Low Power Design Techniques for 3D GFX

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

in

Electronics & Communication Engineering

(Communication Engineering)

By

Pillai Kamlesh R

(13MECC14)

Electronics & Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2015

Low Power Design Techniques for 3D GFX

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

in

Electronics & Communication Engineering

(Communication Engineering)

By

Pillai Kamlesh R

(13MECC14)

Under the guidance of

External Project Guide:
Mr.Gurpreet Singh Kalsi
Tech Lead,
Intel Technology India Pvt. Ltd,
Bangalore.

Internal Project Guide:
Prof. Manisha Upadhyay
Associate Professor (EC Dept.),
Institute of Technology,
Nirma University, Ahmedabad.

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2015

Declaration

This is to certify that

1. The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at Nirma University and has not been submitted elsewhere for a degree.

2. Due acknowledgment has been made in the text to all other material used.

- Pillai Kamlesh R

(13MECC14)

Disclaimer

"The content of this report does not represent the technology, opinions, beliefs or positions of Intel Company, its employees, vendors, customers or associates."

Certificate

This is to certify that the Major Project entitled **"Low Power Design Techniques for 3D GFX "** submitted by **Pillai Kamlesh R (13MECC14)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven't been submitted to any other university or institution for award of any degree or diploma.

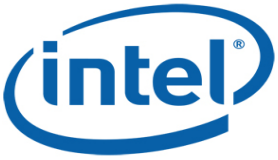
Date: Place: Ahmedabad

Guide **Section Head**

Prof. Manisha Upadhyay **Dr. D. K. Kothari**
(Associate Professor,EC dept.) (Program Coordinator)

HOD

Dr. P. N. Tekwani
(Head of EE Dept)



Certificate

This is to certify that the Major Project **"Low Power Design Techniques for 3D GFX"** submitted by **Pillai Kamlesh R (13MECC14)**,towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Date: Place: Bangalore

Mr. Vishal Mishra,
Manager,
Intel Technology India Pvt.Ltd.,
Bangalore.,

Mr. Gurpreet Singh Kalsi,
Tech Lead,
Intel Technology India Pvt.Ltd.,
Bangalore.

Acknowledgements

I would like to express my gratitude and sincere thanks to **Dr.P.N.Tekwani**, Head of Electrical Engineering Department,**Dr.D.K.Kothari**, Coordinator of M.Tech Comminication Engineering program and **Prof.Manisha Upadhyay** for allowing me to undertake this thesis work and for his guidelines during the review process.

I am deeply indebted to my thesis supervisors, **Mr.Vishal Mishra**, Manager,Intel Technology India Pvt. Ltd. and **Prof.Manisha Upadhyay**, Associate Professor,EC Department, Nirma University for their constant guidance and motivation. I also wish to thank all other team members at Intel for their constant help and support. Without their experience and insights, it would have been very difficult to do quality work.

I would like to extend sincere thanks to my guide **Mr.Gurpreet Singh Kalsi**, Tech Lead,Intel Technology India Pvt. Ltd. for guiding me throughout the project and providing valuable suggestions. He has been a constant source of knowledge and motivation to me.

I wish to thank my friends of my class for their delightful company which kept me in good humor throughout the year.

Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete the masters program successfully.

- Pillai Kamlesh R

13MECC14

Abstract

Extremely small chip consuming least power with maximum compute is something market needs. Extreme Graphics compute power became one of the major requirement for many products. Low power optimization techniques spread across architectural, micro-architectural, design and circuit techniques. This requirement pushes VLSI design engineers to keep innovating and finding solutions to achieve same. The main objective of this thesis is to discuss different low power design techniques and there application to 3D Graphics. Clocking circuitry is one of the major power consuming block in any design. This thesis work covers clock gating techniques like active stall, idle stall, etc. some of the data gating techniques and data rearrangement which will enhance clock gating are also discussed. Tradeoff for different techniques is also covered. Shutting down unused logic is another common practice to save power. Power gating comes with verification challenges. This thesis will cover power gating requirements and some of the industry standard techniques to verify crossing of signals across power domain. Caching is commonly used technique to boost up performance by reducing memory accesses. Graphics use caching concept and different caching policies are applied as per requirement, this thesis covers cache basic concept with different techniques and organization. The content addressable memory (CAM) which is used in the RAM or searching the tag value is addressed and new method of searching the tag value using hardware binary search has been demonstrated in this thesis. The implementation of the Hardware binary search algorithm reduces the overall delay in the CAM circuit by 10-20% and we reduced the active encoder size used in the CAM from 1024: 10 to 512: 9. A hardware design for calculating log base 2 and antilog base 2 using piecewise linear approximation is proposed in the thesis which is very fast as compared to the existing log and antilog circuits which uses look up table and this hardware can produce output in a single clock cycle. But this is achieved with maximum error of 0.55% for log circuit and 0.6% in case of antilog and mean error of 0.12% for 1800 samples between 2 and 20 in log circuit and 0.2%

in antilog circuit. The log and antilog circuits proposed here is used to build Phong Illumination model which is very basic lighting model in 3D graphics, where these errors in log and antilog circuits cannot be detected by human eyes. The future scope of low power designs are also discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

During the desktop computing design era, VLSI design efforts have primarily focused on optimizing the speed in order to realize computationally intensive real-time functions such as video compression, gaming, graphics etc. Because of which today people have semiconductor circuits that successfully integrates various complex signal processing modules and graphics processing units to meet their computation and entertainment demands. These solutions only addressed the real-time problems, but didn't address the increasing demand for portable operation. The handheld devices need to pack all this without consuming much power. There is a strict limitation on power dissipation in the portable electronics devices such as smart phones and tablet computers which should be met by the VLSI designers without affecting the functional requirements. The handheld devices are rapidly making their way into the consumer electronics market. One of the key design constraints for such devices are total power consumption of the devices. Reducing total power consumption in such devices is an important constrain because it is desirable to maximize the run time with least requirements on size, weight allocated to batteries and battery life. Thus most important factor to be considered while designing a system on chip for portable

devices is low power design constraints.

The scaling of technology node increases power-density much more than expected. CMOS technology below 45nm will produce a real challenge for any sort of frequency and voltage scaling. Each new process technology has inherently higher leakage and dynamic current density with minimal improvement in speed. From 90nm to 65nm the dynamic power dissipation is almost same but there is 5% higher leakage power/mm2. The low cost always continues to drive higher levels of integration. But the low cost technological breakthroughs to keep power under control are getting insufficient.

Recent developments in System-on-Chip demands for more power in both memory and logic. Static power is increasing rapidly and Dynamic power is increasing very slowly. Overall, the power is dramatically increasing. If the semiconductor integration continue to follow Moore's Law, the power density inside the chips will reach far higher than the rocket nozzle[1]. The figure 1.1 shows the power requirements and power trends in SoC (system on chip). It's clear from the figure that the power requirement still remains almost constant, whereas the power trend is increasing exponentially.

The modern devices are compared not only based on power performance but also on graphics performance. The consumers demand for very high quality of graphics, which requires large amount of processing and thus will consume more power. The main objective of this thesis report is to develop low power designs for 3D graphics.

1.2 Area of work

The work is basically focused in the area of low power design techniques in the VLSI domain for 3D graphics. Power dissipation is one of the major constrains when it comes to Portability. The device consumer demands more features and extended battery life at a lower cost. More than 70% of users demand for longer talk and stand-by time as a primary smart phone feature. The primary 3G (3rd Generation telecommunication technology) requirement for operators is power efficiency. User

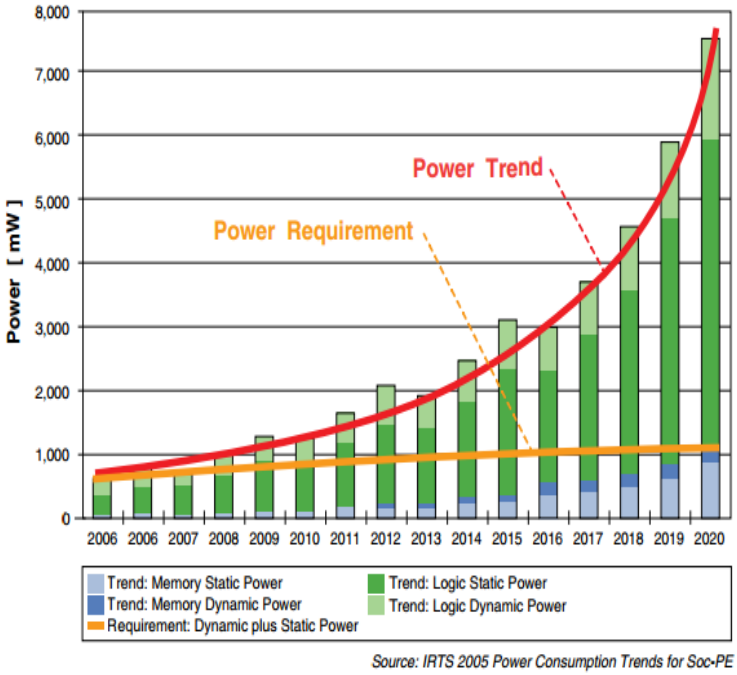


Figure 1.1: Power consumption trend for Soc[1]

wants smaller and sleeker handheld devices. This will require a high level of Silicon integration in advanced processes, but advanced processes tends to have inherently leakage current on the higher side. So there is a concern more on reducing leakage current to reduce power consumption.

The figure 1.2 shows the flow chart of ASIC (Application specific Integrated circuit) flow and the highlighted portion is the area in which the work has been carried out.

In the RTL design, the codes are generally written in Verilog or System Verilog, and VCS/ Cadence LDV tool is used for simulation. Design Compiler-tool (DC - Synopsys) is used for synthesis. Formal verification is done by using a tool called Jasper Gold.

1.3 Problem statement

As the technology is shrinking, the reducing power consumption and over all power management on a chip are the major challenges below 65nm due to complexity. For many designs, optimization of power is as important as timing because of the need for reduced packaging cost and extended battery life. This becomes even more challenging when it comes to the 3D graphics, where large amount of processing is needed. This will consume large amount of power. For power management on a SoC, the leakage current plays an important role as far as low power VLSI designs are concerned. Leakage current is getting increased as an important part of the total power dissipation of integrated circuits below 65nm technology.

1.4 Organization of thesis

The rest of the thesis is organized as follow:
Chapter 2 describes the detailed literature review which includes function of each stages in the Direct3D X graphic pipeline and a brief overview of the low power design techniques which can be used at various stages of the design.
In chapter 3 some low power design techniques in RTL like clock gating, active stall, operation rearrangement which are used to implement low power graphics in Intel has been discussed. It also describes the cache memory organization, different types of cache techniques which can be used depending on the application needs. Along with it CAM (Content Addressable Memory) is also been discussed. The Hardware implementation for binary search algorithm is also explained in this chapter.
In chapter 4 dedicated hardware for performing logarithm of base 2 and antilog base 2 circuits are explained. The Phong Illumination model is based on log and antilog circuit is praposed, which is explained in this chapter.
Finally, in Chapter 5 concluding remarks and scope for the future work in low power design techniques in 3D graphics is presented.

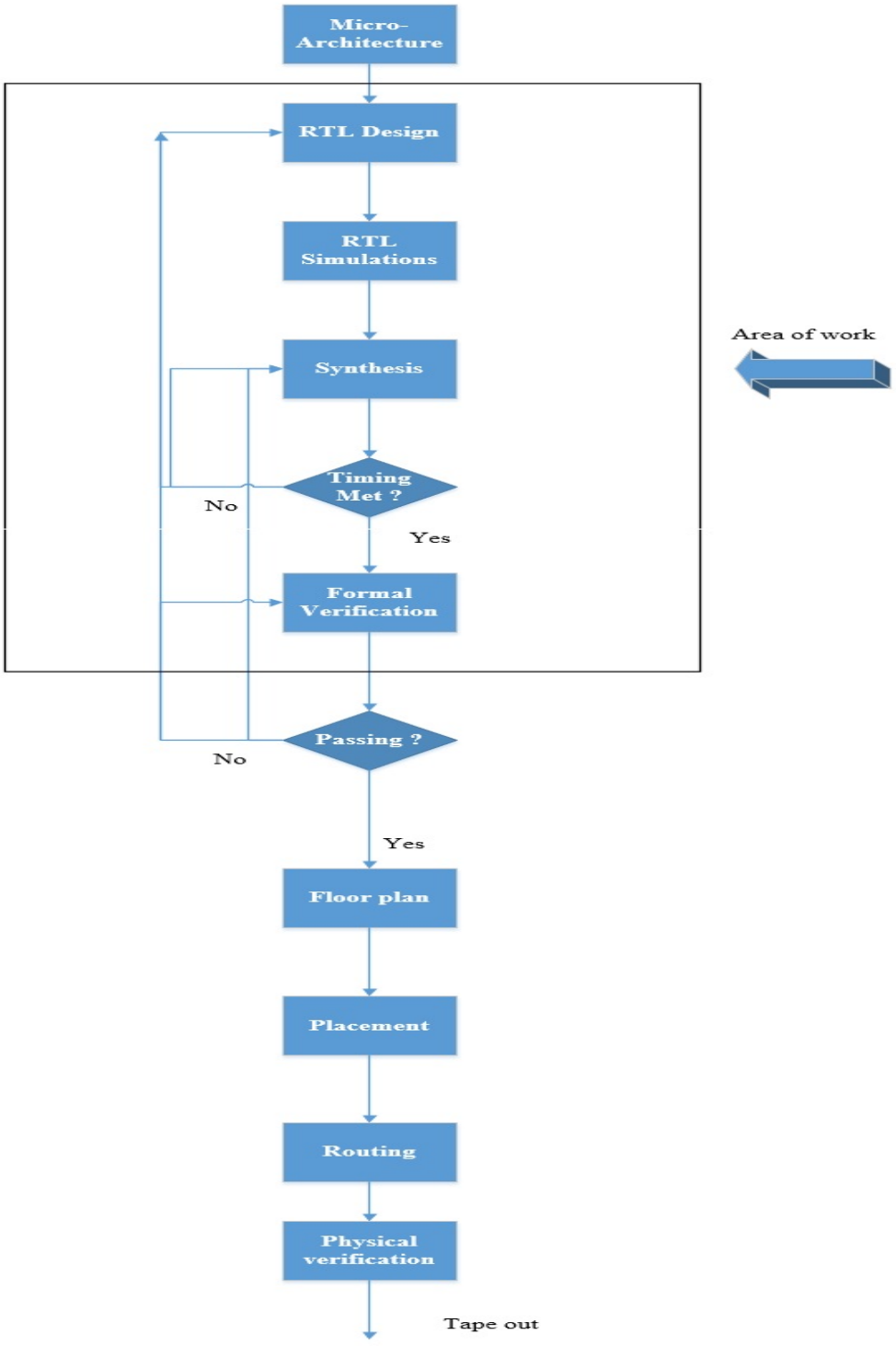


Figure 1.2: Area of work in ASIC flow

Chapter 2

Literature Survey

2.1 Introduction

The requirement for low power devices are increasing rapidly due to the demand of handheld devices. The user requires all high-end applications on the handheld devices which generally runs on the battery. In todays global market devices are compared based on the battery efficiency along with the graphics performance. The devices like smartphones, tablets and other handheld devices requires low power circuits, which will increase their battery efficiency. Moreover, the graphics performance are used as one of the key specification for comparing various devices. The graphics processing requires large amount of computation which will consume more power. In literature many low power design techniques are discussed. These techniques will be discussed in the coming sections.

The on chip memory bandwidth is the real bottleneck, to render a very high quality graphics we will need large amount of memory for processing. To render such high quality graphics we use the pipeline architecture. In pipeline architecture more than one operation is performed in a given time without using much of memory bandwidth and processing at higher rate. In modern day processors integrated GPU is quite common. Whenever we speak of integrated / soc (system on chip) many problem

comes along with it. Generally speaking on chip memory are too costly to build and they will be even more costly when build very close to the processor. No doubt the speed of the operation gets increased, also need to take care about certain drawbacks. These all things apply to the on chip graphics also.
Due to the above fact the 3D graphics is rendered through a pipeline called 3D pipeline/ Rendering pipeline. Basically the 3D pipeline are mapped to Direct3D of Microsoft/ OpenGl architecture in order to take the advantages of API like DirectX / OpenGl. The rendering pipeline is basically a set of stages which can be used in order to overcome the memory bandwidth issue. The API (Application programing Interface) are used to make the code or design machine independent.
In the next few section will focus on the basic terminology used in graphics. The later sections will cover the basic behavior of different stages in the pipeline and also presents the mutual difference between DirectX, OpenGl.

2.2 Basic terminologies in graphics

1. **Primitives:** A simple object to render. Commonly used primitives are triangles, lines and points.

2. **Rendering:** The process of converting primitives into pixels.

3. **Pixel:** Short form for 'Picture element', it's that tiny little dot of light that makes your screen glow much smaller than the punctuation mark at the end of the sentence.

4. **Vertex:** A "corner" of a primitive having a position in space and other attributes (color, texture, coordinates) describing it.

5. **Buffer:** An allocation of memory that stores information to be processed (input buffer) or result of processing (output buffer). Also called a surface.

6. **Vertex buffer:** A buffer that stores vertex position and other attributes for a list of vertices.

