MOBILE 3D GRAPHICS SoC

From Algorithm to Chip

Jeong-Ho Woo
Korea Advanced Institute of Science and Technology, Republic of Korea
Ju-Ho Sohn
LG Electronics Institute of Technology, Republic of Korea
Byeong-Gyu Nam
Samsung Electronics, Republic of Korea
Hoi-Jun Yoo
Korea Advanced Institute of Science and Technology, Republic of Korea

John Wiley & Sons (Asia) Pte Ltd

MOBILE 3D GRAPHICS SoC

MOBILE 3D GRAPHICS SoC

From Algorithm to Chip

Jeong-Ho Woo
Korea Advanced Institute of Science and Technology, Republic of Korea
Ju-Ho Sohn
LG Electronics Institute of Technology, Republic of Korea
Byeong-Gyu Nam
Samsung Electronics, Republic of Korea
Hoi-Jun Yoo
Korea Advanced Institute of Science and Technology, Republic of Korea

John Wiley & Sons (Asia) Pte Ltd

Other Wiley Editorial Offices

John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstrasse 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons Canada Ltd, 5353 Dundas Street West, Suite 400, Toronto, ONT, M9B 6H8, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Mobile 3D graphics SoC : from algorithm to chip / Jeong-Ho Woo ... [et al.].
p. cm.
Includes index.
ISBN 978-0-470-82377-4 (cloth)
1. Computer graphics. 2. Mobile computing. 3. Systems on a chip. 4. Three dimensional display systems.
I. Woo, Jeong-Ho.
T385.M62193 2010
621.3815–dc22

2009049311

ISBN 978-0-470-82377-4 (HB)

Typeset in 10/12pt Times by Thomson Digital, Noida, India.
Printed and bound in Singapore by Markono Print Media Pte Ltd, Singapore.
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Preface

This is a book about low-power high-performance 3D graphics for SoC (system-on-chip). It summarizes the results of 10 years of "ramP" research at KAIST (ramP stands for RAM processor) – a national project that was sponsored by the Korean government for low-power processors integrated with high-density memory. The book is mostly dedicated to 3D graphics processors with less than 500 mW power consumption for small-screen portable applications

Screen images continue to become ever-more dramatic and fantastic. These changes are accelerated by the introduction of more realistic 3D effects. The 3D graphics technology makes vivid realism possible on TV and computer screens, especially for games. Complicated and high-performance processors are required to realize the 3D graphics. Rather than use a general-purpose central processing unit (CPU), dedicated 3D graphic processors have been adopted to run the complicated graphics software.

There is no doubt that all the innovations in PC or desktop machines will be repeated in portable devices. Cellphones and portable game machines now have relatively large screens with enhanced graphics functions. High-performance 3D graphics units are included in the more advanced cellphones and portable game machines, and for these applications a low power consumption is crucial. In spite of the increasing interest in 3D graphics, it is difficult to find a book on portable 3D graphics. Although the principles, algorithms and software issues have been well dealt with for desktop applications, hardware implementation is more critical for portable 3D graphics. We intend to cover the 3D graphics hardware implementation especially emphasizing low power consumption. In addition, we place emphasis on practical design issues and know-how. This book is an introduction to low-power portable 3D graphics for researchers of PC-based high-performance 3D graphics as well as for beginners who want to learn about 3D graphics processors. The HDL file at the end of the book offers readers some first-hand experience of the algorithms, and gives a feel of the hardware implementation issues of low-power 3D graphics.

This book would not have been possible without help from many colleagues and supporters. First we would like to thank Dr Sejeong Park of Mediabridge, Dr Yongha Park of Samsung, Dr Chiwon Yoon of Samsung, and Dr Ramchan Woo of LG for their pioneering efforts in mobile 3D graphics research at the Semiconductor Systems

laboratory in KAIST. Professor Kyuho Park of KAIST, Professor T. Kuroda of Keio University, and Dr Ian Young of Intel helped us to begin our research on low-power 3D graphics. We would like also to thank Professor Young-Joon Park of Seoul National University, Dr Heegook Lee of LG, and Dr Huh Youm of Hynix for their help with the ramP project. Last but not least, we would like to thank James and his team at John Wiley for their care in the birth of this book.

