박사학위논문

Doctoral Thesis

고정 소수점 SIMD Vertex Shader를 이용한 저전력 프로그래머블 3D 그래픽스 프로세서

A Low Power Programmable 3D Graphics Processor with Fixed-point SIMD Vertex Shader

손 주 호 (孫住鎬 Sohn, Ju-Ho)

전자전산학과 전기 및 전자공학 전공

Department of Electrical Engineering and Computer Science

Division of Electrical Engineering

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

2006

고정 소수점 SIMD Vertex Shader를 이용한 저전력 프로그래머블 3D 그래픽스 프로세서

A Low Power Programmable 3D Graphics Processor with Fixed-point SIMD Vertex Shader

A Low Power Programmable 3D Graphics Processor with Fixed-point SIMD Vertex Shader

Advisor: **Professor Yoo, Hoi-Jun**

By

Sohn, Ju-Ho

Department of Electrical Engineering and Computer Science

Division of Electrical Engineering

Korea Advanced Institute of Science and Technology

A thesis submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical Engineering and Computer Science, Division of Electrical Engineering

Deajeon, Korea

2006. 6. 3

Approved by

_____

Professor Yoo, Hoi-Jun

고정 소수점 SIMD Vertex Shader를 이용한 저전력 프로그래머블 3D 그래픽스 프로세서

손 주 호

위 논문은 한국과학기술원 박사학위 논문으로
학위논문 심사위원회에서 심사 통과하였음.

2006년 5월 29일

심사위원장	유 회 준	(인)
심사위원	김 성 대	(인)
심사위원	나 종 범	(인)
심사위원	박 인 철	(인)
심사위원	김 재 민	(인)

DEE
20035146

손 주 호, Sohn, Ju-Ho, A Low Power Programmable 3D Graphics Processor with Fixed-point SIMD Vertex Shader. 고정 소수점 SIMD Vertex Shader를 이용한 저전력 프로그래머블 3D 그래픽스 프로세서. Department of Electrical Engineering and Computer Science, Division of Electrical Engineering. 2006, 99p. Advisor Professor Yoo, Hoi-Jun. Text in English

Abstract

The real time 3D graphics becomes the most attractive application for mobile terminals, in which the battery lifetime and small computing power, however, limit the system resources and memory bandwidth for graphics processing. Besides, since users watch graphics images on a small screen very closely to their eyes, recent mobile 3D graphics are introducing the programmability in both hardware and software for more advanced functionality while achieving low power consumption. In this research, I designed and implemented a programmable graphics processor with fixed-point vertex shader for mobile applications. The proposed architecture has four major features: separation of data transfer flow, full hardware accelerations with stream processing, two level extensions of instruction set architectures, and fixed-point single-instruction-multiple-data (SIMD) processing. The graphics processor contains an ARM10 compatible 32-bit RISC processor, a 128-bit programmable fixed-point SIMD vertex shader, a low power rendering engine with 26kB dedicated graphics cache, and a programmable frequency synthesizer (PFS). Different from conventional graphics hardware, the proposed graphics processor implements ARM10 coprocessor architecture with dual operations so that user-programmable vertex shading is possible for advanced graphics algorithms and various streaming multimedia processing in mobile applications. The circuits and architecture of the graphics processor are optimized

for fixed-point operations and achieve the low power consumption with help of instruction-level power management of the vertex shader and pixel-level clock gating of the rendering engine. The PFS with a fully balanced voltage-controlled oscillator (VCO) controls the clock frequency from 8MHz to 200MHz continuously and adaptively for low power modes by software. The 36mm^2 chip shows 50Mvertices/s and 200Mtexels/s peak graphics performance, dissipating 155mW in 0.18㎛ 6-metal standard CMOS logic process. For more enhancement of stream processing, model of 3D graphics computing is analyzed and SIMD computing elements with hierarchical memory system is revised into the architecture of graphics processor. The implemented graphics processor was successfully demonstrated on the evaluation platform and verified real-time 3D graphics in mobile applications.

Table of Contents

Table of Contents

Table of Contents

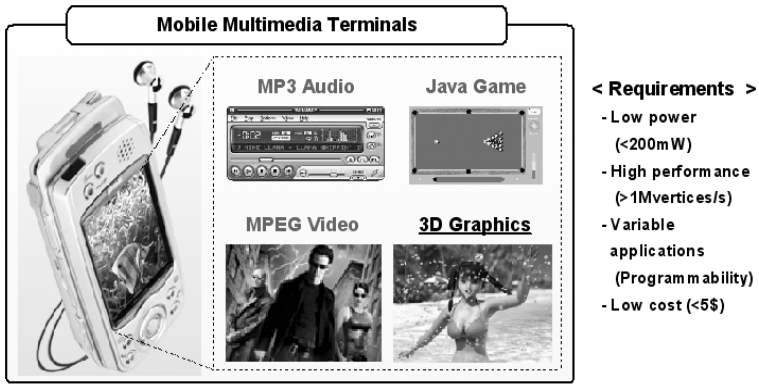
CHAPTER 1
Introduction

1.1 Mobile Multimedia Terminals

The popularity of mobile terminals such as smart cell-phones and wireless personal digital assistants (PDAs) is increasing with the rapid expansion of the mobile electronics market and its migration from text-based applications to various multimedia applications. Today's mobile terminals are evolving to become wireless multimedia centers that allow us to take pictures, watch 2D graphics animations and MPEG4 movies, listen to MP3 music, and enjoy Java games. Among these applications, real-time 3D graphics becomes one of the most attractive applications. It is especially beneficial to games, advertisement, and avatars whose data can be downloaded over the wireless network. Complex 3D scenes can also be represented by lists of vertices, texture images, and corresponding camera movements, yielding high data compression ratios, so as to make 3D graphics advantageous for bandwidth-critical wireless applications.

Since the real-time 3D graphics requires huge computing power and corresponding memory bandwidth, it has been a critical issue even in PC or console platforms during the past ten years [1-3]. Although today's PC graphics accelerators can draw high-quality 3D images with high performance graphics processing unit (GPU), however, handheld devices cannot tolerate those tens-of-watt power monsters. Figure 1.1-1 shows the mobile multimedia terminal and its requirements. For mobile

applications, the low power consumption is the most important issue because of limited battery lifetime. When we use a typical Li-ion battery of 2000 mWh energy capacity, the power budget of graphics system including processing and internal memory access should be limited to less than 200mW for two or three hours seamless operation. The Advanced RISC Mahcine (ARM) processor family that has the reduced instruction set computer (RISC) architecture is widely used as the main platforms for wireless applications because of its high MIPS/Watt [4]. However, these low power RISC platforms have very limited system resources in terms of computation power and memory bandwidth, so that the additional mobile graphics processor should be implemented to consume as little energy as possible in the given platforms. Moreover, since users are watching 3D graphics images on a small screen very closely to their eyes [5], the graphics processor must generate high quality of graphics images with high performance such as more than 1Mvertices/s processing speed. And, the variety of applications in a single hardware requires the programmability for advanced and flexible algorithms such as programmable vertex shading and image processing. Also, the low-cost aspect cannot be ignored because the target system will be carried by everybody's hand.



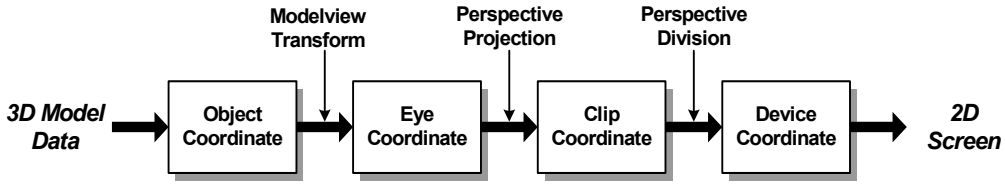
[Figure 1.1-1: Mobile Multimedia Terminal and Its Requirements]

Recently, several researchers have tried to increase the mobile graphics capabilities in mobile applications. Since the rasterization and texture mapping require more

processing complexities than the rest of operations in 3D graphics pipeline [6], most of graphics architectures have mainly focused on the rendering pipeline and achieved the efficient graphics performances [5][7][11-13]. However, since relatively little attention has been given to 3D geometry operations, now they become the performance barriers in 3D graphics pipeline. In the previous architectures, the general-purpose RISC processors with simple integer datapath [11-13] or conventional bus-mapped floating-point datapath [7] were used to process geometry operations. However, simple integer datapath cannot provide the required performance of programmable graphics processing. In the case of conventional floating-point datapath, the performance is also limited due to low operating frequency for limited power consumption.

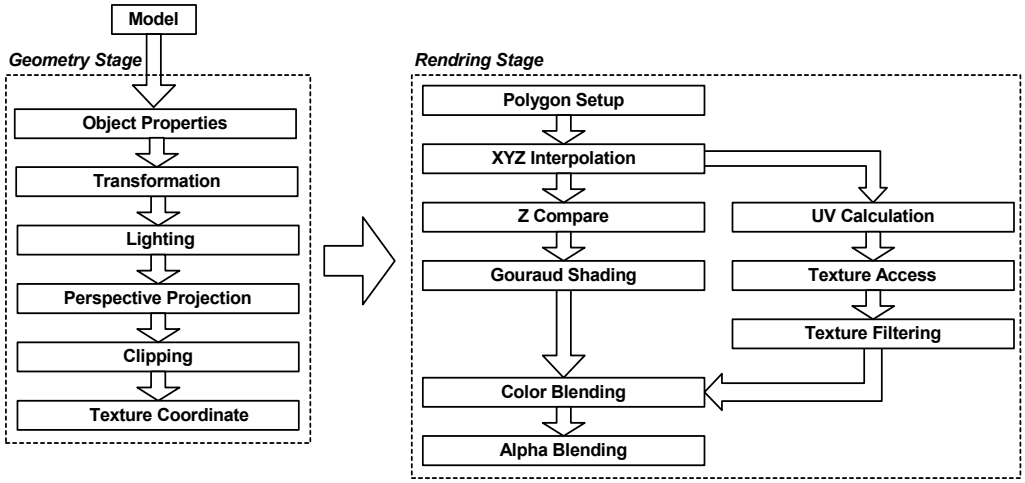
1.2 3D Graphics Pipeline

1.2.1 Traditional Graphics Pipeline



[Figure 1.2-1: Coordinate Transforms in 3D Graphics Pipeline]

The definition of graphics pipeline is the sequence of processes applied to transform a three-dimensional image into a two-dimensional screen and can be considered as the transformation between coordinate systems as shown in Fig. 1.2-1. The pipeline is responsible for processing information initially provided just as properties at the end points (vertices) or control points of the geometric primitives used to describe what is to be rendered. The typical primitives in 3D graphics are lines and triangles. The type of properties provided per vertex include x-y-z coordinates, RGB values, translucency, texture, reflectivity and other characteristics.



[Figure 1.2-2: Traditional Fixed Graphics Pipeline]

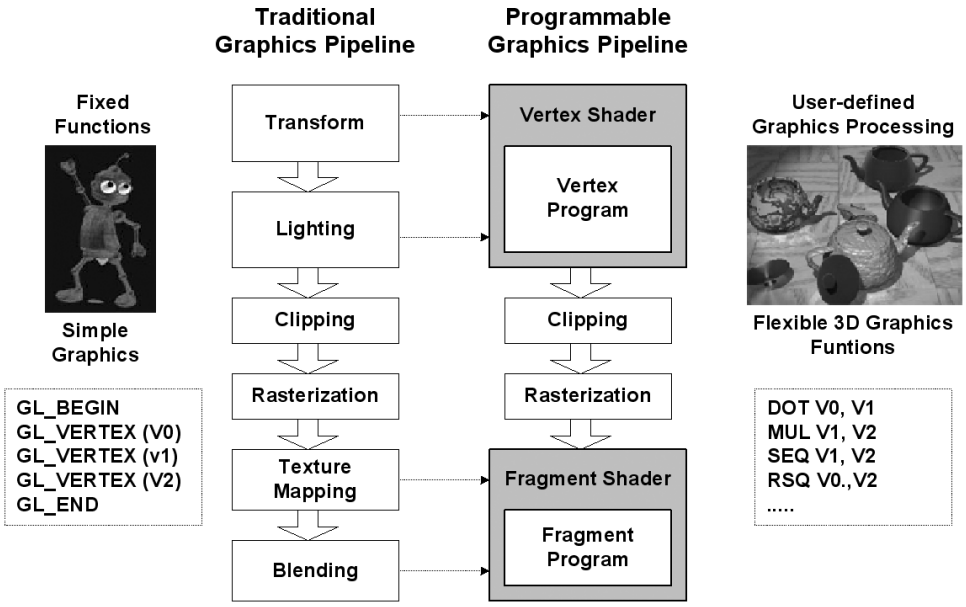
Figure 1.2-2 shows the traditional graphics pipeline. It is composed of geometry operations calculating the attributes of vertices of triangles, and rendering operations filling colors inside the triangles [8]. The geometry stage processes polygon data from input models by performing operations such as transformation, lighting, and perspective projection. Especially, the light effect is calculated by blending ambient, specular, diffuse, and emission component originated by each light source. Therefore it is computation-intensive, but the bottleneck can be relieved by using fast, parallel datapaths such as multi-core vector processors with 3D graphics-optimized instruction set architecture. Software simulation indicates that over 40GOPS is required when calculating the full suite of geometry operations at speed of 1Mvertices/s in conventional embedded RISC processors with floating-point graphics library [6]. The rendering stage takes the output of the geometry stage and draws pixels to the screen buffer. It first sets up triangles in 2D screen from 3D geometry data and performs interpolation to calculate edge coordinates of each triangle. Then it renders each pixel by shading and texture mapping, and also performs alpha-blending for translucent objects and z-comparison for hidden surface removal. The rendering stage operations are

memory-intensive due to frequent accesses of the frame buffer, depth buffer and texture memory. An effective graphics system memory bandwidth of 1GB/s is required to show realistic 3D images with a pixel fill rate of 1Mpixels/s on today's mobile terminals with QVGA screen size.

1.2.2 Programmable Graphics Pipeline

Figure 1.2-3 shows the comparison between traditional graphics pipeline and programmable graphics pipeline. In the traditional graphics pipeline, each unit has specific function with dedicated hardware block. So, it cannot give the flexibilities to various graphics algorithms although it can be fast and efficient in fixed graphics processing. Whether the batch processing such as vertex array is employed or intermediate mode (sample code segment shown in the left of the figure) is chosen, programmers cannot control the behavior of internal graphics pipeline except mode or parameter settings.

In the programmable graphics pipeline, there are vertex shaders and fragment shaders, which are optimized single-instruction-multiple data (SIMD) processors [9]. The vertex shader can execute vertex program, composed of assembly graphics instructions. It enables various user-defined vertex processing of geometry pipeline for flexible 3D graphics functions. The fragment shader is responsible for user-programmable pixel operations for realistic graphics images. It can execute fragment program, composed of assembly graphics instructions optimized for rendering pipeline. Now, programmers can control and calculate any attributes of vertex and pixel by their specific programs on instruction-set architecture (ISA) driven graphics processor. In addition to typical transformation lighting and texture mapping, various graphics effects such as shadow volume creation, vertex blending, motion blur, silhouette rendering and per-pixel phong lighting are made possible.



[Figure 1.2-3: Programmable Graphics Pipeline]

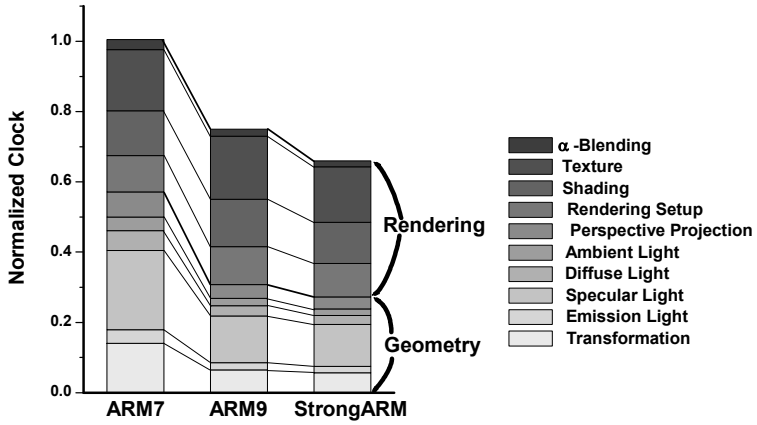
In the case of mobile 3D graphics, the programmable pipeline can be more useful, because we perform the graphics operations by software optimization in the programmable shaders instead of many complex hardware blocks. It eliminates unnecessary steps in graphics pipeline on specific input conditions. Moreover, the programmability in mobile terminals can allow the various multimedia applications to be optimized through software in a single compact and fast hardware.

1.2.3 Cycle Breakdown of Graphics Pipeline

Before designing mobile graphics architecture, cycle usage of each graphics pipeline stage were analyzed on conventional embedded RISC processor architecture such as ARM platforms [6].

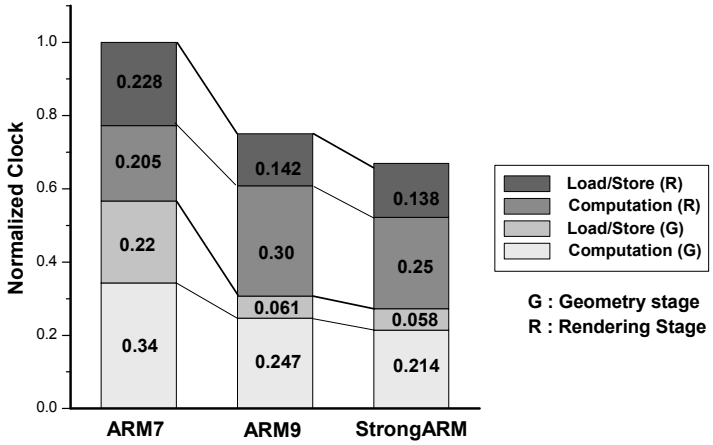
Fig. 1.2-4 shows cycle breakdown of each sequence of 3D pipeline normalized to ARM7 cycle time when conventional software floating-point graphics library is performed. The most time consuming part of the geometry stage is the calculation of specular lighting due to the distance calculation between light source and object as well

as normal vector of the object. To calculate specular lighting, floating-point divisions and square root operations are required. For the rendering stage, texturing consumes most of time, because it uses logarithmic and exponential operations to find level of detail (LOD) value, and it frequently accesses texture memory.



[Figure 1.2-4: Cycle Breakdown of Floating-point Graphics Library]

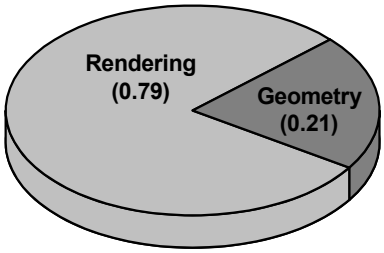
Fig. 1.2-5 shows instruction pattern of the geometry and rendering stage with SRAM interface as memory system. The load/store cycles were counted only when the datapath was owned by memory access instruction. The rendering stage has more memory access cycles than geometry stage in all of the processor types. It means that the memory bandwidth is more critical than computing complexity in the rendering stage. The



[Figure 1.2-5: Cycle Pattern of Graphics Pipeline]

portion of load/store cycles is cut in half for ARM9 and StrongARM that use Harvard architecture in geometry stage. It is because in Harvard architecture, the instruction and data can be fetched simultaneously. Since the required memory bandwidth can't be solved in conventional processor architecture, there is no performance enhancement in rendering stage even if ARM9 or StrongARM is used.

In order to enhance the computing efficiency of ARM's integer datapath, the fixed-point graphics library was also analyzed. The cycle pattern is similar to the case of the floating-point graphics library, However, the performance of geometry stage is more improved than the rendering stage. Figure 1.2-6 shows the cycle breakdown in this case. When using the fixed point library, the 79% of total cycle times is spent in rendering stage and the performance of 3D pipeline is limited by pixel fill rate. It is because the high memory bandwidth required in rendering stage cannot be solved by means of increasing the computing efficiency.



[Figure 1.2-6: Cycle Breakdown of Fixed-point Graphics Library]

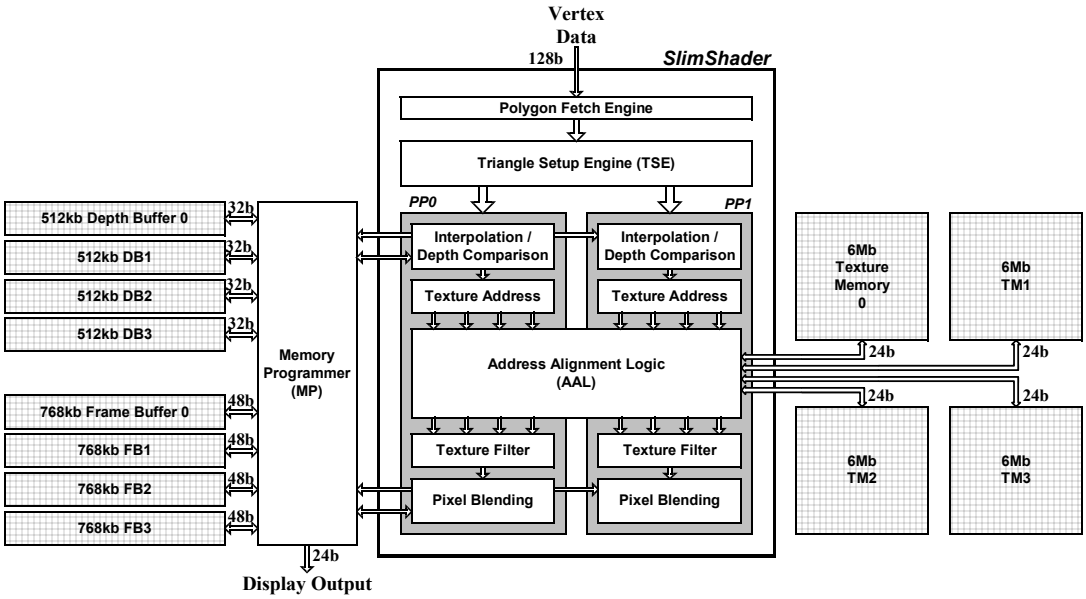
1.3 Related Works

1.3.1 RAMP by KAIST

The memory bandwidth is the most stringent constraint in implementing 3D graphics architecture. Solving the bandwidth bottleneck with traditional approaches such as high-speed crossbar and off-chip DDR-SDRAMs can result in increased power consumption. However, the limited screen resolutions in mobile terminals (e.g., QVGA) imply that the reasonable amount of integrated memory, from tens of kByte to

hundreds of kByte, is sufficient for graphics memories such as highly hit-rated caches or frame buffers [10]. Moreover, integrating whole necessary memory itself with logic in a single die yields more effective architectures or implementation schemes in terms of performance and power consumption. The RAMP design methodology of KAIST is based on the philosophy that memory is no longer a passive device, nor a sub-system. The RAMP (RAM Processor) architecture utilizes embedded DRAM (eDRAM) for 3D rendering in a very efficient manner that avoids connecting the memory with a large number of wires and corresponding crossbar switch. Three RAMP chips were evaluated in order to demonstrate the RAMP architectures and methodology [11-13].

The latest RAMP-IV [13] focuses more on real-time 3D gaming applications, drawing bilinear MIPMAP texture-mapped pixels with special rendering effects at 66Mpixels/s and 264Mtexels/s, as well as supporting shading operations of the previous RAMP architectures. Figure 1.3-1 shows SlimShader architecture developed in RAMP-IV. It consists of a triangle setup engine, an edge processor (EP), two pixel processors (PXPs), and 29Mb of embedded DRAM. The reduced number of PXPs is compensated by using



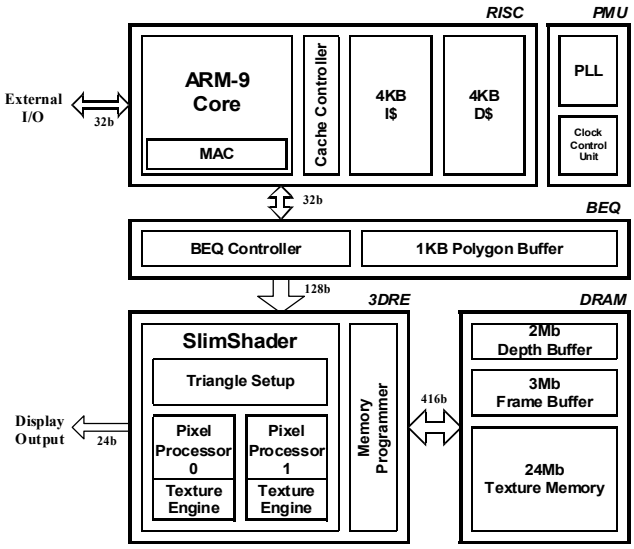
[Figure 1.3-1: SlimShader Architecture]

deeply pipelined PXP structure for high clock frequency of 50MHz. Since texture mapping is crucial function in real-time 3D graphics for more realistic pictures, SlimShader contains two energy-efficient texture engines (TEs). In bilinear filtering, two pixels mapped to texel space require 8 texture memory requests at every cycle, causing huge power consumption. TEs employ Address Alignment Logic (AAL), which uses temporal and spatial localities of texture addresses in MIPMAP-filtering to reduce total memory requests, yielding power saving. For real-time special effects such as fog, anti-aliasing and cartoon shading, memory programmer is implemented in SlimShader, and post-processes the rendered pixels of frame buffer by using dedicated instruction set and SIMD datapath.

RAMP-IV distributes the embedded DRAM over the logic pipeline via different ports, in addition to pixel-parallel distribution. Each pipeline stage can directly and concurrently access the contents of DRAM, just like accessing dedicated local SRAM. Satisfying the pipeline timing is a big challenge in terms of DRAM design as the cycle time (tRC) of embedded DRAMs must be less than 20ns, while commodity SDRAMs are working at 65ns or more. The timing budget of frame and depth buffers is even stricter as the read-data must be written back to the same address within a single cycle for efficient Read-Modify-Write (RMW) transactions. Distributing the DRAMs over the pipeline and accessing one or some of them selectively can reduce the power consumption of memory by 65%. Since the depth of the processed pixel is compared at the first stage of PXP pipeline, the following stages and corresponding memories can be gated off according to comparison result.

The SlimShader architecture is integrated into a RAMP-IV chip together with an ARM9-compatible RISC processor with enhanced multiply-and-accumulate (MAC), 29Mb embedded DRAM, and a power management unit as shown in Figure 1.3-2. The chip was fabricated using 0.16㎛ Hynix 256Mb SDRAM

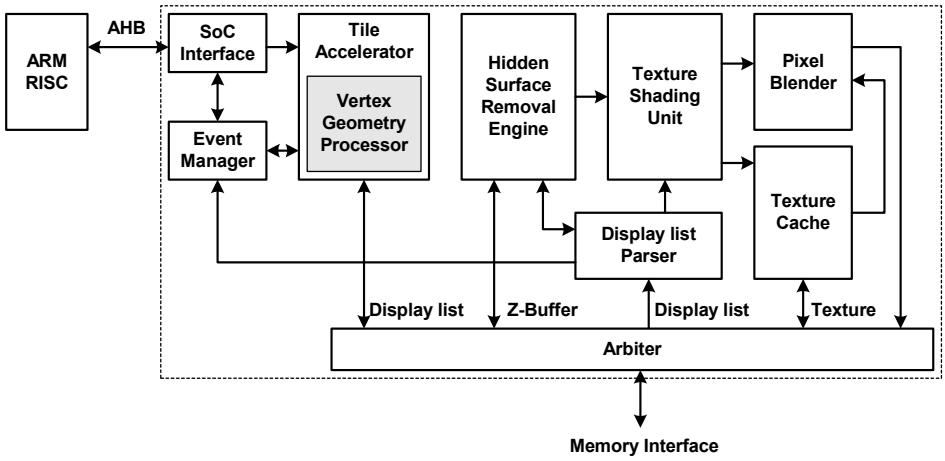
process. Its area and power consumption were 121mm^2 and 10mW respectively. The RAMP-IV chip utilizes a pure DRAM process to reduce the fabrication cost. Although the pure DRAM process has slower logic transistor speed and fewer metal layers, a 133MHz speed could be achieved in the chip's RISC processor. Negligible sub-threshold leakage current of the DRAM process also reduces standby current, which is a critical issue for battery-driven device. However, RAMP-IV accelerates only rendering operations, which occupy 79% of total execution time. Therefore, Amdhal's law [14] tells that this limited functionality guarantees only five times performance speed-up compared with software-only implementation at most in actual cases.