7. **Texture map:** An input buffer that stores rasterized "texture" to apply to primitives.

8. **Render target:** An output buffer that stores the result of rendering step. Can contain color, depth information (color buffer, depth buffer) and can be reused as a texture map in a later rendering step (render to texture).

9. **Frame buffer:** The particular output buffer which contains the color information to be read by the display hardware and shown on the screen.

10. **3D:** The part of the graphics hardware which takes a stream of state and primitive commands describing a 3 - dimensional scene and renders an image to render target. Other parts / behavior of the graphic hardware include display, media, and GPGPU.

11. **Direct X, Direct 3D (D3D):** Microsoft API for programing 3D hardware. A 3D application called the D3D layer, which calls the graphic driver, which controls the graphic hardware. OpenGL is the other main API for programing 3D hardware.

12. **Shader:** A program which operates on some graphics element at some storage stage of the 3D rendering pipeline.

    (a) **Vertex Shader:** Transforms a vertex attributes.

    (b) **Geometry Shader:** Creates new vertices from existing ones.

    (c) **Pixel Shader:** Determine what color to draw a piece of triangle.

2.3 Direct3D X rendering pipeline

The figure 2.1 shows the Direct3D X rendering pipeline. The pipeline is divided into 2 parts one is the rendering pipeline where the actual processing of the vertices or primitives are carried out and the second is resources. The resources are basically, buffers which contains the attributes to be processed. Like vertex attributes are placed in the vertex buffer, their indices are placed in the index buffer and so on. The figure 2.2 shows the basic functionality of the Input assemble. The application developer develops an application, say for example windows application which are generally written in some high level language like C, java etc. these are read by the system call API. The system call API is divided into three parts 1) run time API 2) user mode drivers and 3) gfx schedular. The application is given to run time API which converts the codes into some binary codes which can be processed by the CPU. The shader codes are not compiled here and the application codes are compiled here. The next stage is user mode drivers, in this stage all the .dll file (dynamic link libraries) are linked along with compilation of shader code and linking of shader codes. GFX shedular is the stage which allocates GPU for the application to run. Its the schedular which take care of the execution of multiple applications at a very high speed that user feels like very thing is running in parallel. Once the processing is done in the system call API stage the OS kenel will be invoked. The kenel in return will invoke the device driver through interacting via API calls. The driver converts the in coming API calls to some binary codes (GFx commands) which are placed in the memory and IA fetches them from the main memory.

The GPU as only one memory and many applications fights for the memory. There should be some entity to govern that activity. There should be some entity that can initialize the GPU, flush the registers and may be used to program a watchdog timer which can be used to reset the GPU if it is not responding for a predefined amount of time. These are governed by the kernel.

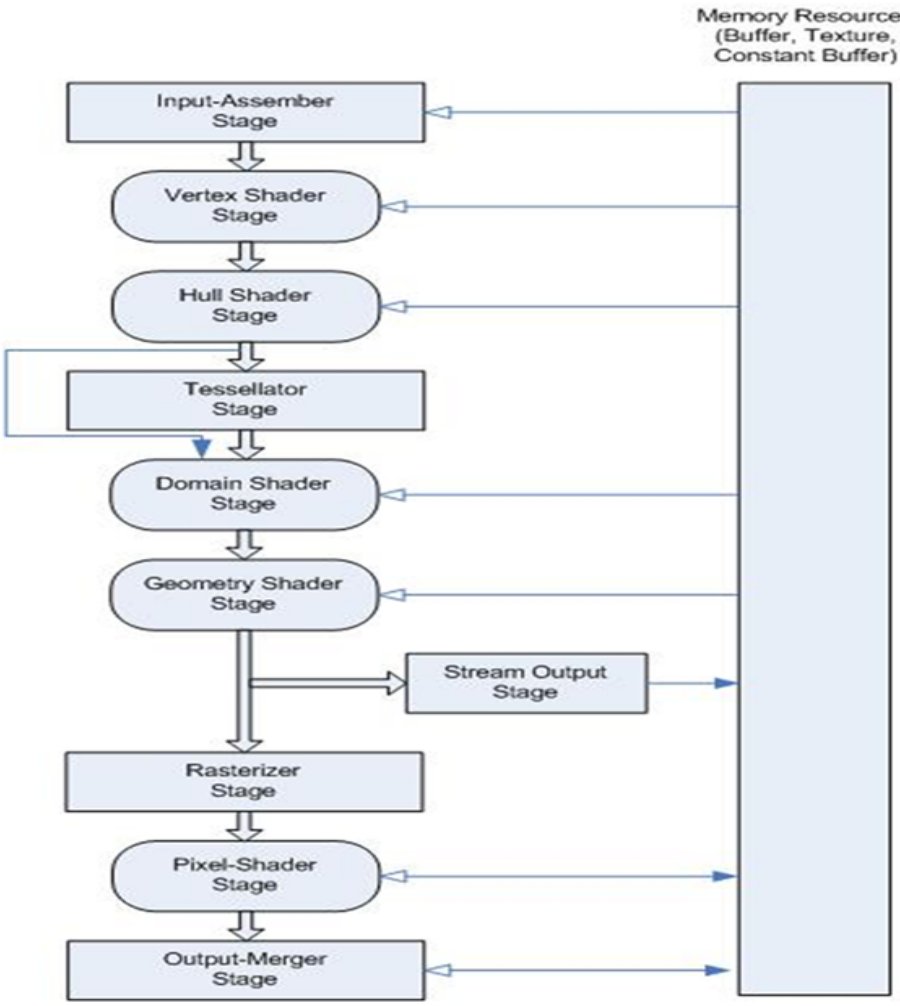


Figure 2.1: Direct3D X rendering pipeline[2, 3].

2.3.1 Input assembler

- The purpose of the input-assembler stage is to read primitive data which are generally points, lines and/or triangles from user-filled buffers and assemble the data into primitives that will be used by the other pipeline stages[2] . The input assembler stage can assemble vertices into several different primitive types such as line lists, triangle strips, or primitives with adjacency.

- The secondary purpose of the IA is to attach system-generated values to help make shaders more efficient. System-generated values are text strings that are also called semantics[2]. All three shader stages are constructed from a common shader core, and the shader core uses system-generated values such as a primitive id, an instance id, or a vertex id so that a shader stage can reduce processing to only those primitives, instances, or vertices that have not already been processed[3].

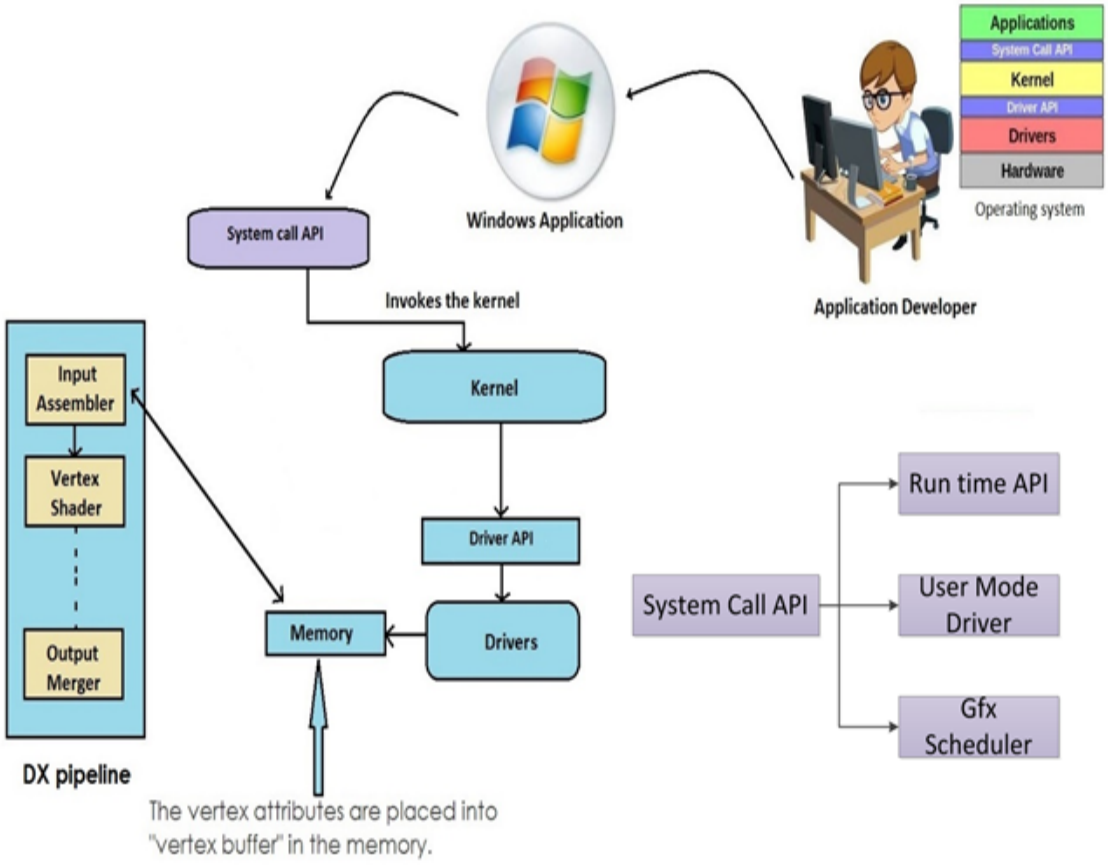


Figure 2.2: Basic function of Input assembler.

2.3.2 Vertex shader

- It's a shader program which operates on vertex and a vertex has its attributes like positions, color and normal vectors and texture coordinates etc. VS (vertex shader) transforms input/pre-shaded attributes to output/post shaded attributes. It does not change the number of vertices i.e. one vertex goes in other one comes out. It operates only on one vertex at a given time, it keeps 1:1 mapping of vertex. During the operations on one vertex, at any given time the vertex shader does not have an access to the attributes of any other vertex[3].

- The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. It always takes a single input vertex and produces a single output vertex. The vertex-shader stage can consume two system generated values from the input assembler (IA): Vertex ID and Instance ID[3]. Since Vertex ID and Instance ID are both meaningful at a vertex level, and IDs generated by hardware can only be fed into the first stage that understands them, these ID values can only be fed into the vertex-shader stage. Major functions of Vertex shader are:

  1. Transformation.
  2. Lightings.
     a Ambient.
     b Diffusion.
     c Specular.
  3. Skinning.

- The vertex-shader (VS) stage processes incoming vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. Vertex shaders always operate on a single input vertex and produce a single output vertex. The vertex shader stage must

always be active for the pipeline to execute and executed on every vertex. If no vertex modification or transformation is required, a pass-through vertex shader must be created and set to the pipeline[2].

- Each vertex shader input vertex can be comprised of up to 16 32-bit vectors (up to 4 components each) and each output vertex can be comprised of as many as 16 32-bit 4-component vectors [2].

- Vertex shaders are always run on all vertices, including adjacent vertices in input primitive topologies with adjacency.

2.3.3 Tessellation

Tessellation is a process of dividing larger primitives in to small triangles depending on the level of detail (LOD). The figure 2.3 explains about the function of different stages involved in the tessellation pipeline. The Tessellation pipeline is divided into 3 stages. 1. Hull Shader. 2. Tessellator and 3. Domain shader.

I. **Hull shader**

Hull shader is the first stage in the tessellation universe. In a tessellation universe, we don't get input vertices and triangles but instead we get input control points and patches. Hull shader is operated once per patch. The ICPs (Input control points) are analogous to triangles that can be defined by between 1 and 32 control points, but usually just 3 or 4. The shader output will be between 1 and 32 control points regardless of the tessellation factor [2]. The control points and the patch constant data can be consumed by a domain shader along with the tessellation factors which will be consumed by tessellation stage. In general hull shader thread takes input control points as an input and produces output control points, tessellation factors and patch constant data as shown in the figure 2.3 and OCPs (Output Control Points) are analogous to post transformation vertices. The tessellation factor describes how much subdividing of the patch

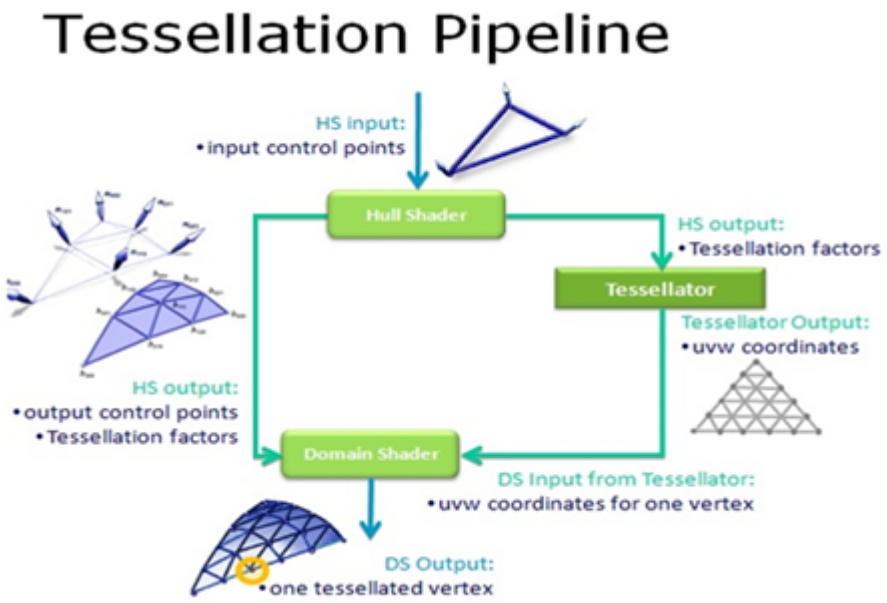


Figure 2.3: Tessellation pipeline.

the tessellator should do. The DX allows the hull shader to be multithreaded into 3 phases:[3]

- Compute each OCP based on all ICPs.
- Compute edge tessellation factor based on ICPs and OCPs.
- Compute interior tessellation factor based on all the above.

A hull shader,which is invoked once per patch, transforms input control points that define a low-order surface into control points that make up a patch [3]. It also does some per patch calculations to provide data for the tessellation stage and the domain stage i.e in (u, v) coordinates. At the simplest black-box level, the hull-shader stage is shown in figure 2.4.

A hull shader is implemented with an HLSL function, and has the following properties: [2]

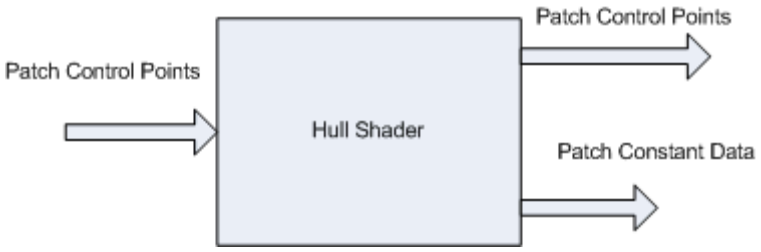


Figure 2.4: Hull shader [2].

- The shader input is between 1 and 32 control points.
- The shader output is between 1 and 32 control points, regardless of the number of tessellation factors. The control-points output from a hull shader can be consumed by the domain-shader stage. Patch constant data can be consumed by a domain shader; tessellation factors can be consumed by the domain shader and the tessellation stage.
- Tessellation factors determine how much to subdivide each patch.
- The shader declares the state required by the tessellator stage. This includes information such as the number of control points, the type of patch face and the type of partitioning to use when tessellating. This information appears as declarations typically at the front of the shader code.
- If the hull-shader stage sets any edge tessellation factor to ≤ 0 or NaN, the patch will be culled. As a result, the tessellator stage may or may not run, the domain shader will not run, and no visible output will be produced for that patch.
- At a deeper level, a hull-shader actually operates in two phases: a control-point phase and a patch-constant phase, which are run in parallel by the hardware. The HLSL compiler extracts the parallelism in a hull shader and encodes it into bytecode that drives the hardware.
- The control-point phase operates once for each control-point, reading the

control points for a patch, and generating one output control point (identified by a ControlPointID).

The patch-constant phase operates once per patch to generate edge tessellation factors and other per-patch constants (data which remains constant throughout a patch). Internally, many patch-constant phases may run at the same time and has read-only access to all input and output control points.

II. **Tessellator**

Tessellator is a fixed function stage initialized by hull shader to the pipeline. The major function of tessellator stage is to subdivide a domain into many smaller objects like triangle, points or lines. The tessellator also operates once per patch using tessellation factor i.e. it takes tessellation factor as an input. These tessellation factors were computed for each edge and the interior of each patch by hull shader[2].
Internally, tessellator operates in two phases :

- The first phase is about processing the tessellation factors, fixing rounding problems, handling very small factors, reducing and combing factors and using 32-bit floating point arithmetic [4].
- The second stage is the core task of the tessellation stage which generates the points or topology lists based on the type of partitioning selected and for this it uses 16 -bit fixed point arithmetic[4].

It generates domain points in (u, v) coordinates as outputs which describes the interior of the patch. Conceptually they're just a bunch of (x, y) offsets at this point in the pipeline. Higher the tessellation factor means more domain points get generated for that edge or interior. Finally these domain points will become vertices in later stages of the pipeline. There are two types of tessellation factor used by the tessellator they are 1) inner tessellation factor and 2) outer

tessellation factor. The figure 2.5 shows the operation of tessellator in quad domain and in tri domain.