1

Introduction

1.1 Mobile 3D Graphics

Mobile devices are leading the second revolution in the computer graphics arena, especially with regard to 3D graphics. The first revolution came with personal computers (PC), and computer graphics have been growing in sophistication since the 1960s. To begin with it was widely used for science and engineering simulations, special effects in movies, and so on, but it was implemented only on specialized graphics workstations. From the late 1980s, as PCs became more widely available, various applications were developed for them and computer graphics moved on to normal PCs – from specific-purpose to normal usage.

Three-dimensional graphics are desirable because they can generate realistic images, create great effects on games, and enable slick effects for user interfaces. So 3D graphics applications have been growing very quickly. Almost all games now use 3D graphics to generate images, and the latest operating systems – such as Windows 7 and OS X – use 3D graphics for attractive user interfaces. This strongly drives the development of 3D graphics hardware. The 3D graphics processing unit (GPU) has been evolving from a fixed-function unit to a massively powerful computing machine and it is becoming a common component of desktop and laptop computers.

A similar revolution is happening right now with mobile devices. The International Telecommunications Union (ITU) reports that 3.3 billion people – half the world's population – used mobile phones in 2008, and Nokia expects that there will be more than 4 billion mobile phone users (more than double the number of personal computers) in the world by 2010 [1]. In addition, mobile devices have been dramatically improved from simple devices to powerful multimedia devices; a typical specification is 24-bit color WVGA (800×480) display screen, more than 1 GOPS (giga-operations per second) computing power, and dedicated multimedia processors including an image signal processor (ISP), video codec and graphics accelerator.

So 3D graphics is no longer a guest on mobile devices. A low-cost software-based implementation is used widely in low-end mobile phones for user interfaces or simple games, while a high-end dedicated GPU-based implementation brings PC games to the mobile device.

Nowadays, 3D graphics are becoming key to the mobile device experience. With the help of 3D graphics, mobile devices have been evolving with fruitful applications ranging from simple personal information management (PIM) systems (managing schedules, writing memos, and sending e-mails or messages), to listening to music, playing back videos, and playing games. Just as with the earlier revolution in the PC arena, 3D graphics can make mobile phone applications richer and more attractive – this is the reason why I have used the phrase "second revolution."