[Figure 1.3-2: RAMP-IV by KAIST]

1.3.2 MBX by PowerVR

MBX is 2D/3D graphics core co-developed by Imagination Technology and ARM to accelerate 2D/3D graphics on ARM-based mobile platform [15]. As shown in Figure 1.3-3, MBX contains a tile rasterizer, a hidden surface removal (HSR) engine, a vertex geometry processor (VGP), a texture shading unit, a pixel



[Figure 1.3-3: PowerVR's MBX]

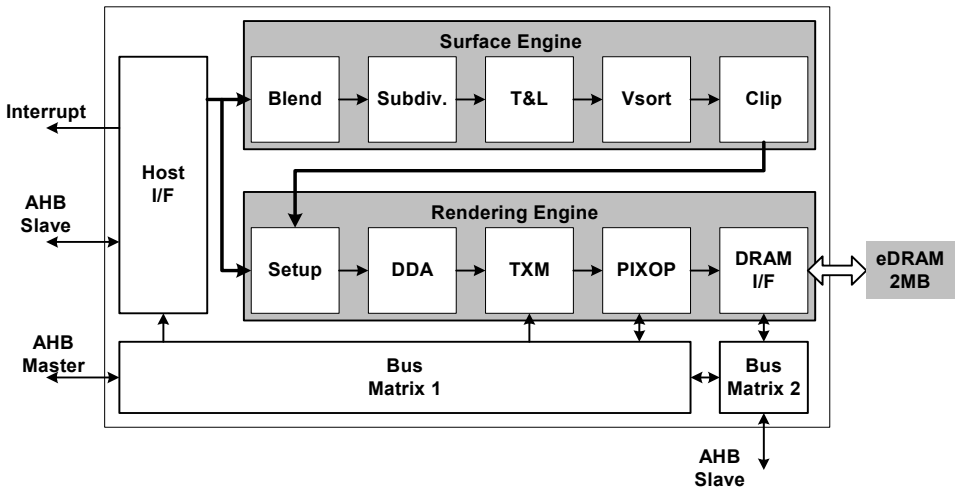
blender and 512kB texture cache. It also has its own memory controller with virtual memory functions to reduce the overhead of host system. Energy consumption is proportional to the number of memory access, so many researchers focus on reducing off-chip bandwidth to enhance the battery lifetime for mobile 3D applications. Unlike the conventional graphics architecture [16], MBX reduces memory accesses by tile-based rendering, in that a scene is partitioned into small tiles or regions and each region is rendered independently. This deferred rendering techniques may reduce the bandwidth to access data for frame and textures, however it needs extra time and bandwidth to setup parameters for tiling itself.

To process the geometry operations, MBX has 4-way SIMD floating point VGP optimized for 3D graphics. VGP can operate with rate of 4 FLOPS at the 120MHz in the 0.13 ㎛ CMOS logic process. It can be used as hardware transformation and lighting engine. MBX is the embedded 2D, 3D and video acceleration cores which can be integrated with the conventional RISC core via system bus. Although it has the all necessary hardware blocks for 3D graphics, the performance can be lower than expected because the bus architecture is used as the interface between the host processor and embedded core. The bus traffic caused by transferring the data

preprocessed by host processor to the embedded core can be bottleneck of performance. Therefore, the sustained pixel fill rate can be only 9Mpixels/s at 100MHz, which is less than 10% of maximum rendering performance. Moreover the cost was increased by complex system architecture.

1.3.3 Playstation Portable by Sony

In 2004, Sony released Playstation Portale™ (PSP) for real-time 3D graphics gaming applications and other multimedia such as MPEG video and MP3 audio in battery-operated consumer electronics products [17]. It contains all necessary hardware blocks required in handheld video gaming system, including a MIPS processor with vector floating-point unit (FPU), 3D graphics module and media processing unit. The PSP features 4MByte of embedded DRAM to boost internal memory bandwidth and support Read-Modify-Write operations for 3D graphics. Figure 1.3-4 shows the 3D graphics module implemented in the PSP. The graphics module consists of surface engine and rendering engine. The surface engine reduces the model data size by supporting high speed tessellation for Bezier and Spline surfaces while increasing reality of graphics images. It also supports hardware transformation and lighting operations, and more advanced graphics algorithms such as geometry and vertex blending for skinning and morphing. Four parallel pixel pipeline of the rendering engine can draw various types of graphics images at the speed 664Mpixels/s at 166MHz operating frequency. Although the conventional bus protocol was used for interfacing with host system, the direct eDRAM controller attached in the rendering engine can reduce the memory bandwidth requirements concentrated to external DDR main memory. This eDRAM controller also. allows host system to access directly video memory for flexible memory operations. The media processing unit equips hardwired H.264 codec for MPEG accelerations and virtual media engine for



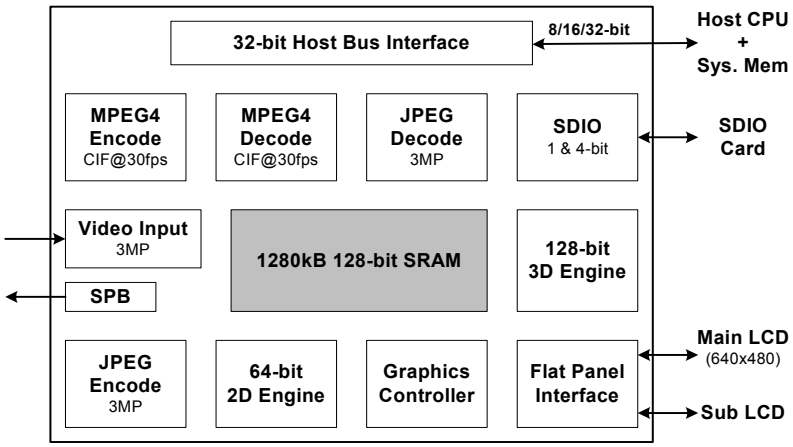
[Figure 1.3-4: 3D Graphics Module in the PSP]

real-time reconfigurable audio/video codec implementation while achieving low power and low cost. However, the relatively high power consumption and complex system architecture of the PSP make it difficult its application in mobile terminals such as cell-phones. More optimizations in both of functionalities and architectures are required.

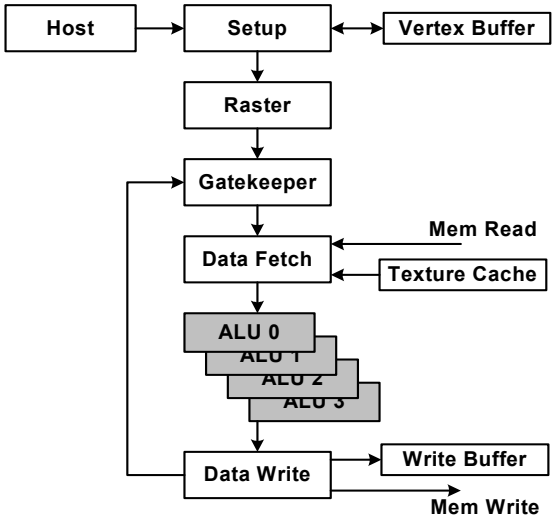
1.3.4 SC10 by nVidia

nVidia's SC10 is a companion chip for handheld devices such as PDAs and cell-phones, accelerating images, video, 2D and 3D graphics [18]. Figure 1.3-5 shows chip block diagram and 3D graphics engine. It operates with assumption of host processor and external memory, and interfaces with host processor from 8-bit to full 32-bit I/O. The equipped full duplex hardware MPEG-4 codec and serial bus interface for camera control with own LCD interface enables various multimedia solutions in a single chip. The SC10 distinguishes itself from other architectures by implementing pixel-level programmability such as blending and combining operations for more realistic graphics images on handheld displays. The embedded 1280kB SRAM provides large vertex cache for reducing external

memory accesses. Although setup unit of the graphics engine relieves the burden of host system by performing simple transform, clip and culling operations, the lack of dedicated geometry engine and slow off-chip host interface limit the performance to less than 1Mvertices/s at 75mW power consumption.



[Figure 1.3-5 (a): nVidia's SC10]



[Figure 1.3-5 (b): Graphics Engine in the SC10]

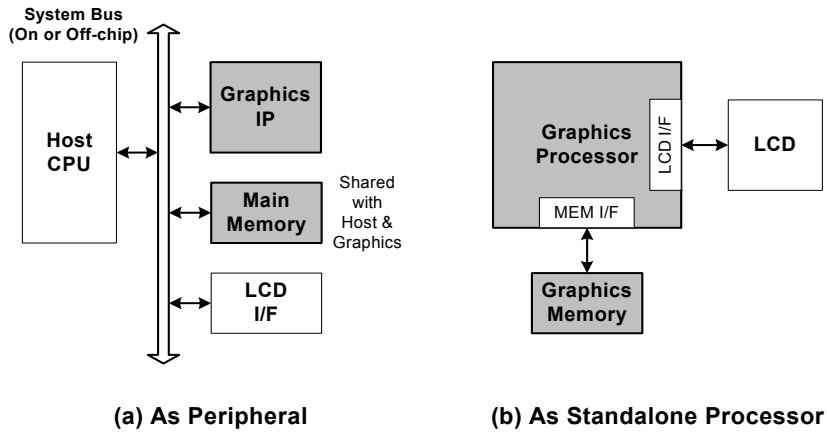
1.3.5 Others

Recently, a hardware rasterization architecture for mobile phones was presented by Akenine-Moller, et al [5]. The proposed architecture focused on reducing

memory accesses to external memory in rendering of textured triangles. The inexpensive multi-sampling anti-aliasing scheme, a new texture filtering method with texture minification and compression, and a scan-line based z-culling scheme shows the relatively moderate performance in software-only implementation on commercial cell-phones. Mitsubishi's Z3D core, intended for mobile phones, utilizes clock gating to achieve the lowest power consumption in spite of a floating-point geometry engine and 1Mbits embedded SRAM [19]. Also, a 3G baseband processor with 3D capability [20] and embedded RISC processor with geometry FPU [21] are trying to realize 3D graphics on mobile platforms.

Standard software graphics APIs for embedded systems have also been released. One example is OpenGL-ES, which is subset of desktop Open-GL [22]. OpenGL-ES adopts optimizations such as fixed-point operations and redundancy eliminations for mobile devices with low processing power, while enabling fully programmable 3D graphics such as vertex and pixel shading

1.4 Architecture Summary of Mobile Multimedia Hardwares



[Figure 1.4-1: Categories of Mobile Multimedia Hardware]

Integration philosophy can categorize the listed mobile multimedia hardwares in the previous section into two parts — *as peripheral intellectual property (IP) or*

Integration / Feature	As peripheral	As standalone processor
Traditional 3D graphics	PowerVR's MBX • OpenGL-ES compatible ARM IP • On-chip bus interface • Tile-based rendering	KAIST's RAMP-IV • Full 3D pipeline with texturing • 28Mb embedded DRAM • Limited functions (GE)
Multiple functions	nVidia's SC10 • 2D/Camera video processor and pixel-shading GPU • Off-chip companion interface • Lack of geometry engine	Sony's PSP • Single chip LSI with H.264, 3D and reconfigurable processor • 4MB embedded DRAM • Not low power consumption

[Table 1.4-1: Summary of Mobile Multimedia hardware]

as standalone processor. Figure 1.4-1 shows the conceptual block diagrams of these categories and Table 1.4-1 summarizes the features of the related works.

Conventional bus architecture and complicated floating-point design is not suitable for mobile multimedia in terms of power consumption and balanced performance.

In the conventional works, PC graphics architecture [1-3], which has its roots in traditional workstation graphics system [16], has been applied to various consumer electronics and battery-operated devices. Hence, PC graphics can be used to consider design issues for mobile terminals. The graphics processing unit (GPU) in the PC system contains large vector floating-point units (FPUs) with special instructions for graphics operations. The main CPU invokes the GPU using the system bus interface. However the available system memory bandwidth is not sufficient to support both the CPU and GPU. In the GPU architecture, several pixel engines work in parallel to boost performance, fetching data from dedicated T$ (texture cache) and P$ (pixel cache) memories. The external memory interface (EMI) merges transactions from cache memories and transfers them to off-chip memories assigned to graphics processing. The memories are connected to the EMI through a high-speed crossbar switch. Burst-mode operations are used to fully utilize the available memory bandwidth. Although each cache element and

FPU can be power-efficient, the massive structure and high-speed crossbar of the GPU cannot be applied directly to mobile terminals that lack sufficient computing power and memory capacity. In addition, modern baseband chips and mobile platforms such as Qualcomm's MSM chip [20] or TI's OMAP [23] employ power-efficient ARM processors of integer datapath, and are implemented as system-on-a-chip (SoC) optimized for battery-operated mobile devices.

This research is proposed as responses to these concerns. It utilizes a simple ARM coprocessor interface or dedicated buffer connected to an energy-efficient fixed-point graphics accelerator with specific local memory.

1.5 Contributions of This Research

Since the rasterization and texture mapping require more processing complexities than the rest of operations in the 3D graphics pipeline, most of graphics architectures have mainly focused on the rendering pipeline and achieved the efficient graphics performances. However, since relatively little attention has been given to 3D geometry operations, now they become the performance barriers in 3D graphics pipeline. Moreover, the previous designs of mobile graphics architectures seem to be too complicated to be applied to PDAs and cell-phones or to provide limited functionalities such as lack of geometry engine and programmability. Especially, the variety of mobile applications requires generality as well as high performance. Therefore, I proposed and implemented programmable 3D graphics processor for mobile application. It can fill the gap between the flexible high performance 3D geometry systems and the low power wireless platforms with limited system resources. The proposed hardware architecture has four major features:

(a) *Separation of data transfer flow* is proposed for efficient hardware and bandwidth utilization. Different from previous works, the ARM coprocessor

architecture, that is an instruction extension mechanism of ARM platform, enables optimized performance throughput while achieving easy programmability.

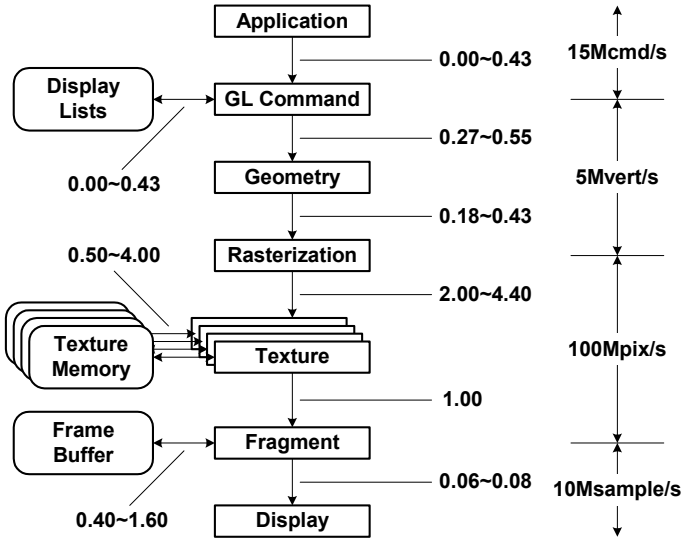
(b) *Full hardware accelerations with stream processing* are achieved to boost-up the sustained performance in compact and fast hardware. Various low power techniques in both of instruction set and its micro-architecture along with clock management are implemented for low power consumption. The producer-consumer locality, that are frequently observed in stream multimedia operations such as 3D graphics, is also considered in hardware design.

(c) *Two level extensions of instruction set architecture* are implemented for programmability and parallel processing. The added multimedia instructions by the coprocessor architecture is once again extended to more optimized graphics instructions by dual operations, in which concurrent operations of graphics coprocessor with main processor are enabled.

(d) *Fixed-point SIMD processing* is employed for low power consumption and low cost implementation. It exploits data level parallelism in graphics processing while keeping the power consumption low.

CHAPTER 2
System Architecture

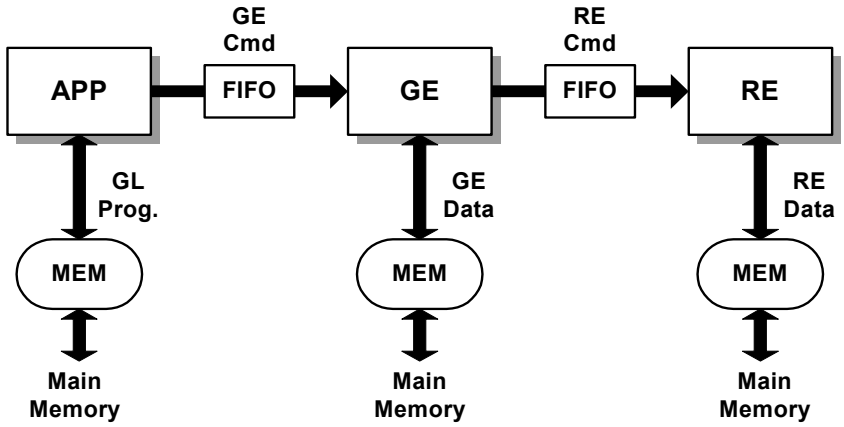
2.1 Model of 3D Graphics Computing



[Figure 2.1-1: Communication Bandwidth in Graphics Pipeline]

The whole system performance of graphics hardware is dependent not only on the performance of the individual hardware acceleration blocks but also on the communication cycles for transferring the graphics data between memory and processing elements. Figure 2.1-1 summarizes communication bandwidth in various points of graphics pipeline operating at 5Mvertices/s 100Mpixels/s with 640x480 display in terms of gigabyte (GB) per second [24]. From the figure, typical mobile graphics requires total bandwidth ranging from 4.8GB/s to

12.5GB/s, however, which is difficult to achieve in conventional 32-bit SDRAM running at 100MHz. Since much of required bandwidth are consumed in local traffic among processing elements, concept of stream processing as well as optimizations arithmetic circuits should be considered to reduce explicit communication costs. Each context in graphics hardware needs appropriate local memory for buffering intermediate data, yielding low external memory requests. So, it can be thought the following model of 3D graphics computing shown in Figure 2.1-2 from a hardware implementation's point of view.



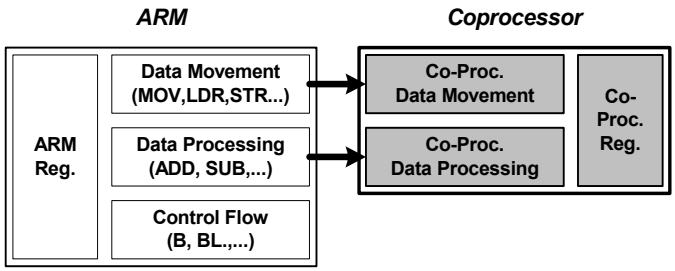
[Figure 2.1-2: Model of 3D Graphics Computing]

There are three processing elements with their own local memory — application (APP), geometry engine (GE) and rendering engine (RE). Although real implementation may contain dedicated graphics memory chip with hardware accelerators, all external data including application program, vertex model and calculated pixel output are conceptually stored in main memory owned by host processor. However, FIFO memory can be used to interconnect among processing elements for intermediate data. These intermediate data may be commands or instructions for next processing elements, or may be temporary output of previous processing element. In real implementation, direct connections or shared bus can be employed in place of FIFO memory.

2.2 Separation of Data Transfer Flow

As described in the previous section, the communications of graphics data in pipeline are crucial concerns in designing hardware, and many implementations use single or multi-layer bus architectures [15][17-21]. However, I proposed a mobile graphics processor architecture using coprocessor interface to implement the model depicted in the figure 2.1-2 [25-26].

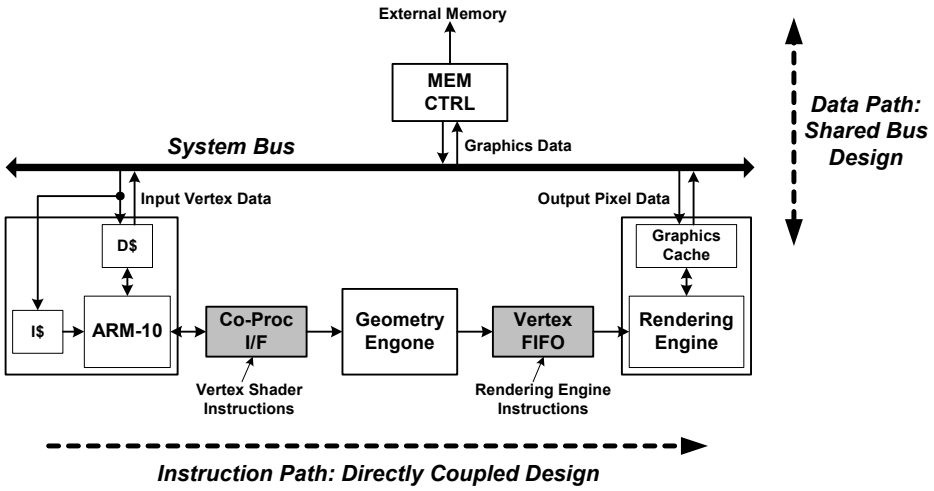
In the modern embedded RISC processor such as ARM platform, the coprocessor is defined as a general mechanism for extension of instruction set architecture [27] as shown in Figure 2.2-1. ARM coprocessors have their own private register set and state, and these are controlled by coprocessor instructions that mirror the ARM's instructions controlling ARM's register set. The ARM has sole responsibility for control flow, so the coprocessor instructions are concerned only with data processing and data movement. Following RISC load-store architectural principles, these categorizes are cleanly separated.



[Figure 2.2-1: Coprocessor Architecture]

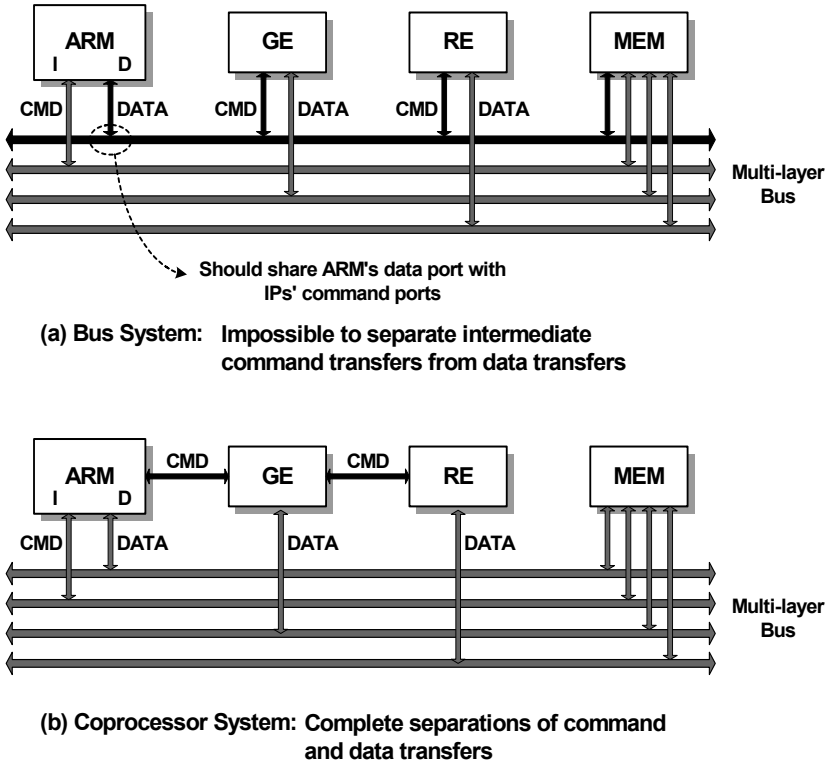
Figure 2.2-2 shows data transfer flow of the proposed mobile graphics processor. In this data flow, hardware-instruction path and graphics-data path are separated physically from each other to improve the stream processing within allowed memory bandwidth. The coprocessor interface, which connects GE to the ARM10 processor, is used for GE instruction transfer. RE is connected to the GE through the vertex FIFO, which is also used for RE instruction transfer. The system bus

interfaces are used only for vertex and pixel data transfers. Since the data cache of the ARM10 processor can be shared with the GE to store the graphics primitives such as input-vertex-model data, the GE does not need additional cache system. The vertex data stored in the data cache of the ARM10 processor are transferred by vertex-attribute-move instruction of the GE, which is mapped into the coprocessor register transfer instruction of the ARM10 processor. The output pixel data are transferred between the graphics cache of the RE and the external memory through the system bus interface. The separation of instruction and data paths increases processing parallelism of the hardware blocks and reduces the required bus arbitration cycles. Therefore, directly coupled design is achieved in instruction transfer path for easy control of processing elements. And, shared bus design is used in data transfer path for easy memory management such as unified memory map architecture and efficient direct memory access (DMA) among memory space.



[Figure 2.2-2: Data Transfer Flow]

The coprocessor architecture shows many benefits over conventional multi-layer bus architecture in implementing model of 3D graphics computing. Figure 2.2-3 visualizes differences between coprocessor and bus architectures. Conventional bus



[Figure 2.2-3: Bus System and Coprocessor System]

architecture implies that additional hardware block attached in memory space should be connected with data port of main processor [28]. This is because modern embedded RISC processor doesn't have dedicated port for memory-mapped components. Therefore, as shown in Figure 2.2-3(a) the command transfers of hardware blocks should use shared bus with main memory transactions, causing inefficient utilizations of processing elements. In addition, the multi-layer bus architecture requires complex interconnections including multi-port arbiters with long and wide global metal wires, yielding high power consumption. Also, concentrated data transactions may cause heavy bus arbitrations, and main processor should always consider thread synchronizations in invoking bus-attached hardware blocks. On the other hand, the coprocessor system depicted in Figure 2.2-3(b) shows the following features.

(a) Direct signal path with short distance in coprocessor interface provides simple interconnections. Coprocessor shares bypassed instruction port with main processor. They don't need the bus arbitrations for hardware accesses contrary to conventional bus-attached hardware accelerators. Therefore, coprocessor interface can reduce the unwanted stalls between main processor and hardware accelerators, and thus relevant power consumption.

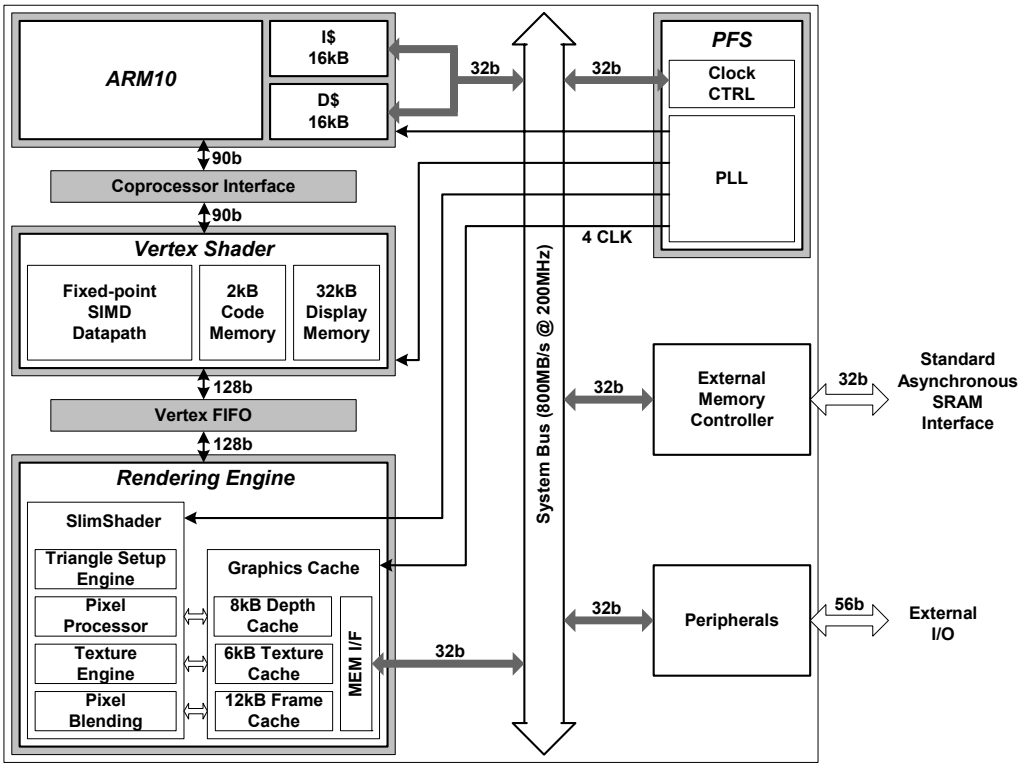
(b) Since the coprocessor operates in lock step with core pipeline of main processor, it can avoid a complex synchronization and provide a single thread of context.