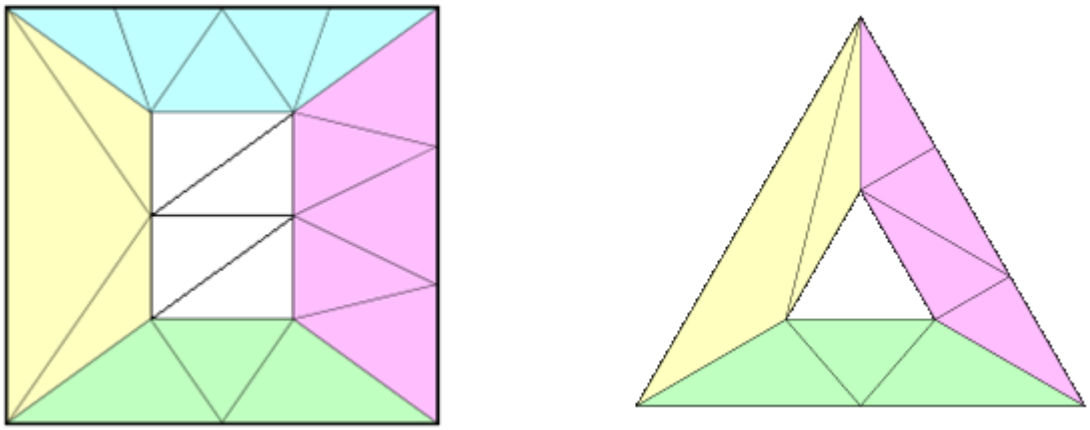


Figure 2.5: Subdivision of a patch by Tessellator

The tessellator tiles a canonical domain in a normalized coordinate system. For example, a quad domain is tessellated to a unit square.
The tessellator operates once per patch using the tessellation factors which specify how finely the domain will be tessellated and the type of partitioning which specifies the algorithm used to slice up a patch that are passed in from the hull-shader stage. The tessellator outputs u v (and optionally w) coordinates and the surface topology to the domain-shader stage [3].
Internally, the tessellator operates in two phases:

- The first phase processes the tessellation factors, fixing rounding problems, handling very small factors, reducing and combining factors, using 32-bit floating-point arithmetic [3].
- The second phase generates point or topology lists based on the type of partitioning selected. This is the core task of the tessellator stage and uses 16-bit fractions with fixed-point arithmetic. Fixed-point arithmetic allows hardware acceleration while maintaining acceptable precision. For

example, given a 64 meter wide patch, this precision can place points at a 2 mm resolution [3].

Table I: Types of partition and their range in Tessellation [2].

Type of Partitioning	**Range**
Fractional_odd	[1...63]
Fractional_even	TessFactor range: [2..64]
Integer	TessFactor range: [1..64]
Pow2	TessFactor range: [1..64]

III. **Domain shader**

Domain shader transforms the domain point in (u, v) coordinates to the real vertices[3]. The input to the domain shader includes domain points, all the outputs of the hull shader, output control points, patch constant points, patch constant data , tessellation factors etc. It calculates the position of the output vertices based on domain points and the output control points. Along with this it can perform other operations like displacement mapping, transformation etc. The real vertices which are produced as the output of the domain shader which are arranged into a geometry by using the indices from the tessellator and will be passed to Geometry shader and beyond in the pipeline.

A domain shader calculates the vertex position of a subdivided point in the output patch. A domain shader is run once per tessellator stage output point and has read-only access to the tessellator stage output (u, v) coordinates and the hull shader output patch, and the hull shader output patch constants, as the figure 2.6 shows.

Properties of the domain shader include: [3]

- A domain shader is invoked once per output coordinate from the tessellator stage.
- A domain shader consumes output control points from the hull-shader stage.

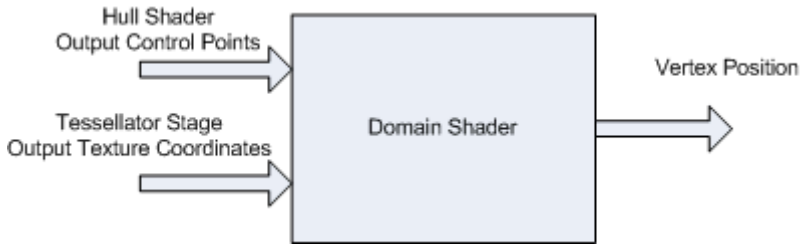


Figure 2.6: Domain shader[2].

- A domain shader outputs the position of a vertex.

Inputs are the hull shader outputs including control points, patch constant data and tessellation factors. The tessellation factors can include the values used by the fixed function tessellator as well as the raw values (before rounding by integer tessellation, for example), which for example, facilitates geomorphing. After the domain shader completes, tessellation function is finished and data continues to the next pipeline stage (geometry shader, pixel shader etc).

2.3.4 Geometry shader

The geometry shader works by taking vertices as the inputs and producing vertices as the outputs similar to vertex shader but geometry shader does not maintain 1:1 mapping between the vertex (input and output). The geometry shader inputs are vertices for a full primitive (two vertices for lines, three vertices for triangles, or single vertex for point) different from the vertex shader which operates on a single vertex. Geometry shaders as a capability to bring in the vertex data for the edge-adjacent primitives as input for example, an additional two vertices for a line or an additional three for a triangle.

The geometry shader can produce an output which has more than one vertex from a single selected topologies like tristrip, linestrip, pointlist etc. The geometry shader

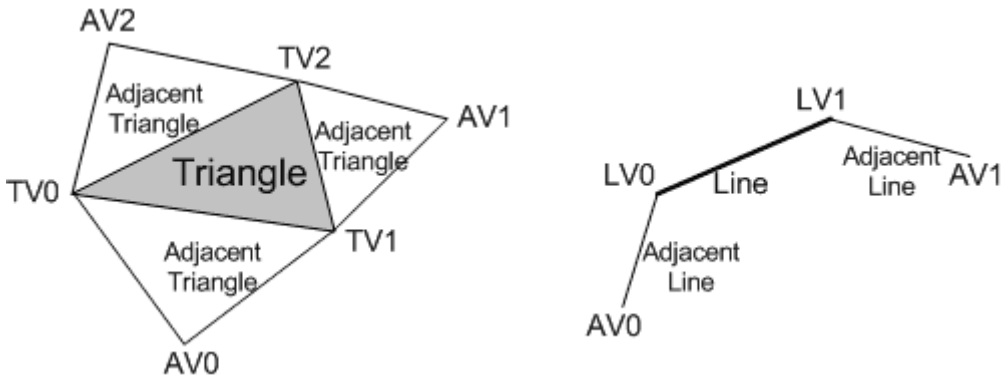


Figure 2.7: Primitives with adjacency [2].

output may be fed to the rasterizer stage and/or to a vertex buffer in memory via a stream output stage, which will be expanded to individual point/line/triangle lists exactly as they would be passed to the rasterizer[3].

Whenever a geometry shader is active, it is invoked once for every primitive including adjacency information those were passed down or generated earlier in the pipeline. At each invocation of the geometry shader takes the input data of invoking primitive, whether that is a single point, a single line, or a single triangle. A triangle strip from earlier stage in the pipeline would result in an invocation of the geometry shader for each individual triangle in the strip as if the strip were expanded out into a triangle list [2]. All the input data for each vertex in the individual primitive is available (i.e. 3 vertices for triangle), along with adjacent vertex data if applicable or available. The geometry shader outputs data one vertex at a given time by appending vertices to an output stream object. The topology of the streams is determined by a fixed declaration, choosing one of: PointStream, LineStream, or TriangleStream as the output for this shader stage. Execution of a geometry shader instance is atomic from other invocations, except that data added to the stream output stage is serial. The outputs of a given invocation of a geometry shader are independent of other invocations even though ordering is respected. When a geometry shader output is identified as a System Interpreted the hardware looks at this data and performs some

behavior dependent on the value, in addition to being able to pass the data itself to the next shader stage for input [2]. When such data output from the geometry shader has meaning to the hardware on a per-primitive basis instead of a per-vertex basis, the per-primitive data is taken from the leading vertex emitted for the primitive. If the geometry shader ends and the primitive is incomplete, the partially completed primitives will be generated and these incomplete primitives are silently discarded. Various algorithms that can be implemented in the geometry shader are as follow:[3]

1. Silhouette Detection
2. Point Sprite Expansion.
3. Dynamic Particle Systems.
4. Fur/Fin Generation.
5. Shadow Volume Generation.
6. Single Pass Render-to-Cubemap.
7. Per-Primitive Material Swapping.

2.3.5 Stram output

The purpose of the stream-output stage is to continuously output or stream vertex data from the geometry-shader stage or the vertex-shader stage, if the geometry-shader stage is inactive to one or more buffers in memory.
The stream-output stage (SO) is located in the pipeline right after the geometry-shader stage and just before the rasterization stage as shown in figure 2.1. Data streamed out to memory can be read back into the pipeline in a subsequent rendering pass, or can be copied to a staging resource, so it can be read by the CPU. The amount of data streamed out can vary depending on the geometry shader or vertex shader output.

When a triangle or line strip is bound to the input-assembler stage, each strip is converted into a list before they are streamed out. Vertices are always written out as complete primitives for example, 3 vertices at a time for triangles; incomplete primitives are never streamed out. Primitive types with adjacency discard the adjacency data before streaming data out.
There are two ways to feed stream-output data into the pipeline: [2]

- Stream-output data can be fed back into the input-assembler stage.
- Stream-output data can be read by programmable shaders using load functions.

If you are streaming data into multiple buffers, each buffer can only capture a single element (up to 4 components) of per-vertex data, with an implied data stride equal to the element width in each buffer (compatible with the way single element buffers can be bound for input into shader stages) [2]. Furthermore, if the buffers have different sizes, writing stops as soon as any one of the buffers is full [DX].If you are streaming data into a single buffer, the buffer can capture up to 64 scalar components of per-vertex data (256 bytes or less) or the vertex stride can be up to 2048 bytes [2].

2.3.6 Rasterizer

Rasterization is the process of finding all the pixels (picture elements) inside the triangle. Generally, we find all the pixels whose centers are inside the triangle and this is done by some specific rules called rasterization rule. The rasterization stage converts vector information composed of shapes or primitives into raster image which is composed of pixels for the purpose of displaying real-time 3D graphics on to the screen. During this stage each primitive is converted into pixels, while interpolating per-vertex values across each primitive[4]. It also includes clipping vertices to the view frustum by performing a dived by z to provide perspective, mapping primitives to a 2D viewport. It also used to determine how to invoke the pixel shader. In general the rasterization stage always performs clipping, a perspective divide to transform

the points into homogeneous space, and maps the vertices to the viewport[4]. The render target into progressively smaller pieces, ultimately down to pixels (hierarchal rasterization or depth test). Each attributes of each triangle vertex such as the x, y, z location, texture coordinates and color needs to be interpolated at each pixel similar to the interpolation at the vertex in the primitives.
The main functions of the rasterizer can be given as:

- Clipping
- View Transform
- Projection transform
- Conversion of vector information (primitives) to raster image(pixels) i.e. rasterization rules.

Clipping

The objects falling between the far clipping plane and near clipping plane are rendered others are clipped in the rasterization stage. The viewing frustum is shown in figure 2.8.

Rasterization rule

In Direct3D normally Triangle rasterization rule which uses top-left rule for converting vertices into pixels[4]. The rule is explained below.
Its quite common that the points specified by the vertices may not match exactly to the pixels on the screen. At this time the Direct3D applies rasterization rule to find which all pixels will be shaded. The figure 2.9 shows the rectangle whose upper left corner is at (0,0) and the bottom right corner is at (5,5). In the top-left filling convention, top refers to the vertical location of horizontal spans, and left refers to the horizontal location of pixels within a span. The next figure shows the right egde and top edge of a triangle in the rectangle.

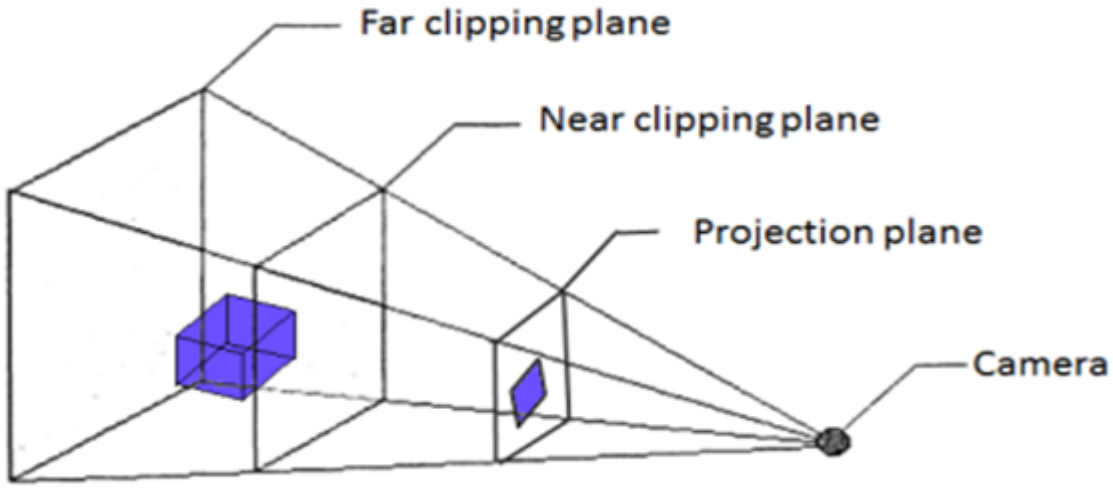


Figure 2.8: Viewing frustum.

The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The figure 2.10 shows two triangles, one at (0, 0), (5, 0), and (5, 5), and the other at (0, 5), (0, 0), and (5, 5). The first triangle in this case gets 15 pixels (shown in black), whereas the second gets only 10 pixels (shown in gray) because the shared edge is the left edge of the first triangle.

If you define a rectangle with its upper-left corner at (0.5, 0.5) and its lower-right corner at (2.5, 4.5), the center point of this rectangle is at (1.5, 2.5). When the Direct3D rasterizer tessellates this rectangle, the center of each pixel is unambiguously inside each of the four triangles, and the top-left filling convention is not needed. The following illustration shows this. The pixels in the rectangle are labeled according to the triangle in which Direct3D includes them. If you move the rectangle in the preceding illustration so that its upper-left corner is at (1.0, 1.0), its lower-right corner at (3.0, 5.0), and its center point at (2.0, 3.0), Direct3D applies the top-left filling convention. Most pixels in this rectangle straddle the border between two or more triangles shown in the figure 2.11. For both rectangles, the same pixels are affected, as shown in the illustration.

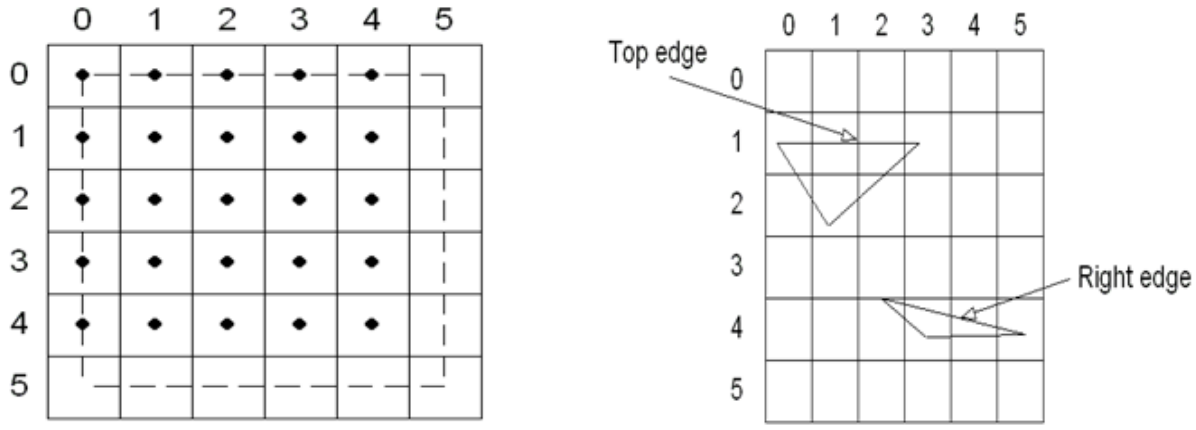


Figure 2.9: Top-left rasterization rule [3].

Table II: Triangle Interpolation [2].

Shading mode	**Description**
Flat	Only the fog factor is interpolated in flat shade mode. For all other interpolated values, the color of the first vertex in the triangle is applied across the entire face.
Gouraud	Linear interpolation is performed between all three vertices.

2.3.7 Output merger

The pixel output stage writes the pixel to the "render target" with an optional blend with previous value in render target. But might not write the pixel at all if it fails the tests like, 1. Is the pixel behind another pixel? (depth testing) and 2. Is the pixel marked not to be written? (alpha testing). Finally, writes them to frame buffer.

2.4 OGL rendering pipeline

The figure 2.12 describes the OpenGL rendering pipeline and figure 2.13 shows the mapping of Direct3D to OGL pipeline.