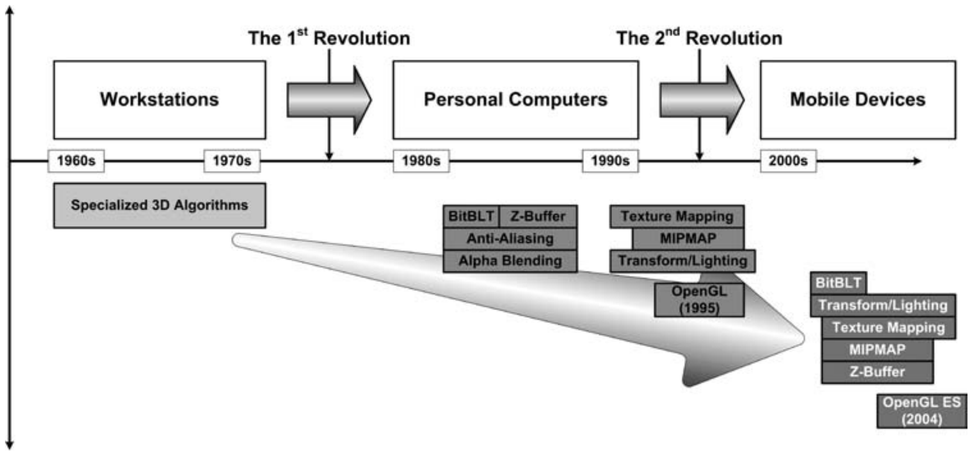


Figure 1.1 A history of 3D graphics

Development of mobile 3D graphics was started basically in the late 1990s (Figure 1.1). Low-power GPU hardware architectures were developed, and the software algorithms of PCs and workstations were modified for mobile devices. Software engines initially drove the market. Among them, two notable solutions – "Fathammer's X Forge" engine and "J-phone's Micro Capsule" – were embedded in Nokia cellular phones and J-phone cellular phones. Those software solutions do provide simple 3D games and avatars, but the graphics performance is limited by the computation power of mobile devices. So new hardware solutions arrived to the market. ATI and nVidia introduced "Imageon" and "GoForce" using their knowledge of the PC market. Besides the traditional GPU vendors like nVidia and ATI, lots of challengers introduced great innovations (Figure 1.2). Imagination Technology's MBX/SGX employs tile-based rendering (discussed in Chapter 5) to reduce data transactions between GPU and memory. Although tile-based rendering is not widely

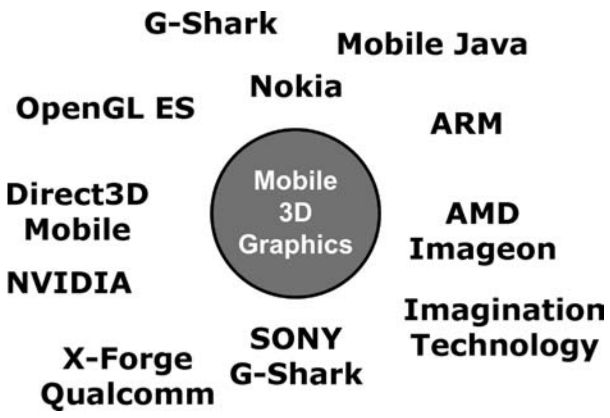


Figure 1.2 Mobile 3D graphics

used on the PC platform, it is very useful in reducing power consumption so that the MBX/SGX has become one of the major mobile GPUs on the market. FalanX and Bitboys developed their own architectures – FalanX Mali and Bitboys Acceleon – and they provided good graphics performance with low power consumption. Although those companies merged into ARM and AMD, respectively, their architectures are still used to develop mobile GPUs in ARM and AMD.

1.2 Mobile Devices and Design Challenges

As mentioned in the previous section, mobile devices have evolved at a rapid pace. To satisfy various user requirements there are lots of types of mobile device, such as personal digital assistant (PDA), mobile navigator, personal multimedia player (PMP), and cellular phone. According to their physical dimensions or multimedia functionality, these various devices can be categorized into several groups, but their system configurations are very similar. Figure 1.3 shows two leading-edge mobile devices and their system block diagram. Recent high-performance mobile devices consist of host processor, system memories (DRAM and Flash memory), an application processor for multimedia processing, and display control. Low-end devices do not have a dedicated application processor, to reduce hardware cost. Evolution of the embedded processor and display devices has led to recent exciting mobile computing.

1.2.1 Mobile Computing Power

In line with Moore's law [2], the embedded processors of mobile devices have been developing from simple microcontroller to multi-core processors and the computing

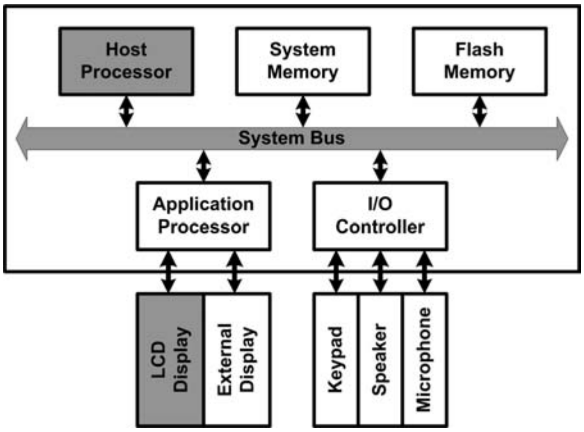


Figure 1.3 Mobile devices and their configuration

power has kept increasing roughly 50% per year. To reduce power consumption, an embedded processor employs RISC (Reduced Instruction Set Computer) architecture, and the computing power already exceeds that of the early Intel Pentium processors.