(c) The data cache of main processor can be shared with coprocessor to store graphics primitives as well.

(d) Since commands of coprocessor are regarded as the extended instruction set architectures of main processor, easy programmability can be achieved.

2.3 Full Hardware Accelerations with Stream Processing

For high sustained performance, I implemented the graphics processor enabling full hardware accelerations of graphics pipeline including geometry stage. Figure 2.3-1 shows the block diagram of the proposed programmable graphics processor. It consists of an ARM10 compatible 32-bit RISC processor with 16kB I/D caches, a 128-bit programmable fixed-point SIMD vertex shader, a low power rendering engine and a programmable frequency synthesizer (PFS). The RISC processor controls the whole system, operating at 200MHz. The vertex shader is implemented as an ARM10 coprocessor and processes all per-vertex and geometry operations such as matrix transformation and lighting calculation by executing vertex programs. The primitive assembly such as clipping and culling is also performed by the vertex shader in collaboration with the RISC processor. Since the vertex shader is configured as an ARM10 coprocessor, the single thread of a



[Figure 2-3-1: Block Diagram of Graphics Processor]

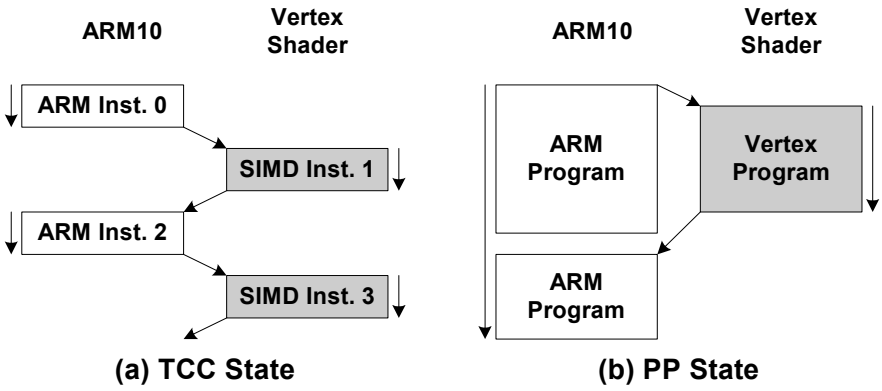
software context running in the ARM10 processor controls the vertex shader by the extended coprocessor instructions. The vertex shader operates at 200MHz in lock with the ARM10 processor, and there is no complex synchronization like bus arbitration. The rendering engine employs a low power 128-bit SlimShader pixel engine [29] with 26kB dedicated graphics cache system. The rendering engine is responsible for the rasterization and the per-pixel operations such as pixel blending and texture mapping. It is connected to the vertex shader through internal vertex FIFO that can store 128-bit wide 8-entries encoded instructions. The rendering engine instruction is composed of transformed vertex coordinates, texture coordinates and lit vertex color. The operating frequency of the rendering engine is as low as 50MHz to reduce power consumption. The PFS reduces the dynamic power consumption of the chip by clock gating and frequency scaling. It

supports four clock domains and clock of each domain can be controlled by software.

Since the data transfer flow depicted in the figure 2.2-2 was implemented in the graphics processor, input vertex stream can be fetched to ARM10's data cache memory, which can be shared as input vertex buffer for the vertex shader, while the vertex shader itself is operating. Also, output vertex stream can be transferred to the rendering engine from the vertex shader without stall of cycles by means of the vertex FIFO. Only final calculated pixel output stream are produced to the external memory, and all intermediate data transfers are separated from traffic of the global system bus. Therefore, full hardware accelerations with this stream processing capability achieves high sustained performance.

2.4 Two Level Extensions of Instruction Set Architecture

For easy programmability and efficient parallel processing in graphics and other multimedia applications, two level extensions of instruction set architecture is realized in the implemented graphics processor. Different from the conventional ARM coprocessor architecture [27][30], the graphics processor has dual operating states as shown in Figure 2.4-1. The first state is tightly coupled coprocessor

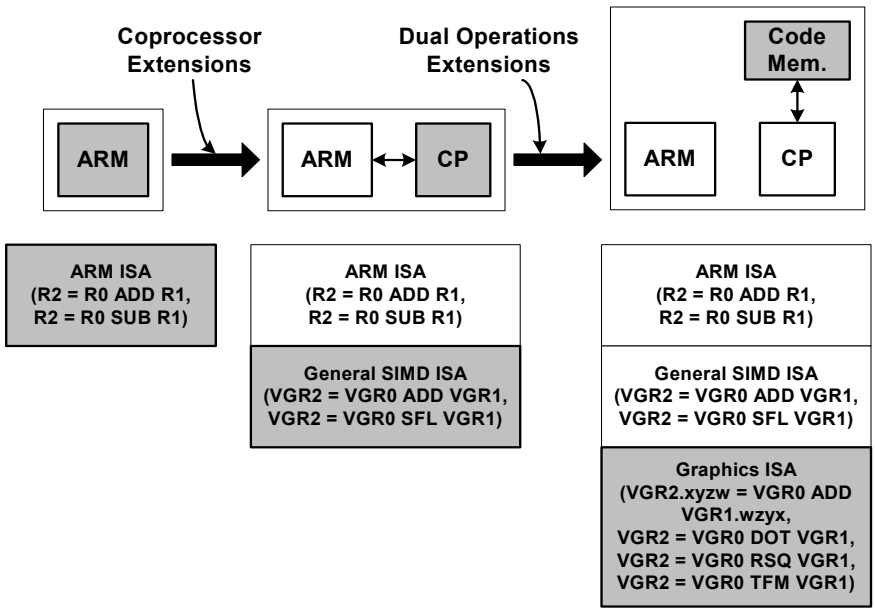


[Figure 2.4-1: Dual Operations]

(TCC) state. In this state, the vertex shader in the graphics processor operates as a normal ARM10 coprocessor. Its instructions are issued by the ARM10 processor as the extended coprocessor instructions, and they are conditionally executed to maximize their execution throughput. They do not affect the memory and registers unless the arithmetic flags (negative, zero, carry out and overflow) of the ARM10 processor satisfy a condition specified in the instructions. In the vertex shader, SIMD control flags such as arithmetic flags, saturation, overflow and underflow are updated after execution of every SIMD data processing instructions and can be moved to program status register (PSR) of the ARM10 processor. The general SIMD instructions such as arithmetic and movement operations are implemented in the TCC state, performing clipping and back-face culling operations in 3D graphics pipeline.

The second state is parallel processor (PP) state [31]. In this state, the vertex shader behaves like an independent processor and it does not need any control from the ARM10 processor. The PP state has a separate graphics instruction set different from the general SIMD instructions of the TCC state. The vertex shader executes the independent vertex program codes while the ARM10 processor performs its main application program or enters even into cache miss. Various user-defined vertex processing operations such as geometry transformation and lighting calculations can be performed for the current vertex input while next vertex data is fetched from the ARM10 processor. In order to maintain the communication protocol of the ARM10 coprocessor interface, the vertex shader drives coprocessor busy (CPbusy) signal to the ARM10 processor in the PP state, blocking next coprocessor instruction from the ARM10 processor for synchronization.

Therefore, I extended the instruction set architecture in two levels for conventional RISC processor as shown in Figure 2.4-2. The general SIMD



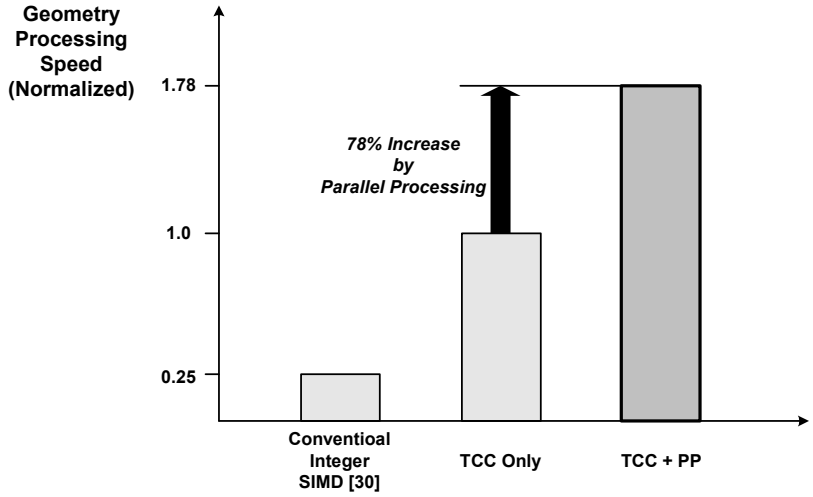
[Figure 2.4-2: Two Level Extensions of ISA]

instructions are firstly extended by the coprocessor architecture. By this extension, various multimedia operations such as 2D image processing and digital signal processing (DSP) applications can be easily programmed in mobile platform. Table 2.4-1 shows performance of various DSP kernels such as filter and dot product in the implemented graphics processor, and which achieves comparable performance with conventional DSP extension [30].

DSP Kernel	This Work	Intel WMX [30]
N sample T tap FIR filter	$\frac{5}{8}NT \quad (T \le 48)$ $\frac{9}{16}NT \quad (T \le 32)$	$\frac{5}{8}NT \quad (T \le 48)$
N sample min or max	$\frac{N}{2}$	$\frac{N}{2}$
N dimensional dot product	$\frac{3}{4}N$	$\frac{3}{4}N$
N dimensional vector sum	N	N
Add-compare-select (4 sample)	3	3
Absolute difference (4 sample)	3	3

[Table 2.4-1: DSP Performance of Graphics Processor]

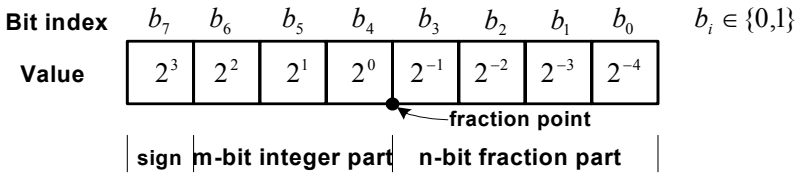
Beyond the first extension, the graphics instructions are secondly extended by the dual operations from the general SIMD instructions. The vertex shading instructions that are highly optimized for graphics applications [9] are enabled in this step by graphics extensions such as write masks and source swizzle. And, more specific data processing instructions such as reciprocal square root and matrix transformation are added to basic SIMD architecture. Moreover, enabled parallelism of the ARM10 processor and the vertex shader by dual operations gives high throughput to seamless input vertex stream. Figure 2.4-3 illustrates performance gain by the dual operations in full 3D geometry calculation. The extended instruction set architecture of the PP state achieves 78% performance improvement compared with case using only the general SIMD instructions of the TCC state.



[Figure 2.4-3: Performance Gain by Dual Operations]

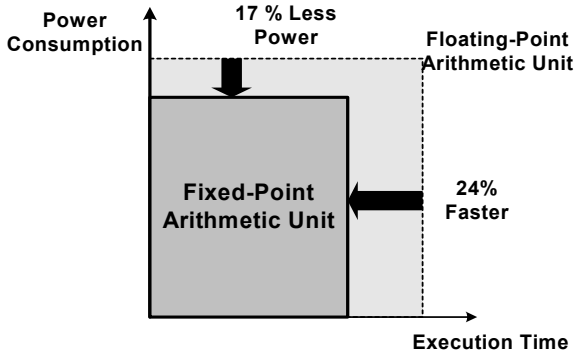
2.5 Fixed-point SIMD Processing

Most of multimedia data such as 3D graphics require real number representation to support various graphics algorithms. In this work, fixed-point number representation shown in Figure 2.5-1 is used instead of floating-point number



[Figure 2.5-1: Fixed-point Number Representation (ex: Q4.4)]

format [32]. Simple integer datapath of fixed-point unit can achieve higher clock frequency while consuming less power than floating-point unit, yielding total energy reduction. For typical 3D matrix transformation, gate level simulation of 4-stage pipelined 32-bit fixed-point multiplier showed 30% higher maximum operating frequency than 6-stage pipelined single-precision floating-point multiplier. In addition, the fixed-point multiplier consumed only 83% power of the floating-point multiplier at the same operating frequency. Consequently, when the fixed-point arithmetic is applied to graphics applications, 36% of total energy consumption can be saved on average (Figure 2.5-2).



[Figure 2.5-2: Energy Reduction of Fixed-point Processing]

To evaluate the accuracy of fixed-point arithmetic in the 3D geometry operations, the following equations can be used to decide the number of bits for fractional part, n_f, of $Q_{m.n}$ fixed-point number [33], where the 'm' is the number of bits representing integer part and 'n' is the number of bits representing fractional part.

For transformation,

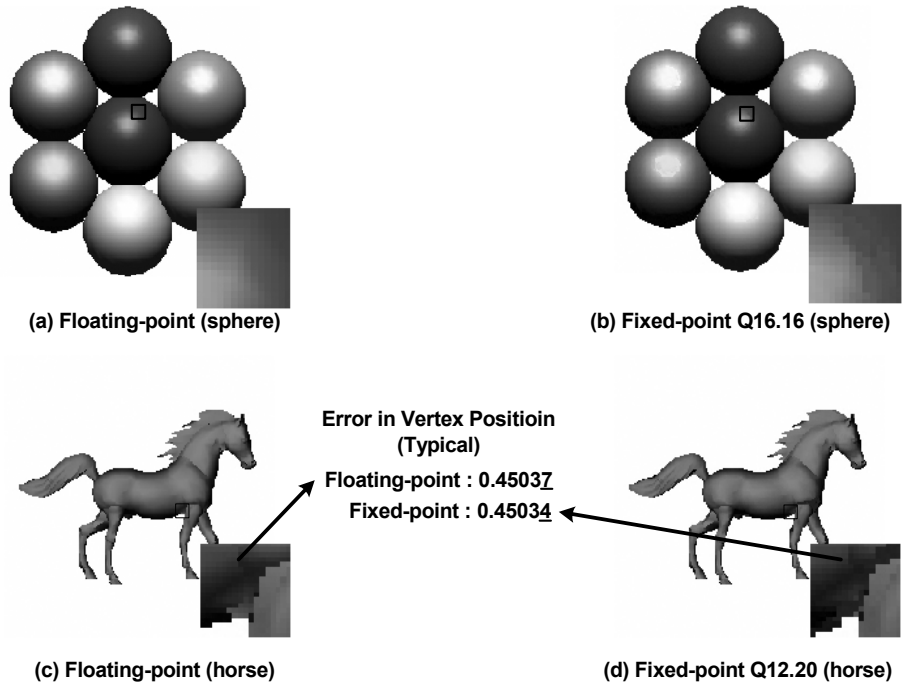
$$n_f = n_a + 3 + \left\lceil \log_2(1 + \frac{\textbf{distance of far plane from eye}}{\textbf{distance of scene vertex to eye}}) \right\rceil$$

for lighting,

$$n_f = n_a + 8 \quad \textbf{or} \quad n_a + 9$$

where 'n_a' is the number of bits required for securing the accuracy in transformation and lighting calculation.

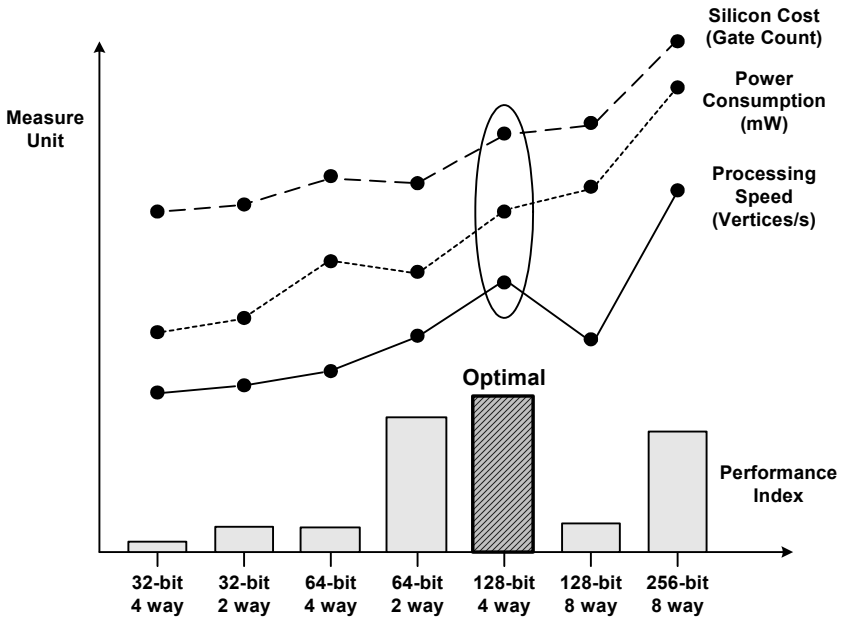
Since the screen resolution and color depth of mobile terminals are relatively small, 32-bit fixed-point system can generate final graphics images with unnoticeable accuracy loss compared with typical floating-point system. For example, minimal 14 or 16 bits are enough for fractional part to represent the vertex data for QVGA (320 x 240) with 16-bit color depth, which is common to the displays of today's mobile devices.



[Figure 2.5-3: Accuracy Comparison of Fixed-point Processing (Sphere: 5068 Vertices, Horse: 6798 Vertices]

In order to measure the accuracy of fixed-point arithmetic in graphics operations, the rendered images of 3D objects using software only floating-point graphics is compared library with the proposed hardware architecture using fixed-point graphics library. In Figure 2.5-3, lighted, smooth-shaded spheres with different material properties and 3D character with animations are rendered to show reliability of the proposed graphics processor. From the results, I can show that under vertex-level accuracy, the maximum transformed distance between floating-point and fixed-point systems is less than 0.000025 for Q12.20 fixed-point format and 0.0002 for Q16.16 fixed-point format.

$$Performance\ Index = \frac{Performance^{2}}{Area * Power\ Consumption}$$

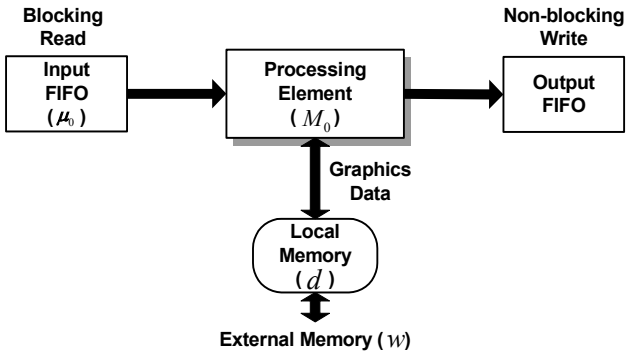


[Figure 2.5-4: Comparison of Various Fixed-point SIMD Configurations]

Many operations in 3D graphics and other multimedia applications shows data-level parallelism, in that same operations are concurrently and independently performed with multiple samples such as position coordinates of vertices or color values of pixels. Before applying SIMD architecture for fixed-point 3D graphics system, I analyzed various SIMD configurations while executing 3D geometry

operations. Figure 2.5-4 visualizes the overall performance index for each SIMD configuration. The performance index was chosen to consider multiple design constraints such as processing speed (vertices/sec), silicon cost (gate counts), and power consumption (mW) at the same time. Since, at least 14 or more bits of fractional part are required to represent fixed-point number in graphics operations, I assumed that each fixed-point operation in this analysis should be performed in 32-bit fixed-point number. Moreover, OpenGL-ES, the standard of embedded graphics library, requests to support Q16.16 fixed-point format in number representation [22]. From the gate-level simulation of each SIMD configuration, 128-bit 4-way fixed-point SIMD configuration was found to achieve the most optimal performance. When the length of SIMD width is less than 64-bit, the area cost caused by arithmetic circuits becomes dominant in performance index. However, as SIMD width is more wider, net interconnection area and relevant power consumption surpass the increase of processing speed. Therefore, I implemented that the proposed architecture utilizes 128-bit wide 4-way SIMD instructions, which allow it to concurrently process up to four 32 bit fixed-point data elements in a single cycle.

2.6 System Analysis



[Figure 2.6-1: Graphics Processing Element]

In order to analyze various characteristics of the graphics computing model and the proposed graphics processor mentioned in the previous sections, I adopted the simplified model of graphics processing elements shown in Figure 2.6-1, where

M_0 is the intrinsic computing power of processing element and,

μ_0 is the issue efficiency of input FIFO and,

d is the capacity of local memory in processing element and,

w is the provided memory bandwidth between local memory and external main memory.

Assuming that total 'n' vertices are divided to batches of size 'b' and each batch is processed with iterating loop for each vertex at one time independently, the batch processing time T_b can be represented as,

$$T_b = C_0 + \frac{m_0}{\mu_0 M_0} * b + T_d \qquad (1)$$

where C_0 and m_0 are constants and T_d is delayed time by external memory transfers from local memory. C_0 is the sum of initialization and epilogue time such as matrix, lighting parameter setting. m_0 is the cycle time consumed in performing one loop iteration for single vertex input.

If the capacity of local memory is sufficient, the processing element can operate while fetching next batch into the local memory simultaneously. Otherwise, the processing element should wait for finishing execution of current batch in order to complete the fetch of next batch. Generally, T_d is the function of the following form.

$$T_d = T_d(d, b, n, w, T) \qquad (2)$$

where T is total amount of time for processing all vertices.

From the above considerations, T_d can be represented as,

for low bandwidth

$$T_d = \left(\frac{C_1 b}{w} - \frac{b}{n} T \right) * \frac{d-b}{d} + \frac{C_1 b}{w} * \frac{b}{d}, \qquad w \leq \frac{nC_1}{T} \tag{3}$$

for medium or high bandwidth

$$T_d = \frac{C_1 b}{w} * \frac{b}{d}, \qquad w \geq \frac{nC_1}{T} \tag{4}$$

where C_1 is constant related with the size of each vertex in batch. The probability that the processing element decides to fetch next batch in the unused space of local memory is proportional to $\frac{d-b}{d}$, and in this case some portions of time for fetching next batch can be overlapped with processing time of current vertex batch. Moreover, if w is high enough, the batch fetching time can be hided completely. In the case that capacity of local memory is not sufficient, the processing element should fetch the next batch in the currently used space of local memory, causing the processing element waiting for finishing graphics operations.

Finally, total execution time can be computed as,

$$T = T_b * \frac{n}{b} \tag{5}$$

After substituting equation (3) and (4) into equation (1) and then using equation (5), I can find that graphics performance (P), processing speed per second, is represented as,

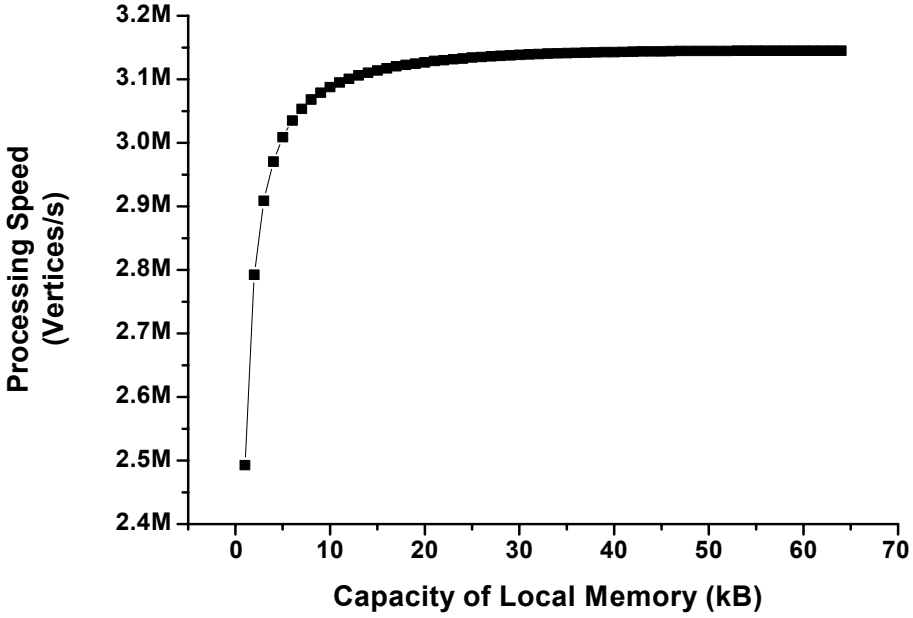
$$P = \frac{n}{T} = \frac{2 - \frac{b}{d}}{\left(\frac{C_0}{b} + \frac{m_0}{\mu_0 M_0} + \frac{C_1}{w} \right)} \qquad w \leq \frac{nC_1}{T} \tag{6}$$

$$P = \frac{n}{T} = \frac{1}{\left(\frac{C_0}{b} + \frac{m_0}{\mu_0 M_0} + \frac{C_1 b}{w} \right)} \qquad w \geq \frac{nC_1}{T} \tag{7}$$

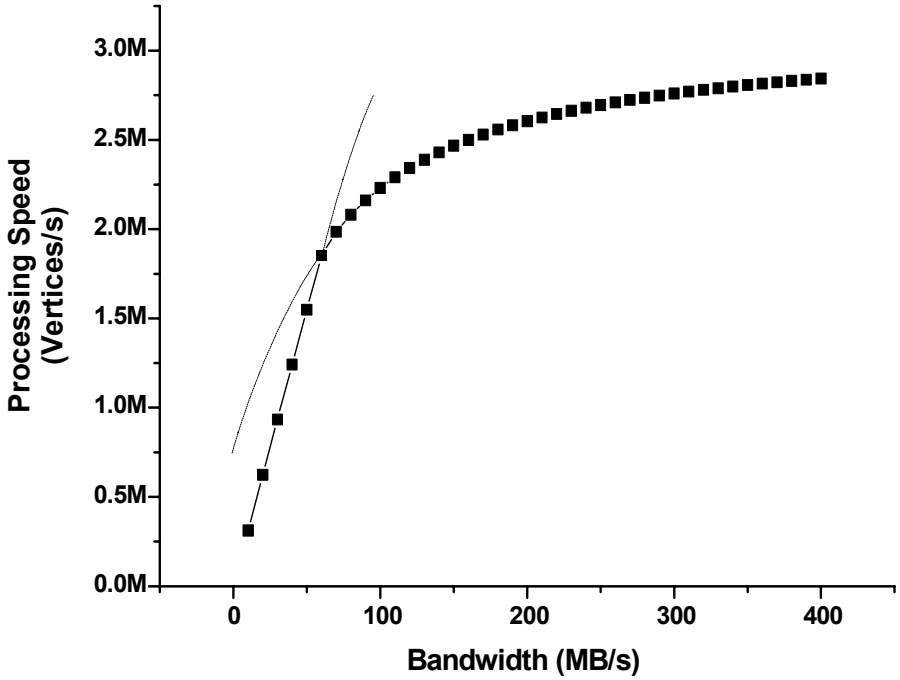
In order to verify the characteristics of above analysis model, I simulated the proposed graphics processor with changing various parameters such as bandwidth and memory capacity. The employed input graphics model is the light-shaded sphere composed of 100k vertices, and each vertex is assumed to be drawn only once. In the implemented graphics processor, the coprocessor interface with the ARM10 processor issues graphics commands such as vertex index to the vertex shader. So, the issue efficiency μ_0 is related with data cache hit ratio, which is about 0.9 roughly in this analysis. Since the rendering engine in the graphics processor is sufficiently fast, vertex FIFO can be regarded as infinitely capacitive output queue for the vertex shader. In a typical mobile device, a 32-bit SDRAM memory running at 100MHz is used as the external main memory I restricted the provided peak bandwidth to be less than 400MB/s.

Figure 2.6-2 shows the relationship between the capacity of local memory and the graphics performance when the provided memory bandwidth is 100MB/s. As the capacity is increased, the performance is also increased. However, the performance is saturated to peak value if the capacity is more increased. In this analysis, 16kB capacity achieves 99% of peak performance and 32kB capacity shows almost maximum performance.