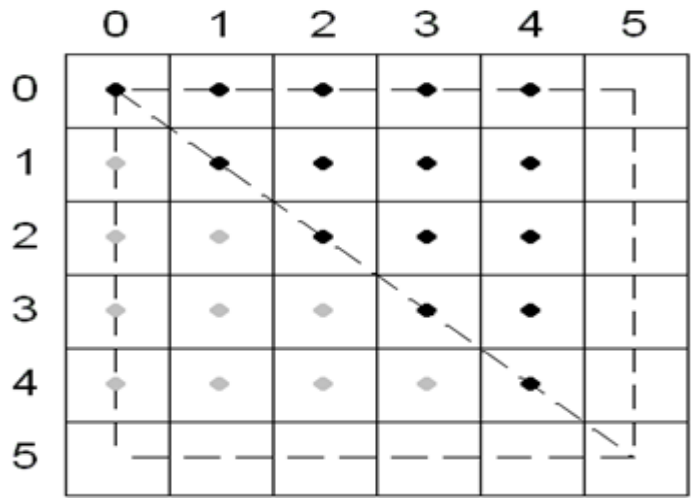


Figure 2.10: Top-left rasterization rule - lit and unlit pixels.

2.5 An overview of low power designs

The amount of power that a chip dissipates per unit area is called its power density, and there are two types of power density digital design 1) dynamic power density and 2) static power density.

Dynamic Power Density

Each and every transistor on a chip dissipates a small amount of power when it is switched, and transistors that are switched rapidly dissipate more power than transistors that are switched slowly. The total amount of power dissipated per unit area due to switching of a chip's transistors is called dynamic power density. There are two factors that work together to cause an increase in dynamic power density they are 1) clockspeed and 2) transistor density.

Increasing a processor's clockspeed involves switching its transistors more rapidly, and as mentioned previously, transistors that are switched more rapidly dissipate more power. Therefore, as a processor's clockspeed rises, its dynamic power density will also increase, because each of those rapidly switching transistors contributes more to the device's total power dissipation.

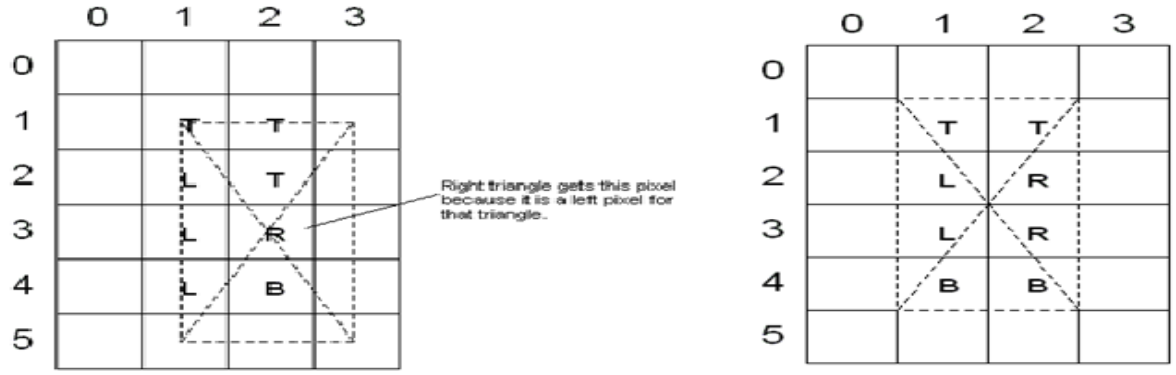


Figure 2.11: Top-left rasterization rule - lit and unlit pixels of a ractangle.

Static power density

In addition to clockspeed-related increases in dynamic power density, chip designers must also contend with the fact that even transistors that aren't switching will still leak current during idle periods, much like how a faucet that is shut off can still leak water if the water pressure behind it is high enough [5]. This leakage current causes an idle transistor to constantly dissipate a noticeable amount of power. The amount of power dissipated per unit area due to leakage current is called static power density[5].

Transistors leak more current as they get smaller, and consequently static power densities begin to rise across the chip when more transistors are placed into the same amount of space. Thus even with relatively low clockspeed devices with very small transistor sizes are still subject to increases in power density if leakage current is not controlled. If a silicon device's overall power density gets high enough, it will begin to overheat and will eventually fail entirely. Thus it's critical that designers of highly integrated devices like modern x86 processors take power efficiency into account when designing a new microarchitecture [5].

To reduce the leakage power following techniques can be used:

1. Reduce the voltage on the transistors.

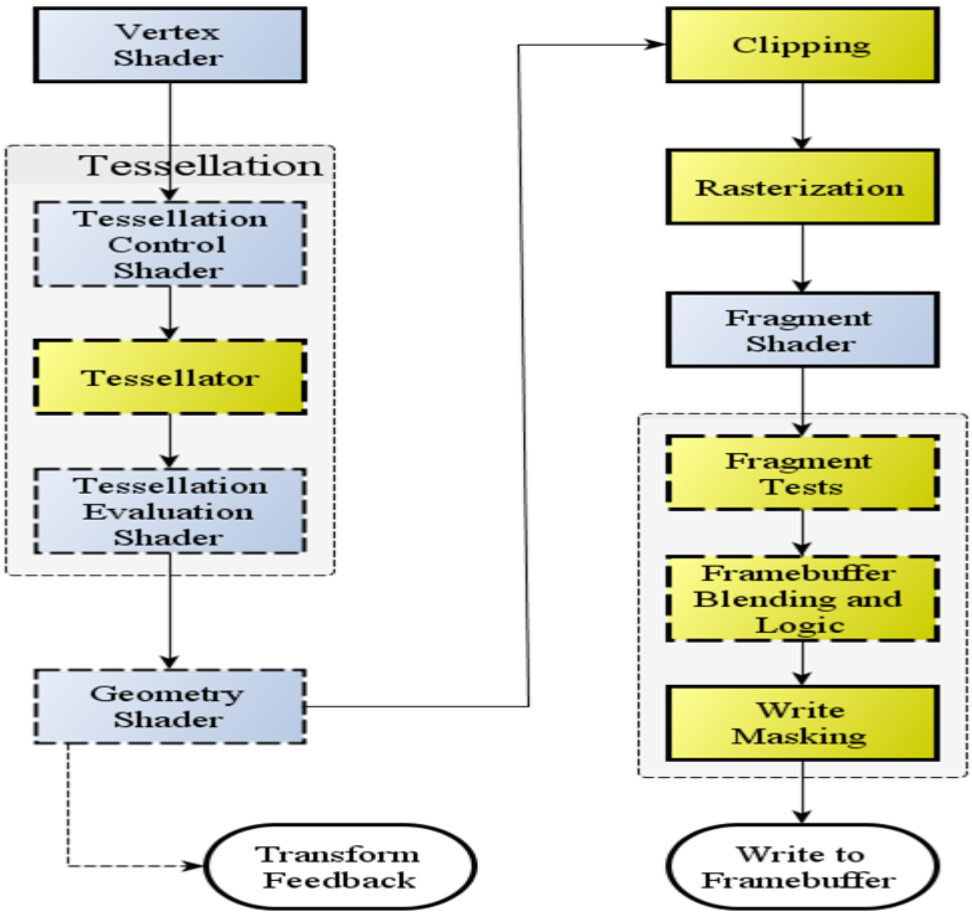


Figure 2.12: OGL (OpenGL) rendering pipeline.

2. Reduce the number of leakage transistors.

3. Reduce the load on the transistors.

In order to reduce the switching power following techniques can be used:

1. Reduce the load on the capacitor.

2. Scale down the switching frequency.

3. Reduce the voltage on the transistors.

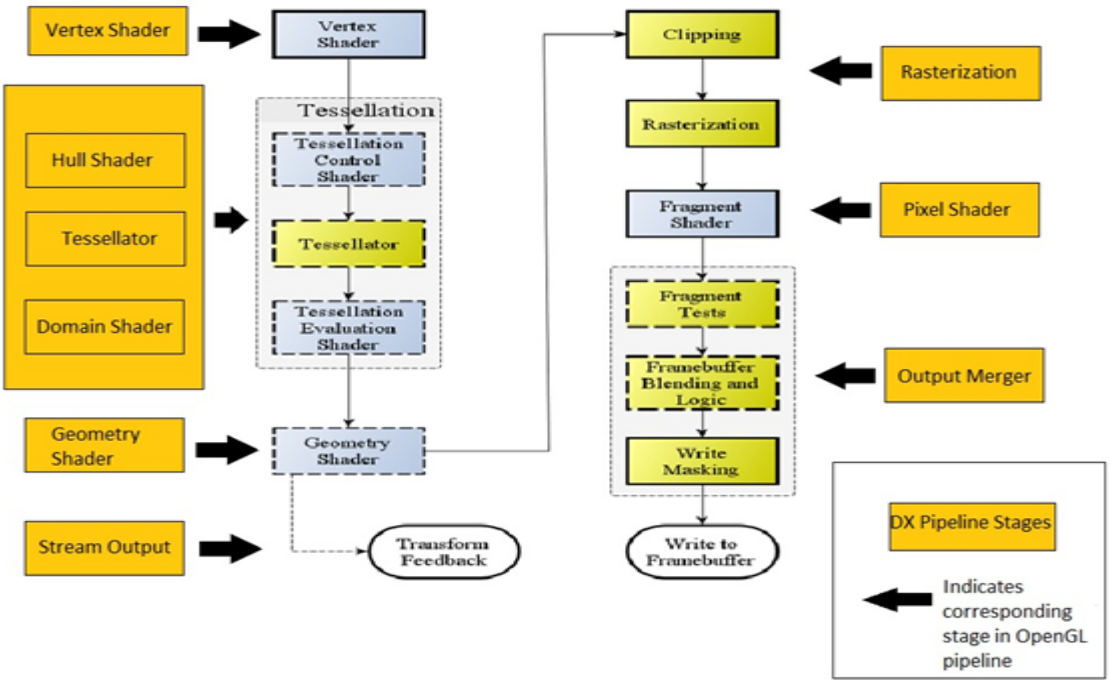


Figure 2.13: Mapping of D3D X pipeline to OGL rendering pipeline stages.

The table-3 below shows different strategies available for optimizing the power consumption:

Effective power management is possible by using the different strategies at various levels in VLSI Design process. So designers need an intelligent approach for optimizing power consumptions in designs.

The table-4 demonstrates some low-power techniques used fro power reduction.

2.6 Summary

An introduction to low power designs and the trend in low power segment was covered in section 2.1. Basic terminologies in the computer graphics were discussed in section

Table III: Strategies to reduce power at various levels.

Design Level	**Strategies**
Operating System Level	Portioning, Power down
Software level	Regularity, locality, concurrency
Architecture level	Pipelining, Redundancy, data encoding
Circuit /Logic level	Logic styles, transistor sizing and energy recovery
Technology Level	Threshold reduction, multi threshold devices

Table IV: Low-power techniques.

Traditional Techniques	**Dynamic Power Reduction**	**Leakage power reduction**	**Other Power reduction Techniques**
Clock Gating	Clock Gating	Minimize usage of low Vt cells	Multi Oxide devices
Power Gating	Power Efficient Techniques	Power Gating	Minimize capacitance by custom design
Variable Frequency	Variable Frequency	Back Biasing	Power efficient circuits
Variable Voltage Supply	Variable Voltage Supply	Reduce Oxide Thickness	
Variable Device Threshold	Variable Island	Use Fin FET	

2.2. A detailed description of Direct3D X was covered in section 2.3. In section 2.4 OGL rendering pipeline and its mapping to Direct3D X stages were explained. An overview of the low power designs were discussed in section 2.5.

Chapter 3

Low Power Design Techniques

3.1 Introduction

Power is emerging as the most critical issue in system-on-chip (SoC) design today. Power management is becoming an increasingly urgent problem in almost every category of design, as power densitymeasured in watts per square millimeter rises at a very fast rate. From a chip-engineering perspective, effective energy management for a SoC must be built into the design starting at the architecture stage; and low-power techniques need to be employed at every stage of the design, from RTL to GDSII [6]. Fred Pollack of Intel first noted a rather alarming trend in his keynote at MICRO-32 in 1999 [1] . He made the now well-known observation that power density is increasing at an alarming rate, approaching that of the hottest man-made objects on the planet, and graphed power density as shown in Figure 3.1 [6].

The design efforts in managing power are rising due to the necessity to design for low-power as well as for performance and costs. This has consequence for engineering productivity, as it impacts schedules and risk. Power management is a must for all designs of 90nm and below [5]. At smaller geometries, aggressive management of leakage current can greatly impact design and implementation and also for some designs and libraries, leakage current exceeds switching currents, thus becoming the

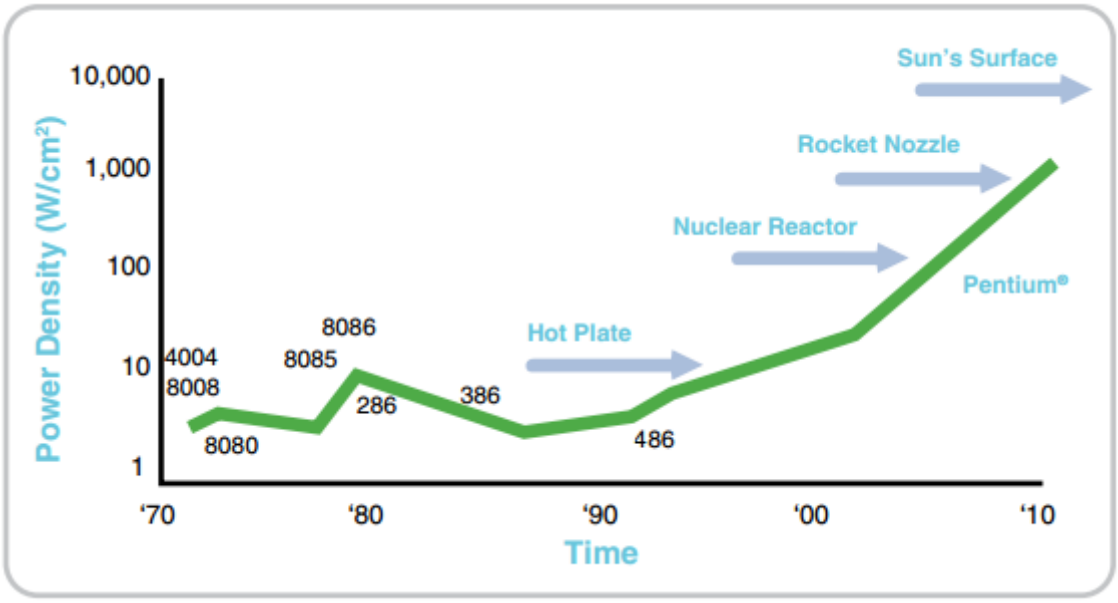


Figure 3.1: Fred Pollack's (Intel) observation on power density[1].

primary source of power dissipation in CMOS technology, as shown in Figure 3.2. Recently, designers were primarily concerned with improving the performance of their designs (throughput, latency, frequency), and reducing silicon area to lower manufacturing costs but now power is replacing performance as the key competitive metric for SoC design. These power challenges affect almost all SoC designs [6].

3.2 Power saving design techniques

3.2.1 Clock gating

Clock gating is a popular technique used in many synchronous circuits for reducing dynamic power dissipation[6]. Clock gating saves power by adding more logic to a circuit on the top of clock tree. Pruning the clock disables portions of the circuitry so that the flip-flops in them do not have to switch states or toggle states. Toggling states consumes power. When not being switched, the switching power consumption

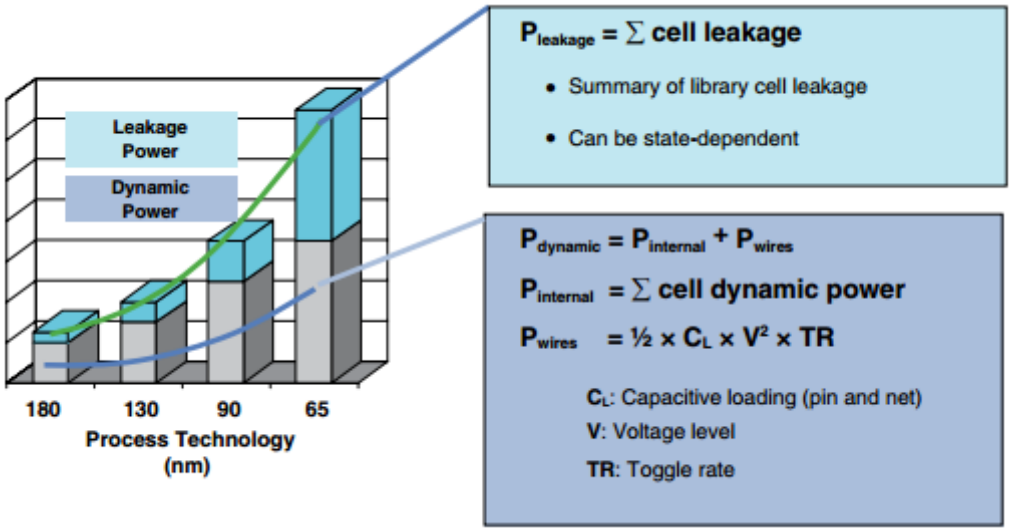


Figure 3.2: Leakage and Dynamic power in various process technology [6].

goes to zero, and only leakage power are incurred.