Typically, recent mobile devices have one or two processors as shown in Figure 1.4. Low-end devices have a single processor so that multimedia applications are implemented in software, while high-end devices have two processors, one for real-time operations and the other for dedicated multimedia operations. The host processor performs fundamental operations such as running the operating system, and personal information management (PIM). Meanwhile the application processor is in charge of

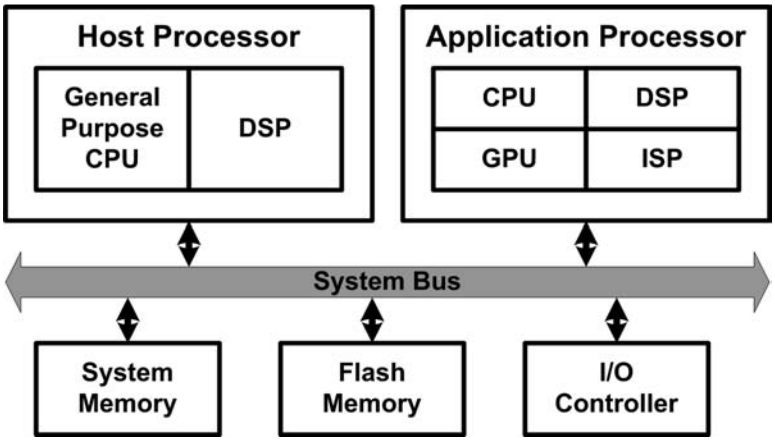


Figure 1.4 Embedded processors and system architecture

high-performance multimedia operations such as MPEG4/H.264 video encoding or real-time 3D graphics. To increase computing power, the newest processors employ multi-core architecture. Some high-performance processors contain both a general-purpose CPU and DSP together, and some application processors consist of more than four processing elements to handle various multimedia operations such as video decoding and 3D graphics processing.

1.2.2 Mobile Display Devices

It is safe to say that evolution of mobile display devices leads the revolution of mobile devices, especially the multimedia type. The first mobile devices had a tiny monotone display that could cope with several numbers or characters. Recent mobile devices support up to VGA (640×480) 24-bit true-color display. The material of the display device is also changing from liquid crystal to AMOLED (Active Mode Organic Light Emitting Diode). The notable advantages of AMPLED are fast response time (about 100 times faster than LCD), and low power consumption. Since it does not require back-lighting like the LCD, the power consumption and weight are reduced, and the thickness is roughly one-third of the LCD. Of course the functionality of the display device is improved too, so that nowadays we can use touch-screens on mobile devices.

1.2.3 Design Challenges

Although the funtionality of mobile devices is greatly improved, there are many design challenges in component design. In short, there are three major challenges.

Physical dimension – The main limitation of mobile devices is definitely their physical size. For portability the principal physical dimension is limited to about 5 inches (12.5 cm), and the latest high-end cellular phones do not exceed 4 inches. That means there is limited footprint on the system board, and components should be designed with small footprint.

Power consumption – Since the mobile device runs on a battery, the power consumption decides the available operating time. As the performance increases it consumes more power owing to the faster clock frequency or richer hardware blocks. Therefore, increasing operating time by reducing power consumption is as important as increasing computing power.

System resources – Mobile devices cannot have rich system resources owing to the physical dimension and power consumption. They cannot utilize a wide-width system bus and cannot use high-performance memory such as DDR2 or DDR3. Despite this, mobile devices provide quite high performance to satisfy user requirements.

To meet these design challenges, many mobile components are designed as SoC (System-on-a-Chip). Since the SoC includes various functional blocks such as processor, memory, and dedicated functional blocks in a single die, we can achieve high performance with low power consumption and small area.

1.3 Introduction to SoC Design

System-on-a-Chip has replaced key roles of VLSI (Very Large Scale Integration) and ULSI (Ultra Large Scale Integration) in mobile devices. The change of the name is a reflection of the shift of the main point from "chip" to "system." You may wonder what "system" means and what the difference is compared with "chip."

Before SoC, the hardware developer considered how to enhance the performance of the components. At that time, the hardware developer, the system developer and the software developer were separated and made their own domains. In the SoC era, those domains are merging. Engineers, be they a hardware engineer or a software engineer, have to consider both hardware issues and software issues and provide a system solution to the target problem with the end application in mind.