Figure 2.6-3 shows the relationship between the provided memory bandwidth and the graphics performance. Similar to the capacity of local memory, the performance is increased as the memory bandwidth is more provided. However, the slope of performance improvement is more declined in the region of high bandwidth than in the region of low bandwidth. This is because the performance is more dependent on the computing power provided by the processing element and the vertex streaming characterized by the batch size and the memory capacity than the memory bandwidth if the bandwidth is sufficiently high. In this case, ideally, the processing element can fetch next vertex from main memory



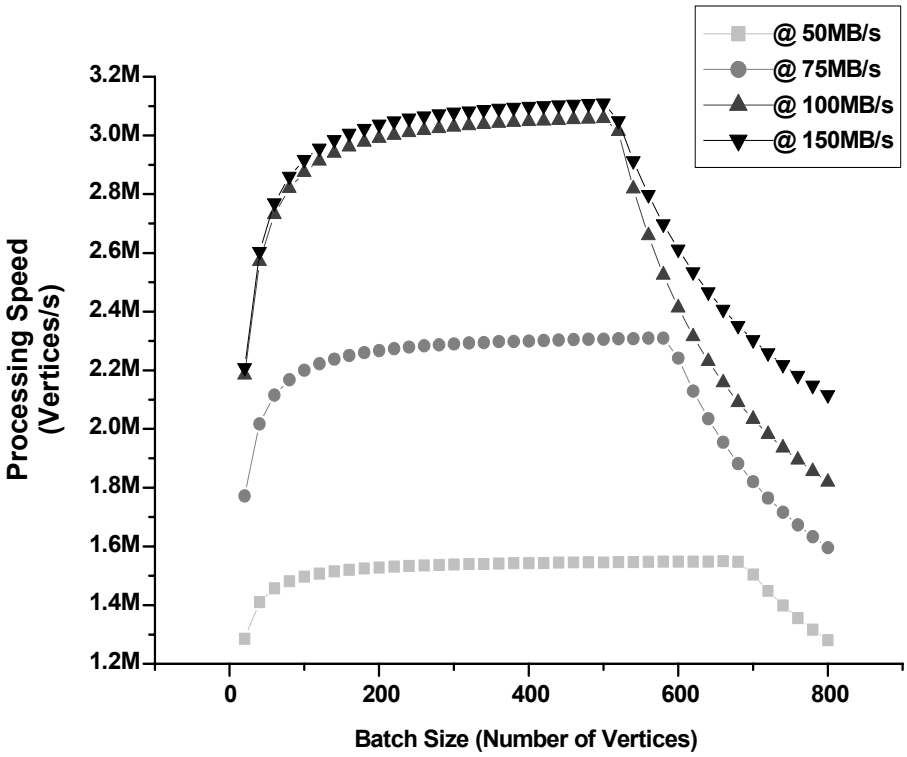
[Figure 2.6-2: Performance versus Capacity of Local Memory]



[Figure 2.6-3: Performance versus Bandwidth]

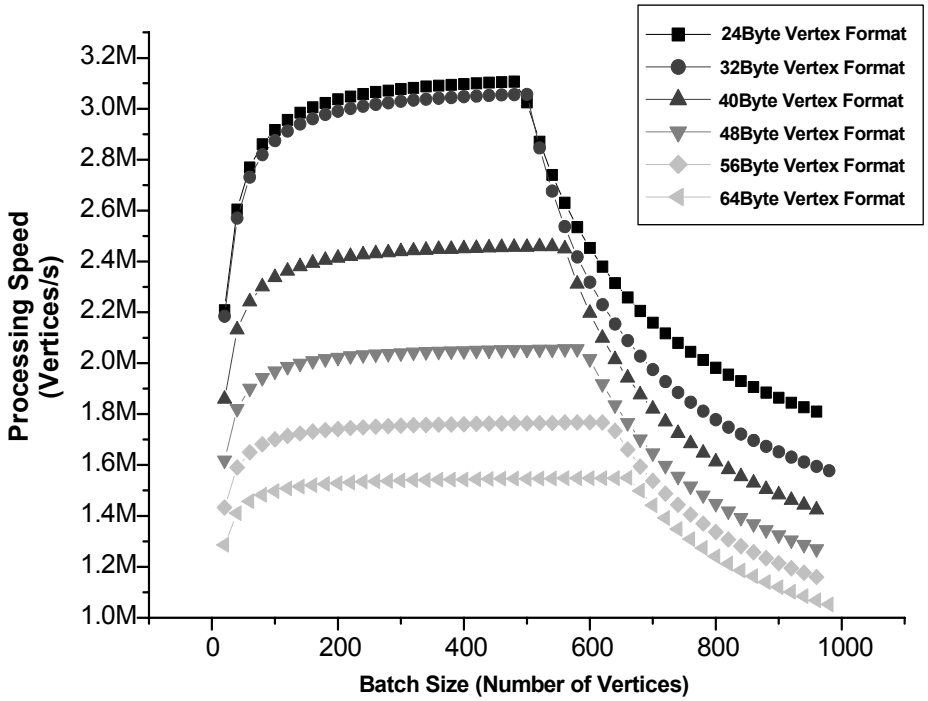
seamlessly while executing current vertex. The simulations recognizes that the memory bandwidth more than about 70MB/s is sufficient for vertex streaming in mobile applications.

Figure 2.6-4 shows the relationship between the batch size and the performance with varying the memory bandwidth. As the batch size is increased, the performance is also increased due to distributions of initialization and epilogue time over each vertex in batch. More increase of the batch size, however, causes the decrease of performance because local memory cannot sufficiently buffer all vertices in the batch. Since smaller batch size yields lower memory transfer time (T_d) which is more easily hided by processing time (T_b), the optimal batch size is inversely proportional to the bandwidth as illustrated in the figure.



[Figure 2.6-4: Performance versus Batch Size]

Finally, Figure 2.6-5 shows the relationship between the performance and batch size with varying the format of vertex data. When using the more smaller format by geometry compression, the graphics processor provides more higher performance.



[Figure 2.6-5:Performance versus Batch Size with Various Vertex Format]

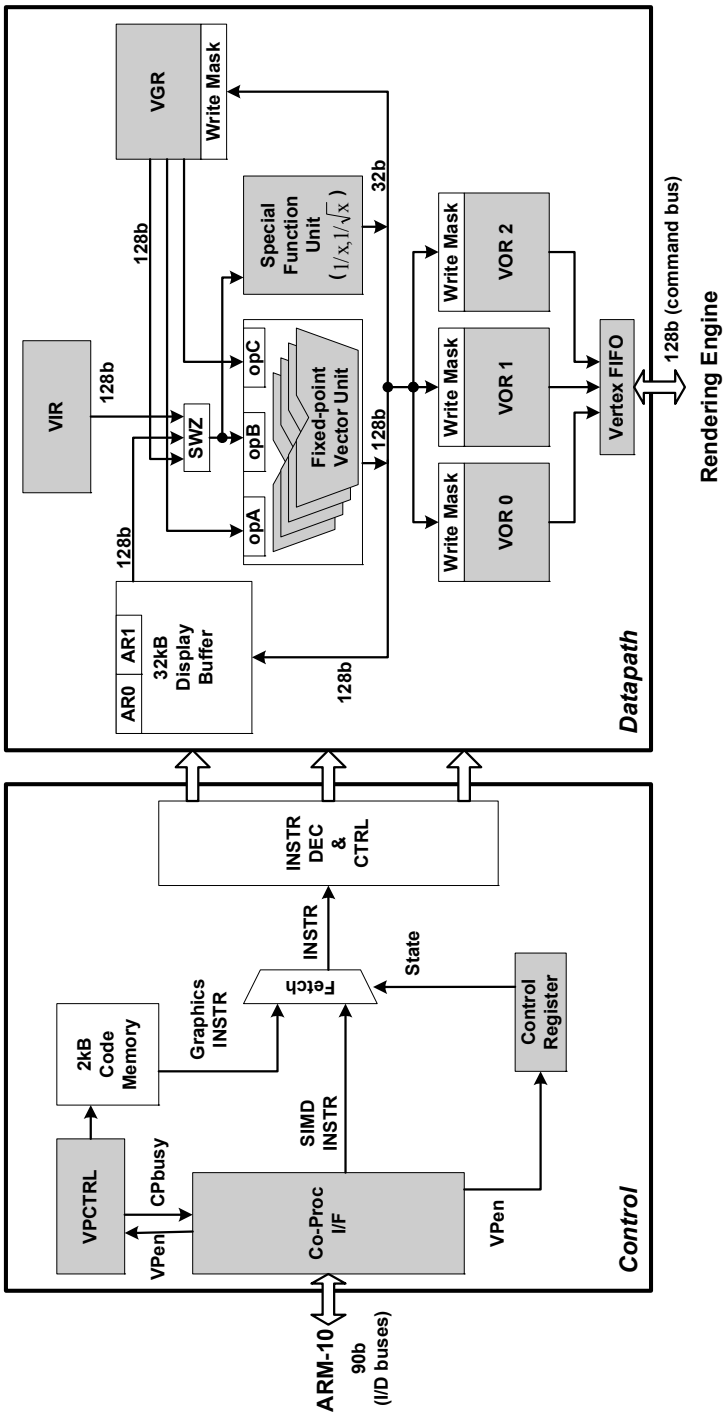
CHAPTER 3
Design of Graphics Processor

3.1 Fixed-point SIMD Vertex Shader

3.1.1 Internal Architecture

Figure 3.1-1 shows the user-programmable fixed-point SIMD vertex shader implemented in the graphics processor [34][35]. The vertex shader is a 128-bit 4-way SIMD ARM10 coprocessor, and it consists of two parts — control and datapath. In the control part, there is a 2kB code memory that stores vertex program codes of graphics instructions. Vertex program control unit (VPCTRL) issues the graphics instructions without control of the ARM10 processor. The general SIMD instructions are transferred through the coprocessor interface and the contents of control register determine its operating state. The two operating states — the TCC state and the PP state share all of the hardware blocks except instruction fetch units.

In the datapath part, there is a fixed-point vector unit that is responsible for all SIMD arithmetic operations such as addition and multiplication. Special function unit (SFU) is responsible for reciprocal (RCP) and reciprocal square root (RSQ) operations. Most of the operations are performed in 32-bit fixed-point numbers, and achieve a single cycle throughput. For streaming graphics processing, the vertex shader contains multiple register files — input vertex registers (VIR), output vertex registers (VOR) and general SIMD registers (VGR). The input vertex



[Figure 3.1-1: Block Diagram of Vertex Shader]

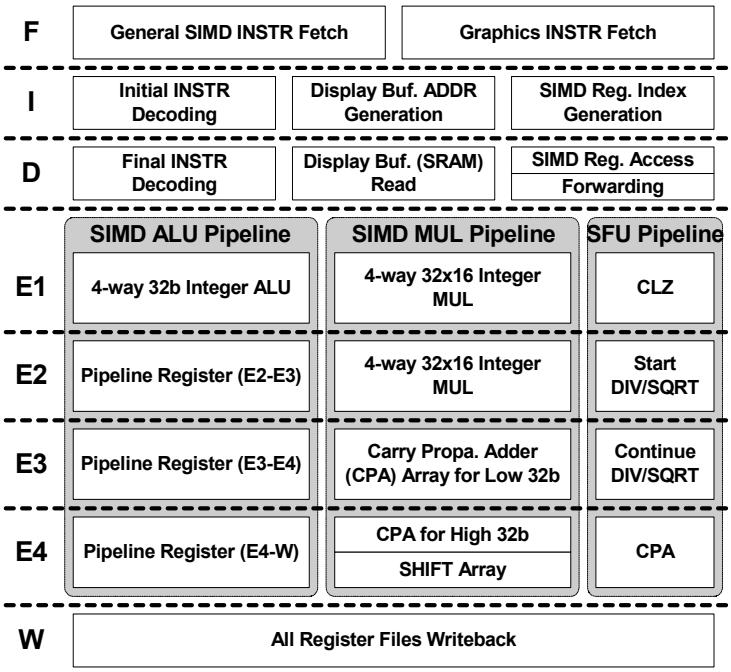
register file, used to hold the vertex attributes such as position and normal vector, is fed into the fixed-point SIMD datapath. The general SIMD register file is used to store temporary results during vertex program execution. The shaded vertex output is transformed into one of the output vertex register files. There are three output vertex register files for caching of vertex data in the primitive assembly and only one of them is accessible in the vertex program. The vertex shader has the display list buffer, implemented by 32kB synchronous SRAM as local memory, to store graphics primitives such as vertex data, reducing the traffic on external memory I/O. Also, the display list buffer can be shared to hold graphics constants at the same time for design simplicity of hardware. To enhance the efficiency of addressing and to avoid the conflicts when accessing display list buffer, the vertex shader has two integer address registers for indexed display list buffer reads. In addition, the display list buffer has the following two features.

(a) Auto increment and decrement addressing modes: the address register can be updated automatically after indexed display list buffer reads, which is useful to manage the vertex streams.

(b) 8 bit / 16 bit unpack with shuffling of vector components: For geometry compression, the 8 bit or 16 bit read data from display list buffer can be sign-extended to the full 32 bit fixed-point numbers, which can be used as the delta difference between one vertex and the next vertex [36]

The eight-stage single-issue pipeline of the vertex shader is illustrated in Figure 3.1-2 The fetch stage transfers one of the general SIMD instructions and the graphics instructions from the coprocessor interface and the code memory, respectively, to the control unit. For programmable shading, operands of the SRAM display buffer and the SIMD register files are accessed at the same time in the decode stage. The SRAM address is generated in the early stage of pipeline, the issue stage. In the execute stage, there are three separated pipelines:

SIMD arithmetic-and-logic (ALU) pipeline, SIMD multiply (MUL) pipeline and SFU pipeline. By using 4-way 32-bit integer multipliers with integer shifter arrays for fixed-point conversion, single-cycle throughput for fixed-point multiply-and-accumulate (MAC) operations can be achieved. To reduce the design complexity, register-forwarding logic between pipeline stages is used only in the general SIMD register file.



[Figure 3.1-2 Pipeline Structure of Vertex Shader]

3.1.2 Instruction Set Architecture

In the graphics processor, the two separate instruction sets — the general SIMD instruction set for the TCC state and the graphics instruction set for the PP state are implemented. The instruction set in the TCC state contains all data processing and movement instructions for the vertex shader. These instructions can accelerate various multimedia functions such as MPEG4 video besides 3D graphics. In the PP state, the instruction set consists of 20 operations, which are the modified subset of today's programmable vertex engine [37]. All these instructions utilize

the fixed-point arithmetic except integer shift instructions for index calculations. Table 3.1-1 shows the instruction set of vertex program that can be executed in the PP state. In the programmer's view, the PP state instructions are the subset of the TCC state instructions with graphics extensions such as source swizzling and write-masks. That is, one input vector operand can be swizzled arbitrarily in the SIMD datapath and all the output writes can be controlled by component-wise write mask bits. Moreover, in the PP state, there are more options for input and output operands such as VIR, VOR and display buffer memory, while the TCC state allows only VGR for input and output operands.

Opcode	Full Name	Description	Latency	Throughput
MUL	Multiply	Vector→Vector	4	1
MAD	Multiply and Add	Vector→Vector	4	1
DP3	3-term Dot Product	Vector→Replicated Scalar	5	2
DP4	4-term Dot Product	Vector→Replicated Scalar	5	2
TRFM	Transform	Vector→Vector	7	4
ADD	Addition	Vector→Vector	1	1
SUB	Subtraction	Vector→Vector	1	1
MOV	Move	Vector→Vector	1	1
RCP	Reciprocal	Scalar→Replicated Scalar	6	3
RSQ	Reciprocal Square Root	Scalar→Replicated Scalar	8	5
MIN	Minimum	Vector→Vector	1	1
MAX	Maximum	Vector→Vector	1	1
SLT	Set Less Than	Vector→Vector	1	1
SGE	Set Greater Than or Equal	Vector→Vector	1	1
SEQ	Set Equal	Vector→Vector	1	1
LSL	Logical Shift Left	Vector→Integer Vector	1	1
ASR	Arithmetic Shift Right	Vector→Integer Vector	1	1
ZERO	Set Zero	Vector	1	1
ARL	Address Register Load	Vector→Integer Scalar	2	2
END	Vertex Program End	Miscellaneous	1	1

[Table 3.1-1: PP State Instructions for Vertex Program]

In the TCC state, the control flow instructions such as branch and return are managed by the main processor and the vertex shader provides only the extended SIMD arithmetic instructions. However all the vertex shader instructions can be conditionally executed like conventional ARM instructions. When implementing the fixed function pipeline of graphics library such as OpenGL which is controlled by global states, the state checking, vertex shading path selection, homogeneous clip space operations and back face culling are handled in the TCC state. The remaining code segments for actual vertex shading operations can be executed without state checking. These operations are carried out in the vertex program of the PP state, which supports the vertex transform paths without branching for simplicity and efficiency of hardware architecture. Even if the control flow instructions are not supported in the PP state, simple if/then/else statement is still possible through SLT, SGE and SEQ instructions.

To save the system resources, the datapath is made simple and efficient without complex hardware blocks. All arithmetic operations including RCP and RSQ are executed on the fixed-point numbers that can have any precisions, and only low power integer arithmetic units are used. They are also fully pipelined, and there is a bypass logic to forward the data to the different stage of pipeline of correct instructions. And, I removed the complex functions such as the logarithmic, exponential and specular power functions, and rather the table look-up is used for these functions. The integer shift instructions of fixed-point numbers are added in order to extract bit fields for index calculations in the lookup table. After shift operations of vertex specific index, the ARL instruction can allow an offset into the lookup table.

The following vertex program (Figure 3.1-3) implements the vertex transformation and full Phong shading. It uses OpenGL lighting equations with assumption of infinite light and viewpoint positions. To calculate the specular

```
# Vertex Transformation and OpenGLLighting
#
# c[0-3]     = modelview matrix (column-wise)
# c[4-7]     = modelview inverse transpose (column-wise)
# c[8-11]    = modelview projection matrix (column-wise)
# c[16]      = light position
# c[17]      = blinn halfway vector
# c[18]      = precomputed specular light * specular mat.
# c[19]      = precomputed diffuse light  * diffuse mat.
# c[20]      = precomputed ambientlight  * ambient mat.
# c[32-47]  = 64 entries lookup table for specular power (column-wise)
# c[48]      = 16th, 32th, 48th and 64th entries of lookup table
# c[49].x    = fraction bit length of fixed-point format
# c[49].y    = fraction bit length - 2
# c[49].z    = fraction bit length - 6
# c[49].w   = fraction bit length + 4
# c[50]      = (0, 1, 2, 3) in integer format

# Vertex transformation to eye space
TRFM VGR0.xyz, VIR[OPOS], c[0];

# Normal vector transform to eye space
TRFM VGR1.xyz, VIR[NRML], c[4];

# Vertex transformation to clip space
TRFM VOR[HPOS], VIR[OPOS], c[8];

# Compute normalized light direction
SUB  VGR0.xyz, VGR0, c[16];
DP3  VGR0.w, VGR0, VGR0;
RSQ  VGR0.w, VGR0.w;
MUL  VGR0.xyz, VGR0, VGR0.w;

# Compute N.L and N.H
DP3  VGR2, VGR1, VGR0;
DP3  VGR3, VGR1, c[17];
# Index calculation of lookup table for specular power function
ASR  VGR4, VGR3, c[49].y;
ASR  VGR5, VGR3, c[49].z;
LSL  VGR6, VGR5, c[49].z;
SUB  VGR6, VGR3, VGR6;
LSL  VGR5, VGR5, c[49].x;
LSL  VGR7, VGR4, c[49].w;
SUB  VGR5, VGR5, VGR7;

# table look-up
ARL  A0.x, VGR5.x;
SEQ  VGR7.x, VGR4, c[50].x;
SEQ  VGR7.y, VGR4, c[50].y;
SEQ  VGR7.z, VGR4, c[50].z;
SEQ  VGR7.w, VGR4, c[50].w;
DP4  VGR4, VGR7, c[A0.x+32];
DP4  VGR5, VGR7, c[A0.x+33];

# Compute specular power using interpolation
SUB  VGR5, VGR5, VGR4
MAD  VGR3, VGR5, VGR6, VGR4;

# Compute light color values
MUL  VGR5.xyz, VGR3, c[18];
MUL  VGR4.xyz, VGR2, c[19];
ADD  VGR5.xyz, VGR4, VGR5
ADD  VOR[COL0].xyz, VGR5, c[20];

# texture coordinate
MOV  VOR[TEX0], VIR[TEX0
END
```

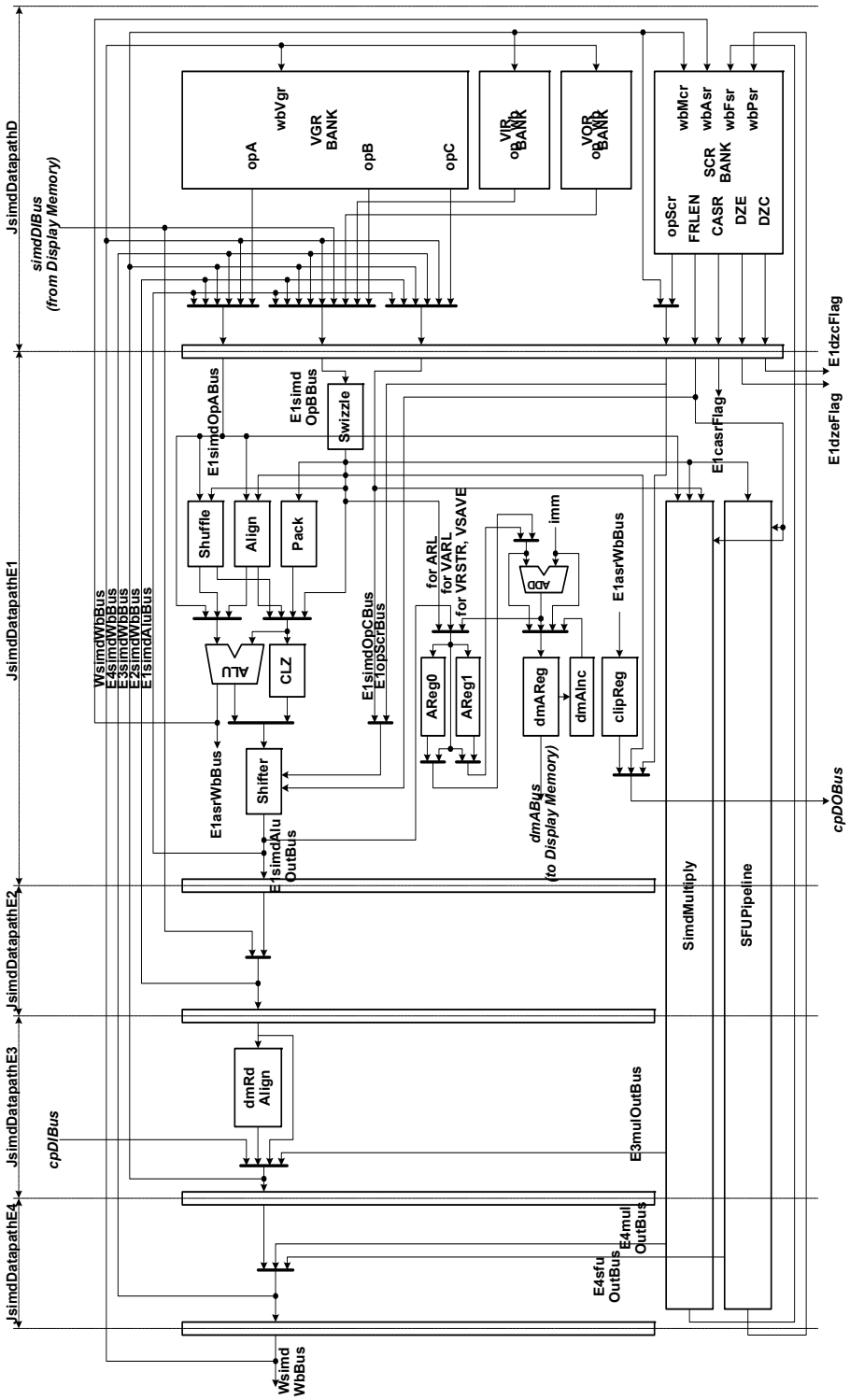
[Figure: 3.1-3: Vertex Program Code for Transformation and Lighting]

power function, I used the lookup table of 64 entries which store the specular coefficients for given shininess value. After calculating dot product of normal vector and the Blinn halfway vector, I used the integer shift instructions for offset values. After the rearrangements of the instructions and the eliminations of false dependencies, the vertex shader running at 200MHz can process these vertices at a rate of 3.6M vertices/sec including view frustum clip check, perspective divide and viewport transform.

3.1.3 SIMD Datapath Design

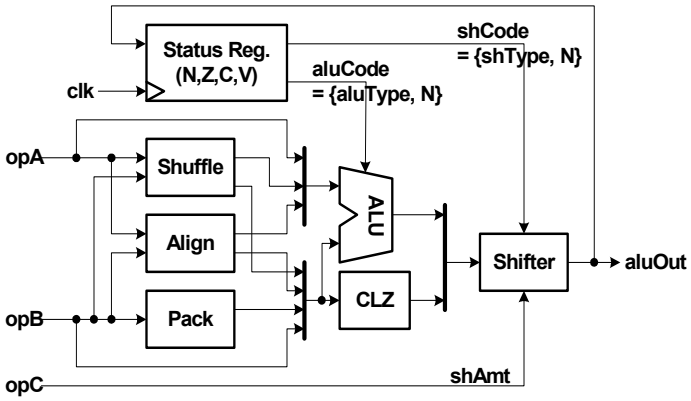
SIMD datapath of the vertex shader consists of SIMD ALU (arithmetic and logic unit), SIMD multiply engine, SFU (special function unit) and SIMD reigster files. Figure 3.1-4 shows the details of SIMD datapath with forwarding paths. Some of operations such as TRFM, DP3, DP4, RSQ and RCP consumes multi-cycles for completions, and decoder unit and finite state machine in the control part generates all necessary signals. SIMD control register bank (SCR) contains informations about processor states and arithmetic flags.

Figure 3.1-5(a) shows the SIMD ALU in the SIMD datapath. It can calculate all fixed-point arithmetic and logic operations including byte shuffle, data packing and operand alignment using only the integer adder and shifter. Although fixed-point arithmetic can provide enough performance in mobile 3D graphics, I designed efficient software floating-point emulations for more wider dynamic range by adding two special instructions — controlled ADD/SUB (CAS) and controlled logical shift (CLS). Control flow instructions such as if-then-else are frequently used in the programming of software floating-point arithmetic routines on conventional integer RISC processors. However, these control flow instructions decrease processing parallelism in SIMD datapath and require many operating cycles. The CAS and CLS instructions change the control flow instructions to

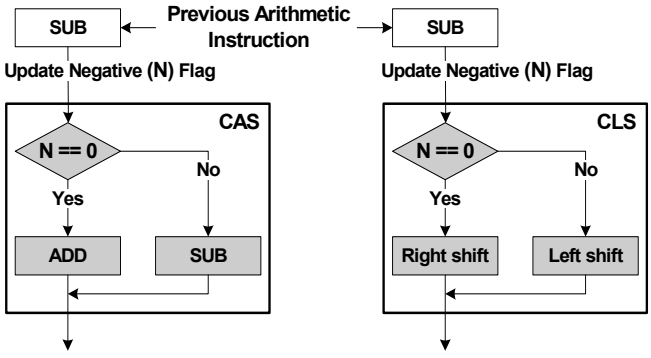


[Figure 3.1-4: Detail Block Diagram of SIMD Datapath]

single cycle SIMD arithmetic operations as shown in Figure 3.1-5(b). After negative flag in arithmetic status register is updated by previous instructions such as SUB, the CAS instruction can be made a single ADD instruction or SUB instruction. The CLS instruction can also made be a single left shift instruction or right shift instruction. These instructions can reduce the unnecessary comparison operations in exponent alignment and normalization of floating-point arithmetic With the floating-point emulation, the graphics processor shows 80MFLOPS peak floating-point performance at 200MHz operating frequency.