Clock gating works by taking the enable conditions attached to registers, and uses them to gate the clocks[6]. Therefore it is imperative that a design must contain these enable conditions in order to use and benefit from clock gating [6]. This clock gating process can also save significant die area as well as power, because it removes large numbers of muxes and replaces them with clock gating logic. This clock gating logic is generally in the form of "Integrated clock gating" (ICG) cells. However, note that the clock gating logic will change the clock tree structure, as clock gating logic will sit in the clock tree.

Clock gating logic can be added into a design in a variety of ways: [6]

- Coded into the RTL code as enable conditions that can be automatically translated into clock gating logic by synthesis tools (fine grain clock gating).
- Inserted into the design manually by the RTL designers (typically as module level clock gating) by instantiating library specific ICG (Integrated Clock Gating) cells to gate the clocks of specific modules or registers.

- Semi-automatically inserted into the RTL by automated clock gating tools. These tools either insert ICG cells into the RTL, or add enable conditions into the RTL code. These typically also offer sequential clock gating optimizations.

The figure 3.3 shows the simple implementation of clock gating. Clock consumes 60-70 percent of total chip power and is expected to significantly increase in the next generation of designs at 45nm and below[6]. This is because of the fact that power is directly proportional to voltage and the frequency of the clock as shown in the following equation:
Power = Capacitance * (Voltage) 2 * (Frequency)

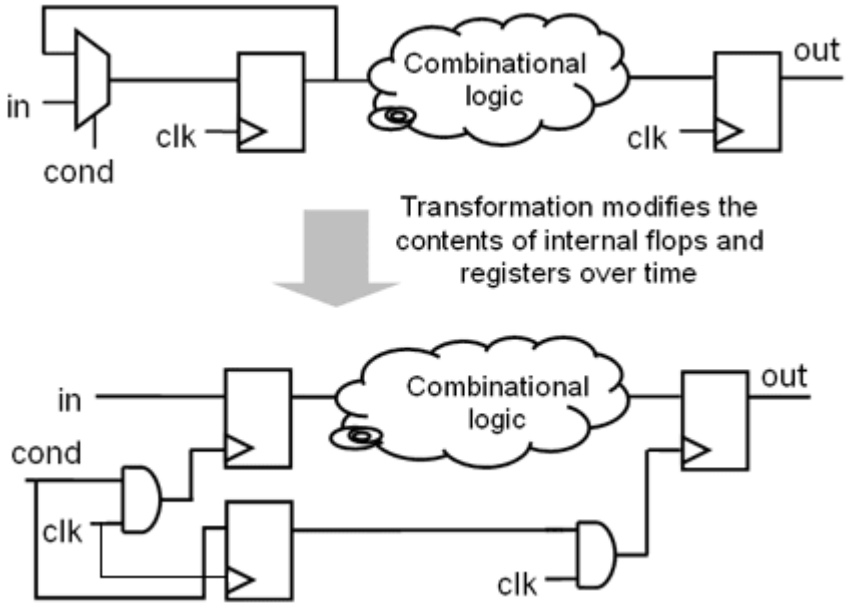


Figure 3.3: A simple clock gating implementation.

Thus, reducing clock power is very important. Clock gating is a key power reduction technique used by many designers and is typically implemented by gate-level power synthesis tools. The figure 3.4 explain the basic advantage of the clock gating.

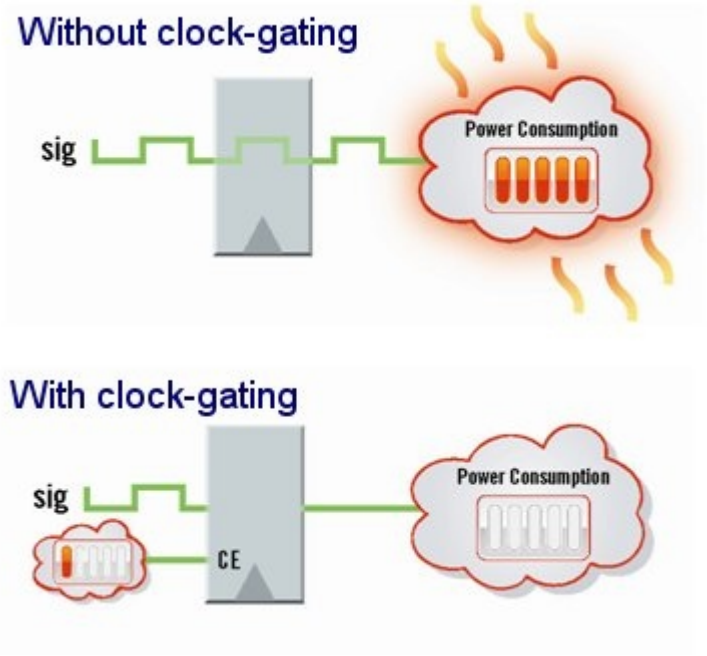


Figure 3.4: Advantage of clock gating.

3.2.2 Active stall

When a module is capable of pushing data in the downstream unit, but there is a stall from that unit, gate the clock of the instantiated clock gate FUB (Functional unit block).

Explanation

The following example is applicable only for the logic which dont use bubble collapse of pipeline stages. When a unit is capable of sending data out but there is a stall from the downstream unit, the design makes sure that the data is held in register. If the width of the register is at least 8, then the power compiler uses this as a local gated clock. But as the main clock gate FUB is still active, there will be clock toggling between the clock gate FUB and the local clock gate AND gate. Consider the re-circulating flop structure of Hold signal is used to hold the data in a register as shown in figure 3.5.

Power compiler will implement the structure as in figure 3.6 where hold is used to

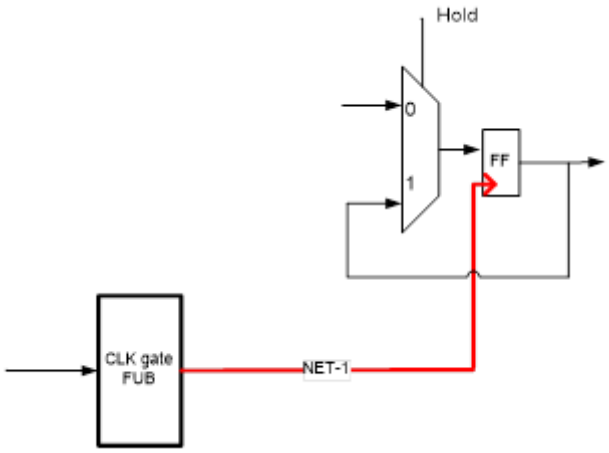


Figure 3.5: Reciculating flop structure.

gate the clock. But the NET-1 will still toggle. So its better to gate the clock of the clock gate FUB as in figure 3.7 instead of gating a local register.

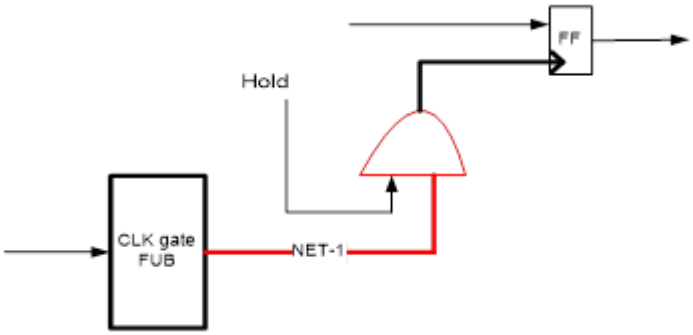


Figure 3.6: Power compiler gated clock for re-circulating flop.

Benefits

There will be no clock toggle in nets between the main clock gate FUB and the clock gating AND gates.

Drawbacks

If the hold signal is coming from the different partition, then the delay in the hold

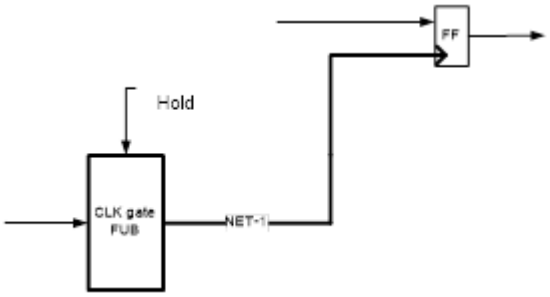


Figure 3.7: Main clock gate fub gated during HOLD.

path will come in the clock enable signal.

Recommendation

Ideally the Hold signal should be removed from the local re-circulating structure and used only in the clock gate FUB. But as the legacy has hold in the select of the re-circulating structure, rather than removing this, we can just add this to the clock gate FUB. If the hold is coming from different partition, then its recommended to change it to credit based mechanism.

3.2.3 Operation rearrangement

Place the high toggling signal towards the end of the logic cone and low toggling signals towards the beginning of the logic cone.

Explanation

In the circuit shown in 3.8 A* is very high toggling signal compared to other signals. This is making most of the logic cone to toggle depending on the select signals.

The placement of the addition with A should be towards the end of the cone so that toggling of A* will cause lesser toggle to the entire logic cone. The circuit is shown below in figure 3.9.

Benefits

As the high toggling signal is placed at the end of the logic cone, the excess toggling is not causing the remaining cone to toggle, and hence power saving.

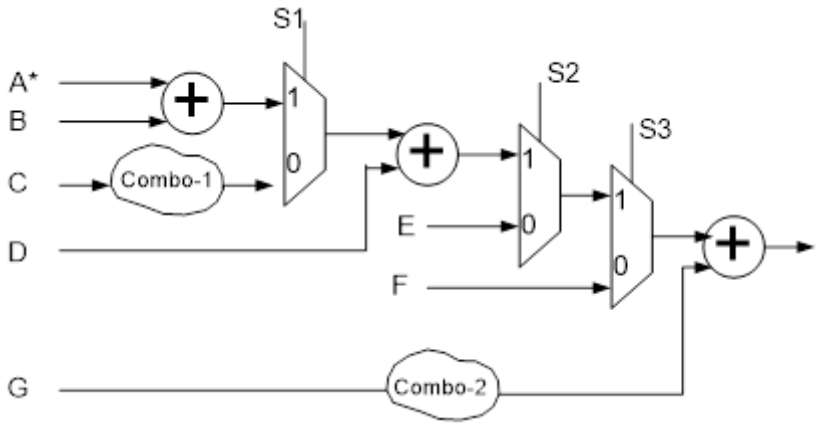


Figure 3.8: Logic rearrangement; power inefficient circuit.

Drawbacks

There can be timing impact due to additional MUX and adder and extra power consumption due the extra MUX.

Recommendation

This method is to be adopted if the affected logic cone is large and the path is not already in timing critical.

3.2.4 Operand isolation/Data gating.

Identify redundant computations of datapath components and isolate them using specific circuitry.

Explanation

Data gating is very commonly used technique where the data/control is forced to some stable value when the logic cone is not in use. This is applied whenever the data cant be made stable using clock gating/flop data gating. But the Operand isolation is a typical type of data gating where a component in a logic cone is isolated by forcing its operands to a stable value wherein the other components of the cone are still functional.

In Figure 3.10, when the MUX select is low, the multiplier output is unused. As the

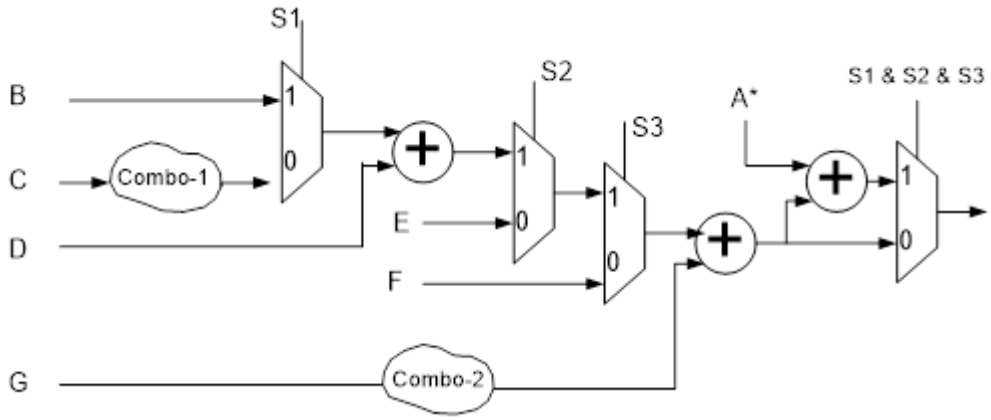


Figure 3.9: Logic rearrangement; logic rearrangement for power inefficiency.

reg-B cant be gated, the multiplier will continuously consume power.

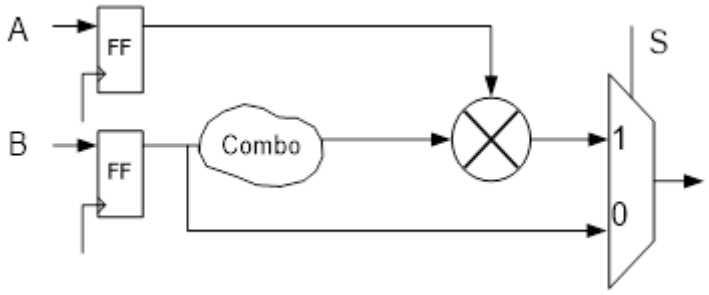


Figure 3.10: Operand isolation; power inefficient circuit.

The select signal should be used to ZERO down the input to the multiplier so that multiplier will not consume power. At the same time, the toggle rate of the MUX must also be considered. If the MUX select is toggling at high rate, then the multiplier input might toggle between 0 and non-0 values at higher than the previous toggle rate.

Benefits

No power consumption in the unused computation unit.

Drawbacks

Extra gate count due to the input blocking AND gates and there can be timing impact

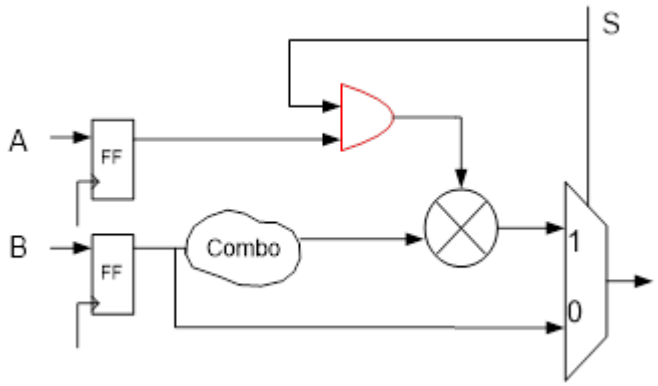


Figure 3.11: Operand isolation; multiplier isolated circuit.

due to additional MUX and adder.

Recommendation

The selection of the blocking input should be such that the load in that input is lower. This will reduce the timing impact due to additional layer of logic. Again, if the toggle rate of the MUX must is high such that the multiplier input might toggle between 0 and non-0 values at higher than the previous toggle rate, this method shouldnt be used.

3.2.5 Common case separation

Identify the most common application which uses a part of the shared logic and implement that using smaller gate count. In these entire common usage mode, clock down the bigger shared logic.

Explanation

Logic A in the figure 3.12 is a shared function. Logic B is a common case behavior of the shared logic which is implemented separately. The input of the shared logic in clock gated while its in common usage mode.

Benefits

In most common usage mode, the smaller logic will consume dynamic power.

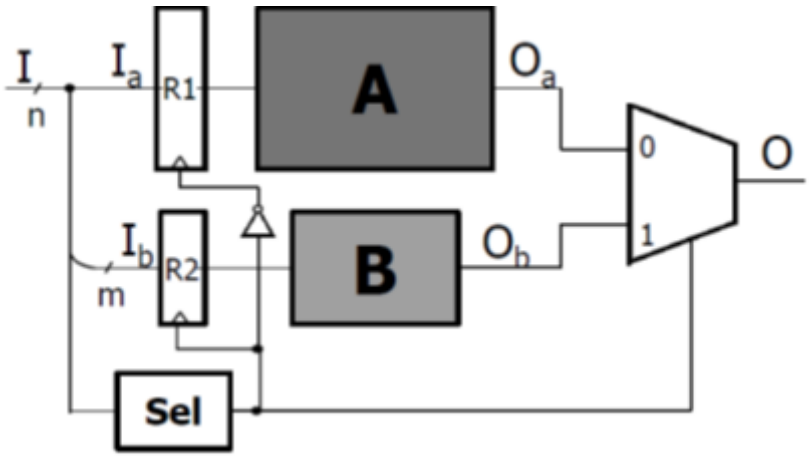


Figure 3.12: Common case separation. A: bigger shared logic and B: common usage logic.

Drawbacks

Additional gate count due to duplicate logic. Continuous leakage power due to this. If the operations are not isolated by registers, then isolation may need forcing input to stable value. This adds logic level and may cause timing impact.

Recommendation

The leakage contribution of the duplicate logic must be considered. If the common case usage is above 50% and the additional gate count is less than 50%, this technique can be used.