Of course, there are many different definitions of SoC according to the viewpoint, but in this book the *system* means "a set of components connected together to achieve a goal as a whole for the satisfaction of the user." To satisfy end-user requirements, the engineer should cover various domains. With regard to the software aspect, the engineer should consider the software interface such as API or device driver, specific algorithms, and compatibility. With regard to the hardware aspect, the engineer should consider functional blocks, communication architecture to supply enough bandwidth to each functional block, memory architecture, and interface logics. Moreover, since such a complicated entity can be handled only by CAD (Computer Aided Design) tools, the engineer should have knowledge of CAD, which covers automatic synthesis of the physical layouts.

Therefore, the discipline of SoC design is intrinsically complicated and covers a variety of areas such as marketing, software, computing system and semiconductor IC design as described in Figure 1.5. SoC development requires expertise in IC technology, CAD, software, and algorithms, as well as management of extended teams and project and customer research.

Initially, the concept of SoC came from the PC bus system. By adopting the same bus architectures as those used in the PC, the processing of embedded applications was to be implemented on a single chip by assembling dedicated hard-wired logic and existing general-purpose processors. As the scale of integration and design complexity increased, the concepts of "design reuse" and "platform-based design" were born. The well-designed functional blocks could be reused in the later SoC.

However, such pre-designed functional blocks, called Intellectual Property (IP), are difficult to reuse with SoC because they were optimally developed for specific purposes, not for general-purpose utilization. In addition, since conventional buses were not suitable for the on-chip environment, there was a need to develop new

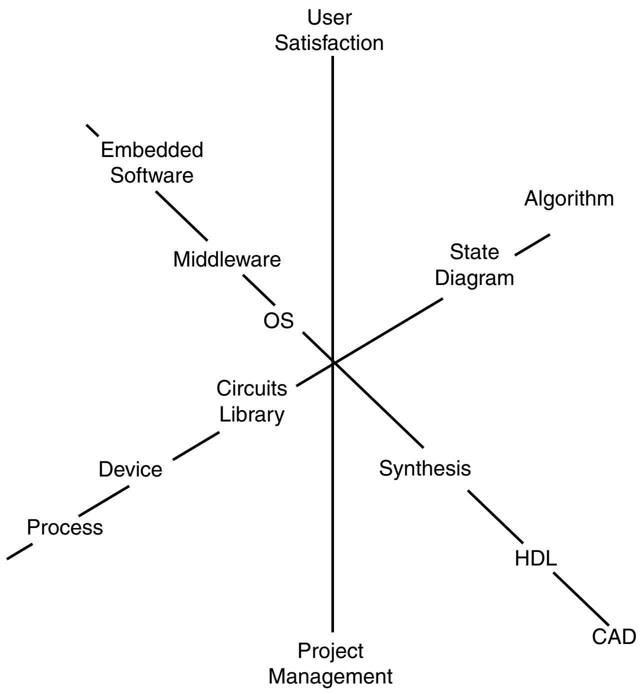


Figure 1.5 Disciplines required for the design of SoC

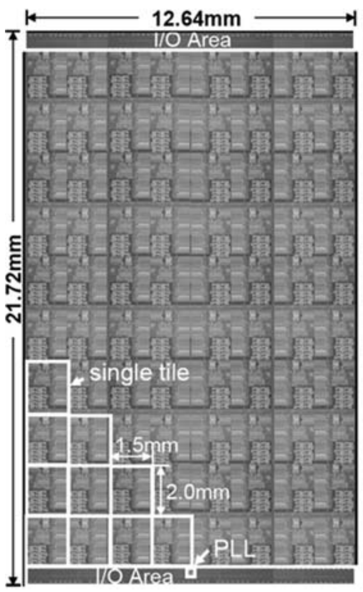
communication architecture with specific characteristics – such as wide bit width, low power, higher clock frequency, and a tailored interface. The details of design reuse and platform-based SoC design are discussed in Chapter 2.