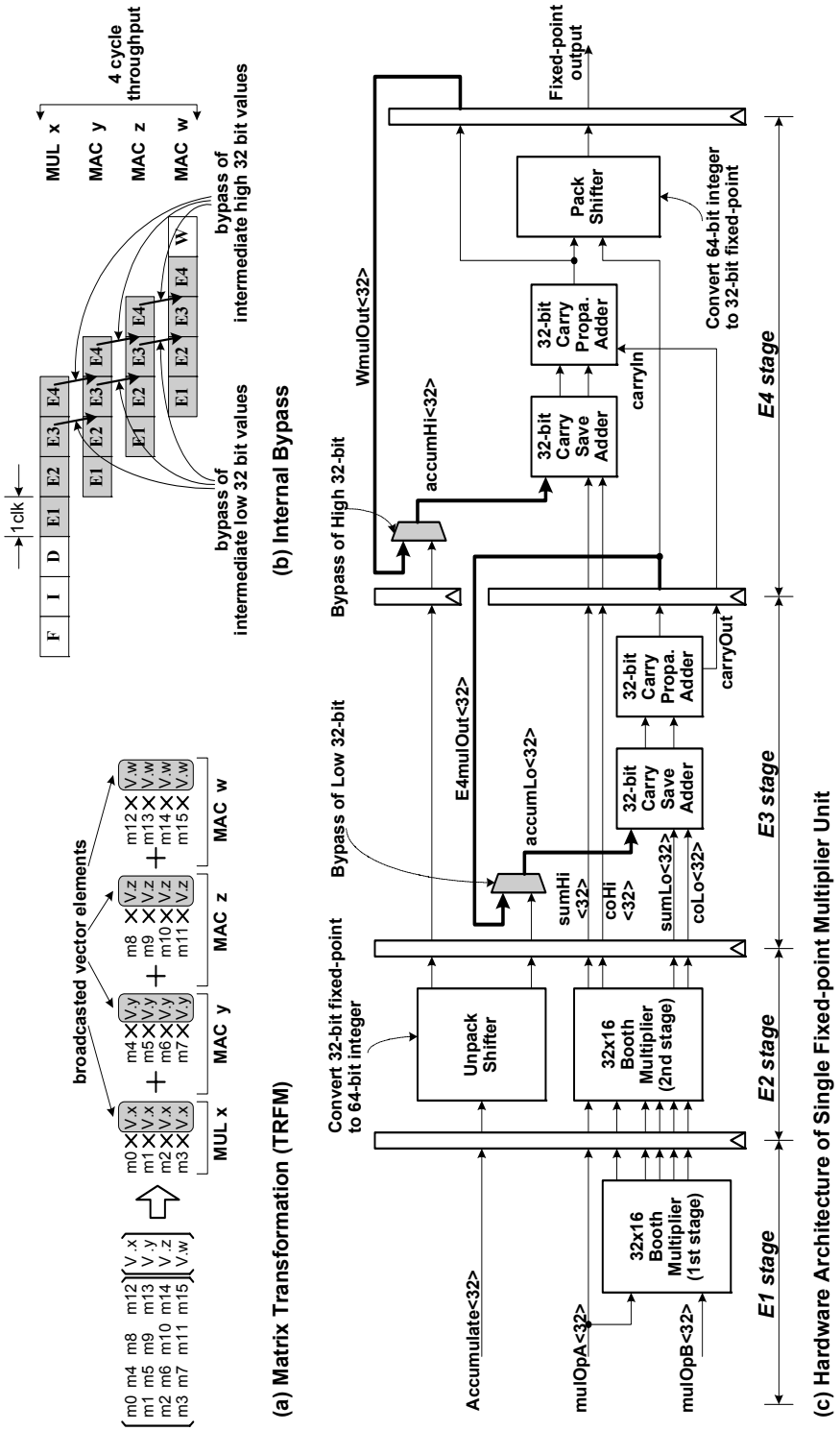


(a) Block Diagram



(b) Two Instructions (CAS,CLS) for Floating-point Emulation

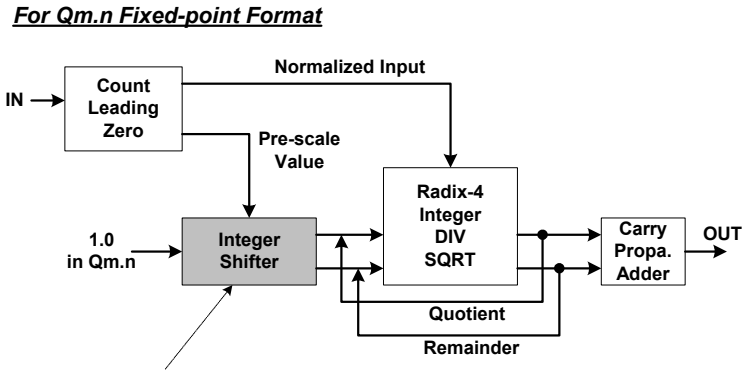
[Figure 3.1-5: SIMD ALU]



[Figure 3.1-6: SIMD Multiply]

Since multiplication-equivalent instructions spend most of time in graphics operations, the throughput of fixed-point MAC operations is designed as a single cycle. In addition, fast 4-cycle matrix transformation (TRFM) is implemented as shown in Figure 3.1-6. By broadcasting vector elements of input vertex, TRFM can be calculated by the first MUL and the following three MAC operations. However, fixed-point MUL and MAC operations require two cycle integer multiplications and two cycle integer additions, leading to 4-cycle latency. To resolve data dependency between theses MUL and MAC operations, it is allowed that intermediate value of the integer multipliers can be bypassed to accumulated input of the integer adders in the SIMD multiply engine (Figure 3.1-6(c)). By this scheme, the graphics processor shows 50Mvertices/s peak graphics performance for parallel projection at 200MHz.

SFU (Figure 3.1-7) calculates the square root and division by using 32-bit radix-4 combined integer division and square root unit. It calculates fixed-point result from fixed-point input number. Integer shifter in SFU pre-scales the input fixed-point number to intermediate 64-bit integer format before actual division and

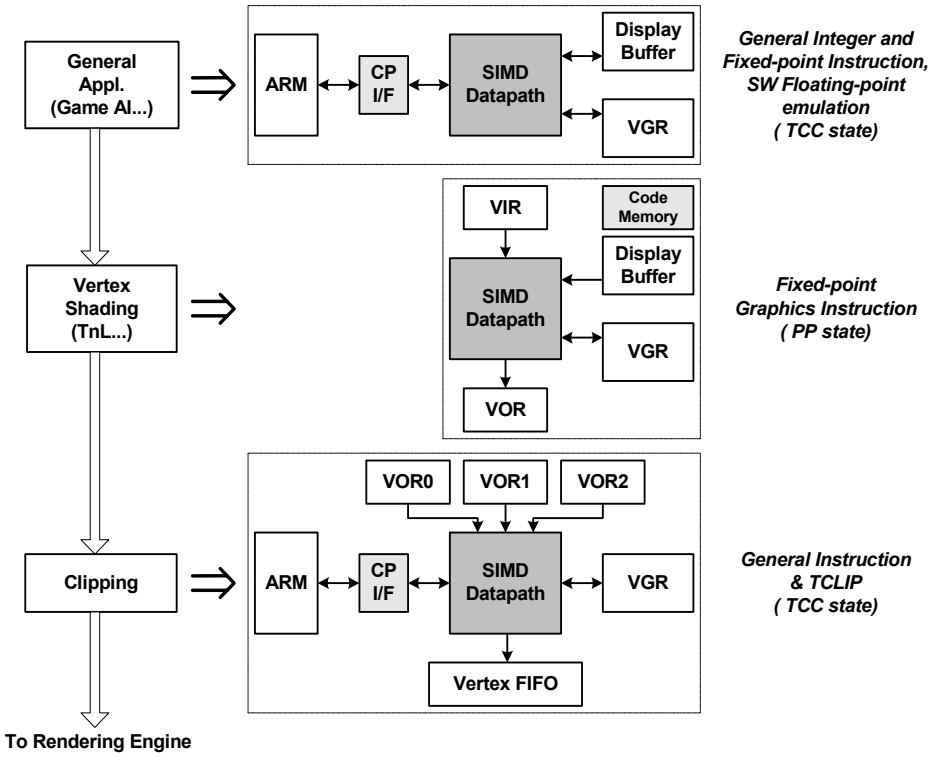


OP	Pre-scale Value
RCP	# of Leading Zero
RSQ	n/2 + # of Leading Zero

[Figure 3.1-7: SFU]

square root operations. Since output fixed-point number is 32-bit value, only MSB 32-bit of the intermediate 64-bit integer value is calculated after counting-leading-zero (CLZ) operation.

3.1.4 Operation Model



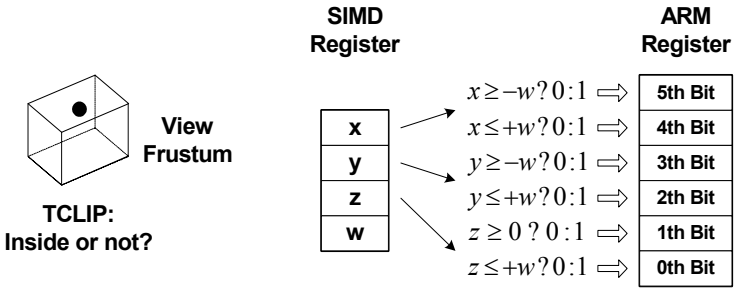
[Figure 3.1-8: Operation Model]

Figure 3.1-8 illustrates operation model of the vertex shader in geometry pipeline. General SIMD integer and fixed-point instructions of the TCC state can be used to program general applications such as artificial intelligence (AI) part of graphics game engine. The graphics parameters such as model-view matrix, camera movement and lighting information can be generated at this step. The efficient floating-point emulation enables calculations requiring more wide dynamic range while consuming less silicon area. Since the display buffer can be readable and writable in the TCC state, the vertex shader can move the vertex

model data in the display buffer to VIR for vertex shading while writing graphics parameters. After that, vertex shading operations exampled in the figure 3.1-3 can be performed by using vertex program instructions of the PP state. Vertex program call instruction in the TCC state changes the processor state and make the vertex shader issue graphics instructions stored in the code memory. After finishing the vertex program, the processor state of vertex shader is automatically changed back to the TCC state. At this time, the vertex shader performs primitive assembly such as polygon clipping. The TCC state contains a special instruction – TCLIP for accelerating polygon clipping by testing whether a given vertex is inside the view frustum in clip coordinates (see the figure 1.2-1). A point inside the frustum in clip coordinates satisfies the following conditions [38].

$$\begin{array}{ll} -w_c \le x_c \le +w_c & +w_c \le x_c \le -w_c \\ -w_c \le y_c \le +w_c \quad \text{for } w_c \ge 0 & +w_c \le y_c \le -w_c \quad \text{for } w_c < 0 \\ 0 \le z_c \le +w_c & +w_c \le z_c \le 0 \end{array}$$

The TCLIP instruction, which is mapped in ARM10's coprocessor register transfer instruction, generates a clip code into one of ARM10's registers from a input vertex stored in one of the general SIMD registers as shown in Figure 3.1-9(a). If the clip code is zero, the given vertex is inside the view frustum. Although the TCLIP instruction deals with the case of positive w only, ARM's conditional execution mechanism allows the clip code to be calculated in the case of negative w as well (Figure 3.1-9(b)). In the operation model, the transformed and lit vertex output is stored in one of there VORs after vertex shading. In the clip stage, the vertex shader first inspects the clip code of this vertex output, and then transfers it to the rendering engine through the vertex FIFO by using conditional execution mechanism only if the vertex is inside. Otherwise, interpolation of vertex across clip boundary of view frustum indicated by the clip code is performed before transferring the vertex output to the rendering engine.



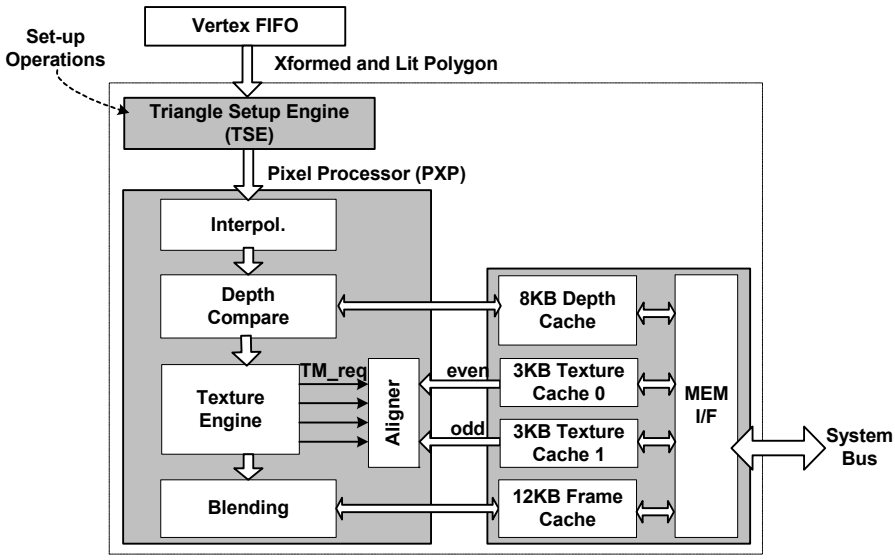
[Figure 3.1-9(a): TCLIP Instruction]

```
# VGR1: transformed and lit vertex output (x,y,z,w)
# R0: clip code

VZERO VGR0;                    // VGR0 = (0,0,0,0)
VSUB VGR2, VGR0, VGR1;         // VGR2 = (-x,-y,-z,-w)

#ARM's CPSR = w part of SIMD CPSR
TEXTC.w R15;

TCLIPLE R0, VGR1;              // in the case of positive w
TCLIPGT R0, VGR2;              // in the case of negative w
```

[Figure 3.1-9(b): Clip Code Calculation]

3.2 Rendering Engine

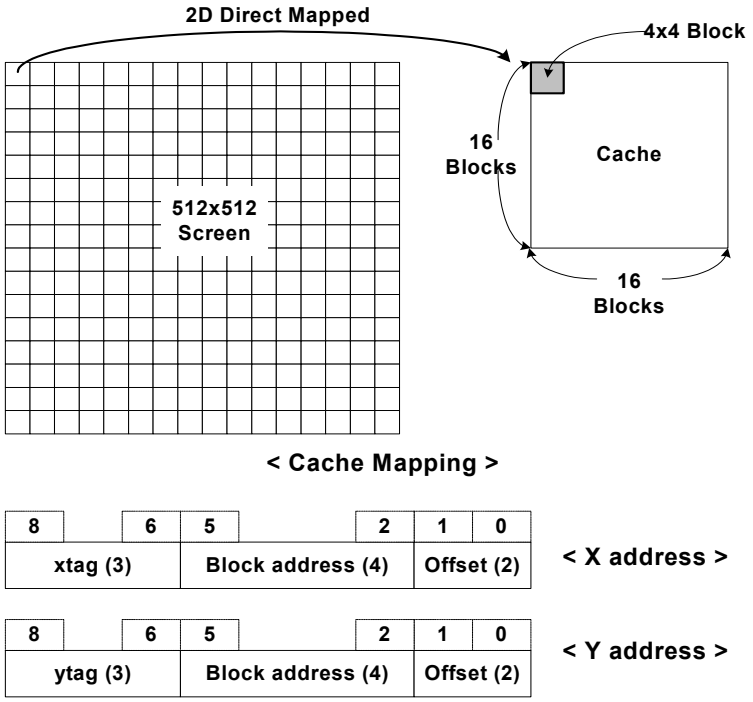
3.2.1 Internal Architecture



[Figure 3.2-1: Internal Architecture of Rendering Engine]

Figure 3.2-1 shows internal architecture of the rendering engine. It consists of a triangle setup engine (TSE), a pixel processor (PXP) and a graphics cache system. The TSE accelerates setup operations by sorting positions of the input triangles, and balances the 3D graphics pipeline between the rendering engine and the vertex shader. The PXP performs the main rendering operations such as shading, depth comparison, texture mapping and pixel blending.

The 26kB graphics cache contains frame, depth and texture caches, and stores frequently accessed pixel data. The frame and depth caches are direct-mapped caches in screen coordinates with two-dimensional array as illustrated in Figure 3.2-2, and show 97.9% and 98.8% average cache hit ratio for frame and depth buffer operations, respectively. In order to prevent conflicts in bilinear MIPMAP filtering, the texture cache are composed of two separate direct-mapped caches having the same screen-mapped coordinates with the frame and depth caches.



[Figure 3.2-2: 2D-screen Mapping in Graphics Cache]

Since the texture cache has separate memory for even and odd mip-level, it works effectively like as 2-way set-associative cache, achieving up to 96.5% hit ratio and average 20% power reduction compared to single direct-mapped cache.

The data access pattern of frame and depth caches are strongly related with rasterization order of the rendering engine in 2D screen, and the cache capacity in this work is relatively higher than in PC graphics system. Therefore, the hit ratio of direct mapped frame and depth caches is similar to associate caches. However, the well-known cache power model, CACTI [39], tells that the power consumption of two-associate cache is 50% higher than one of direct mapped cache in cache hit state. Therefore, the direct mapped cache can be more beneficial in design of depth and frame caches. But, in the case of texture cache design, the two-associate caches or separate direct-mapped caches show two-times smaller miss ratio than conventional direct mapped cache in bilinear MIPMAP texture filtering [40].

3.2.2 Instruction Set and Vertex FIFO

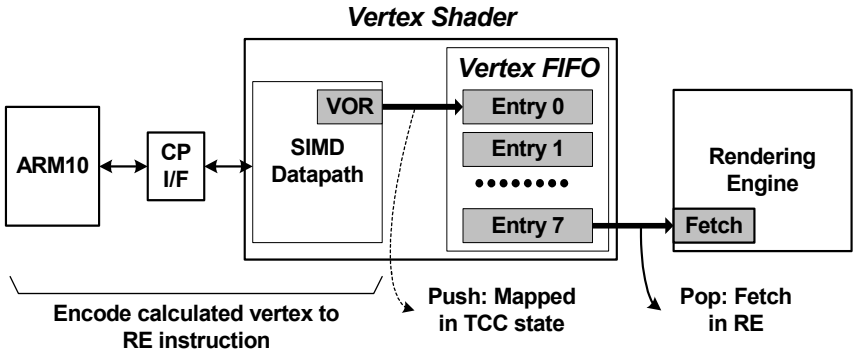
The rendering engine has its own instruction set to control the datapath and execute rendering program. Table 3.2-1 briefs the rendering engine instructions. Since the cycle consumed in transferring vertex data from geometry stage can be performance bottleneck in full graphics pipeline, the rendering engine is optimized to process all necessary information of vertex at every rendering cycle by RDAT instruction. It contains screen coordinates (*X, Y*), 16-bit screen depth (*Z*), color (*R, G, B, A*), and homogeneous texture coordinates (*u, v, 1/w*). Each color component is represented by 8-bit integer to support true-color rendering with alpha-blending. And each screen coordinate (X, Y) contains 9-bit integer to cover 512x512 screen resolution. The homogeneous texture coordinate is represented as 16-bit fixed-point format (8-bit integer + 8-bit fraction) to preserve necessary dynamic

Type	Mnemonic	Description
Rendering	RDAT TRI POS W U V X Y Z A R G B	**Fetch vertex data** TRI: Strip support 00: Intermediate vertex 01: End vertex POS: Reduce bandwidth 0100: 1st vertex 0010: 2nd vertex 0001: 3rd vertex W[16b]=1/W DATA0[16b,16b]=u:v DATA1[9b:9b:6b,8b]=X:Y:A:R DATA2[8b:8b:16b]=G:B:Z (A is valid only if TRI=01)
Texture	TMOD ADDR BLND FILT ID SIZE	**Set texture mapping mode** BLND[4b]: Blending mode 0001: Decal 0010: Modulate FILT[4b]: Filtering method 0001: Point sampling 0010: Bilinear filtering ID[8b]: Texture ID LOD[4b]: LOD Bias 0xxx: Normal mode 1AAA: Set LOD to A SIZE[12b]: Texture Size
Cache	CIVLD CACHE	**Set cache tag information invalid** Target[4b]: Target cache 0001: Depth cache 0010: Frame cache 0100: Texture cache
	CFLUSH CACHE	**Flush cache contents to main memory** Target[4b]: Target cache 0001: Depth cache 0010: Frame cache 0100: Texture cache
	MBASE CACHE ADDR	**Set base address for graphics memory** Target[4b]: Target cache 0001: Depth cache 0010: Frame cache 0100: Texture cache ADDR[32b]: Base address

Table 3.2-1: Rendering Engine Instructions

range and precision for texture calculation. The rendering engine also contains instructions for graphics cache management. Since depth buffer, frame buffer and texture memory are located in memory space of the ARM10 host processor, additional memory management are not required for the rendering engine. Instead, MBASE instruction is designed for setting base address of each graphics memory in internal register of the rendering engine, allowing address translations to be performed inside the rendering engine. And, instructions making caches invalid and flushing frame cache contents to frame buffer are added for initialization and finalization of rendering operations, respectively.

The vertex FIFO is implemented in the vertex shader to buffering vertex data between the vertex shader and the rendering engine as shown in Figure 3.2-3. Since the calculated vertex data is stored in VOR of the vertex shader, the ARM10 processor encodes this vertex data to the rendering engine instruction such as RDAT in collaboration with the vertex shader of the TCC state. Then, the ARM10 processor pushes it to the vertex FIFO by using queue insertion instruction mapped to coprocessor data processing instruction of the TCC state, which allows the ARM10 processor to continue executions of next instructions even in the case of queue full state. After that, the rendering engine can pop its instructions from the vertex FIFO through its fetch logic. If there is no more



[Figure 3.2-3: Vertex FIFO]

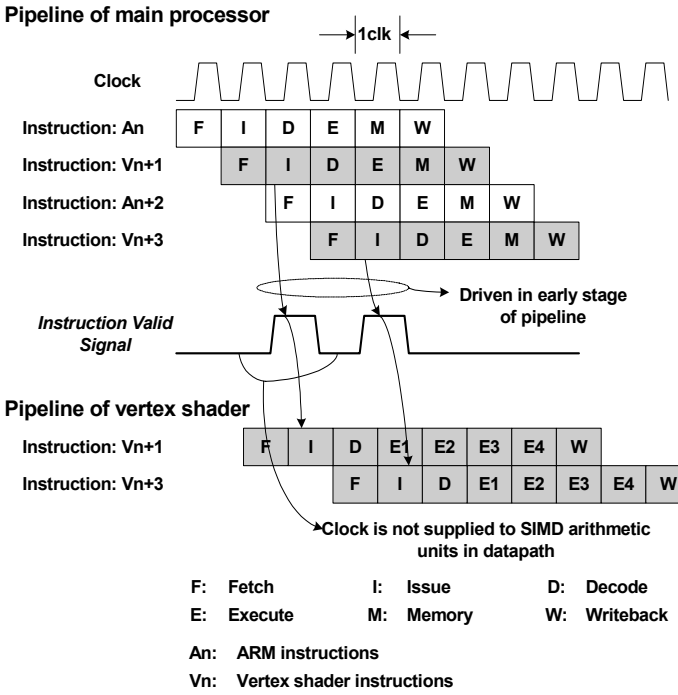
instruction in the vertex FIFO, the rendering engine waits for new instruction and stops its operations.

3.3 Low Power Techniques

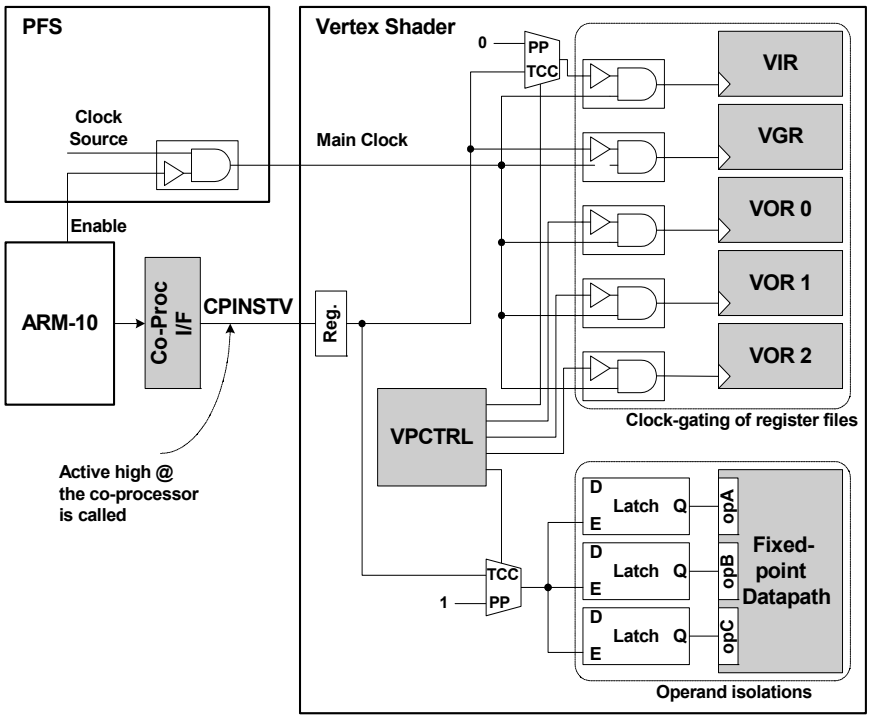
The implemented graphics processor controls its dynamic power consumption at both of micro-level and macro-level.

3.3.1 Instruction-level Power Management

For micro-level power management of the vertex shader, it implements instruction-level power management as shown in Figure 3.3-1. By the definition of the ARM10 coprocessor interface, the ARM10 processor must drive coprocessor instruction valid (CPINSTV) signal to the vertex shader only when the current instruction issued from the ARM10 processor is the valid vertex shader instruction. Using CPINSTV, the clock signals of the SIMD register files can be gated off when the write operations of the register files are not required. The read operations of the register files are still possible in the clock-off state. The write operations of the register files are performed in the writeback stage, and CPINSTV is valid in boundary between the issue and the decode stage of the vertex shader pipeline. Nevertheless, the vertex shader can operate reliably because pipeline registers hold register writeback values before writeback operations, and the register forwarding logic bypasses these valuesto correct input ports of arithmetic units. CPINSTV also reduces the power dissipated in the datapath of SIMD arithmetic units by eliminating the unnecessary signal transitions. Therefore, the coprocessor architecture shows fine-grained power management on an instruction-by-instruction basis. In the vertex shader, since the SIMD register files and datapath consume about 80% of power, about 47% activation ratio in calculating full 3D geometry operations achieves up to 43% power reduction.



(a) Clock-gating in Pipeline Stages

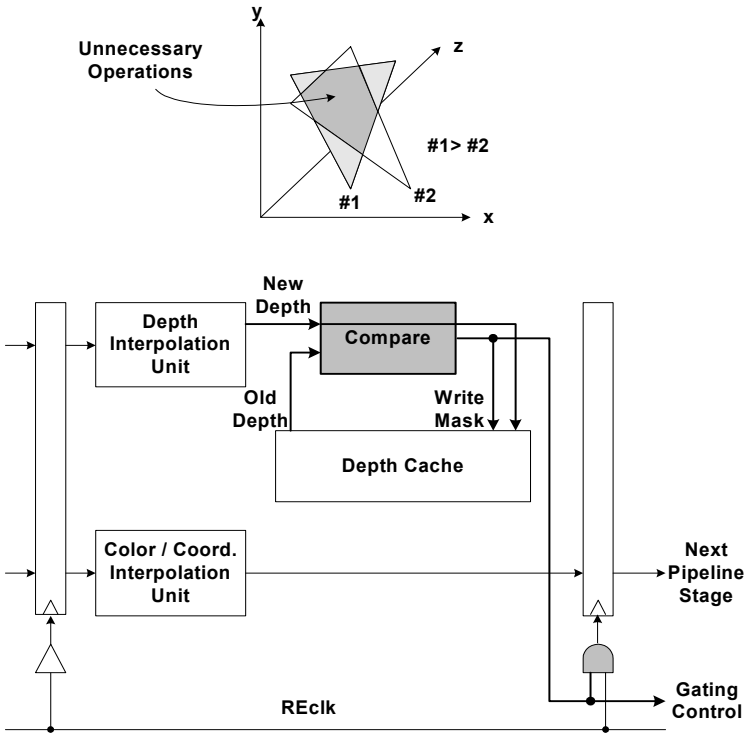


(b) Hardware Implementation

[Figure: 3.3-1: Instruction-wise Power Management]

3.3.2 Pixel-level Clock-gating

For micro-level power management of the rendering engine, it implements pixel-level clock gating. To reduce power consumption, the PXP in the rendering engine allows clock gating, which uses depth-compare results generated in early stage of rendering pipeline as shown in Figure 3.3-2. If a new pixel to be drawn is already covered by the pixels near from the viewpoint, the PXP does not need to process further. To use this property, the depth-compare unit is put into the earlier pipeline stage and the clock signals of the texture and blending units are gated-off to prevent unnecessary shading and texturing. It also reduces the power consumption of the graphics cache system by eliminating the unnecessary requests to each cache. For typical graphics applications that have the depth complexity of two, the pixel-level clock gating of the rendering engine shows average 25% power reduction [29].