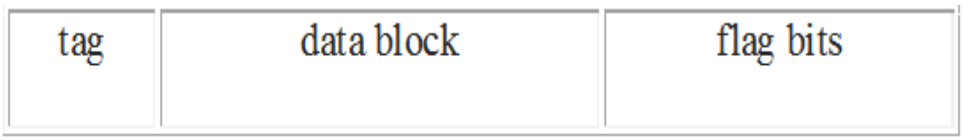
3.3 Cache memory

A CPU cache is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory [7]. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations [7]. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2 etc.) [7]. L3 cache is specifically used only for GPU

processing. Data is transferred between memory and cache are done in blocks of fixed size, called cache lines. When a cache line is copied from memory into the cache, a cache entry will be created. The cache entry will include the copied data as well as the requested memory location called as a tag.

When the processor needs to read or write a location in main memory, it will first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in all the cache lines. If the processor finds that the memory location is in the cache, a cache hit has occurred else cache miss has occurred. In the case of: [7]

- A cache hit, the processor immediately reads or writes the data in the cache line
- A cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.
- Cache row entries usually have the following structure:



- The data block (cache line) contains the actual data fetched from the main memory. The tag contains (part of) the address of the actual data fetched from the main memory. The flag bits are discussed below.
- The "size" of the cache is the amount of main memory data it can hold. This size can be calculated as the number of bytes stored in each data block times the number of blocks stored in the cache. (The number of tag and flag bits is irrelevant to this calculation, although it does affect the physical area of a cache.)

There are three major types of cache memory architectures available in literature.They are as follow:

1. Associative cache
2. Direct mapped cache
3. Set associative cache

3.3.1 Associative cache

A cache where data from any address can be stored in any cache location [7]. The whole address must be used as the tag. All tags must be compared simultaneously (associatively) with the requested address and if one matches then its associated data is accessed [7]. This requires an associative memory to hold the tags which makes this form of cache more expensive.
The figure 3.13 shows the intenal block diagram of associative cache memory.
As shown in figure 3.13 the tag value is divided into 2 parts one part containing block number and other part contains the byte. Say, for example we consider tag value to be of 16 bit then, 13 bit will represent the block number and remaining 3 bit will indicate the byte number. There is an argument register which will hold this value as shown in figure 3.13. The m-bit here indicates the match bit, it will be set if the tag values are matching. The v-bit indicates the existence of the valid tag in the corresponding location. The and gate is used to select the location of the tag in the entire tag location and only one tag location will contain both m-bit and v-bit as one. Thus the output of and gate will be high, which will be fed to the enable pin of the tri-gate buffer, as shown in the figure 3.13. The enabled buffer will pass the corresponding cache-line to the mux. The byte number is used as the select input of the mux. The output of the mux is passed to the CPU.

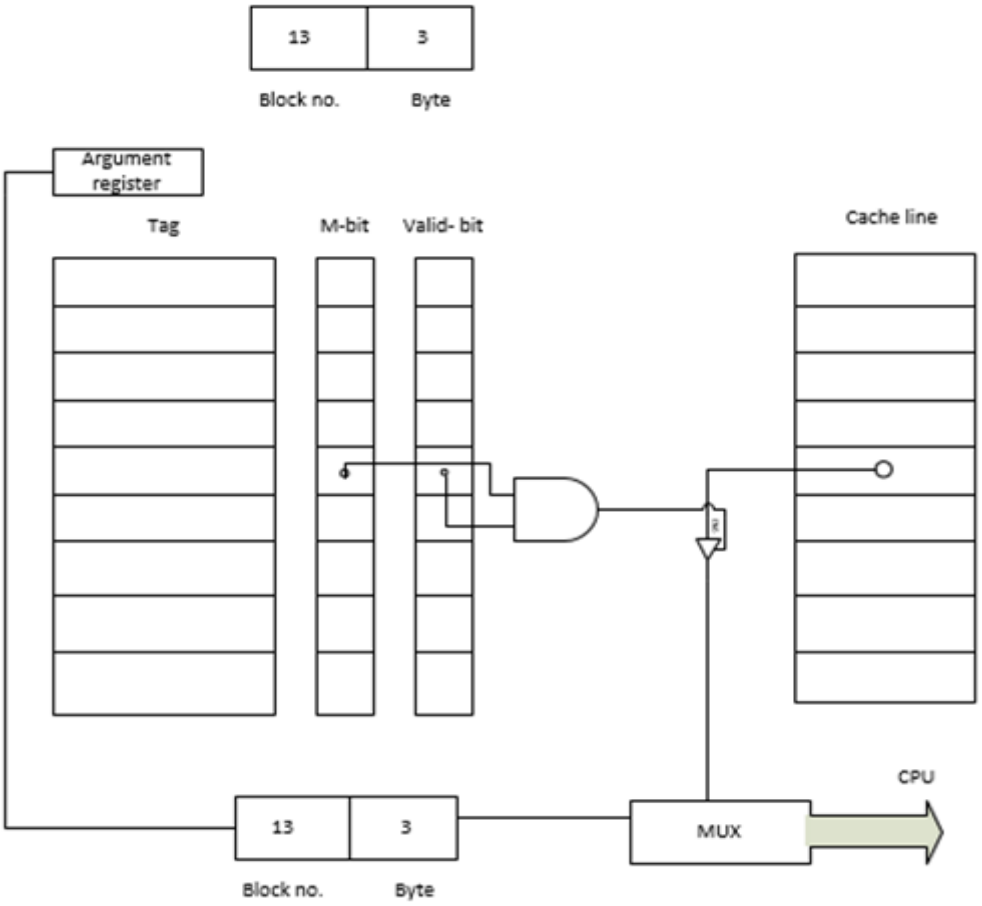


Figure 3.13: Fully associative cache.

3.3.2 Direct mapped cache

In this cache organization, each location in main memory can only go in one entry in the cache [7]. Therefore, a direct-mapped cache can also be called a "one-way set associative" cache. It does not have a replacement policy as such, since there is no choice of which cache entry's contents to evict. This means that if two locations map to the same entry, they may continually knock each other out [7]. Although simpler, a direct-mapped cache needs to be much larger than an associative one to give comparable performance, and it is more unpredictable.

The figure 3.14 shows the organization of the direct mapped cache. Here each main

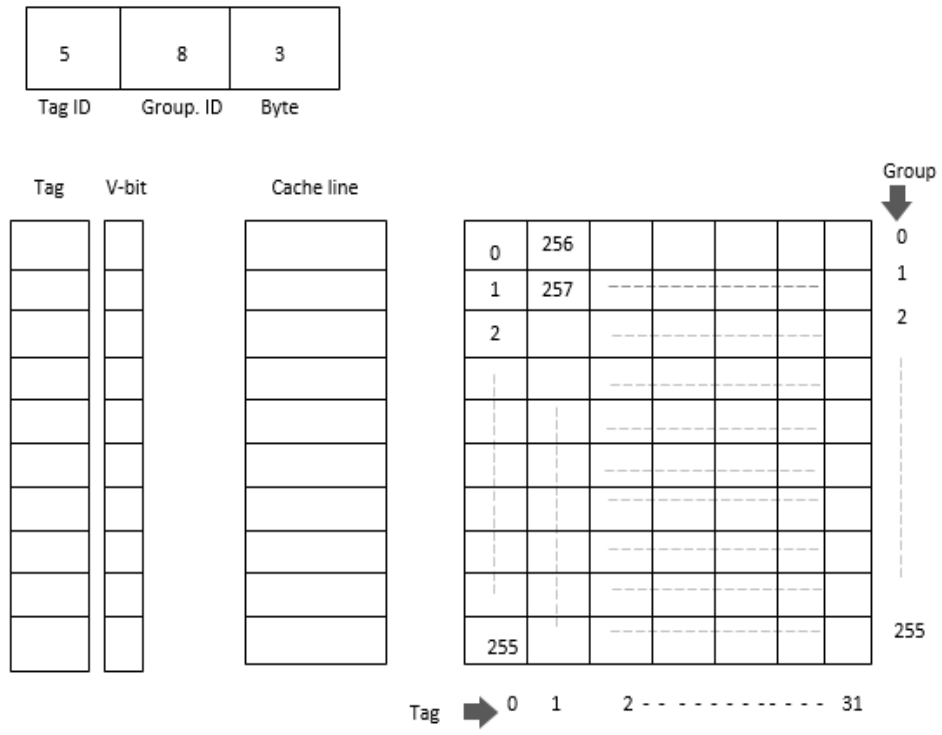


Figure 3.14: Direct mapped cache organization.

memory location can go to only one cache line based on the table shown above. The tag value is divided into 3 parts tag ID, Group ID and byte. The tag ID and group ID are used to select the corresponding cache line as shown in the figure 3.14. The figure 3.15 shows the data flow in the direct mapped cache, here a decoder is used to decode the group ID and it is used along with v-bit to select the correct tag value from the tag locations. The selected tag is compared with the available tag and if they match then it is said to be cache hit else cache miss. If the cache hit occurred then the corresponding data from the cache line will be selected.

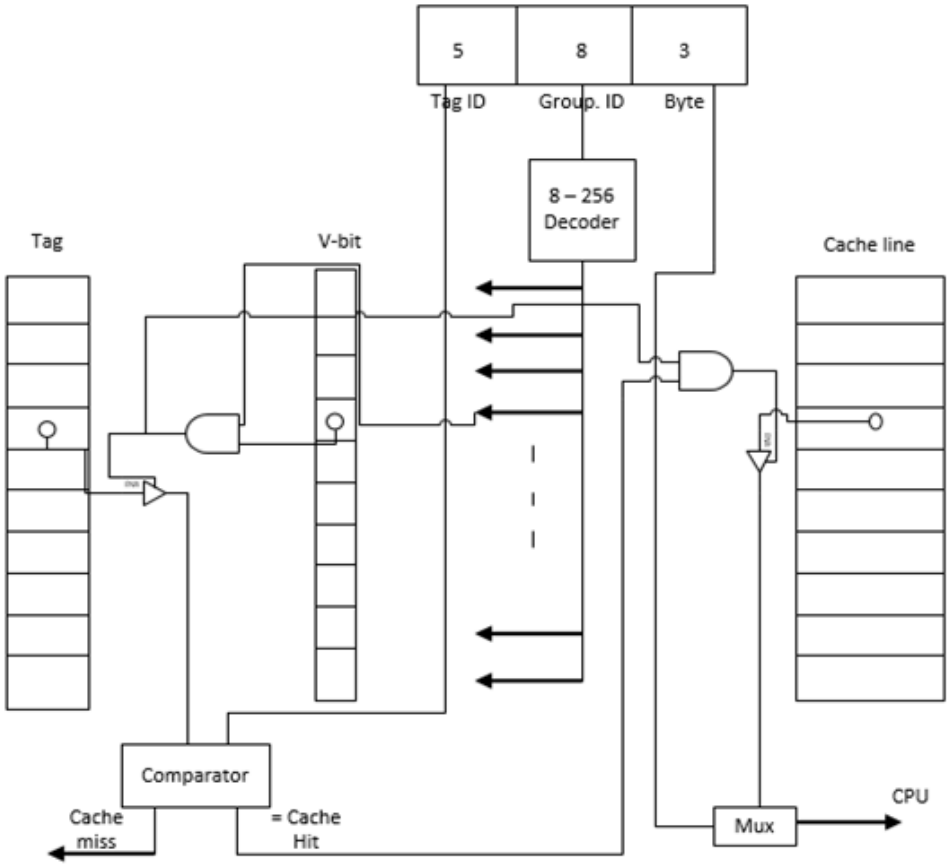


Figure 3.15: Direct mapped cache.

3.3.3 Set associative cache

In set associative the major focus will be on 2 way set associative cache. If each location in main memory can be cached in either of two locations in the cache, one logical question is: which one of the two?. The simplest and most commonly used scheme, shown in figure 3.16, is to use the least significant bits of the memory location's index as the index for the cache memory, and to have two entries for each index [7]. One benefit of this scheme is that the tags stored in the cache do not have to include that part of the main memory address which is implied by the cache memory's index [7]. Since the cache tags have fewer bits, they require fewer transistors, take less space

on the processor circuit board or on the microprocessor chip, and can be read and compared faster [7].

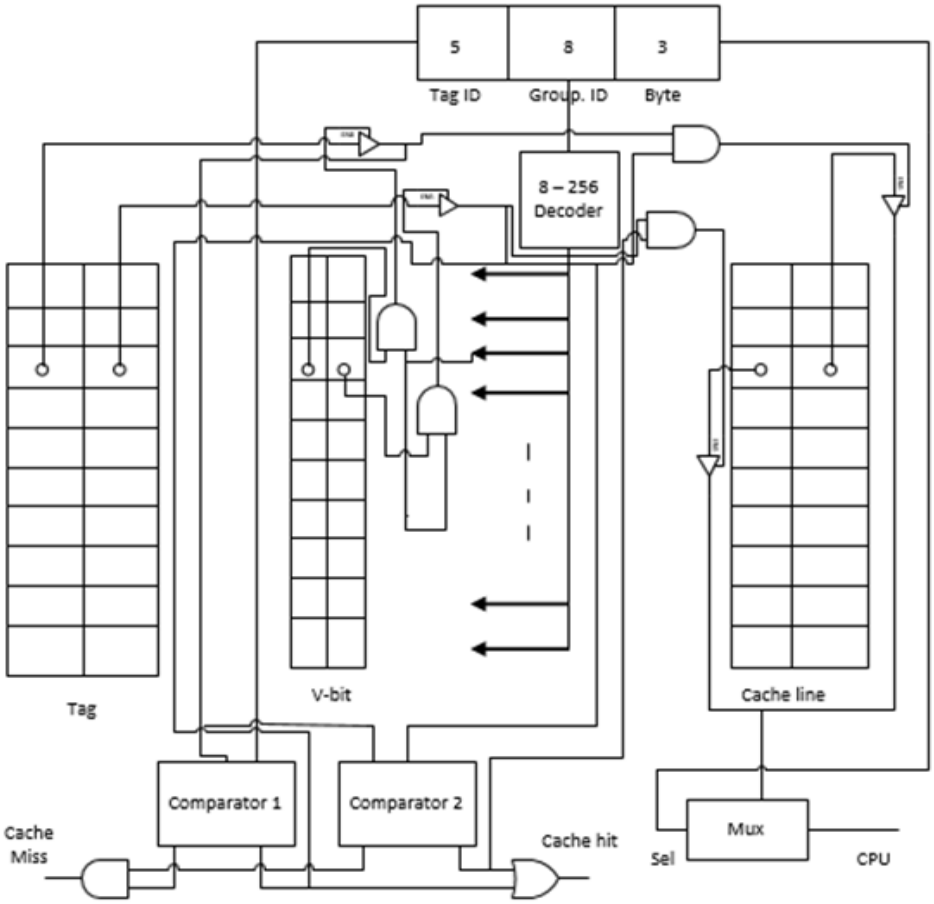


Figure 3.16: Two way set associative cache.

The figure 3.16 shows the 2 way set associative cache memory organization. This method is similar to that of direct mapped but has two fields for every main memory location. Thus it is faster but consumes more hardware. There are two v-bit each one corresponds to one cache line, the decoder output is given to two and gates whose other inputs are two different v-bit. The output of and gate is used to select the corresponding tag locations and compared in two different comparators. The selected tag is compared with the available tag and if they match then it is said to be cache

hit else cache miss. If the cache hit occurred then the corresponding data from the cache line will be selected.

3.4 Content Addressable Memory (CAM)

A CAM is a memory that implements the lookup-table function in a single clock cycle using dedicated comparison circuitry [8]. CAMs are especially popular in network routers for packet forwarding and packet classification, but they are also beneficial in a variety of other applications that require high-speed table lookup like RAM. The main CAM-design challenge is to reduce power consumption associated with the large amount of parallel active circuitry, without sacrificing speed or memory density [8]. However, the speed of a CAM comes at the cost of increased silicon area and power consumption, two design parameters that designers strive to reduce [5]. As CAM applications grow, demanding larger CAM sizes, the power problem further increases. Reducing power consumption, without sacrificing speed or area, is the main thread of recent researching large-capacity CAMs [8].

Figure 3.17 shows a simplified block diagram of a CAM. The input to the system is the search word that is broadcast on to the search lines to the table of stored data. The number of bits in a CAM word is usually large, with existing implementations ranging from 36to 144bits [8]. A typical CAM employs a table size ranging between a few hundred entries to 32K entries, corresponding to an address space ranging from 7 bits to 15 bits. Each stored word has a match-line that indicates whether the search word and stored word are identical (the match case) or not (a mismatch case, or miss). The match-lines are fed to an encoder that generates a binary match location corresponding to the match-line that is in the match state. An encoder is used in systems where only a single match is expected like ram. In CAM applications where more than one word match is possible, a priority encoder is used instead of a simple encoder. A priority encoder selects the highest priority matching location to map to the match result, with words in lower address locations will receive higher priority.

