的[GPU](http://www.eetop.cn/cpu_soc)流水线也是遵照这样的图形流水线来设计，在一段时间里，出于成本效益的考虑，[GPU](http://www.eetop.cn/cpu_soc) 的各个功能单元都是有专门的电路来实现的。

　　不过在新式的[GPU](http://www.eetop.cn/cpu_soc)中，可编程部分（例如顶点程序、片元程序）由于指令集得以统一，所以都采用了同样的计算单元来跑。

公认必须采用固定功能硬件单元来实现的主要是三角形设置、遍历以及输出合并单元，[英特尔](http://www.eetop.cn/cpu_soc)曾经试图在名为 Larrabee 的[GPU](http://www.eetop.cn/cpu_soc)项目里将这些工位采用通用单元来执行，但是最终结果是不了了之，至少说明现阶段或者在未来可见的较长时期里，三角形设置/遍历以及输出合并的最合 理实现方式还是使用固定功能硬件单元。



　　（Intel Larrabee）

　 　在现实中[GPU](http://www.eetop.cn/cpu_soc)并不仅仅是上面讨论的三维处理、计算单元，广义的[GPU](http://www.eetop.cn/cpu_soc)还应该包括视频编解码单元、扫描输出单元、总线单元、存储单元，手机[GPU](http://www.eetop.cn/cpu_soc)现在都 和 [CPU](http://www.eetop.cn/cpu_soc)、基带、周边等单元集成到同一个[芯片](http://www.eetop.cn/semi)里，这样的[芯片](http://www.eetop.cn/semi)被称作 SoC（片上系统），SoC 并非是手机独有的，在此之前的[单片机](http://www.eetop.cn/embedded)（例如洗衣机等里就有，一般几块钱一颗）其实就是 SoC 的一种实现方式。

[**GPU**](http://www.eetop.cn/cpu_soc) **在移动应用中能发挥的作用**

[GPU](http://www.eetop.cn/cpu_soc) 最初是为三维游戏加速设计的，所以它最拿手的自然是三维游戏加速，现在无论是安卓、iOS 都有不少三维游戏大作，例如：劳拉 Go、FIFA 系列、真人快打 X、Hitman Sniper、Marvel Future Fight、Godfire： Rise of Promotheus、Over Kill、Implosion、Battle supremacy 等等。



　 　除了游戏渲染加速外，智能手机的操作系统界面也是采用了[GPU](http://www.eetop.cn/cpu_soc)硬件加速的，例如选单的弹出、桌面平移等。和桌面操作系统使用三维加速相比，移动操作系统 由于受到屏幕空间小的约束，因此三维加速体验带来的空间感拓展是更加不可或缺的。从安卓4.0系统开始，不少用户就感觉到系统流畅性大幅度提高正是因为从 系统层面上线了[GPU](http://www.eetop.cn/cpu_soc)硬件加速。

**浅析**[**GPU**](http://www.eetop.cn/cpu_soc)**的常见术语**

　　在下篇文章中SIMD、Core、[GPU](http://www.eetop.cn/cpu_soc) 中的线程、统一着色器、纹理单元这些[GPU](http://www.eetop.cn/cpu_soc)常见术语对一般读者来说都是相当陌生或者容易产生困扰，在此先进行浅析。

　　SIMD：

　　Single Instruction Multiple Data，单指令多数据流，目前所有的[GPU](http://www.eetop.cn/cpu_soc)在基本功能单元层面都属于SIMD（业界也接受NVIDIA提出的SIMT——单指令多线程），一般是16路SIMD或者32路 SIMD。

　　“内核”或者core：

　 　目前在[GPU](http://www.eetop.cn/cpu_soc)行业或者[GPU](http://www.eetop.cn/cpu_soc)行销上被严重滥用的名词，它被用作指代SIMD一条Lane（计算通道）上的单元集合，里面可能有一个单周期 32位FMA（积和熔加运算）运算器、一个双周期64位FMA运算器等等，[GPU](http://www.eetop.cn/cpu_soc)厂商把[GPU](http://www.eetop.cn/cpu_soc)里的SIMD Lane数加起来就对外宣称有多少个内核。

　　这样的说法是否属于错误宣传还真不好说，因为目前并没有什么法律文件规定怎样的集合才算是一个内核。

　　不过对于计算机科学来说，微架构里对内核约定俗成的看法是它必须有一个PC（程序计算器，它是一个寄存器，其中存放的一般是指向该内核要执行的下一条指令的地址）。

　　这样的话，一个[GPU](http://www.eetop.cn/cpu_soc)内核显然不能是SIMD单元中的一个Lane，它的层级至少应该高一级。

　 　所以，[GPU](http://www.eetop.cn/cpu_soc)上相对严格的“内核”概念单元对应的应该是类似与[AMD](http://www.eetop.cn/cpu_soc) GCN的Compute Unite、NVIDIA Maxwell中的SMM、PowerVR Series 6/7的USC等名词命名的单元集合，在OpenCL中，这个层级的单元集合被称作Compute Unit（计算单元，简称CU，[AMD](http://www.eetop.cn/cpu_soc)的GCN微架构也采用Compute Unit这个术语，完全对应OpenCL的Compute Unit），而[GPU](http://www.eetop.cn/cpu_soc)厂商的行销术语 “内核” 或者“core” 在OpenCL中被称作Process Element，简称PE。

[GPU](http://www.eetop.cn/cpu_soc)中的线程：

　　现在的[GPU](http://www.eetop.cn/cpu_soc)都采用了多层次线程技术，按照硬件开发商提供的文档，对应SIMD Lane的被称作thread（OpenCL中称作work-item，在图形渲染的时候你可以将其看作是屏幕上的一个像素），是最小的线程单位；

　 　往上的一层线程单位在新的OpenCL被称作sub-group，NVIDIA称作warp 或者thread warp，[AMD](http://www.eetop.cn/cpu_soc)称作wavefront，属于[GPU](http://www.eetop.cn/cpu_soc)执行调度的最小硬件线程单位。再往上就是workgroup（NVIDIA 称之为thread block）和NDRange（NVIDIA称之为Grid，由若干个workgroup或者thread block组成）。

　　 Workgroup的对应[GPU](http://www.eetop.cn/cpu_soc)硬件关系是Compute Unit，同一时间里Compute Unit跑的都是一个workgroup，而Grid则对应[GPU](http://www.eetop.cn/cpu_soc)的一个partition（分区，在设备或者说加速器允许的情况下，OpenCL可以把 一个设备分成若干个分区来使用）。

　　你可以把饭粒比作是work-item，而每一口饭则算是一个sub-group，一碗饭看作是一个 workgroup，饭煲看作是NDRange（sorry，我的比喻未必很恰当）。

统一着色器

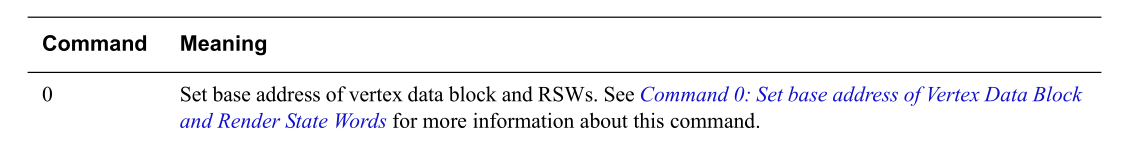
　　在 DX10以后，由于几何、像素的指令格式一样，使得几何和像素处理可以在同样的单元上执行，自此以后台式[GPU](http://www.eetop.cn/cpu_soc)都采用了统一着色器设计，以确保着色器的利用率提高。

　　纹理单元

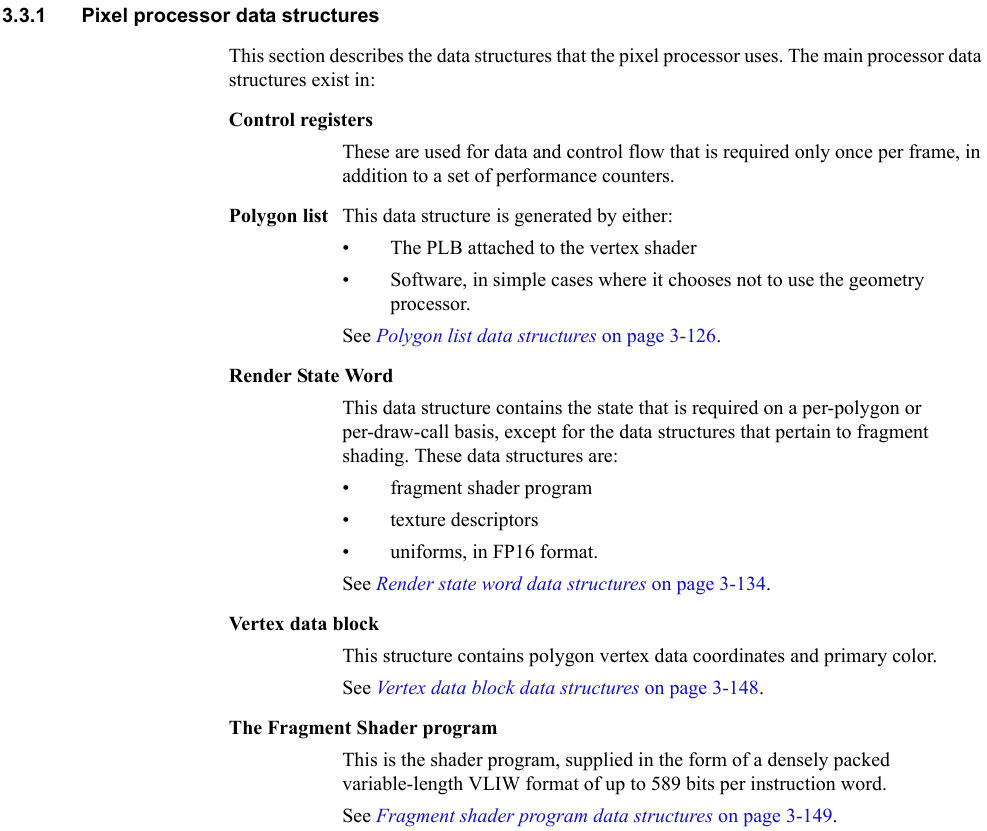
　 　纹理单元或者说纹理映射单元是[GPU](http://www.eetop.cn/cpu_soc)中计算纹理坐标和获得纹理样本的单元。在绘制某个对象的时候，每个纹理单元进行一个纹理取样动作，不同的绘制对象每 次的取样动作都可以更改使用的纹理。纹理单元的性能指标一般用 TexOps/s 来表示，表示每秒的纹理操作数，不同的纹理格式和不同的纹理过滤算法以及硬件实现、内存带宽都会对这个指标产生影响。纹理单元一般和若干个着色器绑定在一 起。