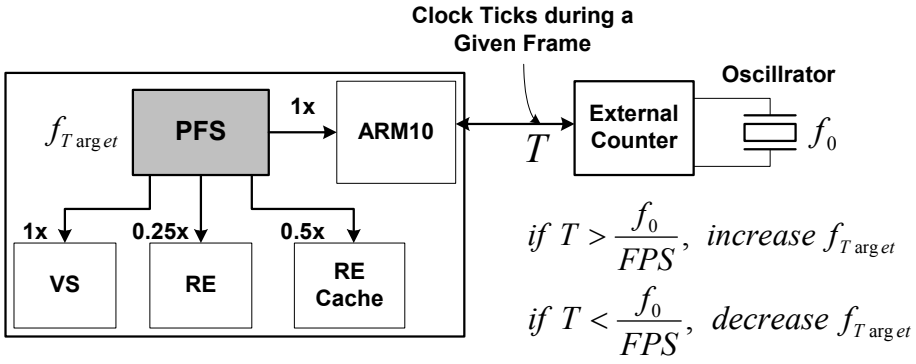
[Figure 3.3-2: Pixel-level Clock-gating]

3.3.3 Programmable Frequency Synthesizer

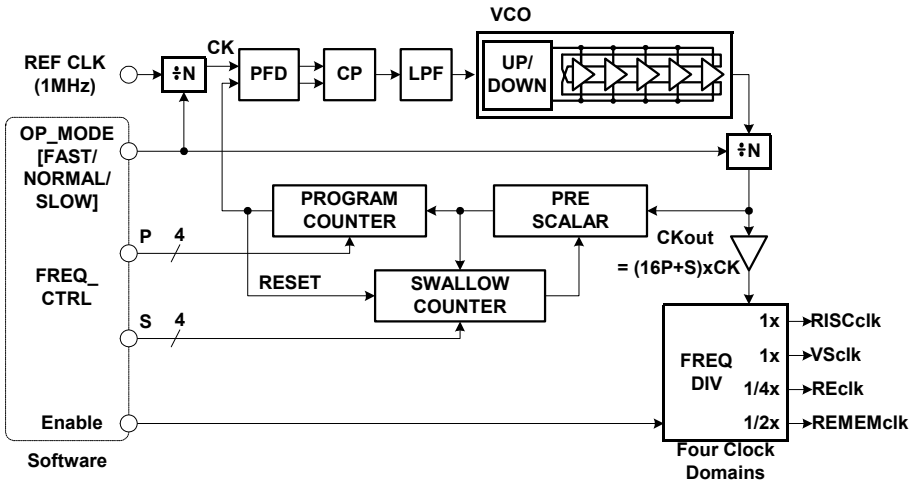
For macro-level power management, the graphics processor contains the programmable frequency synthesizer (PFS) as shown in Figure 3.3-3. Revised from the previous implementation that supported the only abrupt frequency change (2x, 1x, 0.5x) [13], this PFS can continuously and adaptively tune the target frequency with PLL-type frequency synthesizer. Once the operation mode is selected by OP_MODE (FAST / NORMAL / SLOW), FREQ_CTRL sets the target frequency adaptively. The frequency in FAST mode can vary from 32MHz to 271MHz with 1MHz step, NORMAL from 16MHz to 135.5MHz with 500KHz step, and SLOW from 8MHz to 67.75MHz with 250KHz step. The PFS is designed to cover wide frequency scaling range from 8MHz to 271MHz.

Since the 3D graphics applications are executed at a given frame rate, or FPS (Frame Per Second), finite amount of pixels should be drawn within the time slot of a single frame. Once the vertex shader and the rendering engine finish drawing pixels, their datapath do not need clocking for the rest of the time till restarting next frame. Therefore, the host software running on the ARM10 processor measures the average workload of the current frame, and sets the target frequency of PFS adaptively for the next frame. The power-management software counts the clock ticks from external counter when pixels of the given frame are completely drawn. Then, it compares the measured number of clock ticks with pre-determined value that is defined as the frequency of external counter's oscillator divided by required frame rate. If the drawing pixels are completed earlier than the preset threshold, then the software adjusts the chips's frequency for next frame to be slower, and resets the external counter before starting the next frame. For the case that the next frame requires more processing than the present, the software maintains 25% margin in workload monitoring to avoid unwanted slowing-down.

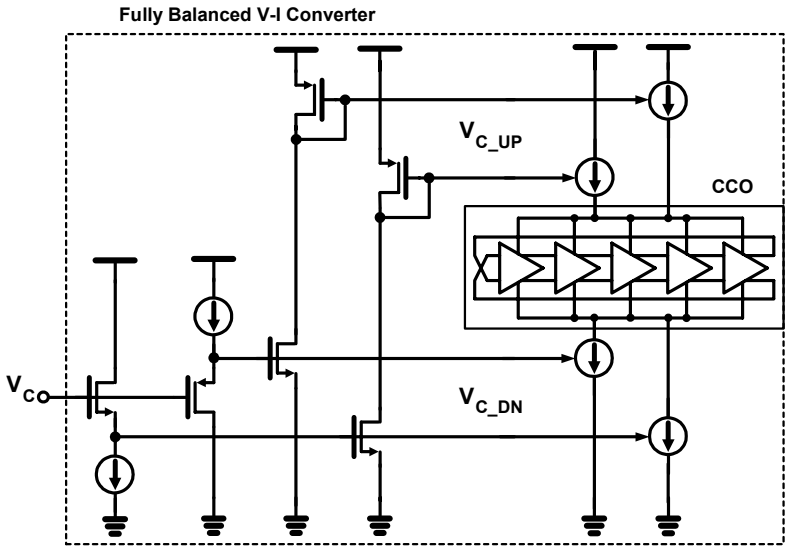
To cover wide frequency scaling range with high tolerance against process



(a) PFS System

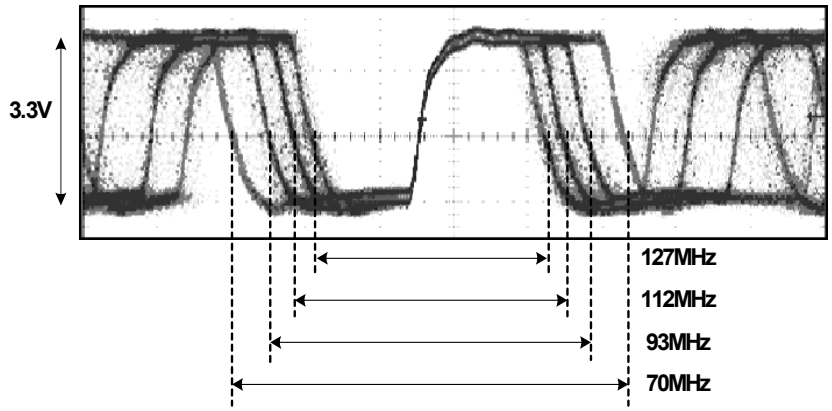


(b) PFS Block Diagram



(c) Fully Balanced VCO

[Figure 3.3-3: Programmable Frequency Synthesizer]



[Figure 3.3-4: Measured Waveform (RISCclk in NORMAL Mode]

variations, the PFS implements the fully balanced voltage-controlled oscillator (VCO) as shown in Figure 3.3-3(c). The proposed VCO consists of a fully balanced V-I converter and a current-controlled oscillator (CCO) with five delay stages. Each stage is designed as fully balanced differential configuration. The V-I converter converts the control voltage of VCO into complementary UP and DN control bias voltages that drive two separated bias current sources. The CCO minimizes the effects of power supply noise and substrate noise. The tuning range of the VCO is 350MHz and ensures wide linearity range and nearly constant gain over the rail-to-rail control voltage variation.

Adaptive variation of the clock frequency [41] is advantageous over the conventional clock gating, which pumps the clock tree at the maximum frequency and pause the clocks to the datapaths by gating off them abruptly after drawing the frame. Even if the datapaths are prevented from transitions, the spine of clock tree is kept pumped thus wastes power in the conventional clock gating.

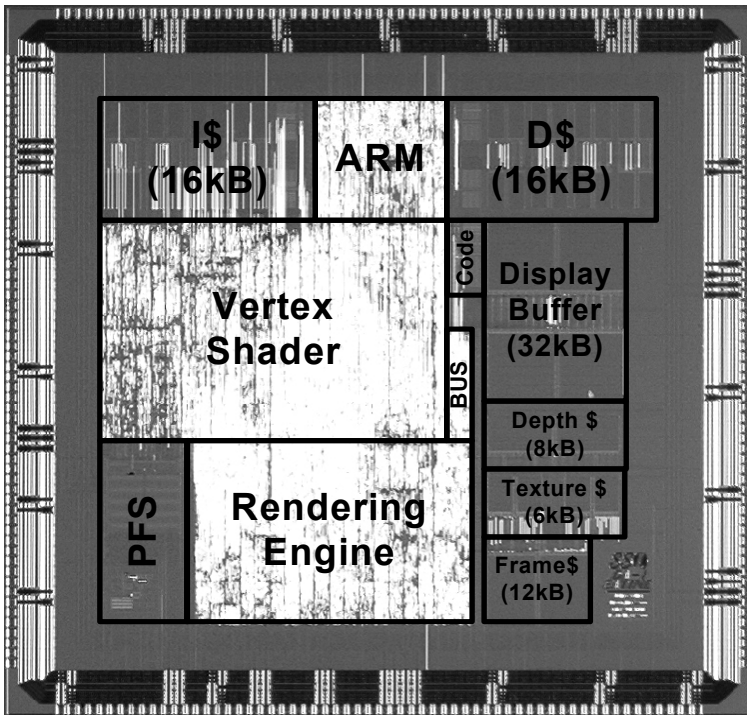
Although the frequency of the clock output (CKout) is continuously changed until being locked to the desired value, the chip can be reliably operated since all logics are designed with fully static circuits and the chip communicates with off-chip devices asynchronously. The PLL locking time is less than 50μs and it

consumes 2mW. Figure 3.3-4 shows the acquisition waveform of the PFS during frequency change in NORMAL mode. As shown in this measurement results, the PFS can provide the clock to each hardware block continuously without unwanted transitions.

CHAPTER 4
Chip Implementation

4.1 Implementation Results

The proposed graphics processor is fabricated in 0.18㎛ 6-metal standard CMOS logic process. The chip size is 36 mm^2 including 2M logic transistors and 96kB SRAM. Figure 4.1-1 shows the die photograph and Table 4.1-1 summarizes its features. By using this chip, various 3D graphics algorithms and other multimedia



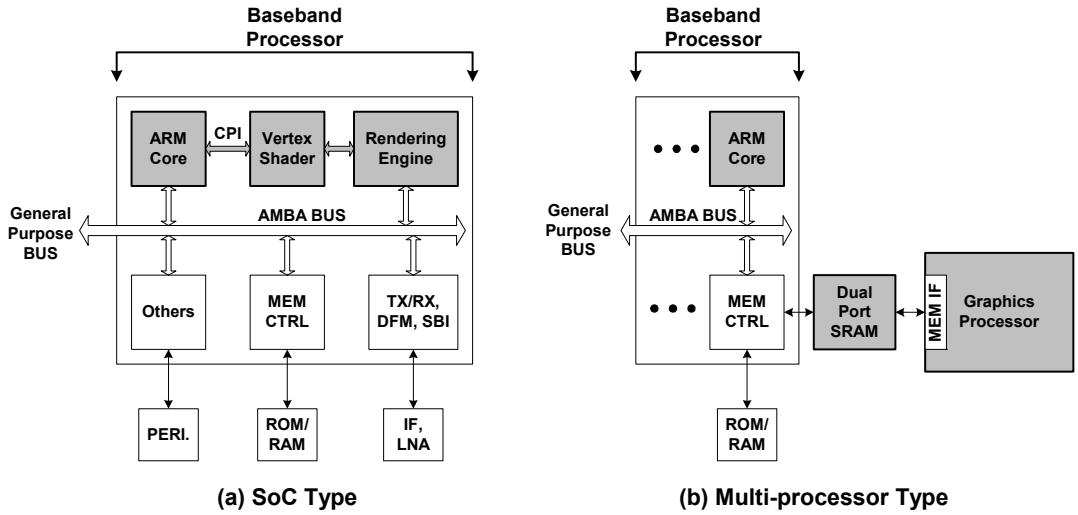
[Figure 4.1-1: Die Photograph]

Process Technology		0.18 um 6-Metal CMOS
Power Supply		1.8V(core), 3.3V(I/O)
Transistor Counts		2M Logic 96kB SRAM
Die Size		4.8mm by 4.8mm (core) 6.0mm by 6.0mm (chip)
Operating Frequency (ARM, VS / RE)		Fast : ~200MHz/50MHz Normal : ~100MHz/25MHz Slow : ~50MHz/12.5MHz
Power Consumption		<155mW
Package		256 pin PBGA
Performance	General	1000MIPS (ARM and vertex shader) 80MFLOPS(software emulation)
	Geometry	50Mvertices/s (Geometry transformation)
	Rendering	50Mpixels/s, 200Mtexels/s (Bilinear MIPMAP filtered pixel)
	Full 3D Pipeline	3.6Mpolygons/s (sustaining) (Including full OpenGL lighting, clip check and texturing)
Graphics Functions	Programmability	Vertex program version 1.1 compatible
	Screen Resolution	up to 512 x 512 pixels
	Triangle Setup	Hardware-accelerated triangle setup engine
	Shading	Gouraud / Flat
	Texture Mapping	Point/Bilinear MIPMAP filtering
	Antialiasing	x2, x4

[Table 4.1-1: Chip Characteristics]

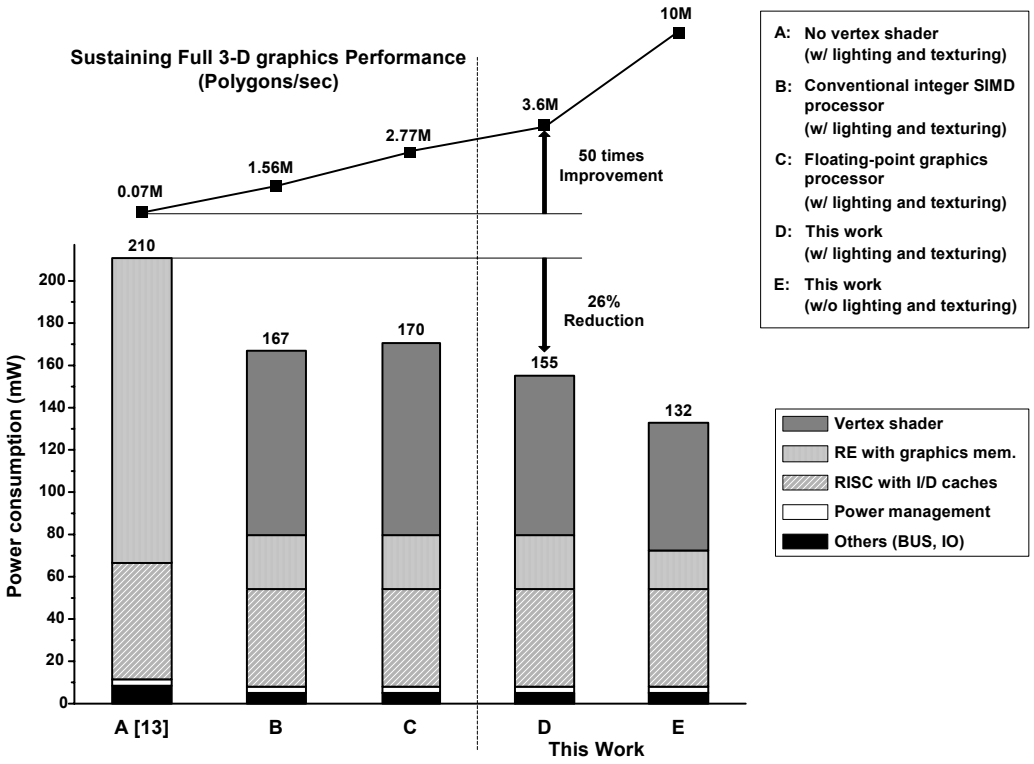
functions can be processed with 50Mvertices/s peak graphics performance, and 24-bit true colored and texture-mapped graphics images can be drawn at the speed of 50Mpixels/s and 200Mtexels/s

The coprocessor architecture of the proposed graphics processor can be easily adopted inside of ARM platform-based mobile SoC (Figure 4.1-2(a)). Or, its standard bi-directional asynchronous SRAM off-chip interface allows it to operate with any existing microprocessor or mobile system chipset. Figure 4.1-2(b) is the integration with existing application processor and baseband processor by utilizing dual-port asynchronous SRAM for shared memory between the graphics processor and host system.

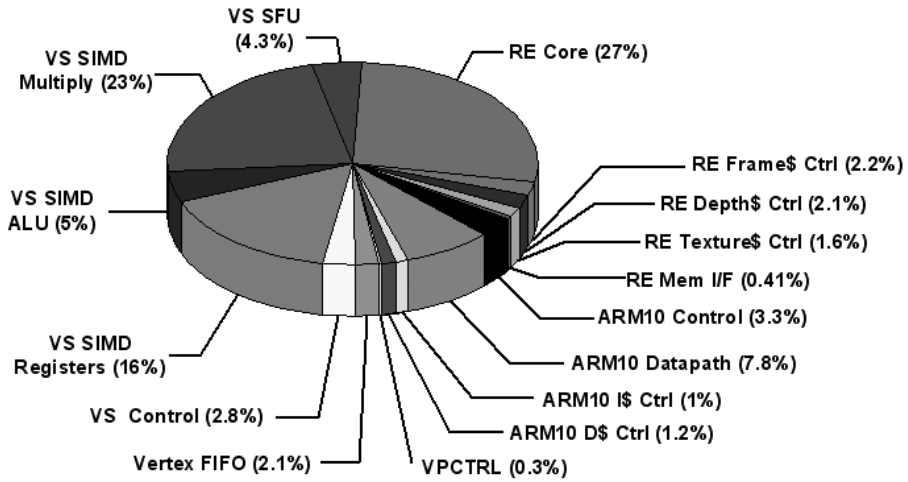


[Figure 4.1-2: Integration of Graphics Processor into Mobile System Chipset]

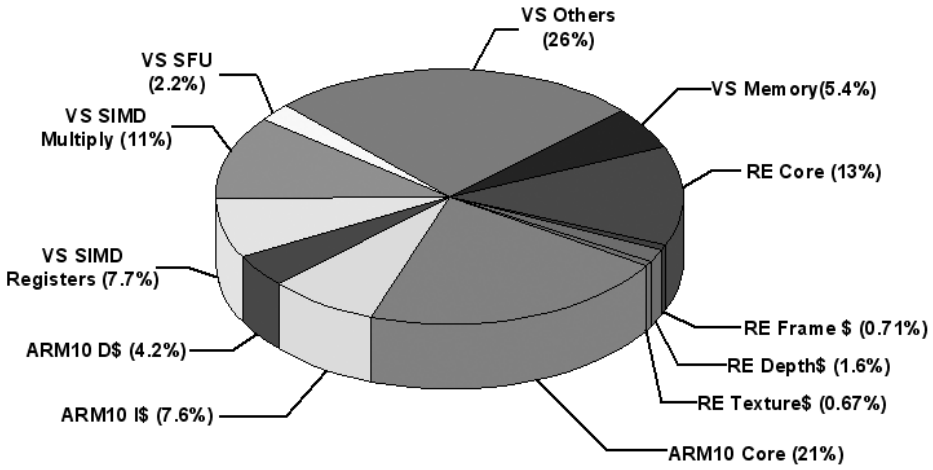
Figure 4.1-3 visualizes the system power consumption and overall full 3D graphics performance for various configurations of the graphics processors. The fixed-point graphics processing and the micro-level (instruction-level and pixel-level) power management reduce the power consumption by 26% compared to the previous implementation [13]. Moreover, parallel operations of the ARM10 processor and the vertex shader by dual operations increase the sustaining graphics performance about 50 times. The additional power dissipated by dual operations is as low as 3mW, because only simple instruction-fetch units are required and remaining hardware blocks are shared by the two operating states. The implemented graphics processor consumes 155mW in continuous calculation of 3.6Mpolygons/s full 3D graphics pipeline including geometry transformation, lighting, clip check, shading and bilinear MIPMAP texture mapping at FAST mode (200MHz RISCclk, VSclk, and 50MHz REclk). For unlighted and non-textured graphics applications, the power consumption is about 132mW and the performance is increased up to 10Mpolygons/s for sustaining input of vertex data. Figure 4.1-4 shows the area and power breakdown of the graphics processor.



[Figure 4.1-3: Performance and Power Consumption of Graphics Processor]



[Figure 4.1-4 (a): Gate Counts Breakdown of Graphics Processor]

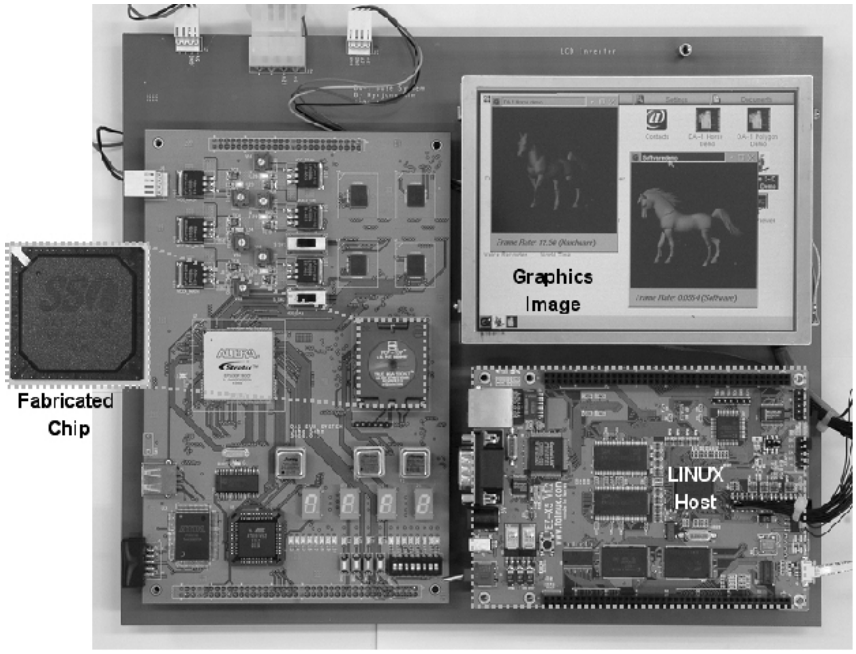


[Figure 4.1-4 (b): Power Consumption Breakdown of Graphics Processor during Operations of Full 3D Pipeline]

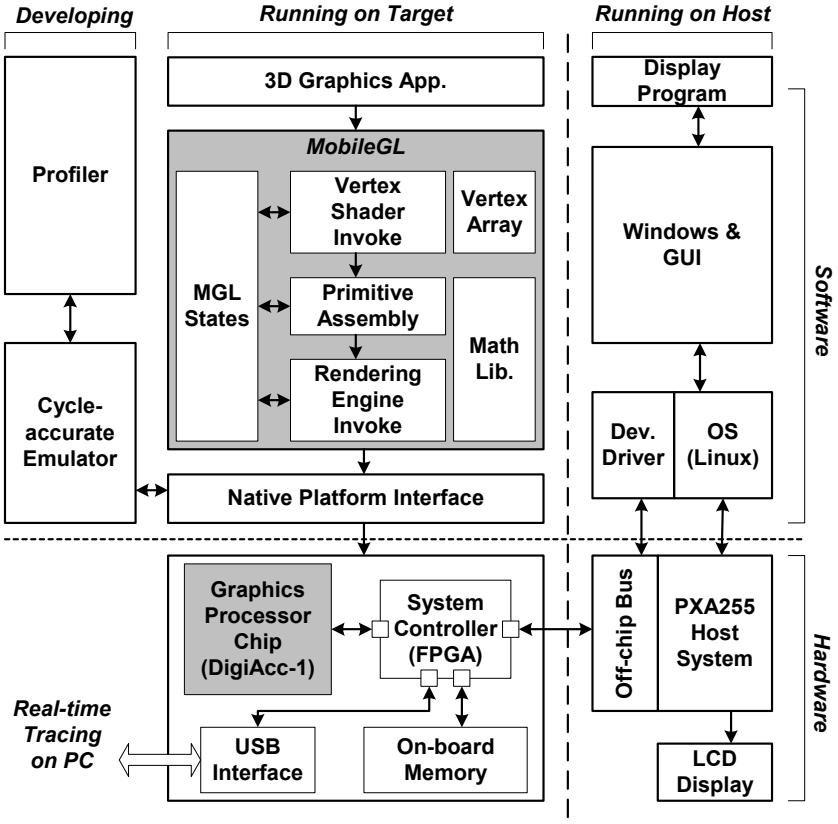
4.2 Evaluation Platform

System evaluation platform, called by REMY platform (Figure 4.2-1), was developed to evaluate and demonstrate mobile 3D graphics using a flexible topology and protocol. The REMY platform incorporates Intel's PXA255 host processor with embedded Linux operating system since the prototype chip doesn't implement subsidiary hardware blocks such as memory management unit and an LCD controller. The host system is used for displaying and accessing the target system while varying the configuration parameters such as external memory capacity and bus protocols. The hardware layer of the REMY platform contains the target system equipped with the fabricated chip and an FPGA system controller. The FPGA chip is responsible for emulating operations of dual-port SRAM and debugging the whole system.

The mobile graphics library, MobileGL, was implemented in the software layer of the REMY platform to simplify development of applications. MobileGL is an OpenGL-ES compatible graphics library optimized with hand-written assembly



(a) Demonstation Board (Full 3-D Operation with Lighting and Transformation)



(b) REMY Block Diagram

[Figure 4.2-1: System Evaluation Platform]

language to improve performance of an ARM-based mobile 3D graphics system. MobileGL consists of a fixed-point math library, vertex shader invocation routines, rendering engine invocation routines, primitive assembly, and state variables with vertex array capability. The native platform interface (NPI) provides intrinsic functions of hardware-dependent programmer's model in assembly and a high-level language for the core of the MobileGL. MobileGL can be ported to various hardware configurations without major architecture modifications by using NPI. The cycle-accurate software emulator of target hardware and the performance profiler were implemented in the REMY platform for performance evaluations and future derivative development.

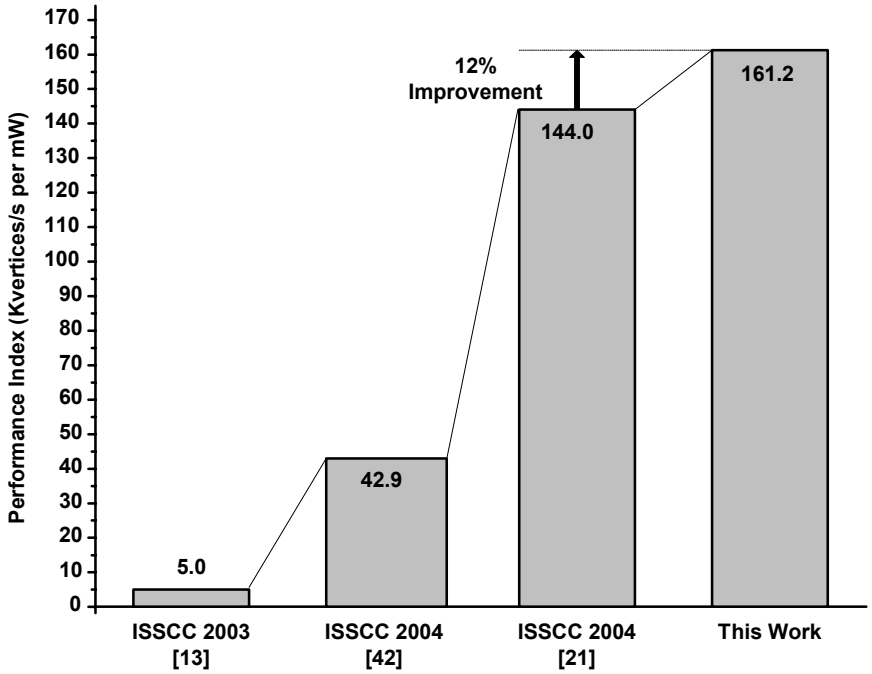
As shown in the figure, the fabricated chip was successfully demonstrated on the REMY platform while showing images of real-time 3D graphics.