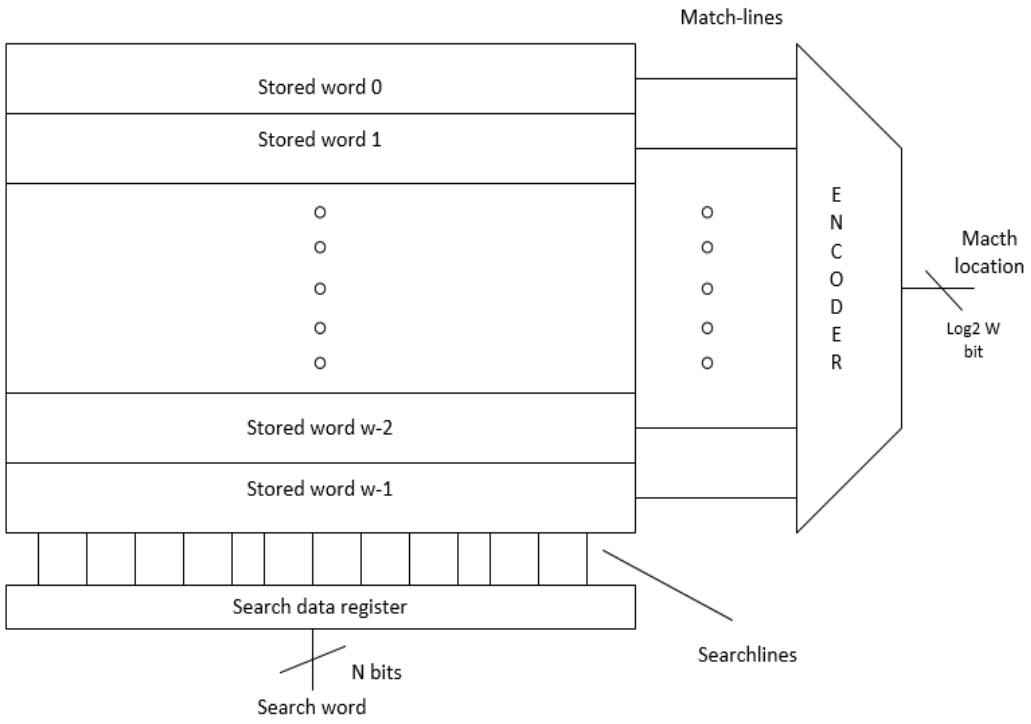


Figure 3.17: Simplified block diagram of CAM.

In addition, there is often a hit signal which is not shown in the figure that flags the case in which there is no matching location in the CAM. The overall function of a CAM is to take a search word and return the matching memory location [8].

The operation of a CAM is like that of the tag portion of a fully associative cache. The tag portion of a cache compares its input, which is an address, to all addresses stored in the tag memory. In the case of match, a single match line goes high, indicating the location of a match. Unlike CAMs, caches do not use priority encoders since only a single match occurs; instead, the match- line directly activates a read of the data portion of the cache associated with the matching tag. Many circuits are common to both CAMs and cache.

3.5 Hardware for binary search algorithm

The binary search or half interval search algorithm finds the position of a specified input value within the array that has sorted values. For example, L = 1, 3, 4, 6, 8, 9, 11 and K=4.

The steps of the algorithm are as follow:

Step1=> select l = 1 and h = 11

Step2=> find i by adding l and h and dividing it by 2,

i = (l+h)/2 = (1+11)/2 = 6 (if not an integer, then round to nearest integer)

Step 3 => Now, i and K are compared

If K > i , then keep the value of h as it is i.e. h = 11 and update the value of l = i. Now got back to step 2.

If K< i, then keep the value of l as it is i.e. l = 1 and update the value of h = i. Now got back to step 2.

If K = i then the key is found from the array. This is shown out.

The figure 3.18 shows the hardware for binary search algorithm [9]

The multiplexers in the design are used to load the initial values in the circuit, the sel input is used to select the value to be loaded or to update. As shown in the algorithm, the initial values are loaded which can be done using two multiplexers one for h and one for l. The other input of the multiplexers are used to update the new value as mentioned in the step 3 of the algorithm. The D flip-flop is used to store the value and then passed to the adder. The added value is divided by 2 and value of i is obtained as mentioned in the step 2 of the algorithm. This is used in the cam part of the RAM (random access memory) to find the tag value.

The main idea behind this hardware implementation is to optimize the number of gates used in the CAM circuit. We assumed that the date are sorted in either increasing order or decreasing order throughout the implementation. The encoder circuit used in the cam circuit is generally very big, say for example 1024 to 10 which will have known amount of propagation delay, by using this binary search we can reduce

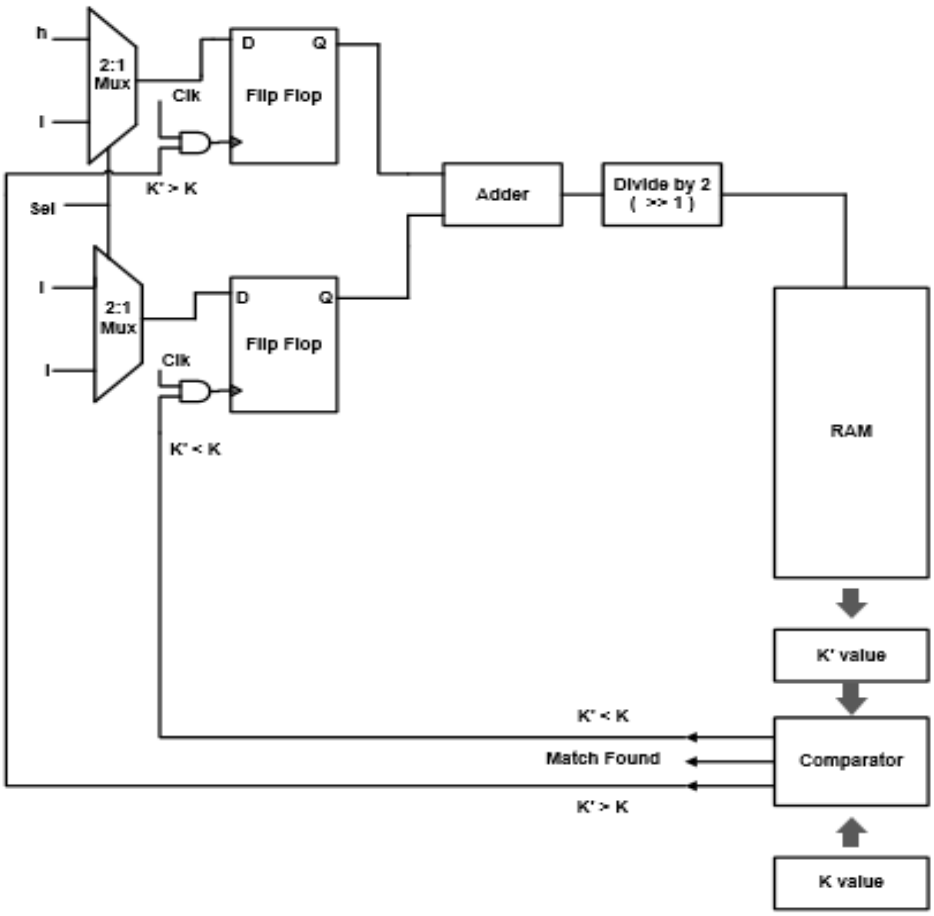


Figure 3.18: Hardware for binary search algorithm

the propagation delay because we will be needing only 512 to 9 bit encoder. Thus the propagation delay will be less compared to the previous case. In this case we divide the encoder circuit in 2 parts say 1024 to 10 bit encoder will be replaced by the two 512 to 9 bit encoder. This seems to be increase in the hardware but at any given time any one of the two encoder will be active thus there will be significant reduction in the power consumption and improvement in the delay of the circuit.

3.6 Summary

In this chapter an explanation for different low power design techniques and their implementation was presented (section 3.2).The important parts of graphics hardware, like cache memory (section 3.3) and CAM (section 3.4) are also covered. A dedicated hardware for binary search algorithm and its implementation was explained (section 3.5).

Chapter 4

Phong's Illumination Model

4.1 Logarithm base2 circuit

4.1.1 Introduction

The floating point multiplication is one of the main operation performed by the CPU. In the graphics also it is used more often, but the problem associated with it is speed of the operation. To increase the speed the operation we propose a new hardware design for multiplication using a logarithm of base 2. Again the log base 2 implementation is one of the challenging task, for this we use piecewise linear approximation model. By this we convert all the input floating point numbers to fixed point log base2 number, and after this all the multiplication operations are carried out using addition. Since we use addition for our multiplication operation the speed increases and the gate count also reduces. One more advantage of this hardware is that even the floating point exponential operations can also be calculated by using a fixed point multiplication which is much less complex as compared to floating point exponent.

4.1.2 Floating point number

In general the floating point numbers are of 32 bit in IEEE 754 single precision format. Single-precision floating-point format is a computer number format that occupies 4

bytes (32 bits) in computer memory and represents a wide dynamic range of values by using a floating point [10].
The IEEE 754 standard specifies a binary32 as having:

- Sign bit: 1 bit
- Exponent width: 8 bits
- Significant precision: 24 bits (23 explicitly stored)

This gives from 6 to 9 significant decimal digits precision, if a decimal string with at most 6 significant decimal is converted to IEEE 754 single precision and then converted back to the same number of significant decimal, then the final string should match the original; and if an IEEE 754 single precision is converted to a decimal string with at least 9 significant decimal and then converted back to single, then the final number must match the original.
Sign bit determines the sign of the number, which is the sign of the significant as well. Exponent is either an 8 bit signed integer from 128 to 127 (2's Complement) or an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 binary32 definition [11]. If the unsigned integer format is used, the exponent value used in the arithmetic is the exponent shifted by a bias for the IEEE 754 binary32 case, an exponent value of 127 represents the actual zero (i.e. for 2e 127 to be one, e must be 127) [10].
The true significant includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zero. Thus only 23 fraction bits of the significant appear in the memory format but the total precision is 24 bits (equivalent to log10(224) 7.225 decimal digits). The bits are laid out as shown in the figure4.1. The logarithm base 2 hardware is implemented using piecewise linear approximation model. The base-2 logarithm of the unbiased FP input (1+m)2e-127 is computed as e+log2(1+m), where log2(1+m) $(1 \leq (1+m) < 2)$ is approximated using 5 linear intervals with

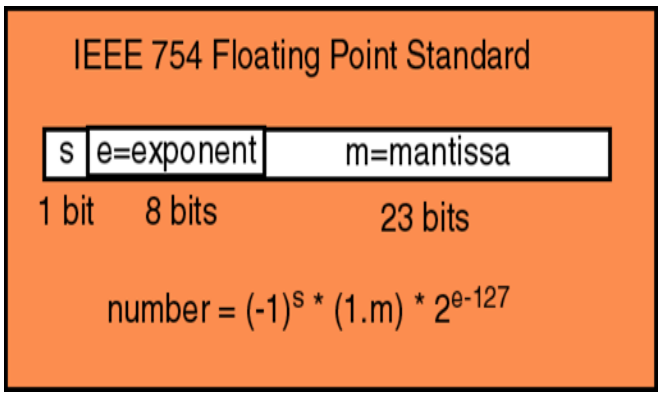


Figure 4.1: IEEE 754 -single precision format.

low Hamming weight coefficients and constants chosen for high accuracy and efficient hardware implementation [11]. The coefficients are realized using 2/3/4-bit right shifts of the mantissa based on its range. The hardware design is shown in the figure 4.2.

The piecewise linear approximation model is approximated using 5 linear intervals as shown below: [10]

$$\log_2(1+m) = \begin{cases} m + \frac{3m}{8} & 0 \le m < 0.125 \\ m + \frac{m}{4} + \frac{1}{64} & 0.125 \le m < 0.25 \\ m + \frac{m}{16} + \frac{1}{16} & 0.25 \le m < 0.5 \\ m - \frac{m}{8} + \frac{1}{8} - \frac{1}{32} & 0.5 \le m < 0.75 \\ m - \frac{m}{4} + \frac{1}{4} & 0.75 \le m < 1 \end{cases}$$

4.1.3 working

Here m is 23- bit mantissa part of 32-bit floating point representation, the above linear equation can be implemented as follow:

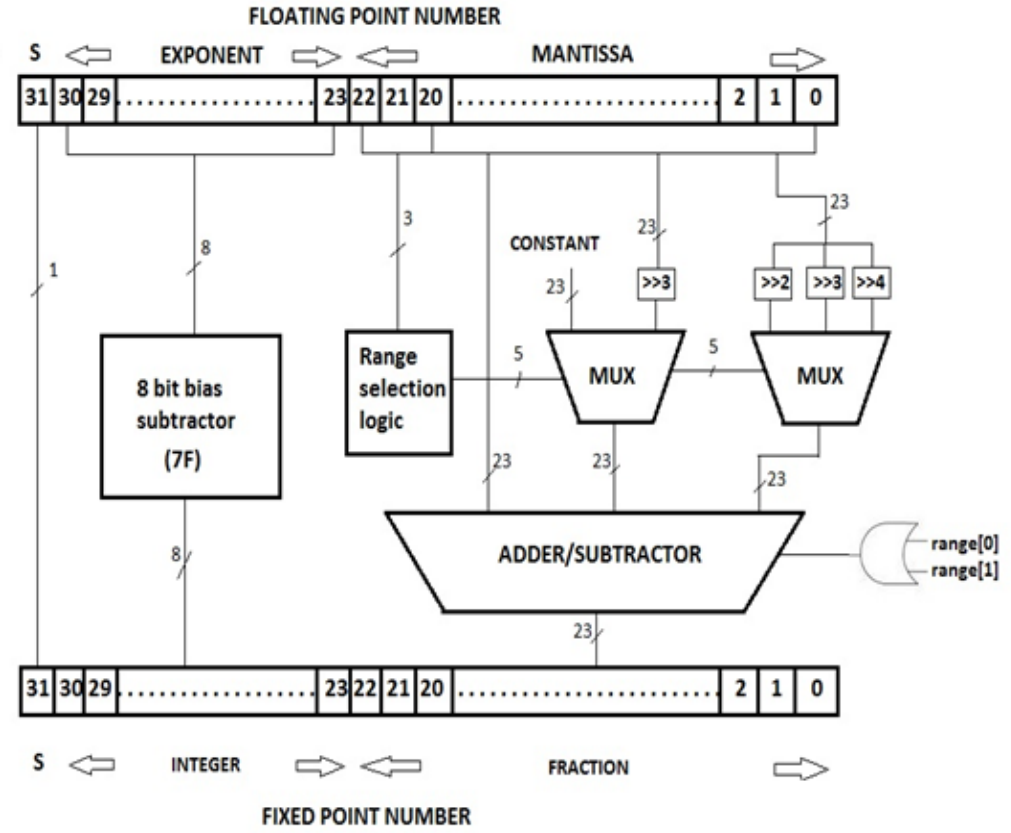


Figure 4.2: Logarithm base 2 circuit.

The sign bit is taken as it is, now the 8 bit biased exponent is converted into unbiased exponent by subtracting 7F (hex equivalent of 127) from the biased exponent. The range of the equation is selected by using decoder logic, which is shown in the figure 4.2. There are two multiplexers used in the design one is for selecting the constant value and other one is used for selecting the shifted version of mantissa value. The fraction of mantissa is implemented using shift registers (right shift). The output of two multiplexers are added with the mantissa value m, which forms the 23 bit fraction of fixed point representation of log2 value. This 23 bit is concatenated with sign bit and unbiased exponent to form a complete 32-bit fixed point number.

The figure 4.2 shows the log base 2 implementation, range selection logic is implemented using a decoder logic which is shown below. The mux logic is used to imple-

ment constant terms and shifted mantissa terms in the linear equation. A 2:1 mux is used to implement constant logic in which range[4] is used as the select line because the range[4] doesnt have any constant term in its linear equation but instead has two shifted version of mantissa, thus we have one of the two input of the mux as >>3 and other one is constant term.
The other mux is used to select the shift register output of mantissa in which select lines are chosen as shown in figure 4.2. Finally, the output of two muxes are passed to the adder/subtractor unit along with mantissa value in which or gate logic is used to implement the negative portion in the linear equation. The output value is concatenation of sign bit, 8-bit unbiased exponent and the output of adder/subtractor unit which represents the fixed point log base2 number.

4.1.4 simulation result

The figure 4.3 shows the simulation result for the log base2 circuit.

4.1.5 conclusion

This circuit can be operated in a single clock cycle and the max error due to piecewise linear approximation in this model is about 0.55 % as compared to floating point single precision value. This model has a very low mean error about 0.12 % as compared to floating point single precision value. The error percentage for different values are plotted below.