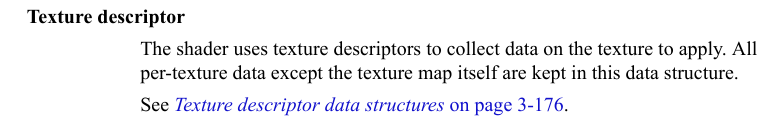
### 硬编码

#### mali结构体



##### Pixel Processor data structures

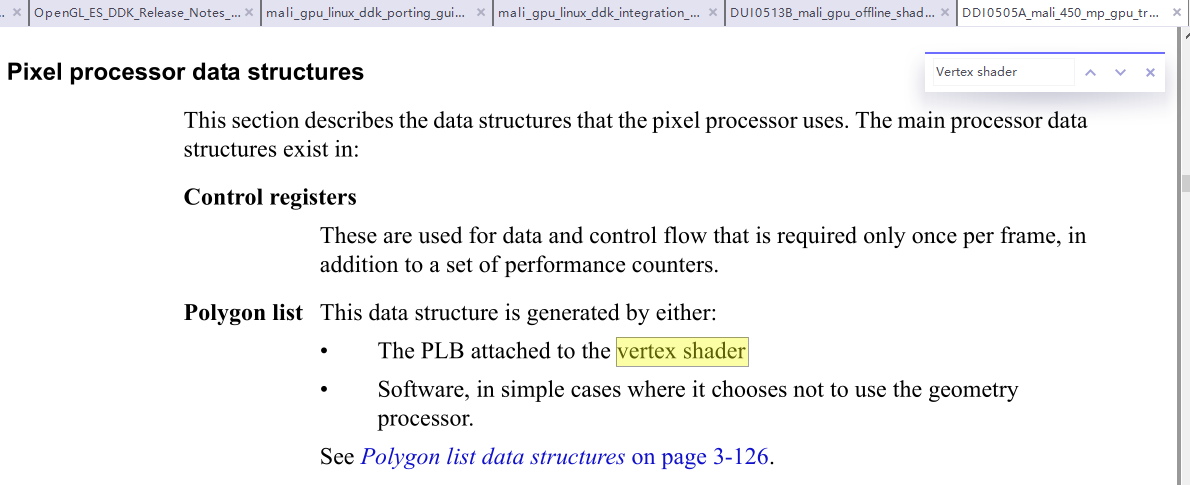




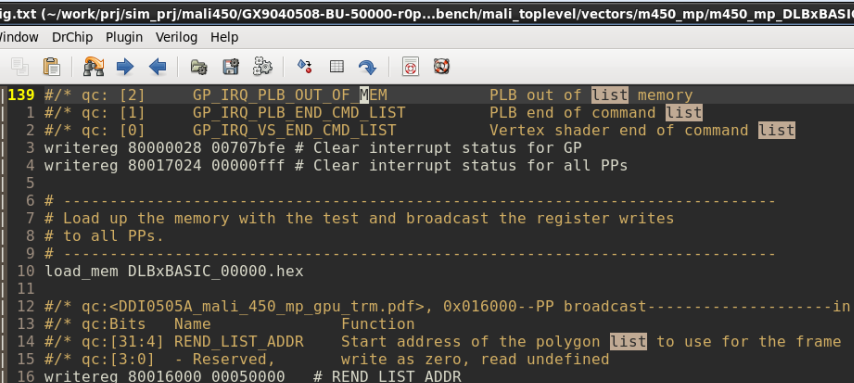
###### Control registers

###### Polygon list （仿真没使用）

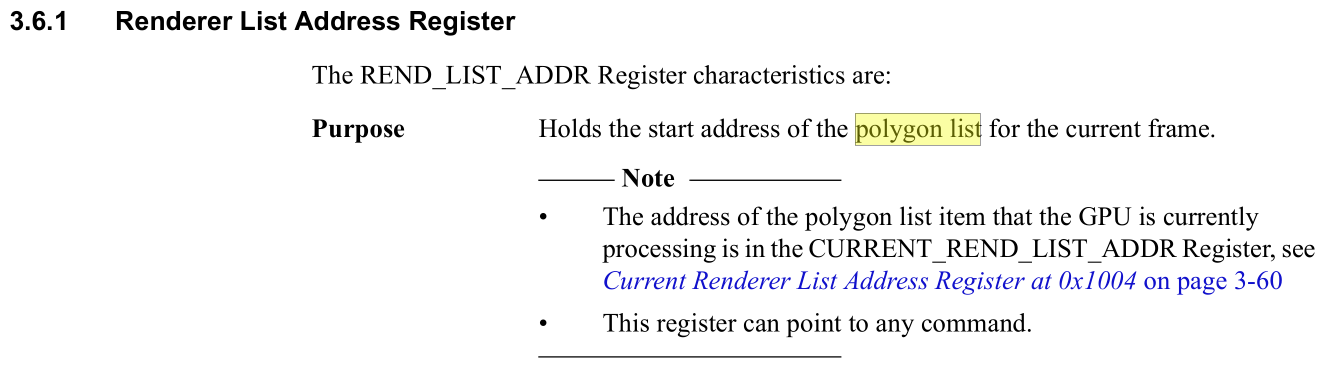
用例 DLBx就没有用到Geometry Processor，因此是用软件的方法进行工作的



Polygon list 的内容 对应地址是 REND\_LIST\_ADDR

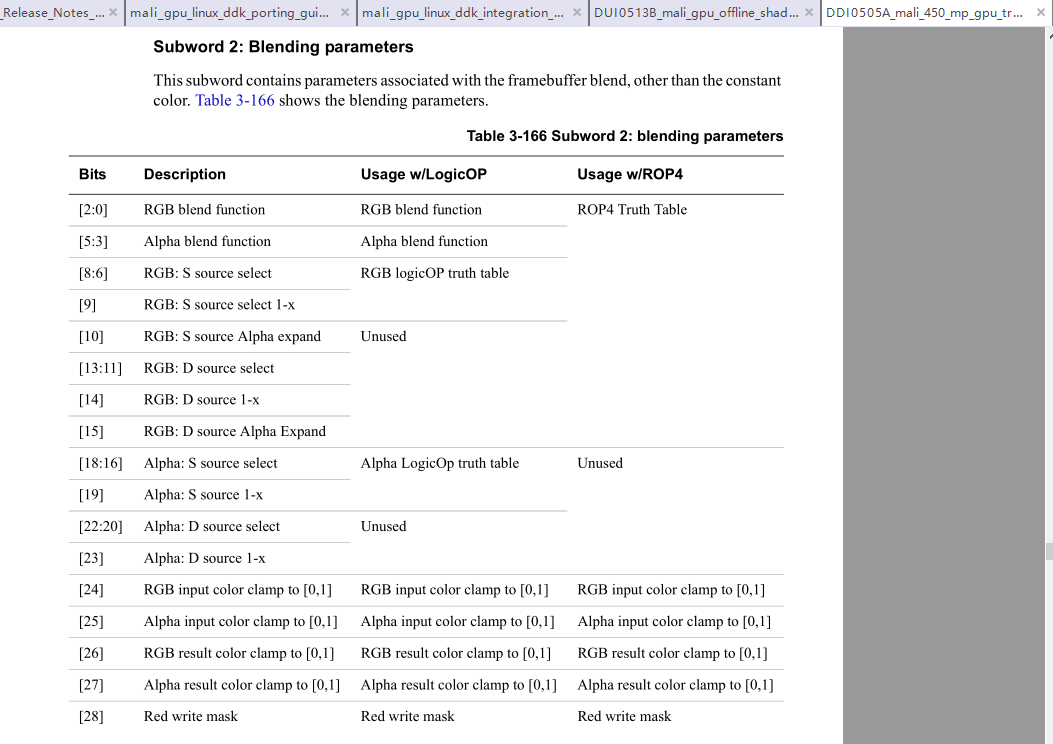


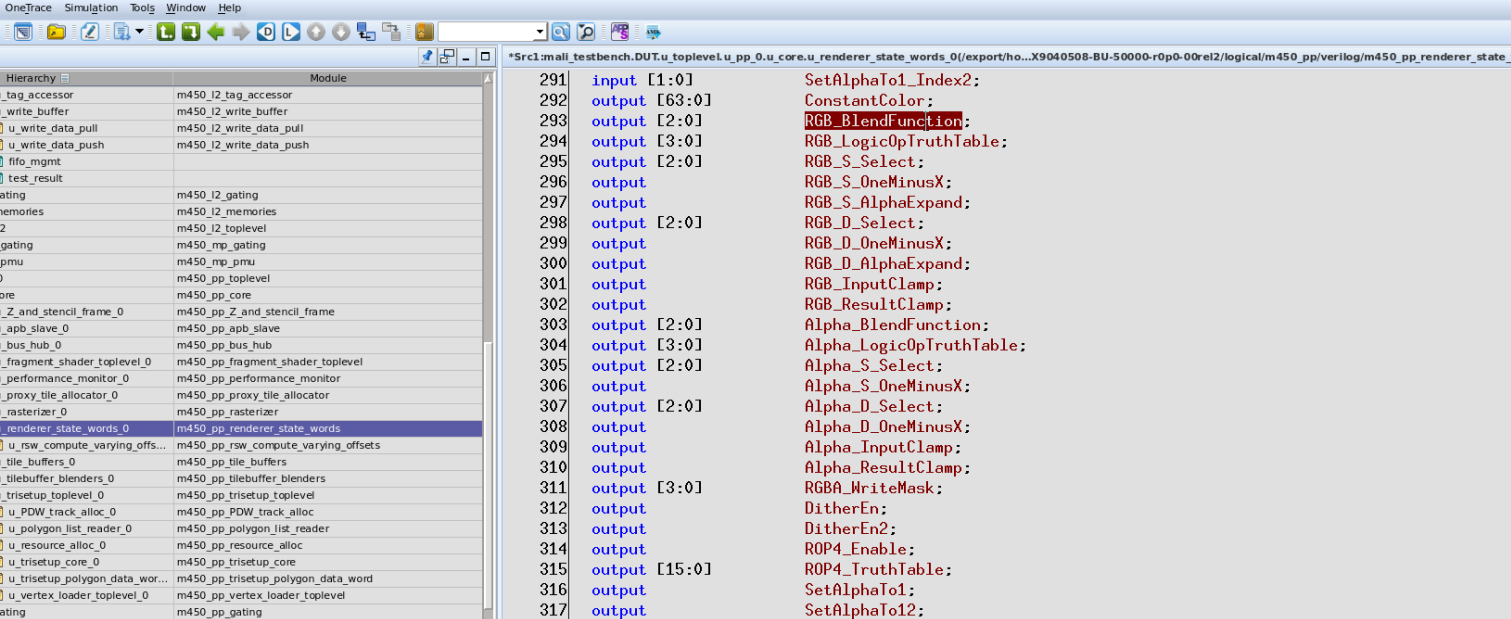
地址是0x50000 ，但是仿真时并没用这个地址。

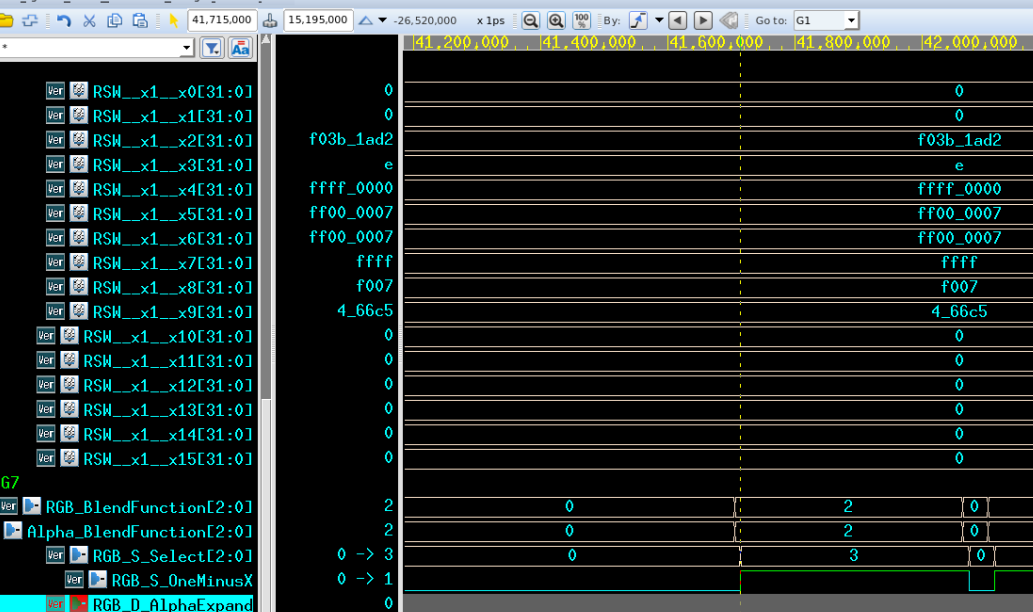


###### Render State Word data structures

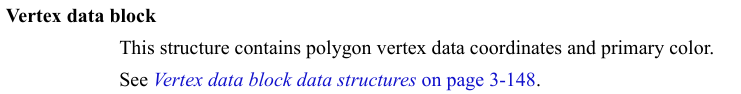
文档和代码对照关系

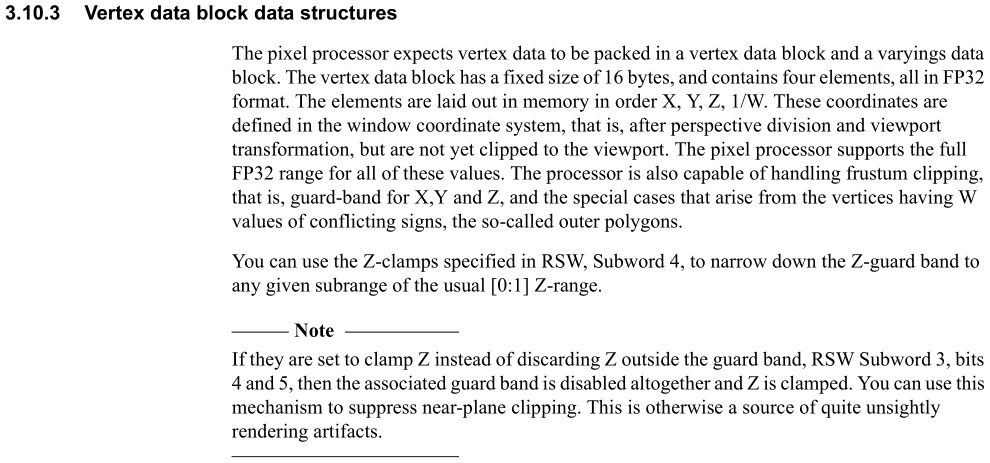






###### Vertex data block





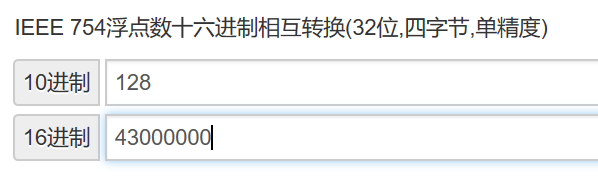


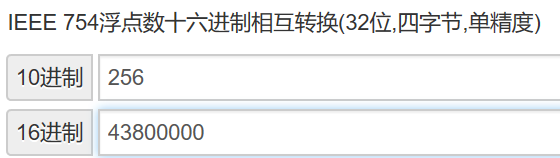
X0 = 4300\_0000 = 128 (DEC) Y0=0, Z0=0, 1/W0=1