4.3 Performance Comparison

The graphics performance in the mobile terminals cannot be compared directly in terms of processing speed such as vertex calculation rate because the power consumption must be taken into account as well. Although PC graphics hardware provides many advanced features with high calculation rate, the power consumption is too much to apply it to mobile platform. For the comparison of the implemented graphics processor with other previous architectures, the following performance index is used instead of only processing speed.

$$Performance\ \ Index = \frac{Vertex\ \ Processing\ \ Rate(Vertices/s)}{Power\ \ Consumption} = VXPS/mW$$

It is analogous to MIPS/mW in embedded RISC processor. Based on the graphics index, the proposed graphics processor shows 161.2kVXPS/mW as shown in Figure 4.3-1, which is significantly higher than other implementations.



[Figure 4.3-1: Performance Comparison]

Energy consumption is proportional to the number of memory access, so many researchers focus on reducing off-chip bandwidth to enhance the battery lifetime for mobile 3D applications. PowerVR's MBX architecture reduces the memory accesses with tile-based rendering, but the performance is limited by the system bus and the tiling overhead. Mitsubishi's Z3D core, intended for mobile phones, utilizes clock gating to achieve the lowest power consumption in spite of a floating-point geometry engine and 1Mbits embedded SRAM. However, its performance and functionality are constrained by the low operating frequency required by its limited power budget. The Playstation Portable (PSP), developed by SONY, contains all necessary hardware blocks required for various applications in a handheld video game system, including a MIPS processor with vector FPU, media processing unit, rendering engine and surface engine. The PSP features 2Mb of embedded DRAM to boost internal memory bandwidth and support Read-Modify-Write operations for 3D graphics. The rendering engine and surface

engine can execute more advanced graphics algorithm such as tessellation, skinning and morphing. The PSP also enables H.264 decoding for mobile video applications. However, the relatively high power consumption of the PSP limits its application in mobile terminals such as cell-phones. nVidia's SC10 provides complete hardware acceleration for mobile multimedia. It supports 2D/3D graphics and MPEG4 video with camera functions. The SC10 distinguishes itself from other architectures by implementing pixel-level programmability such as blending and combining operations for more realistic graphics images on handheld displays. However, the SC10 lacks a geometry engine for balanced performance. Table 4.3-1 summarizes the performance comparison and supported features of various graphics architectures.

The design consideration in the proposed graphics processor is to show the high energy-efficiency that is achievable by scaling and optimizing a processor's graphics functionality. The main design focus is on a simple programmable architecture optimized for mobile platforms, such as ARM processors, while achieving high performance with low power consumption.

Architecture	Hardware Acceleration	3D Feature	2D Feature	Integration Interface	Power Consumption	Performance	Area w/o PAD (@0.18um)	Graphics Index
PowerVR's MBX HR-S	Rendering + Geometry (Option)	Lighting, Multitexturing, Bump Mapping	N/A	AMBA	208mW @120MHz	2.5Mvertices/s 480Mpixels/s	16$_{mm^2}$	12.0 KVXPS/mW
Mitsubishi's Z3D	Geometry + Rendering	Texturing, Multilight, Shading	Rectangle Fill, Bit Block Transfer	Off-Chip Bus	38mW @30MHz	185Kvertices/s 5.1Mpixels/s	30$_{mm^2}$	4.9 KVXPS/mW
SONY's PSP		Surface Engine, Vertex Blending, Multitexturing	H/W H.264 Decode, AAC/MP3 Audio Codec	Stand alone	500mW @166MHz	35Mvertices/s 664Mpixel/s	<150$_{mm^2}$	70 KVXPS/mW
ATI's Imageon2300		Texturing, Vertex Fog	MPEG4 Decoder JPEG Codec	N/A	N/A @100MHz	1Mvertices/s 100Mpixels/s	N/A	N/A
nVidia's SC10	Rendering Only	Shading, Multitexturing	H/W MPEG4 Codec H/W JPEG Codec 64bit 2D Engine	Off-Chip Bus	75mW @72MHz	1Mvertices/s 72Mpixels/s	~70$_{mm^2}$	13.3 KVXPS/mW
This Work	Geometry + Rendering	Vertex Programming, Shading, Texturing	General Purpose Integer SIMD	ARM10 Coprocessor	155mW @200MHz	50Mvertices/s 50Mpixels/s	22$_{mm^2}$	161.2 KVXPS/mW
nVidia's GeForce 6800 (PC graphics)	Geometry + Rendering	Rich Vertex & Pixel Shading	N/A	AGP Bus	<100W	600Mvertices/s	<400$_{mm^2}$	>6 KVXPS/mW

[Table 4.3-1: Summary of Various Graphics Architectures]

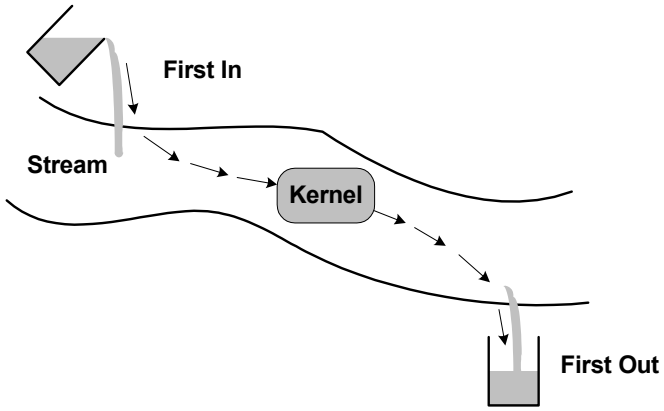
CHAPTER 5
Enhancing Stream Processing

5.1 Data Stream Architecture

5.1.1 Concepts of Stream Processing

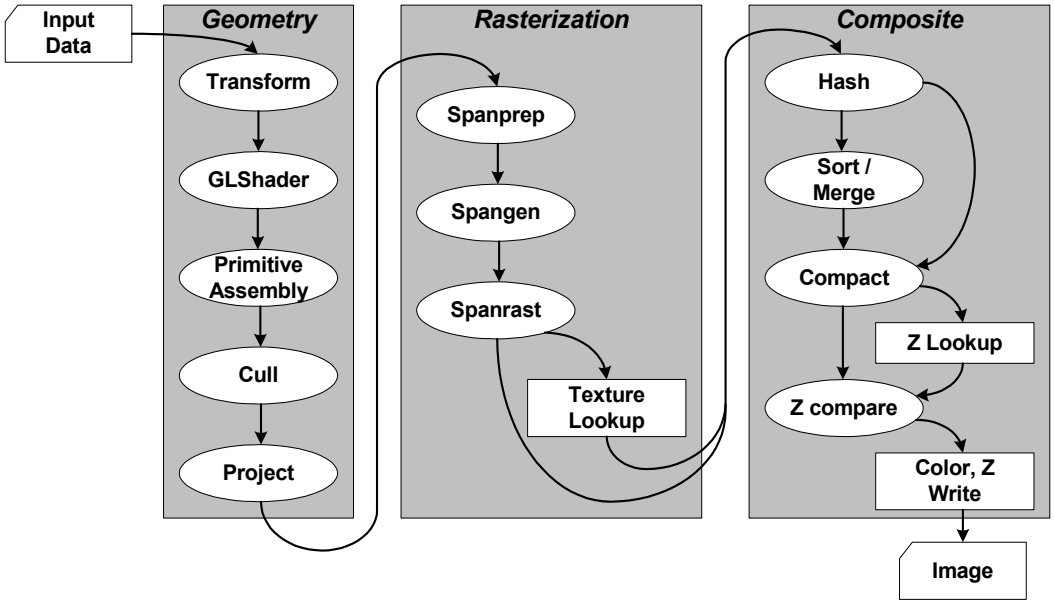
Improving technology of VLSI system makes performance of arithmetic units sufficiently high while bandwidth is still insufficient. Many architectures use cache system [43] or embedded memory [11-13][17] system for compensating bandwidth requirements. However, cache architecture cannot provide enough benefits in multimedia signal processing, in that, generally, primitives are processed once and then discarded. And, embedded memory architectures show low scalability and high physical design complexity, causing adaptation of screen and primitive size changes difficult. In a few years, stream processing is being implemented to exploit locality in signal processing [44-47]. In stream processing, data are organized as streams and all computations as kernels. A stream is defined as single type's collection of data records requiring same computation, and a kernel is defined as a function applied to each element in a stream. A stream processor executes a kernel over all elements of an input stream and places results into an output stream as illustrated in Figure 5.1-1. Therefore, it exploits data parallelism to make computing element busy as well as data locality to increase arithmetic intensity (the ratio of arithmetic to bandwidth) [45].

Producer-consumer locality occurs when one component of a system is producing



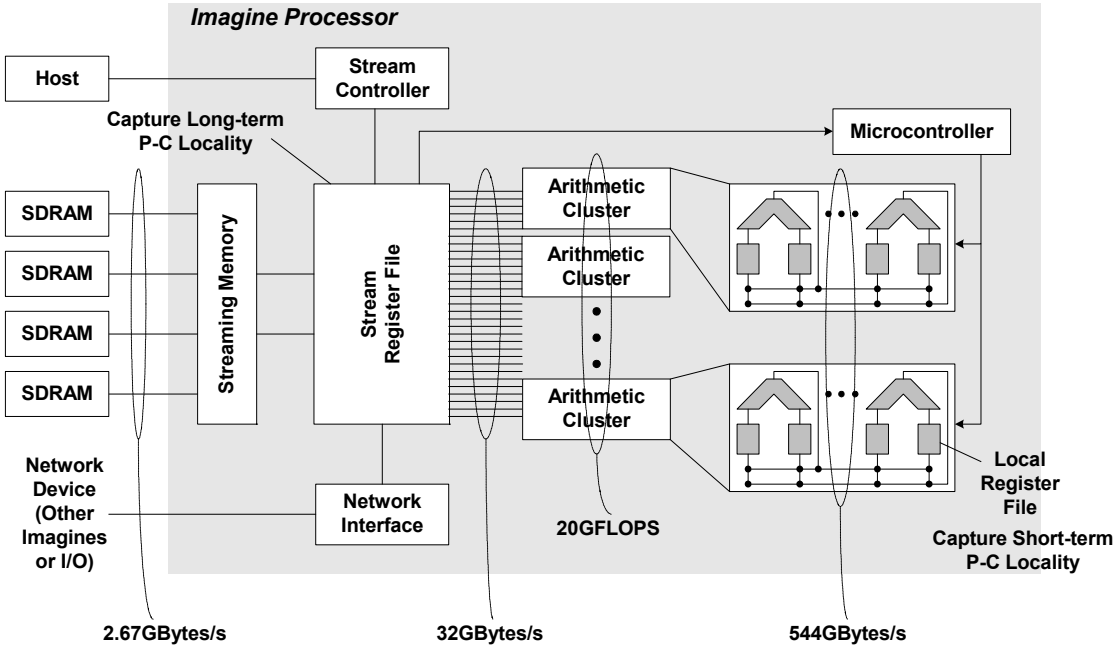
[Figure 5.1-1: Stream Processing]

something that is immediately consumed by another component of the system. This locality features multimedia signal processing itself, expecially 3D graphics. In the graphics pipeline, each stage generates output results that are immediately used by next stage as shown in Figure 5.1-2 [48]. The heterogeneous streams that require non-identical operations on each element by specified conditions such as culling can be splitted to separate homogeneous streams by conditions.



[Figure 5.1-2: Stream Representation of Graphics Pipeline]

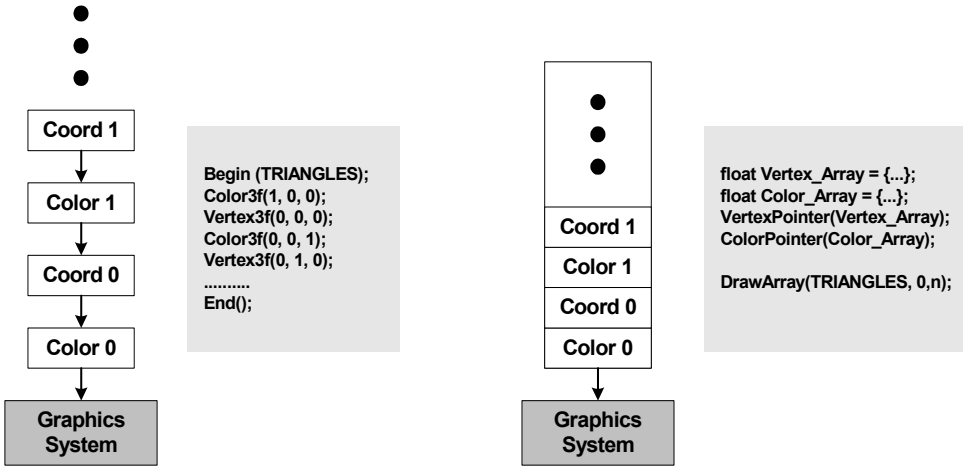
Figure 5.1-3 shows the Imagine Stream Processor developed by Stanford University [44]. In view of hardware implementation, stream processor requires two features – high throughput computing elements for data parallelism and hierarchical memory system for capturing producer-consumer locality. Large ALU clusters or SIMD computing elements execute stream kernels. Stream register file (SRF), organized by 128KByte SRAM and 22 stream buffers, captures producer-consumer locality generated by the ALU clusters. Only reduced global data are transferred through external SDRAM controller. The stream buffers stores temporarily fragments of generated streams from eight cluster ports, eight network ports, four external memory system ports, one microcontroller port and one host system port before accessing SRAM, which yielding high bandwidth for the ALU clusters. Although this massive architecture causes hugh power consumption, the concepts of stream processing should be taken into account for designing mobile graphics hardware for enhancing performance.



[Figure 5.1-3: Imagine Stream Processor]

5.1.2 Stream Processing in 3D Graphics

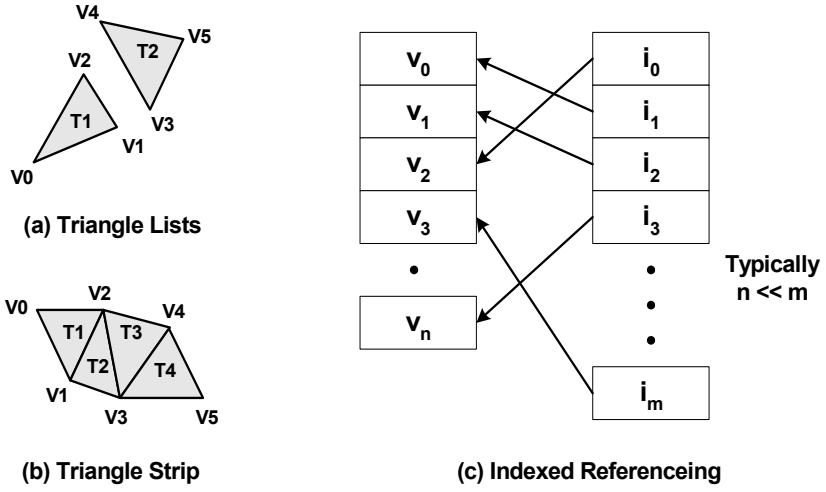
Although graphics pipeline can be represented effectively as stream processing, modern graphics system such as OpenGL [49] processes graphics data in immediate mode basically. In this mode, each parameter of primitive is issued immediately to graphics system by application programming interface (API) function call, allowing representation of graphics primitives to match application's own data structure. However, each API call interrupts graphics system and thus reduces efficiency for stream processing. As a response of this point, OpenGL supports vertex array functions to enable batch processing, reducing function call overhead. Vertex array is defined as place where a block of vertex records such as coordinates and colors may be stored in array format. Figure 5.1-2 shows the difference between intermediate mode and vertex array mode. Since each element in vertex array requires same operations repeatedly, graphics hardware can use virtue of stream processing in vertex array mode.



[Figure 5.1-2: Immediate Mode and Vertex Array Mode]

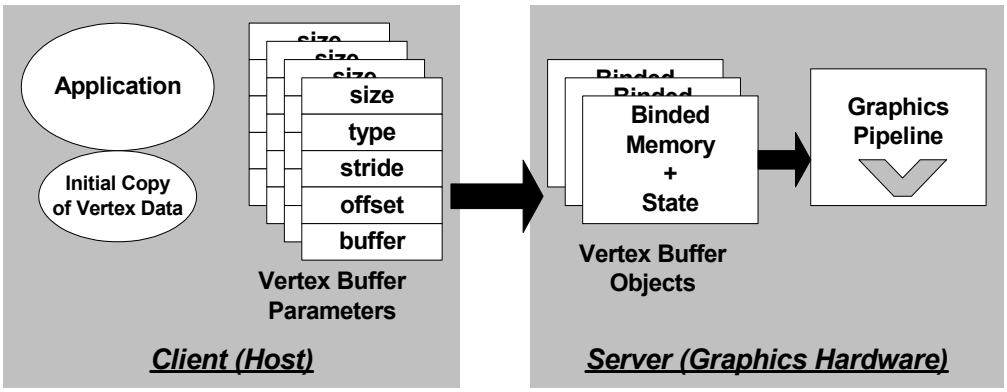
In order to enhance efficiency in processing vertex array, each element of vertex array can be indexed and these indices can be used to reference actual vertex

data. Since most of 3D graphics models are topologically same with a sphere, some portions of vertex data can be used multiple times in representing models. Thus, indexing vertex records and referencing indices instead of vertex data themselves can reduce the total bandwidth consumed when handling long sequence of triangles as shown in Figure 5.1-3. Since many implementations of graphics system contain cache memory, reused vertices can be resided in cache memory after first referencing. The indices can be used for tag information in such cache operations. Moreover, because triangle strips represent each additional triangle by adding just one vertex, strips can further reduce bandwidth. The most optimal case is indexed strip.



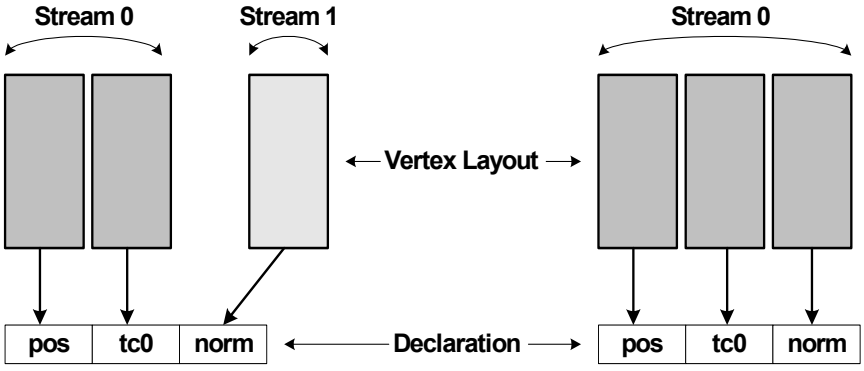
[Figure 5.1-3: Indexed Drawing]

Modern graphics system such as OpenGL adopts client-server model. That is, a program (the client) issues commands, and these commands are interpreted and processed by the graphics system (the server). The server may or may not operate on the same computer as the client. In this sense, the graphics system is "network-transparent." A server may maintain a number of contexts, each of which is an encapsulation of current state of graphics system. A client may choose to connect to any one of these contexts. This separation gives much



[Figure 5.1-4: Vertex Buffer Object]

flexibility to implementations of graphics system. The implementations don't have to consider client-specific features and only provide encapsulated graphics functionality. The original implementation of vertex array is client-side features, in that actual graphics data should be copied to server-side before processing. For more efficient memory management, vertex buffer object (VBO) is introduced as shown in Figure 5.1-4 [50]. In VBO, the storage of graphics data is resided in server-side, that is graphics hardware. The graphics hardware can choose optimal places for these storages and match them to its own memory system structure. The client-side has only state parameters such as size, hints and mapping pointer. These state information can be shared between graphics contexts like texture



[Figure 5.1-5: Vertex Declaration from Multiple Streams]

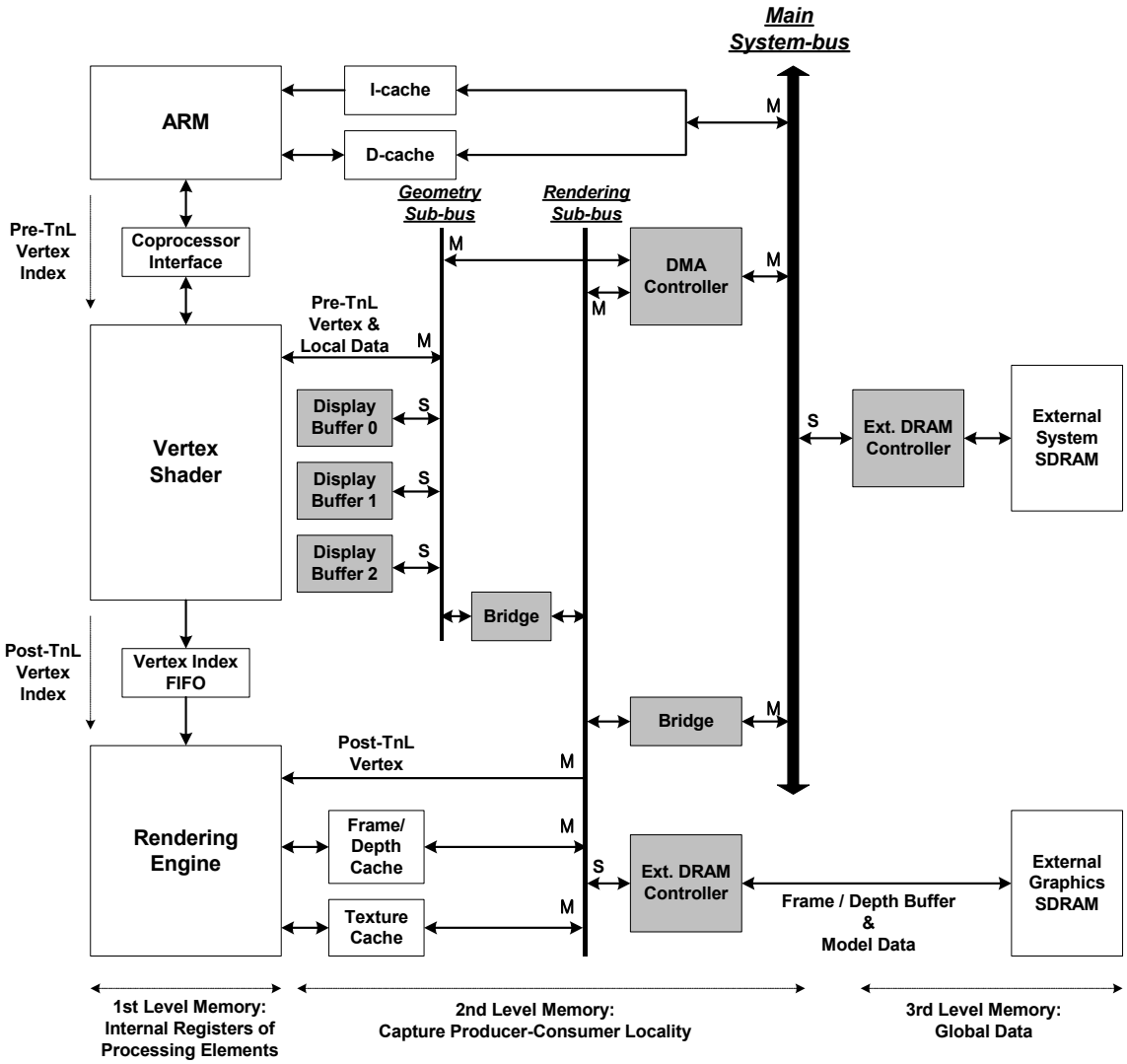
objects. Therefore, the VBO and indexed operations can improve stram processing in graphics hardware moderately. It is not restricted that each vertex data is stored in a single VBO. Figure 5.1-5 illustrates the situation that a group of vertex records are combined from multiple VBOs.

5.2 Enhancing Stream Processing in Graphics Processor

5.2.1 Architecture Revision

Figure 5.2-1 is the revised architecture of implemented graphics processor in previous chapters for enhancing stream processing. Although the separation of data transfer flow by coprocessor architecture gives benefits to stream processing by improving parallelism of processing elements, the revised architecture contains hierarchical memory system to capture producer-consumer locality more effectively. The data transfer path composed of a single display buffer connected directly to the vertex shader in the figure 2.2-2 is replaced by local multi-layer bus architecture including DMA engine and separated display buffer memories (gray region in the figure 5.2-1). The revised architecture can be explained by the following features.

(a) Separated from main system bus, it has two bus layers for geometry sub-system and rendering sub-system respectively. It can help the local traffic generated by producer-consumer locality to be completely captured inside and separated from global traffic that is transferred through main system bus. Also, the two bus layers are assigned to each graphics pipeline wholly and connected through bus bridges, increasing bandwidth throughput. Since the figure 2.1-1 in the chapter 2 tells that transferring data from application to geometry stage and transferring data from geometry stage to rendering stage altogether require half gigabyte bandwidth per second, each sub-bus operating at 200MHz with 4-byte or 8-byte data width can provide sufficient performance. Because these added



[Figure 5.2-1: Enhanced Graphics Processor for Stream Processing ("M" means that this port is a bus-master, and "S" means that this port is a bus-slave)]

two-layer bus architecture is installed locally in the graphics sub-system instead of in the global system bus, hardware cost such as power consumption and area can be more reduced than conventional global multi-layer bus system.

(b) The revised architecture provides multiple (three) separated display buffer memories for managing multiple streams. This feature can be utilized in implementing VBOs with allowing optimal choice of memory locations.

Generally, coordinates are assigned to each vertex while other parameters such as normal vector and colors may be reused among multiple vertices. Therefore, some VBOs can have different lifetime such as dynamic or stream types from others of static type. Like illustration in the figure 5.1-5, display buffer 0 can be assigned for coordinates of vertices before geometry operations (Pre-TnL) while display buffer 1 is for other parameters of vertices. The vertex shader can use display buffer 2 for storing vertex data after finishing geometry operations (Post-TnL). As indicated in the figure 2.6-2 in the chapter 2, about 16kB is sufficient for the capacity of each display buffer memory because single vertex data can be distributed in two or more display buffer memories.

(c) The coprocessor interface and vertex FIFO that are originally used for the command path in the figure 2.2-2 are utilized to transfer vertex indices between processing elements. The coprocessor interface is used for transferring Pre-TnL vertex indices while the vertex FIFO is for Post-TnL vertex indices. As described in the figure 5.1-3, the indexed drawing reduces total bandwidth and are well matched to the revised architecture of graphics processor. Using indices as commands can improve issue efficiency of graphics processing elements because the required bytes of indices are much smaller than the size of vertex data.

(d) The revised architecture suggests that external SDRAM can be attached to the rendering sub-bus. This external SDRAM can store frame buffer data and some texture images. It can also contain initial vertex model data before graphics processing. In most cases, the graphics SDRAM can be stacked on die of the graphics processor in a single chip package. If graphics pipeline is represented as stream processing, the final frame buffer data can be resided in the graphics SDRAM and separated from the initial input data stream, which is stored in the system's main SDRAM.

(e) In the revised architecture, a single two port DMA engine connects the

two-layer graphics sub-bus to the main system bus via a single master port. The DMA engine is controlled by main ARM processor and thus every slave port in the graphics sub-bus, that are all display buffer memories and the external graphics SDRAM, are accessible in a common memory space of the ARM processor. This configuration gives flexibility to managing memories. Moreover, since the dual operations described in the section 2.4 allows the ARM processor to operate in parallel with the vertex shader and the rendering engine, the DMA engine controlled by the ARM processor can show maximum throughput in transferring data between the display buffer memories and the system's main memory. Also, the frame buffer data in the external graphics SDRAM can be accessed directly to the main SDRAM through the DMA engine.

(f) The geometry sub-bus and the rendering sub-bus are connected to each other a bus bridge, and another bus bridge is used to connect the rendering sub-bus to the main system bus. The first bus bridge allows the rendering engine to access the display buffer memories, causing local traffic generated from geometry stage to be captured directly to rendering stage. The second bus bridge makes the rendering engine to access system's main memory directly for texture data or to use the main SDRAM as frame buffer in the absence of the graphics SDRAM.

Entirely, the ARM processor with graphics elements are wrapped by three bus master ports. The coprocessor architecture makes the graphics processor to be extended easily for enhancing stream processing with low cost by using typical SoC building blocks such as two bus arbiters, two bus bridges and one DMA engine.