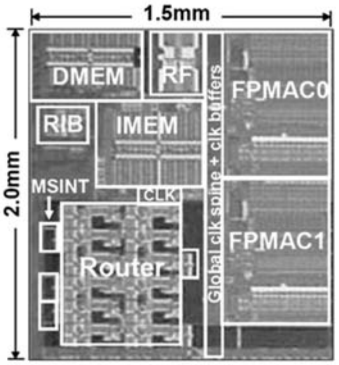
Figure 1.6 shows an example of SoC. Intel's research chip [3] has 80 CPUs inside.

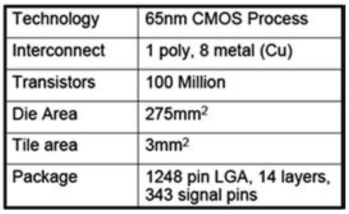
1.4 About this Book

This book describes design issues in mobile 3D graphics hardware. PC graphics hardware architecture with its shortcomings in the mobile environment is described, and several low-power techniques for mobile GPU and its real implementation are discussed.

Chapter 1 introduces the current mobile devices and mobile 3D graphics compared with desktop or arcade-type solutions. Chapter 2 discusses the general chip implementation issue, such as how to design the SoC, and includes an explanation of SoC platforms. The SoC design paradigm, system architecture, and low-power SoC design are addressed in detail. Chapter 3 deals with basic 3D graphics, the fixed-function 3D graphics pipeline, the application-geometry rendering procedure, and the programmable 3D graphics pipeline. In Chapter 4 we articulate the differences between conventional and mobile 3D graphics, and introduce the principles of mobile 3D graphics and standard mobile 3D graphics APIs.







Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm^2
Tile area	3mm^2
Package	1248 pin LGA, 14 layers, 343 signal pins

Figure 1.6 Example of an SoC implementation: Intel's 80-core processor and its unit CPU. The Intel logo is a registered trademark of Intel Corporation

The design of 3D graphic processors is discussed in Chapters 5–7. Chapter 5 explains the hardware design techniques for mobile 3D graphics, such as low-power rasterizer, low-power texture unit, and several hardware schemes for low-power shaders. Chapter 6 covers the real chip implementation of mobile 3D graphics hardware. For academic architecture, KAIST RAMP architecture is introduced and the industrial architectures, SONY PSP and Imagination Technology SGX, are also described. Chapter 7 has a detailed explanation of the low-power rasterization unit with RTL code. In this chapter, readers can grasp the basic concept of how to design low-power 3D graphics processors. The future of mobile 3D graphics is very promising because people will carry more and more portable equipment in the future with high-performance displays. Finally, Chapter 8 looks at the future of mobile 3D graphics.

We also include appendices to introduce to chip design by verilog HDL The reader can run the verilog file to check the algorithms explained in the earlier chapters and get a taste of real 3D graphics chip design.

References

1 Tolga Capin, et, al., "The State of the Art in Mobile Graphics Research", IEEE Computer Graphics and Applications, Vol. 28, Issue 4, 2008, pp. 74–84.
2 Gordon E. Moore, "Cramming more components onto integrated circuits", Electronics, vol. 38, no. 8, 1965.
3 J. Held, et al, "From a Few Cores to Many: A Tera-scale Computing Research Overview," white paper, Intel Corporation, www.intel.com.

2

Application Platform

2.1 SoC Design Paradigms

2.1.1 Platform and Set-based Design

2.1.1.1 Definition of a Platform

Two steps are encountered in any design process: “planning” and “making.” Certain procedures are followed when we want to perform meaningful tasks towards building a target structure. As the target structure takes on more complexity, well-established design procedures are essential. This applies in SoC design, which is strongly driven by its target applications such as multimedia and mobile communications. SoC engineers have to consider factors like quality, cost and delivery (QCD). In that sense, their design procedures naturally seek the reuse of previously developed techniques and materials at every possible design step.