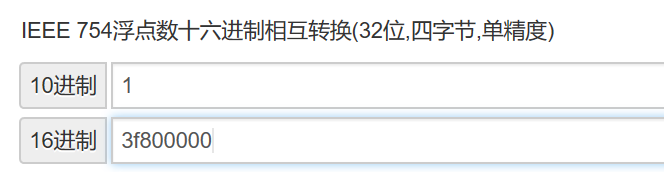
X1 = 4380\_0000 = 256 (DEC) Y1=256, Z1=0, 1/W1=1

X2 = 0000\_0000 =0 (DEC) Y2=256, Z2=0, 1/W2=1

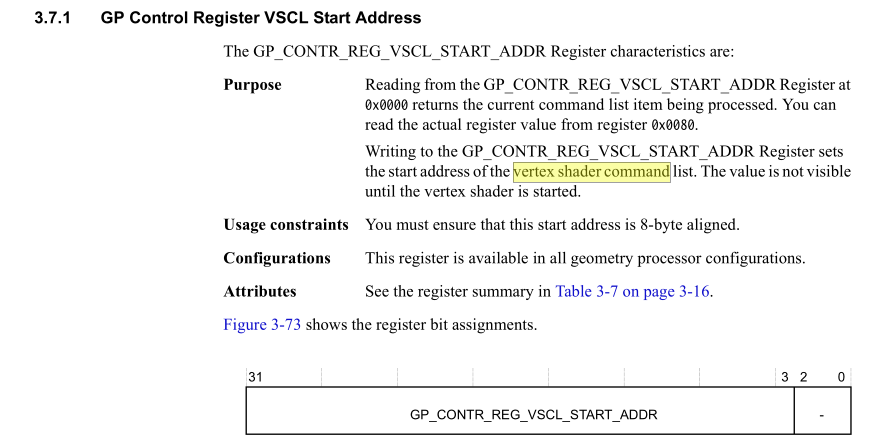
在线进制转换工具网址：<http://lostphp.com/hexconvert/>



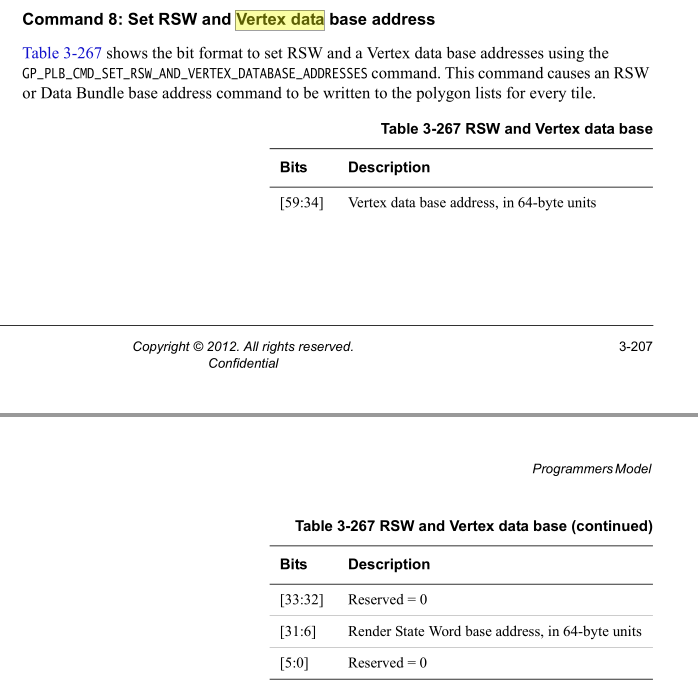




找寻Vertex data base address

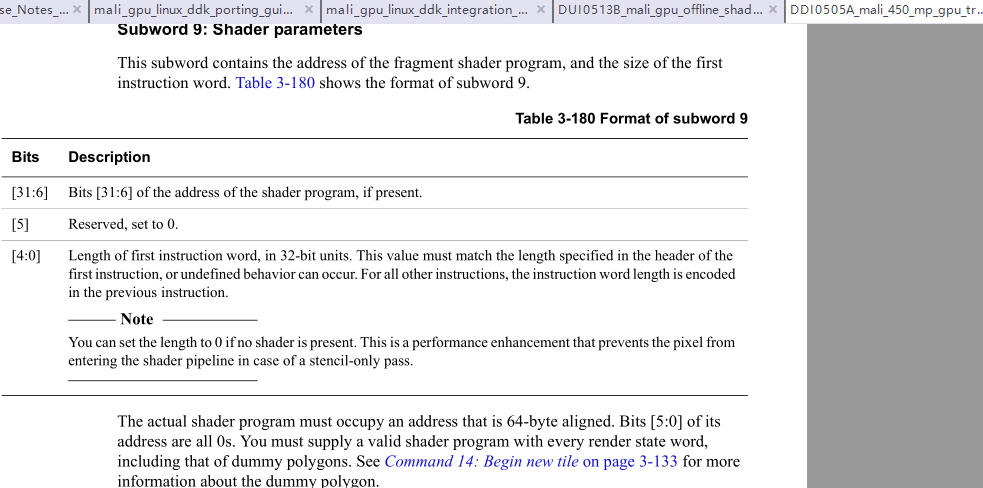




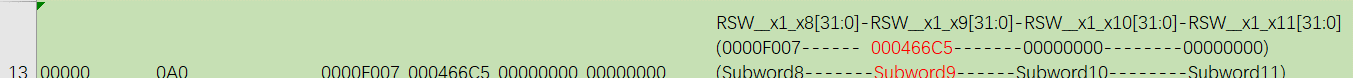


###### Fragment Shader Program data structures

S1:根据手册查到shader program 地址，存在与RSW寄存器的SubWord9中



S2：查程序机器码可见，对应地址值0x466c0,总共有5个单元



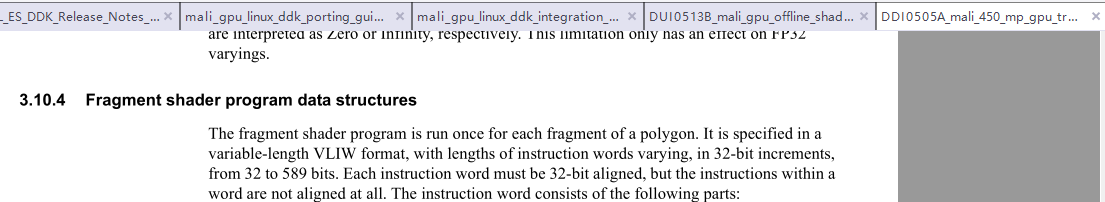
S3：对照机器码，果然是5个32bit指令



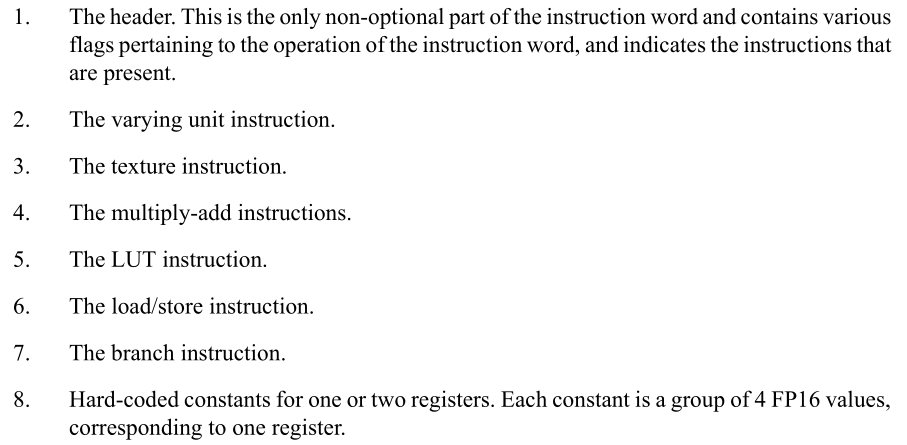
S4：指令分析

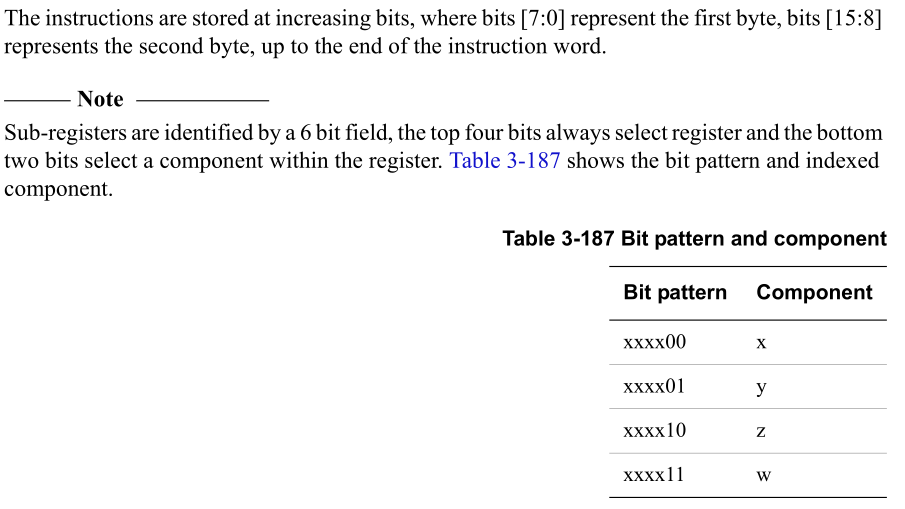
###### Shader program指令分析

手册说明，指令 words从32 到589bit，我们例子是32x5=160bit



Shader包含如下子指令

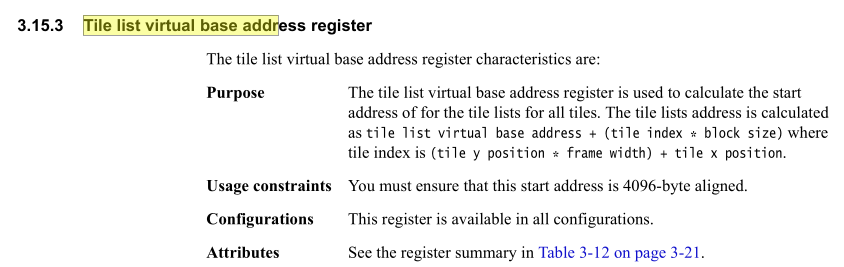




Header

见下文

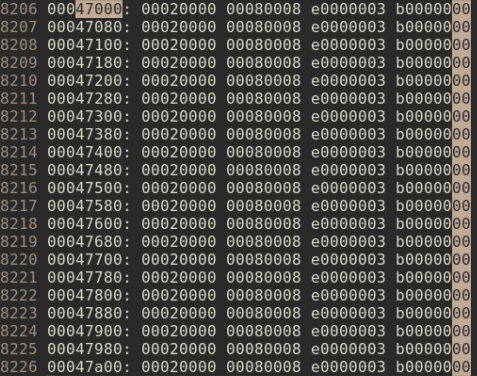
#### Tile list virtual base address register