5.2.2 Performance Limitation

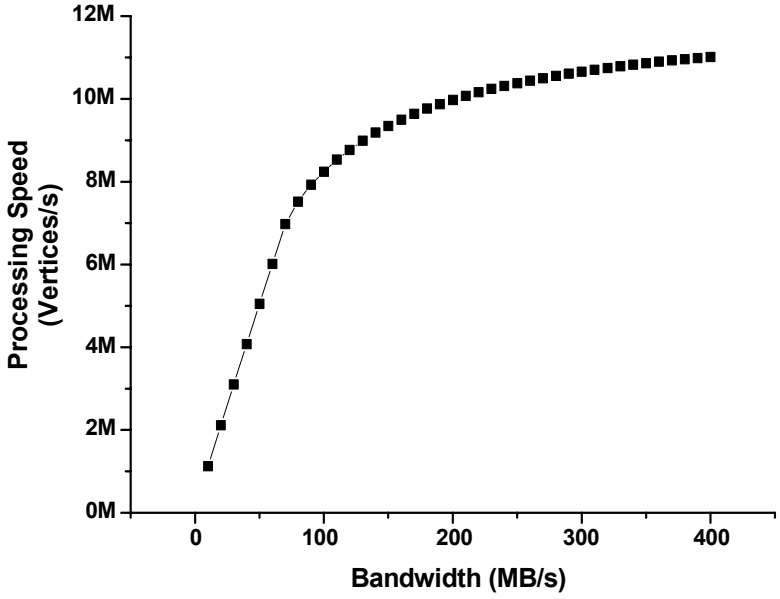
The performance of the revised architecture can be estimated from amount of provided bandwidth. The revised architecture can be regarded to have separate

DRAMs to geometry and rendering stages. In a typical mobile applications, a 32-bit SDRAM chip running 100MHz (PC100) is used as external memory and gives a suitable bandwidth 200MB/s with approximately 50% of efficiency. For rendering stage, the required bandwidth for each buffer can be calculated by the following relations.

$$BW = Pixel\ Fill\ Rate^{*}\ Cache\ Miss\ Rate^{*}\ Block\ Size^{*}\ Pixel\ Size$$

From the description in the section 3.2.1, the required bandwidth for frame buffer and depth buffer when filling 100Mpixels/s are 100.8MB/s and 38.4MB/s respectively. Thus, if remaining bandwidth of the graphics SDRAM or the system main memory are used for texture memories, pixel fill rate can be reached up to tens of Mpixels per second.

For geometry stage, vertex data and related index can be fetched from the system main memory. The indices are issued from the ARM processor via the coprocessor interface and the vertex data are transferred from the DMA engine to the display buffer memories. It is needed to restrict the provided bandwidth to the



[Figure 5.2-2: Performance Limitation of Mobile Graphics Hardware]

vertex shader to about 100MB/s in order to allocate the remains to the ARM processor or other devices. Figure 5.2-2 shows the relationship between the provided memory bandwidth and the graphics performance when indexed drawing is performed with composting vertex data from two streams. In this case, it is assumed that the processing power of the vertex shader can provide the speed of 25Mvertices/s theoretically. In this graph, the best achievable performance does not exceed about 10Mvertices/s. Therefore, the processing capability of mobile graphics system should be designed to match that performance limitation in most cases.

CHAPTER 6
Conclusions and Further Work

6.1 Conclusions

A low power graphics processor is designed, implemented and demonstrated for mobile 2D and 3D graphics and various multimedia applications. Most graphics architectures for mobile applications have mainly focused on rasterization and texture mapping due to high processing requirements. In order to balance 3D graphics pipeline within the limited system resources, I integrated simple and efficient programmable architecture for vertex shading as well as low power rendering engine instead of using dedicated hardware engine with complex functions.

The proposed graphics processor has four major features: (a) Separation of data transfer flow is proposed for efficient hardware and bandwidth utilization. Different from previous works, the ARM coprocessor architecture enables optimized performance throughput while achieving easy programmability by separating command transfer path from data transfer path. (b) Full hardware accelerations with stream processing is achieved to boost-up the sustained performance in compact and fast hardware. The producer-consumer locality, that are frequently observed in stream multimedia operations such as 3D graphics, is also considered in hardware design. (c) Two level extensions of instruction set

architecture are implemented for programmability and parallel processing. The added multimedia instructions by the coprocessor architecture is once again extended to more optimized graphics instructions by dual operations, in which concurrent operations of the graphics coprocessor with the main processor is enabled. And, (d) Fixed-point SIMD processing is employed for low power consumption and low cost implementation. It exploits data level parallelism in graphics processing while keeping the power consumption low.

The graphics processor contains an ARM10 compatible 32-bit RISC processor, a 128-bit programmable fixed-point single-instruction-multiple-data (SIMD) vertex shader, a low power rendering engine with 26kB dedicated graphics cache, and a programmable frequency synthesizer (PFS). The circuits and architecture of the graphics processor are optimized for fixed-point operations and achieve the low power consumption with help of instruction-level power management of the vertex shader and pixel-level clock gating of the rendering engine. SIMD datapath of the vertex shader achieves a single cycle throughput for most graphics instructions for geometry operations. Also, the vertex shader can accelerate primitive assembly such as clipping and culling by conditional executions and clip code instruction. The rendering engine performs the rasterization and the per-pixel operations such as pixel blending and texture mapping with energy-efficient graphics cache system. The PFS with a fully balanced voltage-controlled oscillator (VCO) controls the clock frequency from 8MHz to 200MHz continuously and adaptively for low power modes by software. The 36mm^2 chip shows 50Mvertices/s and 200Mtexels/s peak graphics performance for parallel projection, dissipating 155mW in 0.18㎛ 6-metal standard CMOS logic process. In sustained operations, the implemented graphics processor can calculate full 3D graphics pipeline including geometry transformation, lighting, clip check, shading and bilinear

MIPMAP texture mapping at the speed of 3.6Mpolygons/s

For explanations and optimizations of mobile graphics architecture, model of 3D graphics computing is described and simulated. Various parameters such as memory capacity, bandwidth and batch size are analyzed to show influences in overall performance. The implemented graphics processor with 32kB display buffer is verified to show sufficient performance while consuming 100MHz bandwidth. Since graphics pipeline can be represented effectively as stream processing, I revised the architecture of graphics processor so that multimedia streams can be more effectively processed. SIMD computing elements and hierarchical memory system with multi-layer sub-bus and DMA engine replace the data transfer path composed of a single display buffer connected directly to the vertex shader. In the case of indexed drawing, the overall performance is expected to be improved up to 8Mvertices/s and 100Mpixels/s.

The system evaluation platform is also developed to evaluate and demonstrate mobile 3D graphics using a flexible topology and protocol. It includes software graphics library with programming interface of the graphics processor for simplifying application development.

The implemented graphics processor was successfully demonstrated on the evaluation platform and verified real-time 3D graphics in mobile applications.

6.2 Further Works

Since software architecture for programmable shading and stream processing is not sufficiently defined for mobile platforms, further research on optimal software layers for mobile graphics system are required for highly effective performance

achievement. In the area of hardware research, more advanced bus architecture and memory system such as accelerated graphics port (AGP) in PC graphics system can be considered to be applied for mobile multimedia system. And, the focus on the design of a fragment shader is required to generate more photo-realistic pixels with high sustained throughput. Finally, the combination of vertex shader, pixel shader and even 2D video engine in a single hardware architecture will be studied.

Summary

국문 요약

고정 소수점 SIMD Vertex Shader를 이용한 저전력 프로그래머블 3D 그래픽스 프로세서

실시간 3차원 그래픽스는 휴대용 터미널에서 가장 흥미 있는 응용분야가 되고 있다. 그렇지만, 휴대용 단말기에서는 한정된 전지 사용시간과 계산 성능이 그래픽스 처리를 위한 시스템 자원과 메모리 대역폭을 제한하고 있다. 게다가, 사용자들은 PC 그래픽스와는 상대적으로 작은 화면에서 그래픽스 이미지를 매우 가까이서 사용하고 있기 때문에, 최근의 휴대용 3차원 그래픽스는 하드웨어와 소프트웨어에서 적은 소비전력으로 더욱 향상된 기능을 제공하기 위해 프로그램 처리 능력을 제공하고 있다. 본 연구에서는 휴대용 응용분야를 위해 고정 소수점 vertex shader를 이용한 프로그래머블 그래픽스 프로세서를 설계하고 구현하였다. 제안된 구조는 데이터 흐름의 분리, 스트림 처리를 고려한 완전한 하드웨어 가속, 두 단계의 명령어 구조 확장 그리고 저전력 소비를 위한 고정 소수점 SIMD 연산구조의 네 가지 특징을 가지고 있다. 설계된 그래픽스 프로세서는 32 비트 ARM10 호환의 RISC 프로세서, 128 비트의 고정소수점 SIMD vertex shader, 26kB의 별도 그래픽스 캐쉬를 장착한 저전력 rendering engine, 그리고 프로그래머블 주파수 합성기를 포함하고 있다. 일반적인 그래픽스 하드웨어와는 달리, 제안된 그래픽스 프로세서는 ARM10 보조 프로세서 인터페이스를 사용하여 집적하였고, "이중 동작"기능을 구현하여 진보된 그래픽스 알고리즘과 다양한 스트리밍 멀티미디어 응용을 실현

하기 위한 프로그래머블 vertex shading을 가능하게 하였다. 저전력 소비를 위해 그래픽스 프로세서의 회로와 내부 구조는 고정 소수점 연산에 최적화하여 설계되었다. 또한 vertex shader의 명령어 단위 전력 제어 기술과 rendering engine의 픽셀 단위 clock gating 기술을 사용하여 전력 소비 효율을 더욱 향상시켰다. 완전 균형 전압조정발진기를 내장한 프로그래머블 주파수 합성기는 소프트웨어의 제어에 따라 전체 칩의 동작 주파수를 8MHz부터 200MHz까지 연속적이고 적응성 있게 조절한다. 이 칩은 최고 155mW의 전력을 소비하면서 최대 50Mvertices/s and 200Mtexels/s의 그래픽스 성능을 나타낸다. 0.18㎛ 6-metal 표준 CMOS 공정으로 사용하여 칩을 제작하였고 36mm^2 면적을 차지하였다. 더 향상된 스트림 처리 성능을 위해, 설계된 그래픽스 프로세서를 계층적 메모리 구조를 가지는 SIMD 연산장치로 새로이 확장하였다. 구현된 그래픽스 프로세는 평가보드를 통해 성공적으로 시연되었고, 휴대용 응용분야에서 실시간 3차원 그래픽스의 구현을 입증하였다.

Bibliography

[1] John S. Montrym, et al, "NVIDIA GeForce 6800," in Proceedings. of HotChips 16, 2004

[2] Gordon Elder, "ATI Radeon 9700: Architecture and 3D Performance," in Hot3D of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2002

[3] Aurangeb Khan, et al, "A 150-MHz Graphics Rendering Processor with 256-Mb Embedded DRAM," IEEE Journal of Solid-State Circuits, Vol. 36, No. 11, pp. 1775-1784, Nov. 2001

[4] David Clark, "Mobile processors begin to grow up", IEEE Computer Magazine, Vol. 35, Issue 3, pp. 22-24, March, 2002

[5] Tomas Akenine-Moller, et al, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones," in Proceedings of ACM SIGGRAPH, pp. 801-808, 2003

[6] Ju-Ho Sohn, et al, "Optimization of Portable System Architecture for Real-time 3D Graphics," in Proceedings of IEEE International Symposium on Circuits and System, pp. 1769-1772, 2002

[7] Donghyun Kim, et al, "An SoC with 1.3Gtexels/s 3-D Graphics Full Pipeline for Consumer Applications," IEEE Journal of Solid-State Circuits, Vol. 41, No. 1, pp. 71-84, Jan. 2006

[8] Alan Watt, "3D Computer Graphics," 3rd edition, Addison-Wesley, 2000

[9] Kris Gray, "Microsoft DirectX 9 Programmable Graphics Pipeline," Microsoft Press, 2003

[10] Ju-Ho Sohn, et al, "Low-power 3D Graphics Processors for Mobile Terminals,"

IEEE Communications Magazine, Vol. 43, No. 12, pp. 90-99, Dec. 2005

[11] Yong-Ha Park, et al, "A 7.1GB/s Low Power Rendering Engine in 2D Array Embedded Memory Logic CMOS for Portable Multimedia System," IEEE Journal of Solid-State Circuits, Vol. 36, No. 6, pp.944-955, Jun. 2001

[12] Chi-Weon Yoon, et al, "A 80/20MHz 160mW Multimedia processor Integrated with Embedded DRAM, MPEG4 and 3D Rendering Engine for Mobile Applications," IEEE Journal of Solid-State Circuits, Vol. 36, No. 11, pp. 1758-1767, Nov. 2001

[13] Ramchan Woo, et al, "A 210mW Graphics LSI Implementing Full 3D Pipeline with 264Mtexels/s Texturing for Mobile Multimedia Applications, " IEEE Journal of Solid-State Circuits, Vol. 39, No. 2, pp. 358-367, Feb. 2004

[14] David A. Patterson, et al, "Computer Architecture: A Quantitative Approach," 2nd edition, Morgan Kaufmann Publishers, 1996

[15] "ARM MBX HR-S 3D Graphics Core Technical Overview," Technical Document, ARM DTO-0003B, 2002

[16] John S. Montrym, et al, "InfinityReality: A Real-time Graphics System," in Proceedings of ACM SIGGRAPH, pp. 293-302, 1997

[17] Masanobu Okabe, et al, "A 90nm Embedded DRAM Single Chip LSI with a 3D Graphics, H.264 Codec Engine, and a Reconfigurable Processor," in Proceedings of HotChips 16, 2004

[18] Edward Hutchins, et al, "SC10: A Video Processor and Pixel Shading GPU For Handheld Devices," in Proceedings of HotChips 16, 2004

[19] Masatoshi Kameyama, et al, "3D Graphics LSI Core for Mobile Phone: Z3D," in Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 60-67, 2003

[20] Gregory Uvieghara, et al, "A Highly Integrated 3G CDMA2000 1X Cellular Baseband Chip with GSM/AMPS/GPS/Bluetooth//Multimedia Capabilities and ZIF RF Support," in Digest of Technical Papers of IEEE International Solid-State Circuits

Conference, pp. 422-423, 2004

[21] Fumio Arakawa, "An Embedded Processor Core for Consumer Appliances with 2.8GFLOPS and 36Mpolygons/s," in Digest of Technical Papers of IEEE International Solid-State Circuits Conference, pp. 334-335, 2004

[22] OpenGL-ES, available at http://www.khronos.org/opengles/

[23] OMAP, available at http://focus.ti.com/omap/docs/omaphomepage.tsp

[24] Matthew Eldridge, "Designing Graphics Architectures Around Scalability and Communication," Ph.D. Dissertation, Stanford University, Jun. 2001

[25] Ju-Ho Sohn, et al, "A 50Mvertices/s Graphics Processor with Fixed-point Programmable Vertex Shader for Mobile Applications," in Digest of Technical Papers of IEEE International Solid-State Circuits Conference, pp. 192-193, 2005

[26] Ju-Ho Sohn, et al, "A 155mW, 50Mvertices/s Graphics Processor with Fixed-point Programmable Vertex Shader for Mobile Applications," IEEE Journal of Solid-State Circuits, Vol. 41, No. 5, pp. 1081-1091, May. 2006

[27] Steve Furber, "ARM: System-on-chip Architecture," 2nd edition, Addison-Wesley Press, 2000

[28] Ian Thornton, "ARM PrimeXsys Wireless Platform," White Paper, available at http://www.arm.com

[29] Ramchan Woo, "A Low-power 3D Rendering Engine with Two Texture Units and 29Mb Embedded DRAM for 3G Multimedia Terminals," IEEE Journal of Solid-State Circuits, Vol. 39, No. 7, pp. 1101-1109, Jul. 2004

[30] Intel Wireless MMX Technology, available at http://www.intel.com

[31] Prashant P. Gandhi, "SA1500: A 300MHz RISC CPU with Attached Media Processor," in Proceedings of HotChips 10, 1998

[32] G. K. Golli, et al, "3D Graphics Optimization for ARM Architecture," presented at Game Developer Conference 2002, March, 2002

[33] Xuejun Hao, et al, "Variable-precision rendering." in Proceedings of the 2001

Symposium on Interactive 3D Graphics, pp. 149-158, 2001

[34] Ju-Ho Sohn, et al, "A Programmable Vertex Shader with Fixed-point SIMD Datapath for Low Power Wireless Applications," in Proceedings SIGGRAPH /Eurographics Workshop on Graphics Hardware, pp. 107-114, 2004

[35] Ju-Ho Sohn, et al, "A Fixed-point Multimedia Coprocessor for 50Mvertices/s Programmable SIMD Vertex Shader for Mobile Applications," in Proceedings of IEEE European Solid-State Circuits Conference, pp. 207-210, 2005

[36] Michael Deering, "Geometry compression", in Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, pp. 13-20, 1995

[37] Erik Lindholm, et al, "A User-programmable Vertex Engine," in Proceedings of ACM SIGGRAPH, pp. 149-158, 2001

[38] Bengt-Olaf Schneider, "Efficient Polygon Clipping for an SIMD Pipeline," IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 3, Jul.-Sep. 1998

[39] HP Western Research Lab., CACTI, available:
http://research.compaq.com/wrl/people/jouppi/CACTI.html

[40] Ziyad S. Hakura, et al, "The Design and Analysis of a Cache Architecture for Texture Mapping," in Proceedings of the 24th International Symposium on Computer Architecture, pp. 108-120, 1997

[41] Norman J. Rohrer, et al, "A 64-bit Microprocessor in 130nm and 90nm Technologies with Power Management Features," IEEE Journal of Solid-State Circuits, Vol. 40, No. 1, pp. 19-27 Jan. 2005

[42] Masatoshi Imai, et al, "A 109.5mW 1.2V 600Mtexel/s 3-D Graphics Engine," in Digest of Technical Papers of IEEE International Solid-State Circuits Conference, pp. 332-333, 2004

[43] Donglok Kim, et al, "Data Cache and Direct Memory Access in Programming Media Processors," IEEE Micro, Vol. 21, No. 4, pp. 33-42, Jul. 2001

[44] Brucek Khailany, "Imagine: Media Processing with Streams", IEEE Micro, Vol.

21, No. 2, pp. 35-46, Mar. 2001

[45] William J. Dally, "Merrimac: Supercomputing with Streams," in Proceedings of ACM/IEEE Supercomputing (SC) 2003

[46] Michael B. Taylor, et al, "The RAW Microprocessor: A Computational Fabric for Software Circuits and General-purpose Programs," IEEE Micro, Vol. 22, No. 2, pp. 35-46, Mar. 2002

[47] Ian Buck, et al, "Brook for GPUs: Stream Computing on Graphics Hardware," in Proceedings of ACM SIGGRAPH, 2004

[48] John D. Owens, et al, "Polygon Rendering on a Stream Architecture," in Proceedings SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 23-32, 2000

[49] Masoo Woo, et al, "OpenGL Programming Guide," 3rd edition, OpenGL Architecture Review Board, Addison-Wesley, 1999

[50] Kurt Akeley, "Buffer Objects," presented at Game Developer Conference 2003, March, 2003

감사의 글

짧지 않은 대학원 생활을 돌이켜 보면, 힘들 때 마다 용기와 지혜를 주신 많은 분들이 생각납니다. 6년 전, 처음 실험실 문을 열고 들어올 때부터 지금까지 한결같은 지지와 정성으로 제가 많은 것을 배워 스스로 나아가도록 지도해 주신 유회준 지도교수님께 먼저 이 지면을 빌어 머리 숙여 감사의 마음을 전합니다. 그리고 이 논문이 완성되도록 많은 지도와 조언을 해주신 김성대 교수님, 나종범 교수님, 박인철 교수님, 그리고 김재민 교수님께도 감사의 마음을 전합니다.

세계 최고의 엔지니어가 되기 위해 인생의 가장 아름다운 시기를 함께 보내며 많은 고민을 함께 하고 더 많은 기쁨을 함께 나누었던 반도체 시스템 연구실의 가족들에게 깊은 감사를 드리며, 제 자신이 그 일원이었다는 점이 무엇보다도 자랑스러웠다고 말씀드리고 싶습니다. 석사 신입생 때부터 학문에 대한 지식 뿐 아니라 엔지니어가 갖추어야 할 수많은 가치를 애정으로 일깨워 준 존경하는 선배, 람찬이 형, 그리고 이번 칩을 만들면서 힘든 시기를 같이 해쳐나가며 인생에 대한 많은 고민을 함께 해 준 세중이 형에게 감사의 마음을 전합니다. 그리고 제 밑에서 누구보다도 많은 노력을 보태준 후배 정호에게도 부족한 선배로서 매우 고마웠다는 말을 전하고 싶습니다. 일일이 이름을 열거하여 표현하기에는 너무나 많은 도움을 준 연구실의 선배님들, 동기들 그리고 후배들에게 앞으로 더 넓은 세상으로 나아가 실험실에서의 가슴 벅찬 기억을 뒤로 하며 더욱 노력하는 엔지니어가 되겠다는 다짐을 드립니다.

고등학교 시절부터 힘이 되어준 경기과학고등학교 자랑스러운 13기 친구들

에게도 감사의 마음을 전합니다. 평생을 두고 힘이 되어줄 성현이, 이제 새로운 세상으로 자신의 무한한 능력을 보여줄 도헌이 그리고 이름 하나 하나를 부르면 우리들의 추억들이 가슴을 따뜻하게 해주는 소중한 분들, 모두 제게 너무나도 귀중한 친구들입니다.

철없는 막내 동생을 특별한 사랑으로 많은 기회를 양보하면서 저를 아껴준 큰 누나와 작은 누나, 그리고 친형처럼 보살펴 주신 두 분 매형께도 감사의 마음을 전합니다. 끝없는 사랑과 헌신으로 저를 있게 하시고 삶의 의미가 되어주신 아버지, 어머니께 세상의 글로 표현할 수 없을 만큼 감사의 마음을 전합니다. 마지막으로 하늘에 계신 할아버지께 감사의 마음을 전합니다.

JU-HO SOHN

2006-05-07

sohnjuho@eeinfo.kaist.ac.kr

Education

Korea Advanced Institute of Science and Technology (KAIST)

- Full Scholarship from KAIST

2003/03 ~ 2006/08 *Ph.D. in Electrical Engineering*

Dissertation: A Low Power Programmable 3D Graphics Processor with Fixed-point SIMD Vertex Shader

2001/03 ~ 2003/02 *M.S. in Electrical Engineering*

Dissertation: Design and Optimization of Geometry Acceleration for Portable 3D Graphics

1997/03 ~ 2001/02 *B.S. in Electrical Engineering – Summa Cum Laude*

Major: Electrical Engineering, Minor: Physics

Overall GPA: 3.95/4.30, Major GPA: 3.87/4.3

Working Experience

Korea Advanced Institute of Science and Technology (KAIST)

2001/03 ~ Present *Research Assistant* – Perform research mainly focused on various aspects of circuits, architecture and system design, chip and software implementation. Major research area includes mobile 3D computer graphics.

2001/03 ~ Present *Teaching Assistant* – Assistant teaching for Microelectronics Circuits, Computer Architecture, SoC Design Course

Research Projects

DA (Digital Accessory) Project

Development of 3D Graphics Accelerator IP for Mobile Application Processor SoC

Sponsored by Samsung Electronics

2005/09 ~ 2006/08 Technical Advisor

2004/09 ~ 2005/08 Chief Researcher, Team Leader

Responsible for Software Graphics Library and Platform Development

Dept. of EECS, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, Korea, 305-701
Contacts: +82-42-869-8068, sohnjuho@eeinfo.kaist.ac.kr

2003/09 ~ 2004/08	Responsible for Full Chip Architecture Design and Backend Process
	Responsible for Programmable Graphics Engine RTL Design
2003/03 ~ 2003/08	Responsible for ARM10 Compatible RISC Processor RTL Design

MobileGL-C1

Development of 3D Graphics Library for Wireless Cellular Phones

Sponsored by MCRES

2004/03 ~ 2004/08	Responsible for Library Specification and Code Optimization for ARM7/ARM9

RAMP (RAM Processor) Project

Development of Application Specific Embedded Memory Logic Design Technology

Sponsored by Korea Ministry of Science and Technology, Korea Ministry of Commerce, Industry and Energy

2002/10 ~ 2003/02	Responsible for Evaluation Platform Development
2002/06 ~ 2002/09	Responsible for Software Graphics Library Development
2001/09 ~ 2002/05	Responsible for Buffer Controller RTL Design and SRAM Full-custom Design

X-Switch (Extra High Speed Switch) Project

Development of Hardwired Network Processor using Embedded Memory Process

Sponsored by Samsung Electronics

2001/03 ~ 2001/08	Responsible for ARM7 Compatible RISC Processor Design Using SystemC

International Journal Papers (First-authored Papers Only)

JSSC	**A 155mW, 50Mvertices/s Graphics Processor with Fixed-point Programmable**
2006	**Vertex Shader for Mobile Applications**
	Ju-Ho Sohn, Jeong-Ho Woo, Min-Wuk Lee, Hye-Jung Kim, Ramchan Woo and Hoi-Jun Yoo
	IEEE Journal of Solid State Circuits, Vol. 41, No. 5, May 2006
COMM	**Low Power 3D Graphics Processors for Mobile Terminals**
2005	Ju-Ho Sohn, Yong-Ha Park, Chi-Weon-Yoon, Ramchan Woo, Se-Jeong Park and Hoi-Jun Yoo
	IEEE Communications Magazine, Vol. 43, No. 12, December 2005

International Conference Papers (First-authored Papers Only)

DATE	**Design and Test of Fixed-point Multimedia Co-processor for Mobile**
2006	**Applications**

Dept. of EECS, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, Korea, 305-701
Contacts: +82-42-869-8068, sohnjuho@eeinfo.kaist.ac.kr

	Ju-Ho Sohn, Jeong-Ho Woo, Jerald Yoo and Hoi-Jun Yoo
	Design, Automation and Test in Europe, 2006
ESSCIRC	**A Fixed-point Multimedia Co-processor with 50Mvertices/s Programmable**
2005	**SIMD Vertex Shader for Mobile Applications**
	Ju-Ho Sohn, Jeong-Ho Woo, Ramchan Woo and Hoi-Jun Yoo
	IEEE European Solid-State Circuits Conference, 2005
ISSCC	**A 50Mvertices/s Graphics Processor with Fixed-point Programmable Vertex**
2005	**Shader for Mobile Applications**
	Ju-Ho Sohn, Jeong-Ho Woo, Min-Wuk Lee, Hye-Jung Kim, Ramchan Woo and Hoi-Jun Yoo
	IEEE International Solid-State Circuits Conference, 2005
HWWS	**A Programmable Vertex Shader with Fixed-point SIMD Datapath for Mobile**
2004	**Applications**
	Ju-Ho Sohn, Ramchan Woo and Hoi-Jun Yoo
	ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, 2004
ISCAS	**Optimization of Portable System Architecture for Real-Time 3D Graphics**
2002	Ju-Ho Sohn, Ramchan Woo and Hoi-Jun Yoo
	IEEE International Symposium of Circuits and Systems, 2002

Patents

1. Apparatus for Accelerating Multimedia Processing by Using Co-processor

Ju-Ho Sohn, Ramchan Woo, Hoi-Jun Yoo

Korean Patent Number: 1004659130000

2. Apparatus for Accelerating Multimedia Processing by Using Co-processor

Ju-Ho Sohn, Ramchan Woo, Hoi-Jun Yoo

Korean Patent Number: 1004636420000

3. Apparatus for Controlling Buffer Memory in Computer System

Ju-Ho Sohn, Ramchan Woo, Hoi-Jun Yoo

Korean Patent Number: 1004480710000

Research Interests

1. Mobile 2D/3D Graphics Architectures and Their Software/Hardware Implementations

2. Multimedia Signal Processing in Consumer Electronics

2. Computer Architecture (Streaming Processor, Embedded RISC Processor)

Dept. of EECS, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, Korea, 305-701
Contacts: +82-42-869-8068, sohnjuho@eeinfo.kaist.ac.kr

Skillful Tools

1. High-level Design: C/C++, JAVA, SystemC
2. Graphics Library: OpenGL / OpenGL-ES
3. Logic Design: Verilog-XL, Synopsys Design Compiler, Astro P&R Tools
4. Circuit Design: Cadence OPUS, EPIC nanosim, Hspice, Calibre DRC/LVS
5. Software Programming: Windows MFC, Windows WDM, Windows COM+, Linux QT, ARM ADT/ADS

Language

1. Korean as a Domestic Language
2. Proficient English
3. Beginning Japanese

Dept. of EECS, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejeon, Korea, 305-701
Contacts: +82-42-869-8068, sohnjuho@eeinfo.kaist.ac.kr