In a popular English dictionary, a “system” is defined as a *set* and a *way of working* in a fixed plan with networks of components. In addition to this, SoC requires one more idea, which is the integration of components on a single semiconductor chip. So it follows that we need to focus on two concepts: the fixed plan, and integration. We can catch the concept of predetermined architecture from the fixed plan; and integration involves the network and component-based design. Considering that modern digital gadgets require not only hardware (HW) components but also software (SW) programs, we can begin to see what the “platform” means in SoC design.

The platform is a set of standalone modules that become the basis of the system. These standalone modules are pre-integrated and combine HW and SW components – we call them the “reference architectures.” They are also well-verified and have well-defined external interfaces. The platform guides what designers do, and this guidance determines the design flow. The platform concept helps us to design a more complicated and less buggy system within limited QCD factors by reusing and upgrading pre-built HW and SW components.

In this part of the chapter we will discuss what the platform is and what it does. We will explain how the platform can be extracted from earlier design examples and how it can be used for a new design. The concept of modeling and its relationship with the platform will also be examined. We then go on to discuss the system architecture and software design in detail in the following sections.

2.1.1.2 Platformization

Sometimes, use of the word "platform" seems to be a little confused. Many engineers tend to think that a platform is a kind of restriction. However, we need to consider a platform in two respects: its philosophy and its management. By philosophy we mean how a platform is derived from the ideas, theory and history of pre-designed samples. It also encompasses how we can use the derived platform for new designs. By management we mean the directions the platform – and the designs guided from that platform – should be evolved and maintained. We should avoid trying to make a design tool such as a design wizard program while developing the platform. The platform is not intended to generate design examples automatically. Instead, it is better to approach the platform as a design methodology, which is a set-based design. That can lead us to the many benefits of design planning and our design procedures.

When developing a platform for a given design set and research area, we will try to analyze pre-designed examples and extract some common ideas in those designs. The ideas may include target design specifications with a primary feature set, external interfaces and internal architecture. After collecting these common ideas, we can make the basic standalone modules and define the platform by reusing the individual components and arranging them under categories and levels of primary features. This procedure resembles inductive reasoning, which derives general principles from particular facts and instances. In this process, it is very important to categorize the primary features and link them to each specification level (such as low-, middle-, or high-performance levels) when building the reference architectures. Actually, when we design something, we are first given the target specifications and primary feature set to be designed. The detailed architecture and design plan doesn't matter for this step. We should plan to design our target based on previous examples and theory by using previous knowledge and experience. The platform is then the collection of our design history and theories. So, categorization and arrangement of primary features are the guideline to distinguish the reference architectures in the platform.

Now we are ready for our new design. The specification and external interfaces of our new design target may contain some parts of the pre-developed platforms. Some other parts may differ. However, we can usually find one best-matched standalone module in the platforms as a reference for our next design target. The parts that are common with the reference design can be reused in the new design. Some other parts might be developed by reusing and expanding the internal architecture and interface definitions of the reference design. This is the *set-based design approach* and resembles deductive reasoning, which generates specific facts and conclusions (our new design) from the general premises (our platform). Therefore, *platformization* can be understood as inductive and deductive reasoning, which helps us to develop a new, more complex design with very controlled and acceptable resources.

Figure 2.1 illustrates the design process. We have mentioned that the common ideas extracted from previous designs and theories contain the specification, external interface and internal architectures. In real designs, the specification and definitions of external interfaces tend to influence and decide the internal architectures to a certain extent. The definitions of internal architecture contain the following components.