从0x47000-0x4ef80,总共256行，每行16个字节，实际对应每行16\*8=128字节。

累计 128\*256字节，

每16个字节值都是 0x0002\_0000 0008\_0008 e000\_0003 b000\_0000



… …



对应实际波形结果：

只访问方位 0x47000 – 0x48fe0 = 8K字节

### 指令

#### Neon指令

##### 编译器

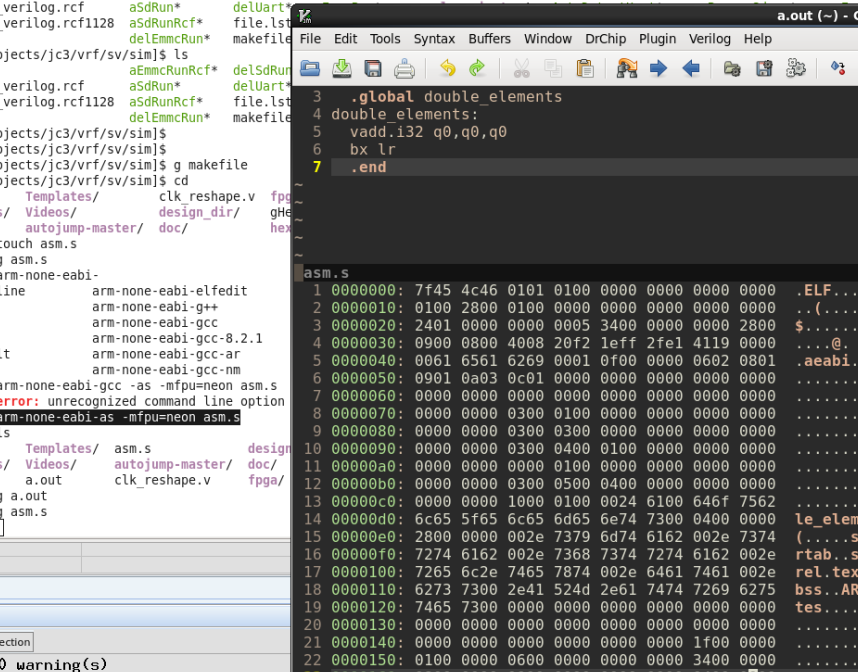
见下文

###### 实例1

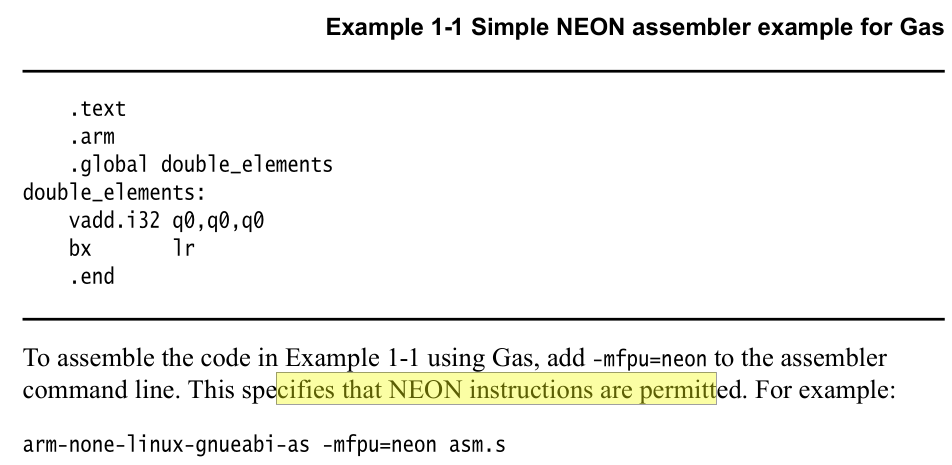
方法1使用GCC编译器

使用命令arm-none-eabi-as -mfpu=neon asm.s 即可生成机器码

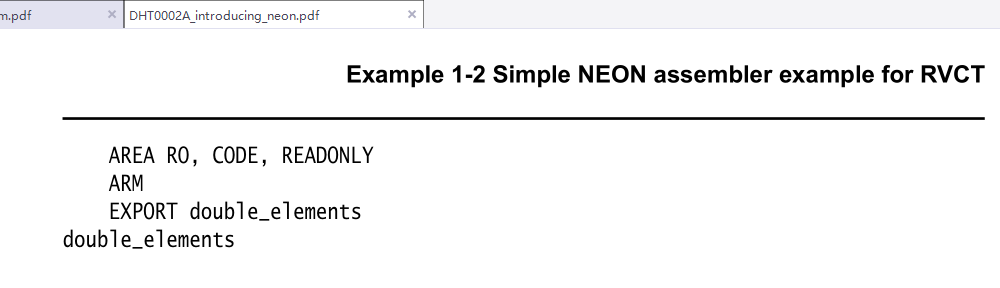
让编译变得非常简单。

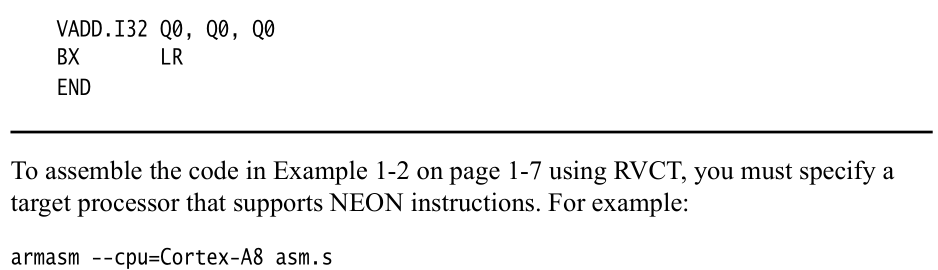


参考文档：

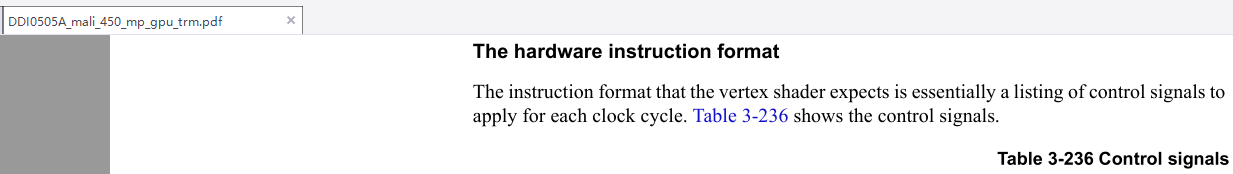


方法2使用RVCT编译器





#### Hardware指令









### Design

#### 集成

##### 配置建议

ARM mali450MP2配置建议:

1.     双核pixel processor

2.     64KB L2

3.     128bit总线

4.     Disable PMU, PMU的由soc上实现。

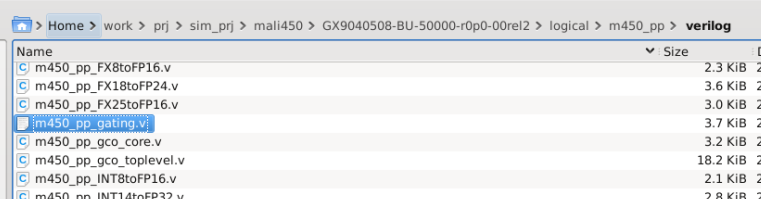
时钟clock gate单元和Memory替换。我们先在smic40上评估一下面积。