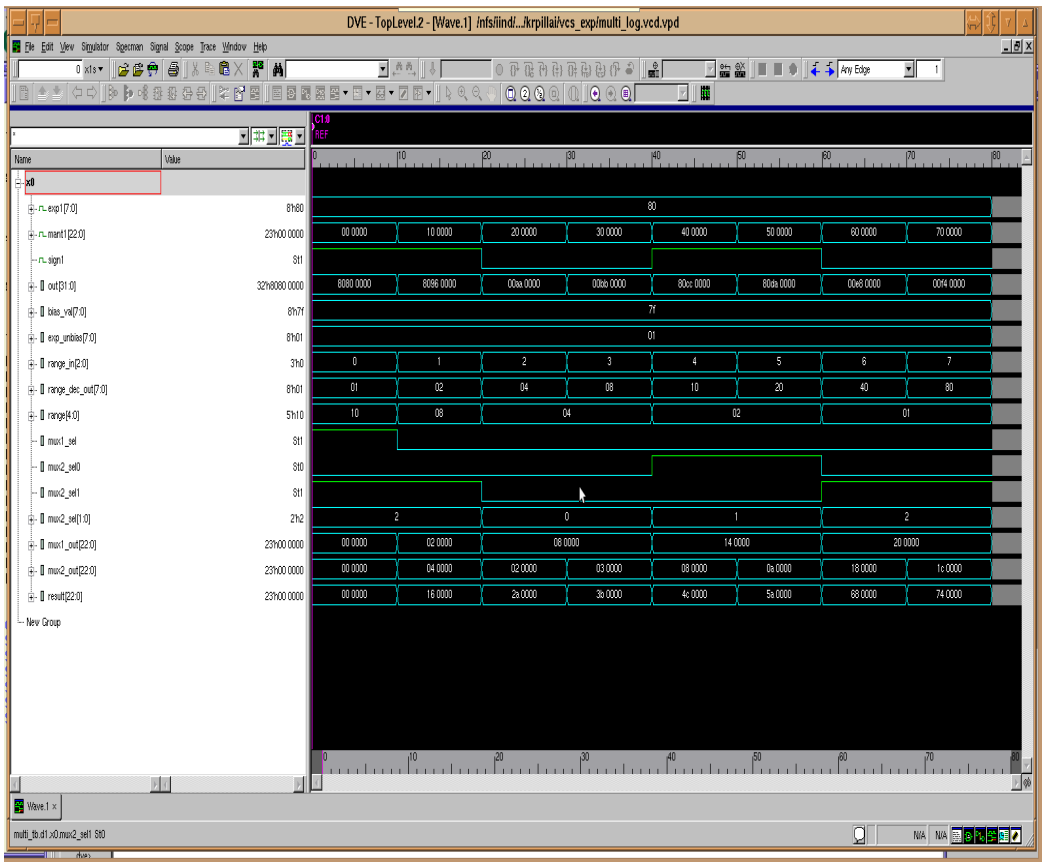


Figure 4.3: Simulation result of Logarithm base 2 circuit.

4.2 Anti-Logarithm base2 circuit

4.2.1 Introduction

In the previous section we introduced a dedicated hardware for calculating Log base2. Now there is a need for converting this number back to natural form. Thus we required an antilog base 2 circuit which can do this job. Again we use the piecewise linear approximation techniques to calculate the antilog values to the particular intervals. This is explained in the following subsections.

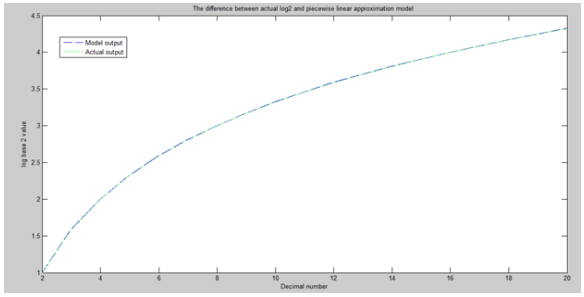


Figure 4.4: The difference between actual log2 and piecewise linear approximation of log2.

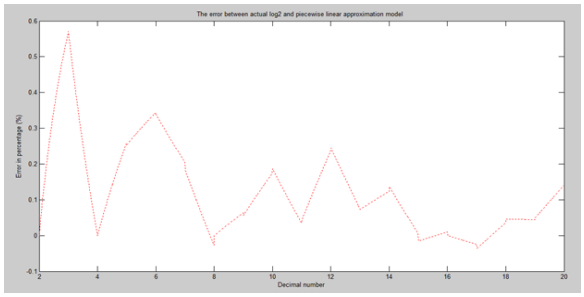
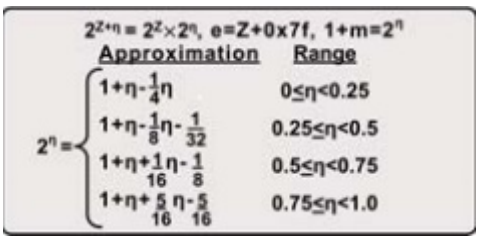


Figure 4.5: Error percentage between fixed point and floating point in log2 multi model for 1800 samples.

4.2.2 Implementation

The Fig. 4.6 shows the anti-log base 2 hardware implemenatation. The circuit looks similar to that of log base 2 and the working is also same. The Fixed point number obtained as an output of the log circuit is take as an input for this circuit. The signed bit (s) is taken as it is and stored into the signed bit position of the final result. Now the 8 bit unbiased exponent value is converted to biased exponent by adding the 8 bit bias value i.e. 7F. The remaining fractional part of 23 bit is divided into 4 intervals using piecewise linear approximation and converted to the form which can be implemented using hardware blocks. This is shown as below:

Here m is 23- bit mantissa part of 32-bit floating point representation, the above linear equation can be implemented as follow:



The sign bit is taken as it is, now the 8 bit biased exponent is converted into unbiased exponent by subtracting 7F (hex equivalent of 127) from the biased exponent. The range of the equation is selected by using decoder logic, which is similar to that in log circuit. There are three multiplexers used in the design one is for selecting the constant value and other two are used for selecting the shifted version of mantissa value. The fraction of mantissa is implemented using shift registers (right shift). The output of all multiplexers are added with the mantissa value m, which forms the 23 bit fraction of floating point representation of antilog2 value. This 23 bit is concatenated with sign bit and unbiased exponent to form a complete 32-bit floating point number.

The figure 4.6 shows the log base 2 implementation, range selection logic is implemented using a decoder logic which is shown below. The mux logic is used to implement constant terms and shifted mantissa terms in the linear equation. A 2:1 mux is used to implement constant logic in which range[4] is used as the select line because the range[4] doesnt have any constant term in its linear equation but instead has two shifted version of mantissa, thus we have one of the two input of the mux as >>3 and other one is constant term.

The other mux is used to select the shift register output of mantissa in which select lines are chosen as shown in figure 4.6. Finally, the output of two muxes are passed to the adder/subtractor unit along with mantissa value in which or gate logic is used to implement the negative portion in the linear equation. The output value is concatenation of sign bit, 8-bit biased exponent and the output of adder/subtractor unit which represents the floating point antilog base2 number.

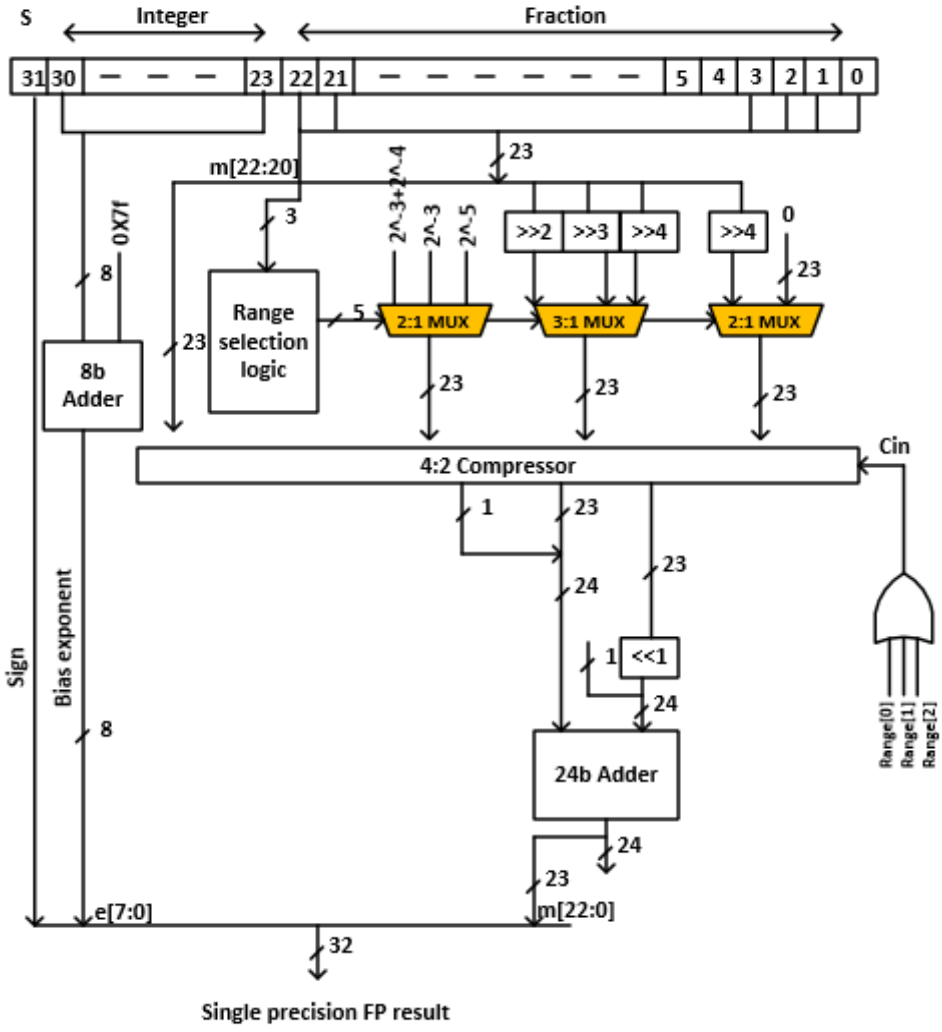


Figure 4.6: Anti-Log base 2 circuit

The error pecentage and its comparison can be found as follow:[11] The circuit has a maximum error of 0.6% and a mean error of 0.2%, which is very less as compared to the log circuit. Moreover this much error is not visible to the human eyes when implemented in lighting model of GPU. Thus these circuits are suitable for the usage in the graphics architecture.

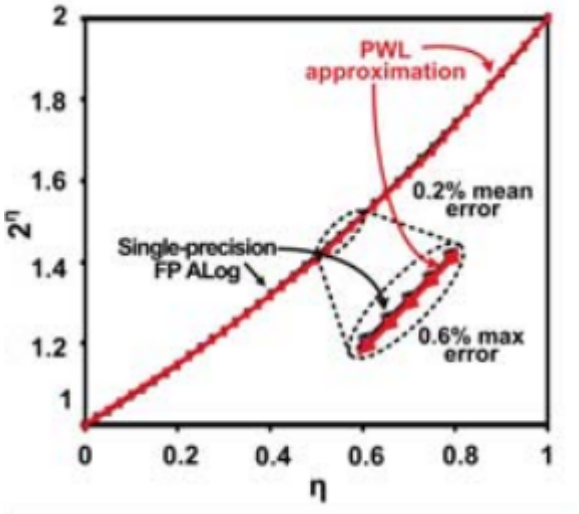


Figure 4.7: Error percentage in the antilog circuit [10]

4.3 Phong Illumination Model

4.3.1 Introduction

The Phongs Illumination model is the most basic lighting model in 3D graphics. All the lighting models used by the GPUs say from lower range segment to very high range segment, in one way or other use Phong illumination model. Many lighting models used today are actually adding more terms into the phong model. Thus the proposed hardware can be used to implement all types of lighting models.

4.3.2 Implementation

The figure 4.8 shows the implementation of Phong model, using the log and antilog circuits discussed in the above sections.

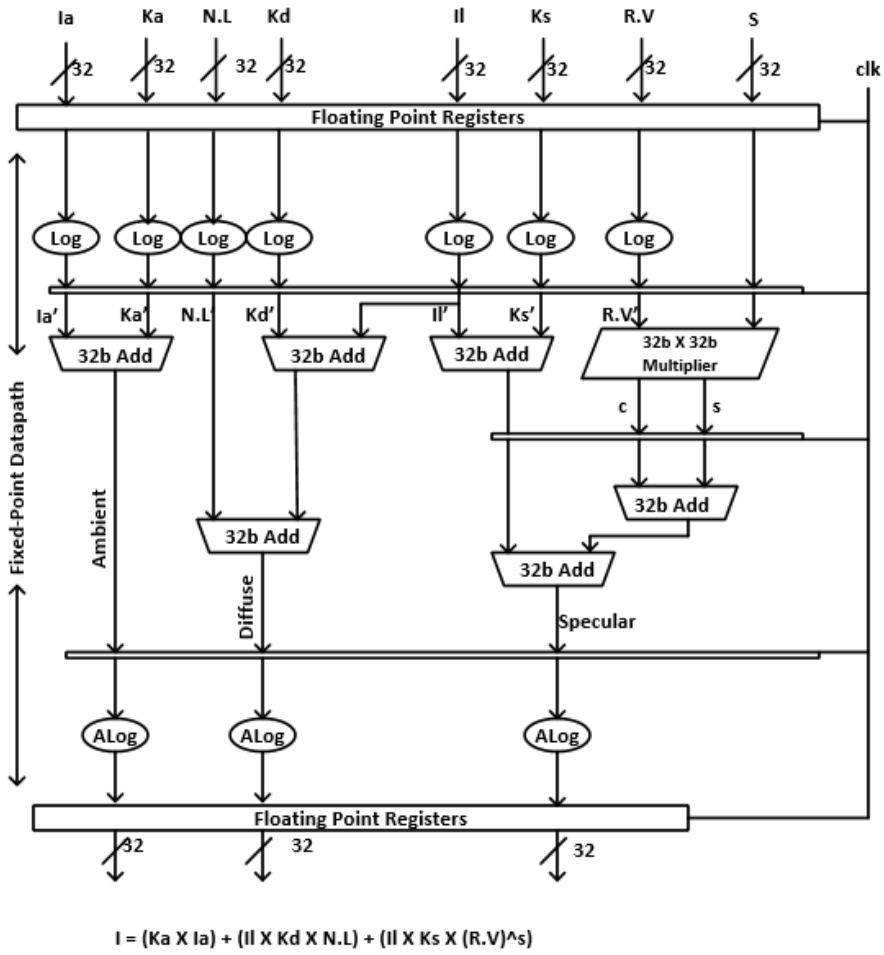


Figure 4.8: Phong Illumination Model

Ambient: - Ambient light can refer to available light in the environment.

Diffuse: - It is the reflection of light equally in all directions.

Specular: - It is the reflection of light at a particular direction only.

All these component combined in an empirical formula which we call Phong illumination model. The equation is shown in the figure 4.8

4.4 Summary

A dedicated hardware for producing logarithm base 2 and antilog base 2 circuit implementations were explained in this chapter in section 4.1 and 4.2 respectively. Using these circuits we proposed a new hardware for Phong Illumination model in section 4.3.

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

As the technology is shrinking, the reducing power consumption and over all power management on a chip are the major challenges below 65nm due to complexity. For many designs, optimization of power is as important as timing because of the need for reduced packaging cost and extended battery life. This becomes even more challenging when it comes to the 3D graphics, where large amount of processing is needed. This report focuses on the different low power design techniques which can be implemented for 3D graphics. The low power design techniques discussed here have their own pros and cons which should be taken care while choosing a design technique, and proper trade-off should be made. Different types of hardware are also discussed in this report which are proposed to reduce the power consumption in the existing designs. These design techniques are basically used in the RTL stage of the design so it will be more efficient and can reduce 20 to 70 % of the power dissipation in the design. All such method are available but it is up to the designer to select one or combination of more methods to optimize the design without affecting the functionality of the design.

5.2 Future Scope

In future, more low power design techniques can be developed and discussed, which can be used in the implementation of 3D graphics hardware. Thus, these techniques will result in implementing low power GPUs which ultimately result in better graphics performance in the handheld devices.

References

[1] F. J. Pollack."*New microarchitecture challenges in the coming generations of CMOS process technologies* ". IEEE 32nd Annual International Symposium on Microarchitecture, Haifa,Israel, 16-18 Nov. 1999.

[2] Dennis Crain, Dale Rogerson. "*Graphics pipeline*". Internet: http://msdn.microsoft.com/en-us/library/windows/desktop/ff476882%28v=vs.85%29.aspx, Jun.08,2014 [Aug.23, 2014].

[3] Microsoft documents.

[4] W.D. Hearn and M. Baker. "*Computer Graphics principles and practices*", 3rd edition, Pearson education, 2002.pp. 123- 197.

[5] Kanika Kaur and Arti Noor. "*Strategies and Methodologies for Low Power VLSI Designs: A Review*", International Journal of Advances in Engineering and Technology, May 2011.

[6] Intel documents.

[7] Vranesic Hamacher. "*Computer Organization*", 5th edition, McGraw Hill Education (India) Private Limited, 2011. pp. 237-332.

[8] Kostas Pagiamtzis and Ali Sheikholeslami. "*Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey*", IEEE Journal of Solid-State Circuits, Vol.41,No.3. pp. 712-727. 2006.

[9] F.K.Hanna and A.K.Misra, "*Hardware Realisation Of Binary Search Algorithm*", IEEE PROC.,vol.127, Pt.E,No.4, JULY 1980. pp. 148 151.

[10] F. Sheikh, S. Mathew, M. Anders, H. Kaul, S. Hsu, A. Agarwal, R. Krishnamurthy, S. Borkar. "*A 2.05 GVertices/s 151 mW Lighting Accelerator for 3D Graphics Vertex and Pixel Shading in 32 nm CMOS*", IEEE Journal of Solid-State Circuits, Volume: 48, Issue: 1, pp. 128 139. Jan. 2013.

[11] Y-S. Kwonl. "*A Hardware Accelerator for the Specular Intensity of Phong Illumination Model in 3-Dimensional Graphics*", IEEE Asia and South Pacific Design Automation Conf, pp. 559-564, 2000.