##### CLOCK GATING RESET配置

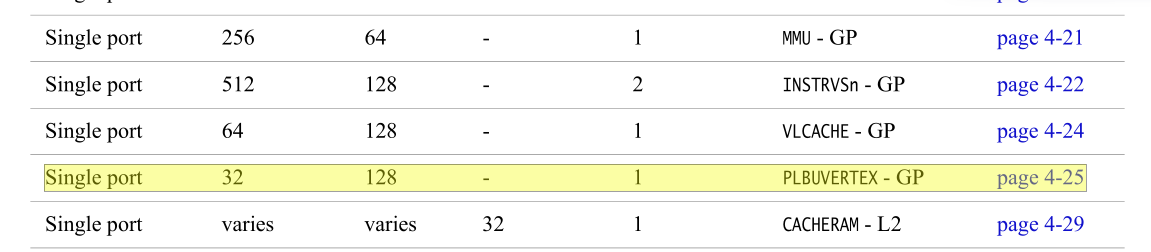


注意要替换代码的位置，以models的这个为准

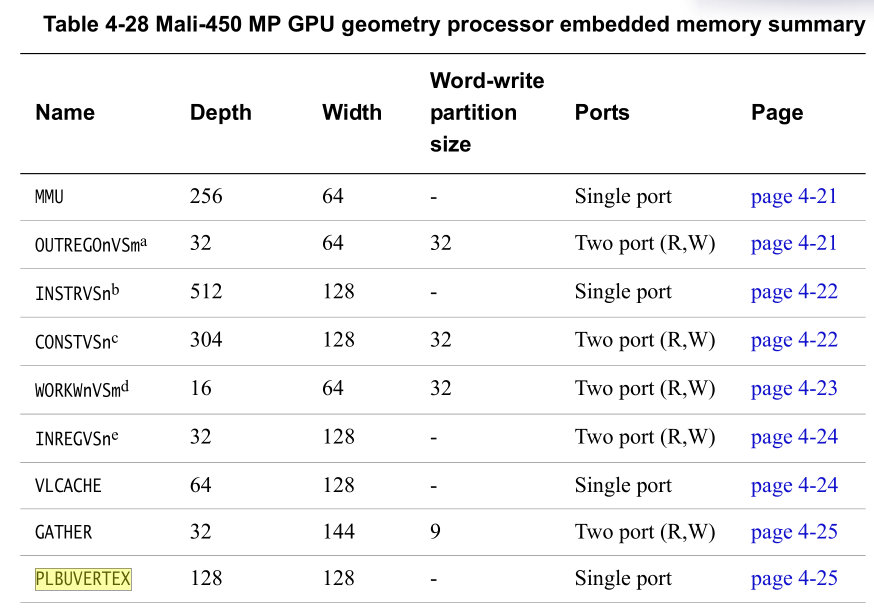
错误的位置，这个是没有集成门的地方，不好，综合工具给推断成latch了。



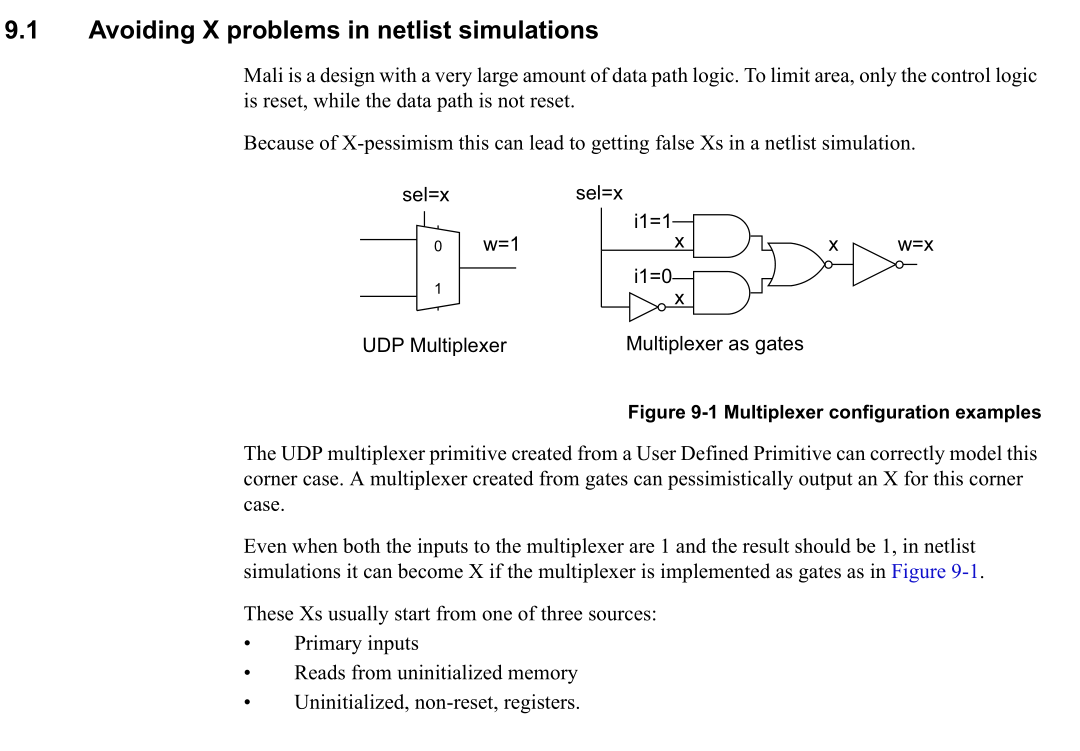
##### 错误位置



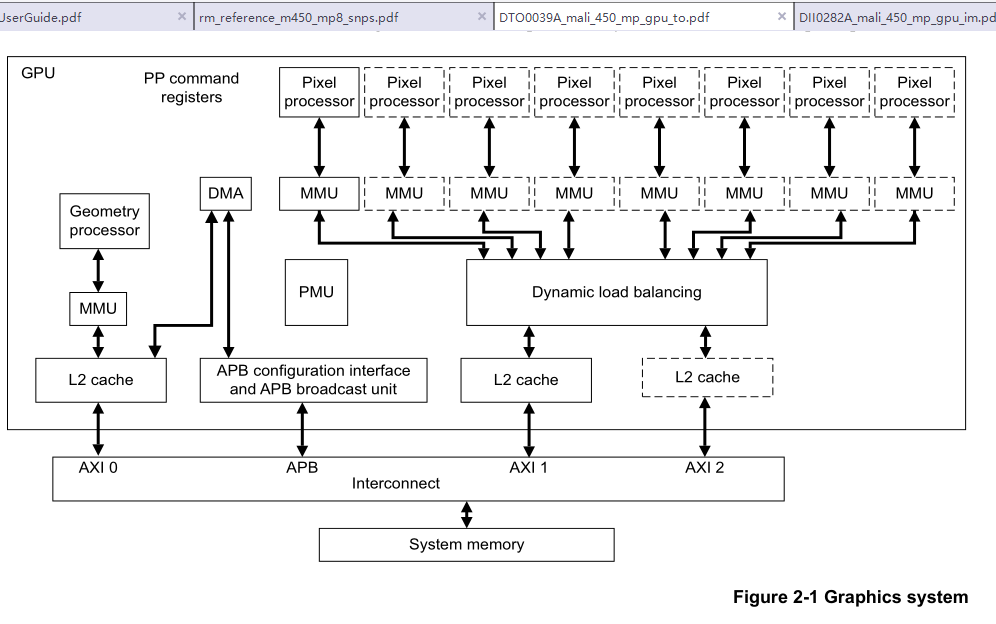




##### 后仿时如何避免许多x



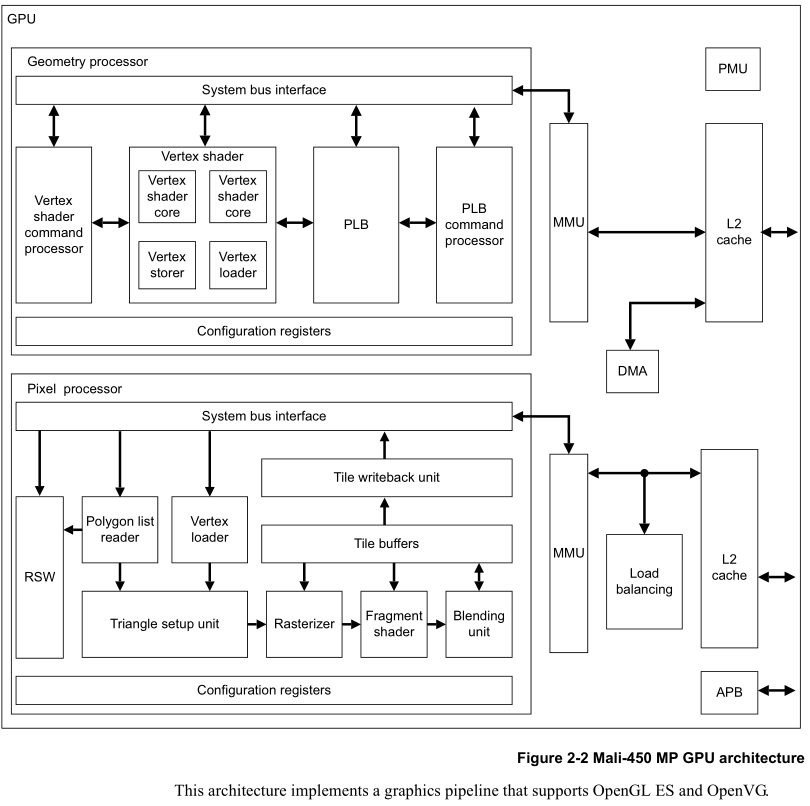
#### T O (Technical Overview)



APB用来配置DMA，用DMA来搬移数据到 L2 Cache中，给Geometry服务。

#### GP和PP

见下文



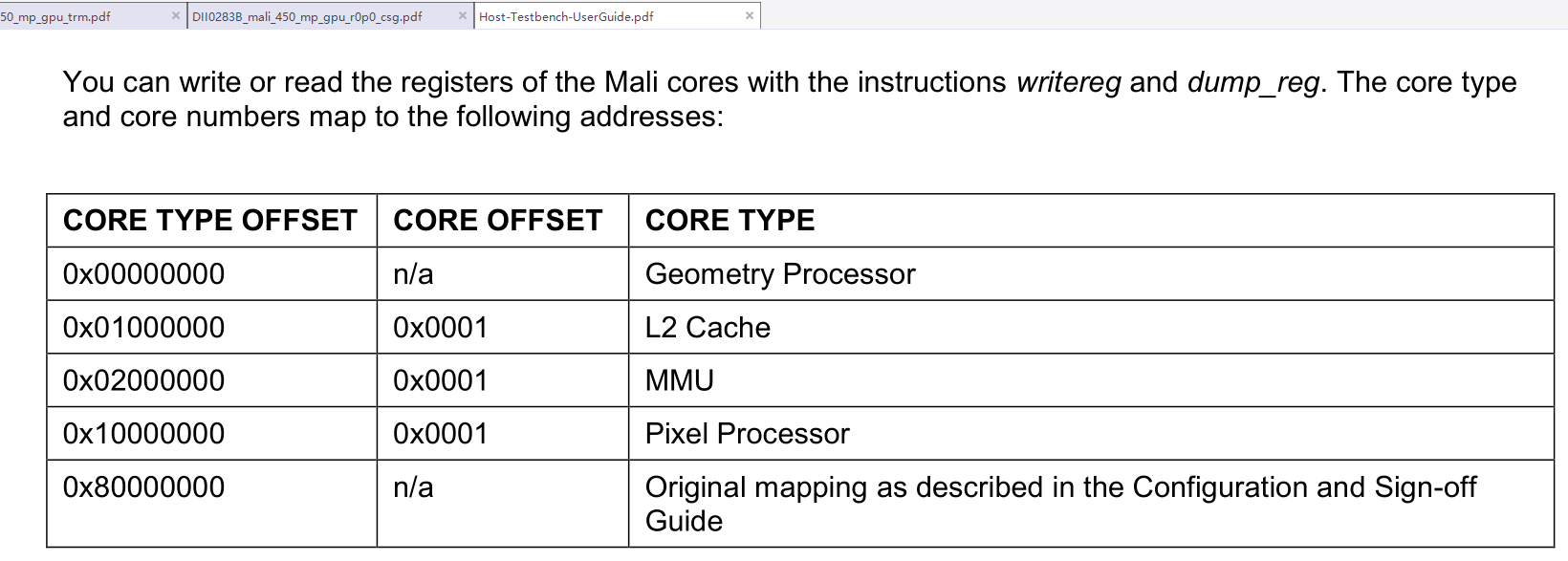
### Test Bench

见下文

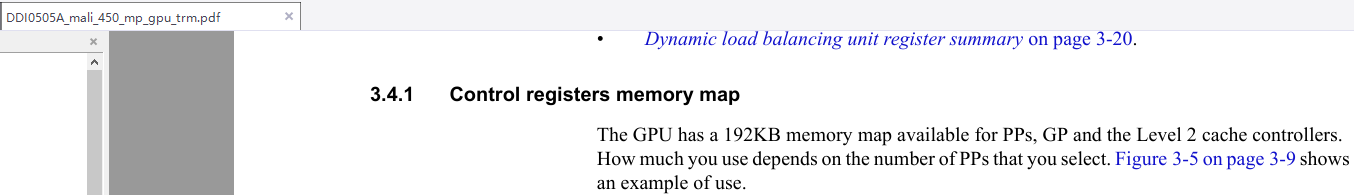
#### Mali ADDRESS

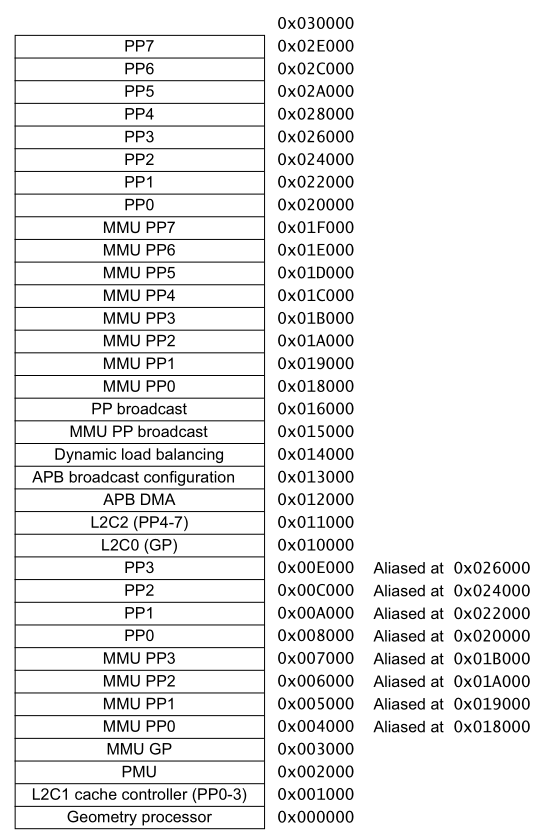
##### Testbench ADDR

见下文



##### 控制寄存器的memory map



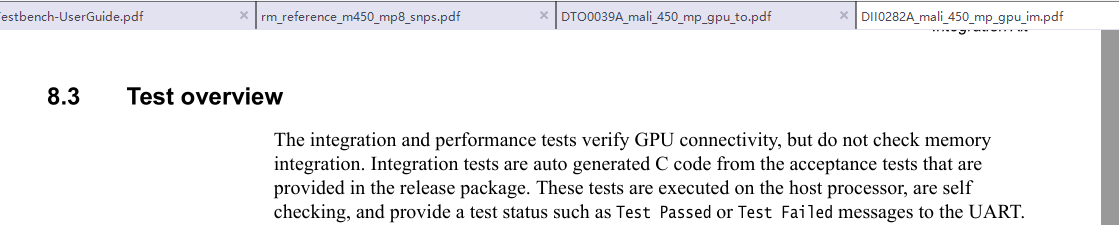


#### RTL Test Bench

见下文

##### Testcase

给出的用例只测试GPU的连通性，无法check memory integration。

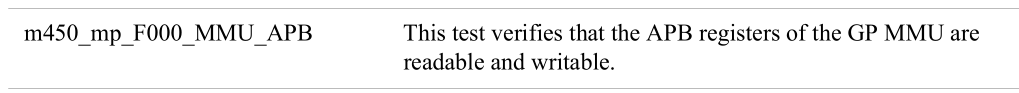


###### M450\_mp

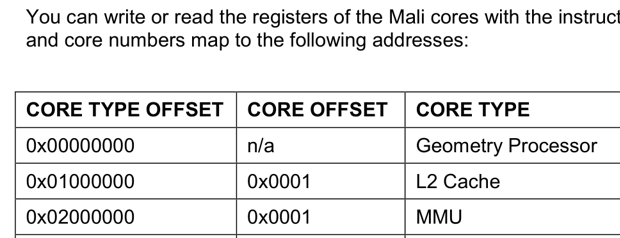
###### M450\_gp

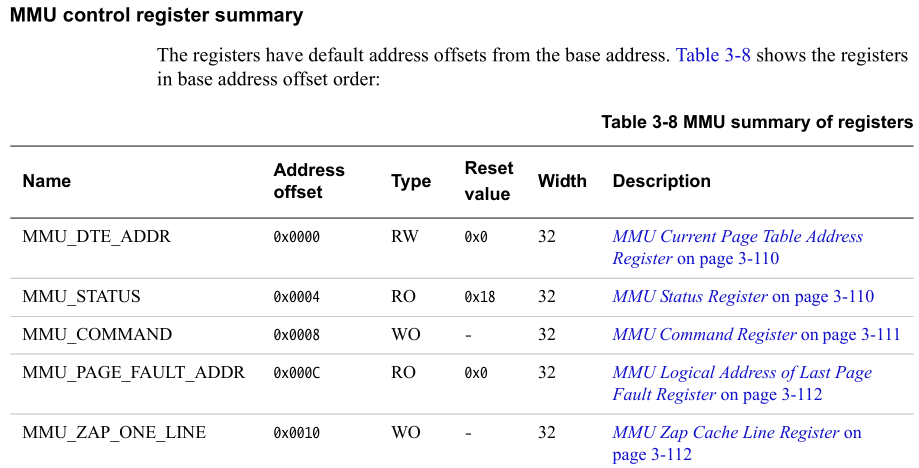
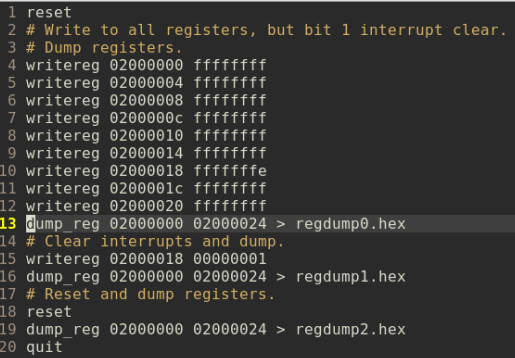
m450\_mp\_F000 \_APB

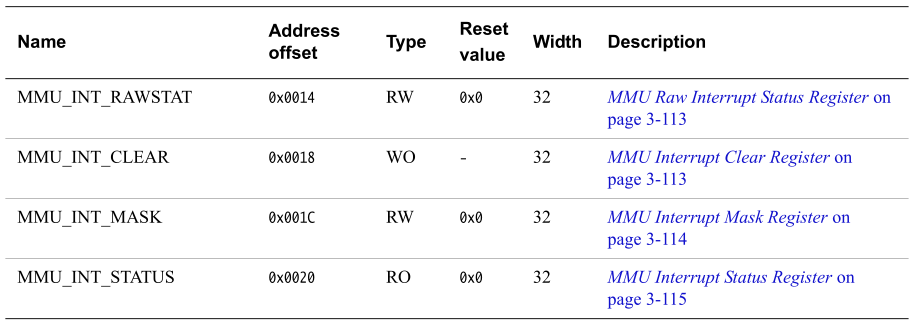




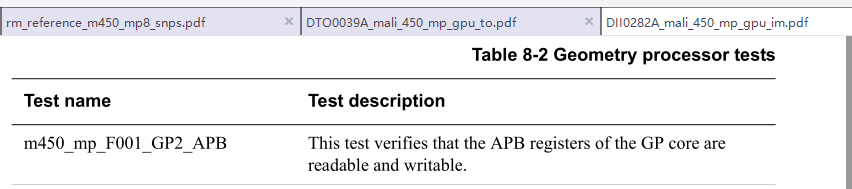
基地址0x0200\_0000 这个是MMU地址

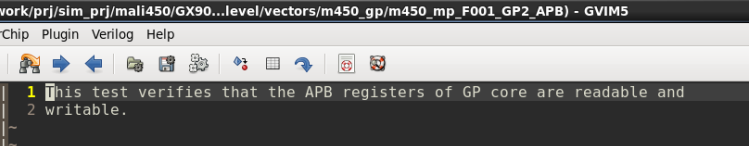




m450\_mp\_F001\_GP2\_APB





###### M450\_pp

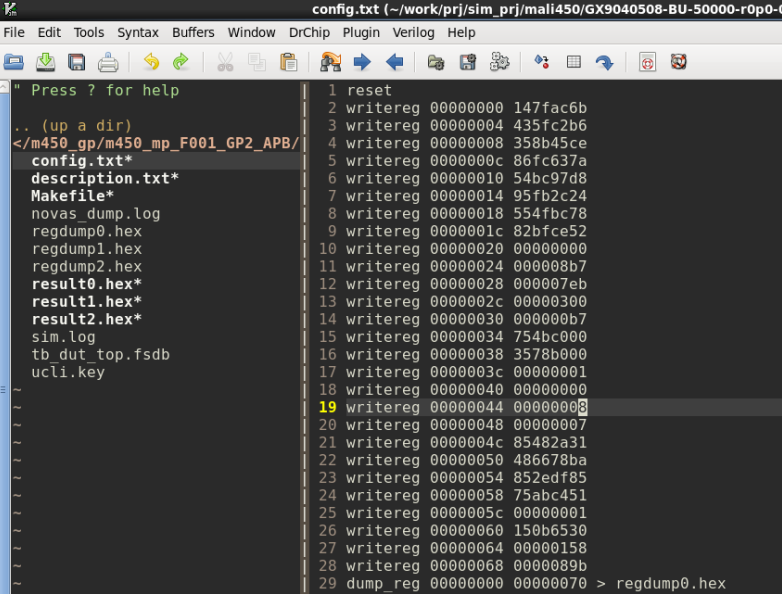
Pp表示pixel processer

m450\_mp\_F010\_PP0\_MMU\_APB

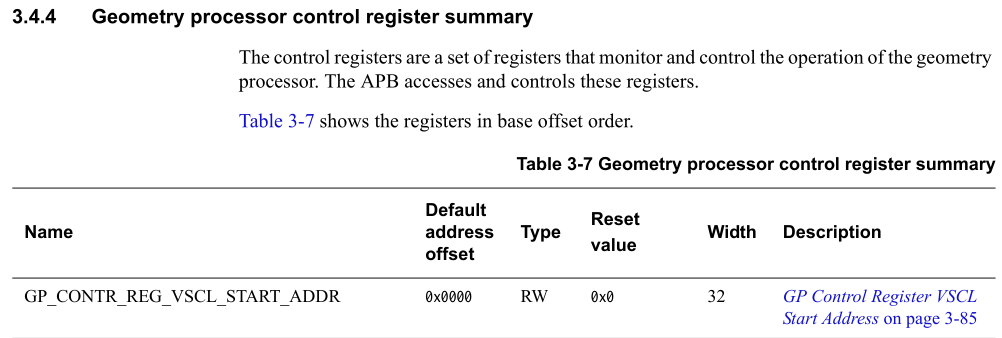
Function：



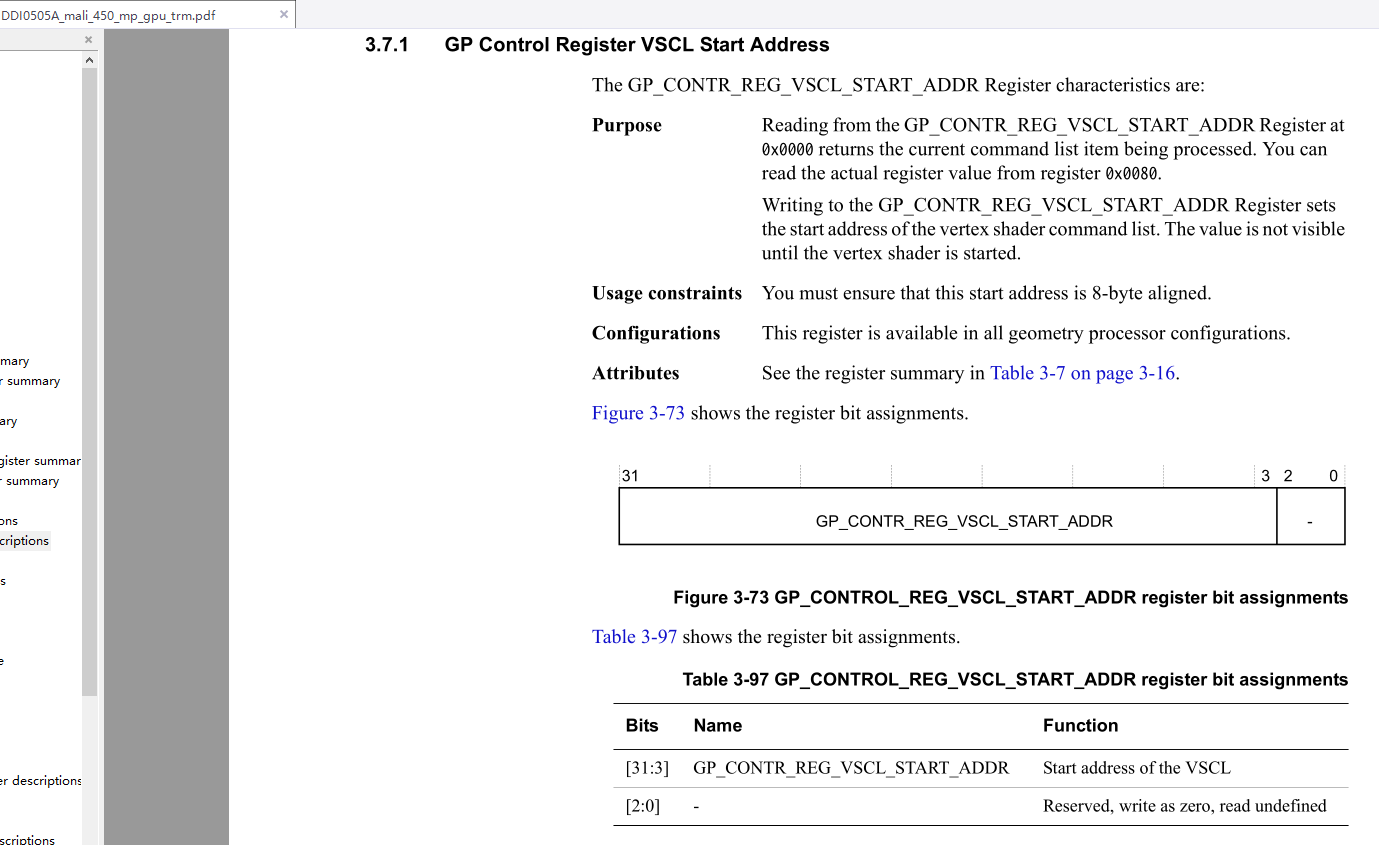
Config.v



内容：



细节：



wave

见下文

### DRIVER

介绍的驱动以DX910-BU上的OpenGLES驱动为主。

#### DRIVER

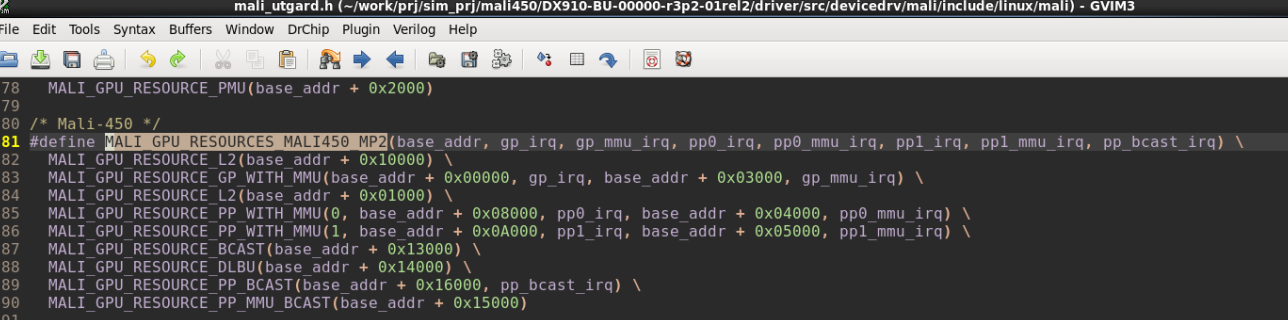
见下文

##### Mali\_gpu\_linux\_ddk\_integration\_guide

见下文

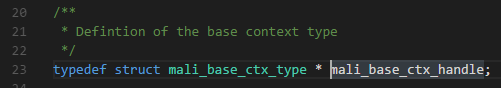
###### Mali的GPU资源信息mali\_utgard.h

见下文



#### 自定义数据类型

##### mali\_base\_ctx\_handle 自有关键字

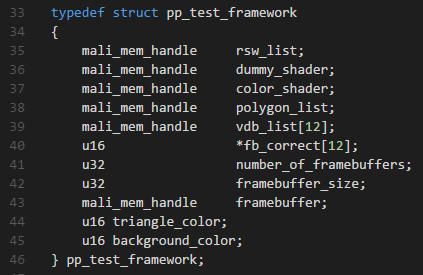


##### pp\_test\_framework\_handle，pp\_test\_framework

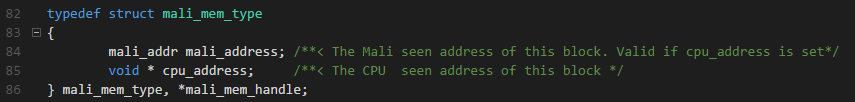
pp\_test\_framework\_handle，是一个指针类型的句柄



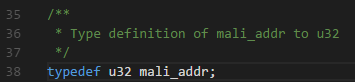
就是一个结构体，这个东西和Mali450的trm文件可一一对应。



##### mali\_mem\_type 或mali\_mem\_handle

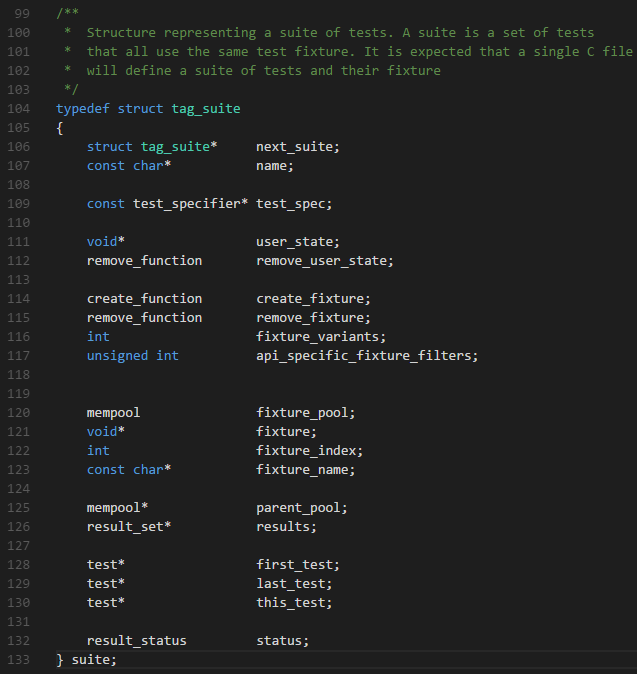


##### mali\_addr



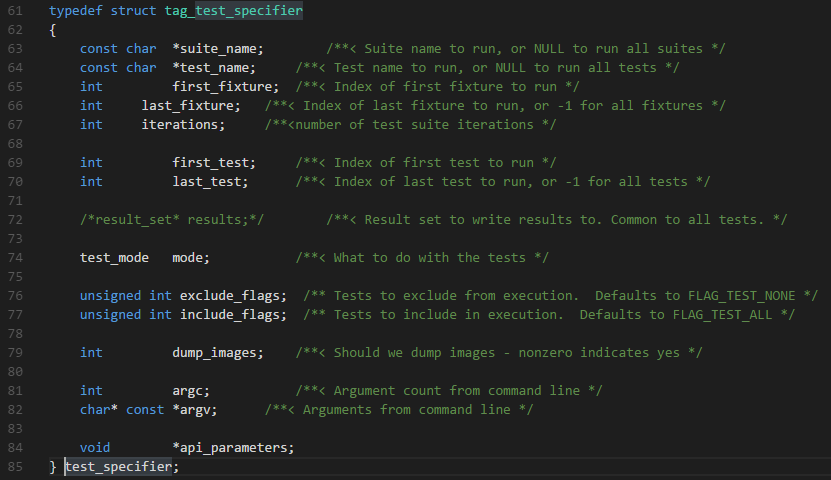
##### suite

它是一个链表



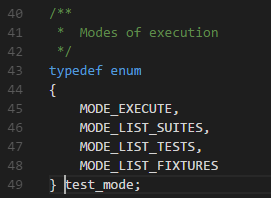
##### test\_specifier

test\_specifier ，这东西可配置成dump\_images.



##### test\_mode

子类型：test\_mode 是enum 枚举类型，枚举本质就是整形。



##### remove\_function

子类型 函数指针么？



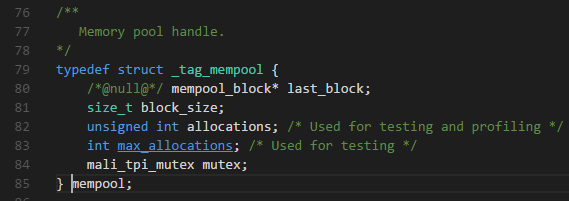


##### create\_function



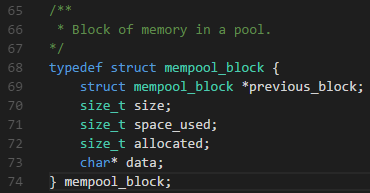
也是函数指针。

##### mempool



##### mempool\_block

他是一个链表，给出了内存大小，使用空间，数据首地址，分配情况等等信息。



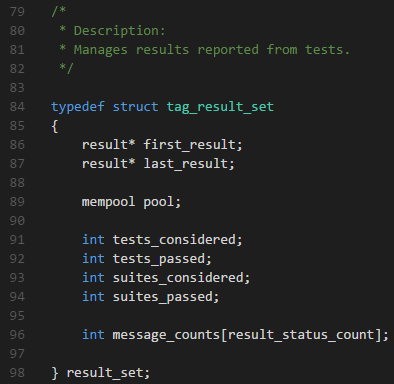
size\_t中，这里应该指的是无符号整形。



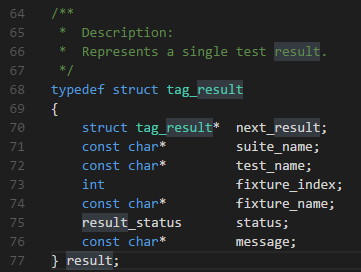
##### mali\_tpi\_mutex自有关键字



##### result\_set

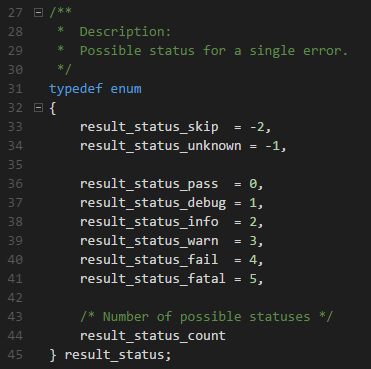


##### result

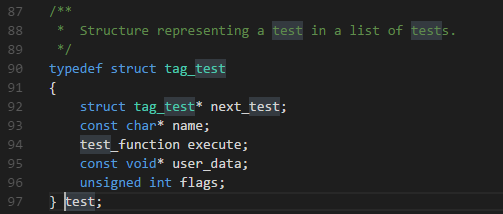


##### result\_status

枚举

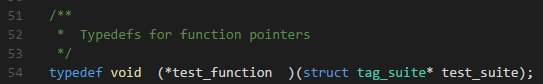


##### test



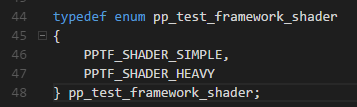
##### test\_function

函数指针，参见remove\_function有函数指针详细说明



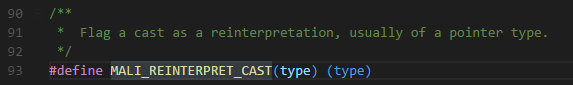
##### pp\_test\_framework\_shader

枚举，超级简单



##### MALI\_REINTERPRET\_CAST

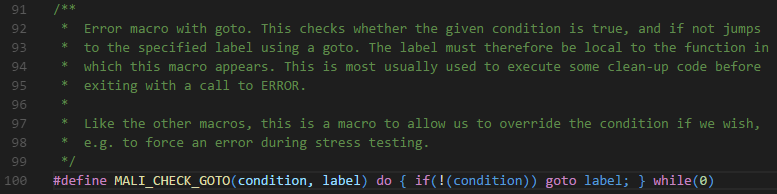
个人理解是一种函数类型强转的命令，这个太恐怖了，始得函数之间没有秘密可言啊，谨慎使用吧。





#### 自定义宏

##### MALI\_CHECK\_GOTO(condition,label)



用法举例：注意function\_failed，是goto语句中的标号哦。

如果MALI\_ERR\_NO\_ERROR 和 函数pp\_util\_internal\_\_allocate\_fram\_buffer返回值不同时，则这个宏执行到function\_failed标记，否则就是正常状态喽。

也就是 相等时，正常。

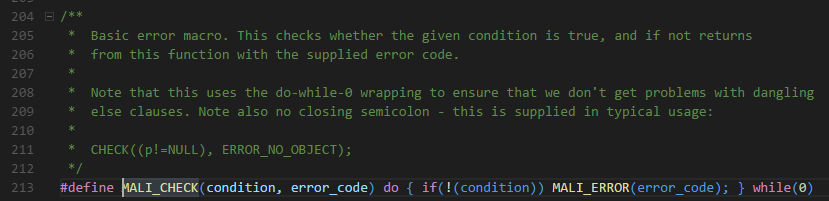




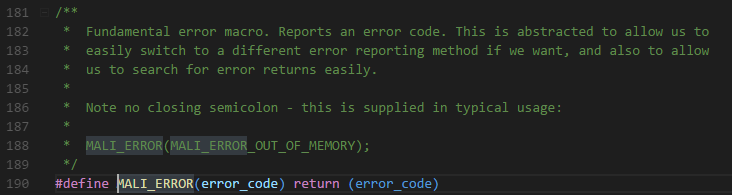
##### MALI\_CHECK\_NON\_NULL(pointer,error\_code)



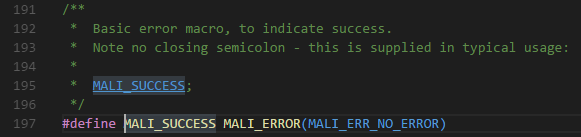
##### MALI\_CHECK(condition,error\_code)



##### MALI\_ERROR(error\_code)



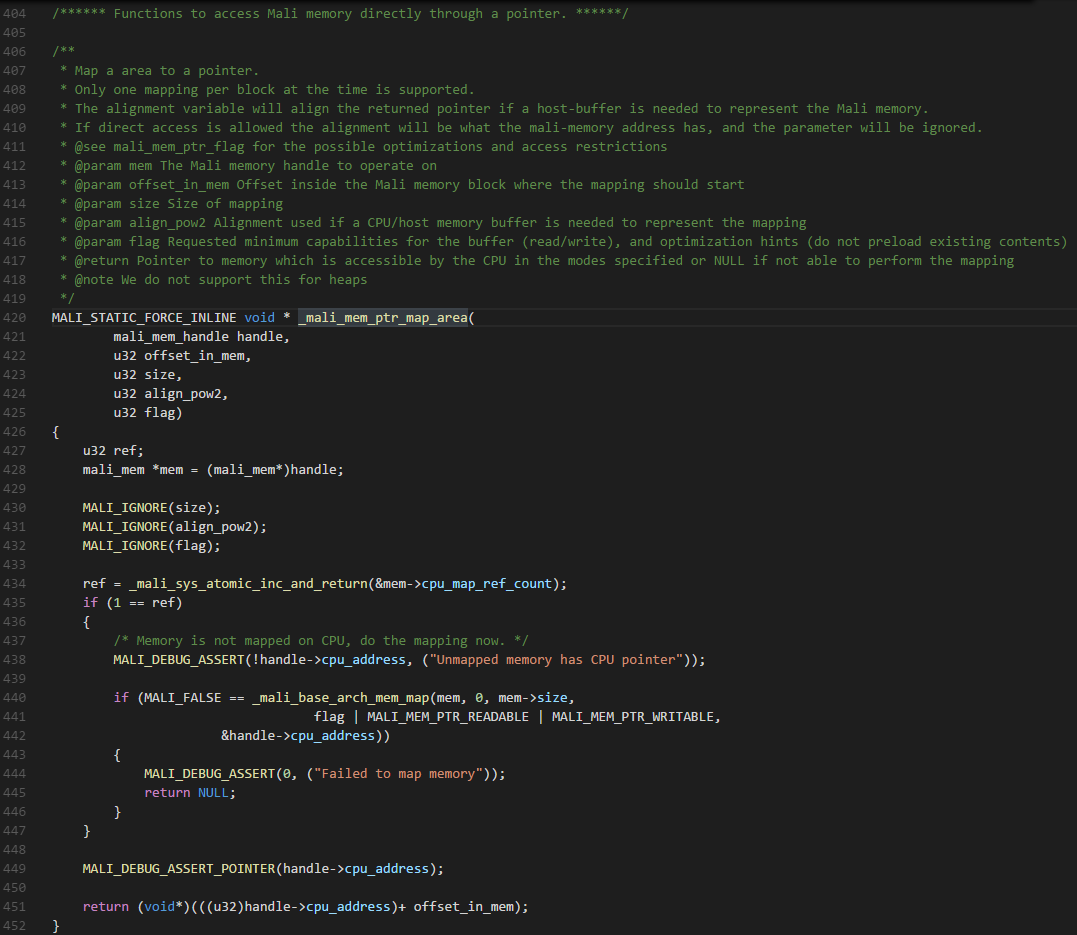
##### MALI\_SUCCESS



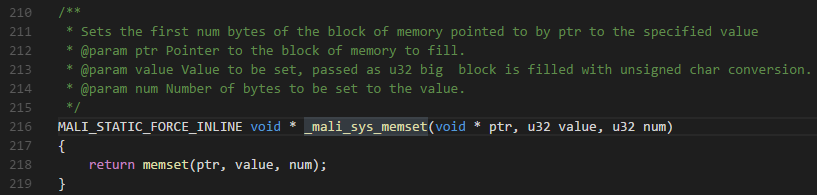
#### 自定义函数

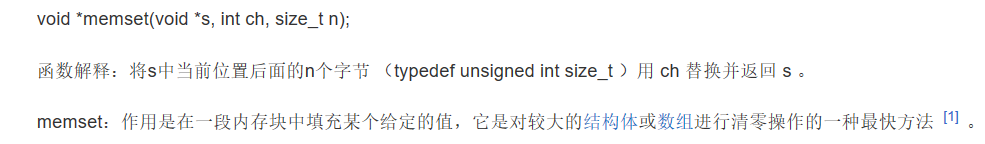
##### \_\_mali\_mem\_ptr\_map\_area (… … )

返回Mali直接访问和CPU共用缓存的指针。注意第451行，返回值是cpu\_address+偏移量。

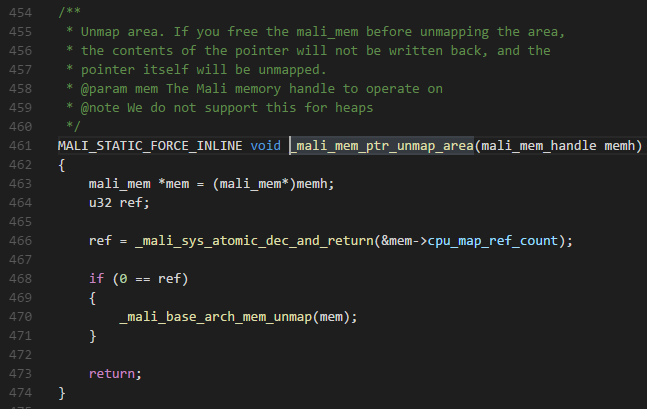


##### \_\_mali\_sys\_memset (void \*ptr,u32 value,u32 num)





##### \_\_mali\_mem\_ptr\_unmap\_area (mali\_mem\_handle memh)



##### pp\_util\_internal\_\_alloate\_frame\_buffer ()

###### 接口形式

见下文，很明显，这个函数的功能是分配帧缓存。



###### 参数说明

共4个参数

###### 调用关系

根据pf02测试函数，num\_fbs=1,表示只有一个帧buff，test\_suite和pptf是被赋值对象。



###### 函数主体

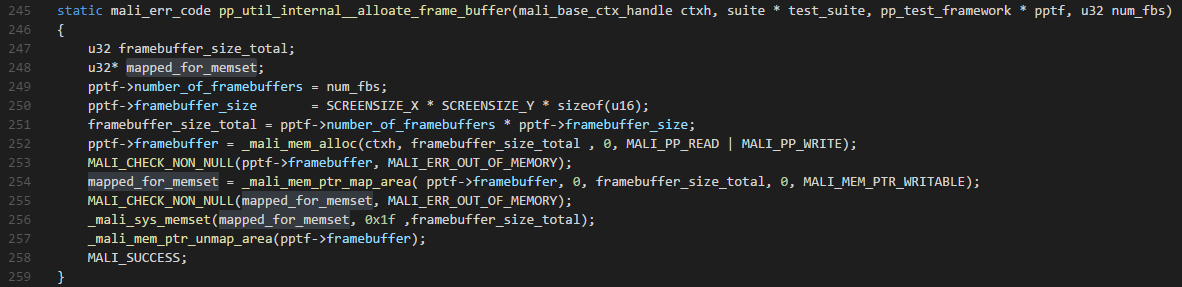
第246-250行，对pptf两个成员变量赋值，注意framebuffer\_size是250行

这里，两个值都是16.



因此framebuffer\_size\_total = 16 \* 16 \* 2 \* 1 = 512 字节

第251-252行，分配内存空间，返回地址指向pptf->framebuffer。



第253行，检查内存是否分配成功

第254-255行，设置mali指针和cpu指针一致。

第256行，关键，将和CPU共用的地址，都写成0x1f，共写512（framebuffer\_size\_total）字节。

第257行，释放framebuffer。

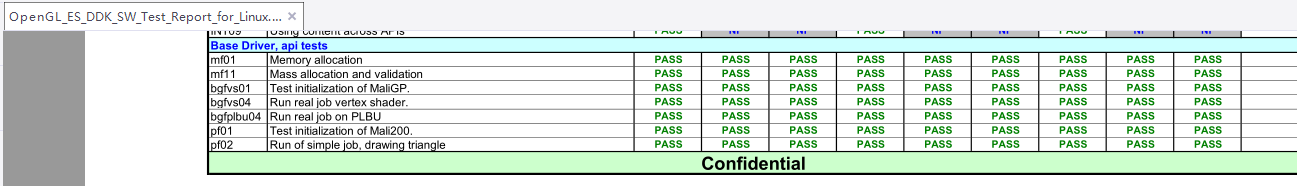
第258行，返回成功。

完毕

#### TestBench Example

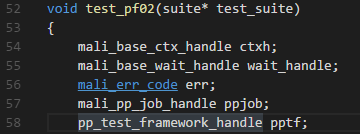
##### pf02

根据PDF文档说明，pf02用例可以画一个三角形，分析一下该程序执行方法。



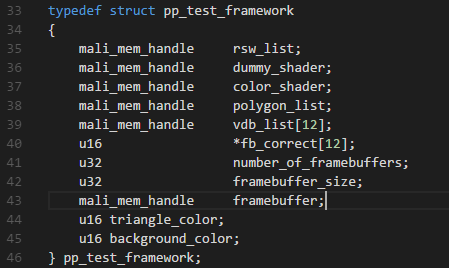
###### 第52-58行

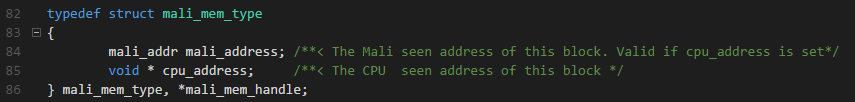
代码啃读：

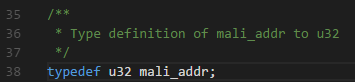


pp\_test\_framework 结构体

第58行解释：

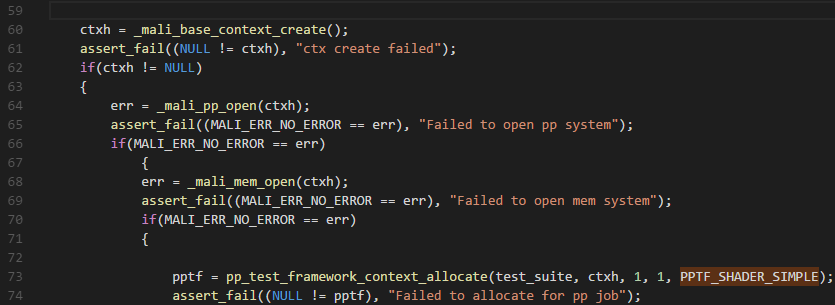






###### 第59-72行

代码啃读，前面都是创建句柄，关键函数是pp\_test\_framework\_context\_allocate



###### 第73行pp\_test\_framework\_context\_allocate（）函数

接口形式

见下文



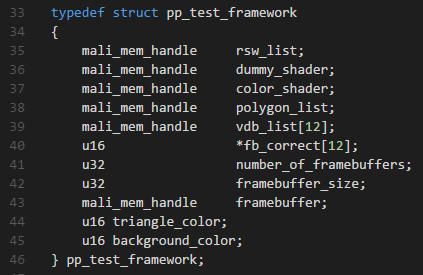
参数说明

返回参数pp\_test\_framework\_handle

返回参数：pp\_test\_framework\_handle，是一个指针类型的句柄

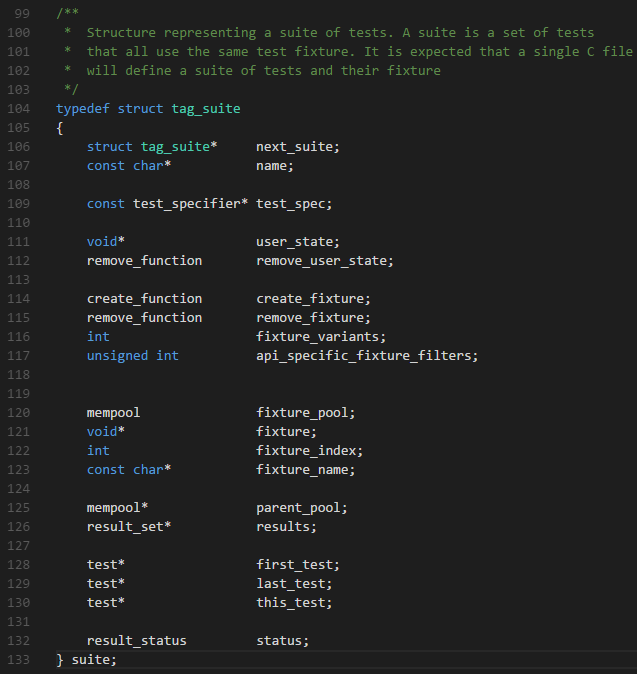


就是一个结构体，这个东西和Mali450的trm文件可一一对应。



第1参数suite \*类型

这货是个链表。



第2参数mali\_base\_ctx\_handle类型

mali\_base\_ctx\_handle ctxh，传入句柄

第3，4参数类型u32





两个值都是1

第5参数类型pp\_test\_framework\_shader

pp\_test\_framework\_shader shader，是个枚举，超级简单。

调用关系

见下文



返回的pptf，正好包含了 shader的数据结构，也就是硬件mali加速器可以识别的指令结构了。

函数主体

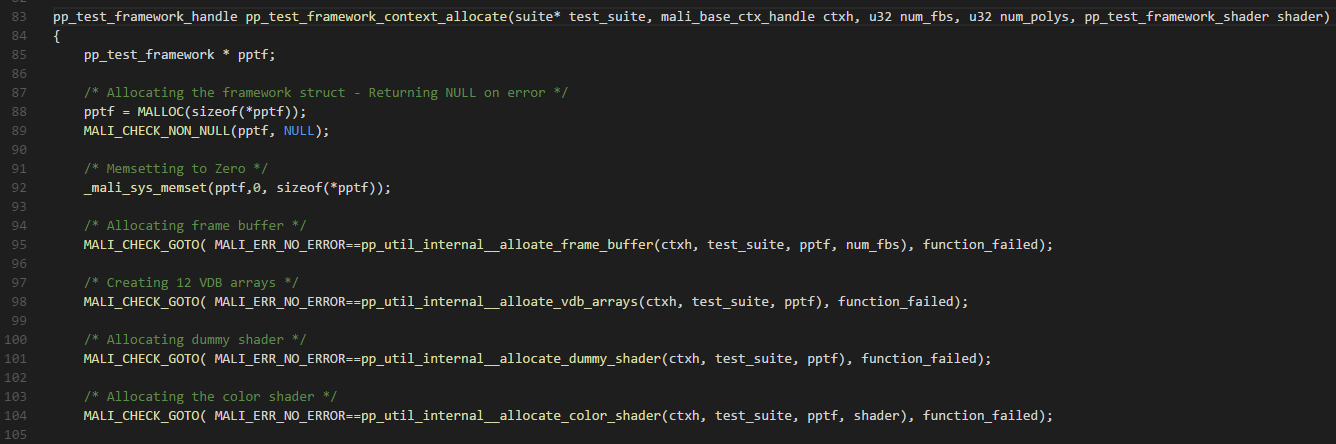
这个函数主要功能是给pptf结构体赋值，这个pptf结构体就是硬件电路mali450的数据结构，可参见trm文件



第88-89行，给pptf分配内存空间，并检查是否分配成功，如果pptf返回值不为NULL，表明分配ok了。

第90-93行，利用memset，对分配的内存清零。

第94-96行，分配frambuff



###### 第74-76行



没有实际意义

###### 第77行ppjob=pp\_test\_framework\_job\_create(ctxh,pptf,0);

创建一个job