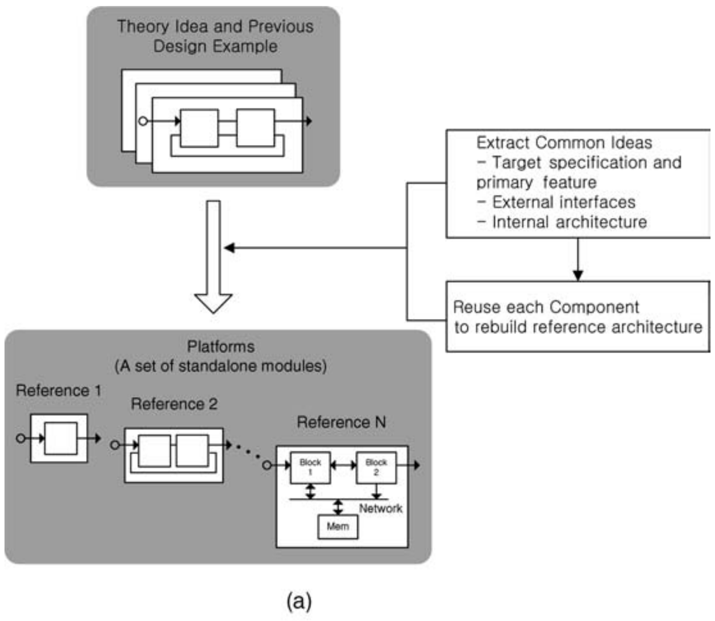
Primary processing elements – What kinds of task are required and what are the related computing units?
Memory architecture – What kinds of processing result are stored for next time and how many memories are required?
Internal network – How can the processing elements and memory components be connected and interfaced with each other? And how are the internal elements connected to external interfaces?
Programmer's model – How can software developers use the HW devices to complete the functions of the target design?

In set-based design, the reference architecture is applied as a starting point for the target design. As shown in the figure, there are four options: "As-is," "Modified," "New design," and "Removed." "As-is" means the reuse of components. Since the reference architecture is not the final design output, additions and modifications are always necessary. However, the set-based design approach can help us concentrate on the updated parts and reduce design costs.

2.1.1.3 Mobile 3D Graphics Example

The operational sequence, or *pipeline*, of mobile 3D graphics consists of geometry and rendering stages, which are explained in Chapter . In this subsection, the platformization of mobile 3D graphics will be briefly described as an example of the earlier discussion.

There are many design examples in mobile 3D graphics [1–3]. Like many multimedia applications, the Advanced RISC Machines (ARM) processor family that has the reduced instruction-set computer (RISC) architecture is most widely used as a main host processor because of its good performance and low power consumption [4]. Many mobile 3D graphics designs employ the ARM architecture with appropriate hardware accelerators. These HW accelerators can be divided into fully hard-wired logic and programmable architecture. As more functionality is required,


Theory Idea and Previous Design Example
Extract Common Ideas
- Target specification and primary feature
- External interfaces
- Internal architecture
Reuse each Component to rebuild reference architecture
Platforms (A set of standalone modules)
Reference 1
Reference 2
Reference N
Block 1
Block 2
Network
Mem
(a)

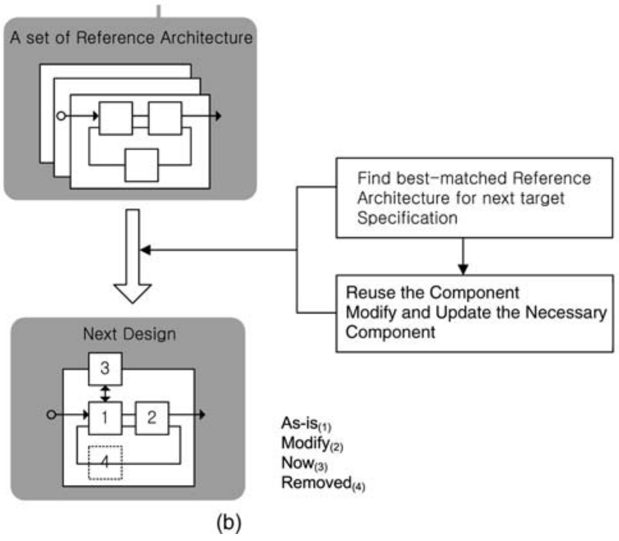

A set of Reference Architecture
Find best-matched Reference Architecture for next target Specification
Reuse the Component
Modify and Update the Necessary Component
Next Design
3
1
2
4
As-is(1)
Modify(2)
Now(3)
Removed(4)
(b)

Figure 2.1 (a) Platform and (b) set-based design

more programmability is integrated. In the software part, we have the industry-standard API, OpenGL-ES, for mobile 3D graphics [5]. It is also evolving from the application of compact and efficient architecture into integrating more programmability in the next version.

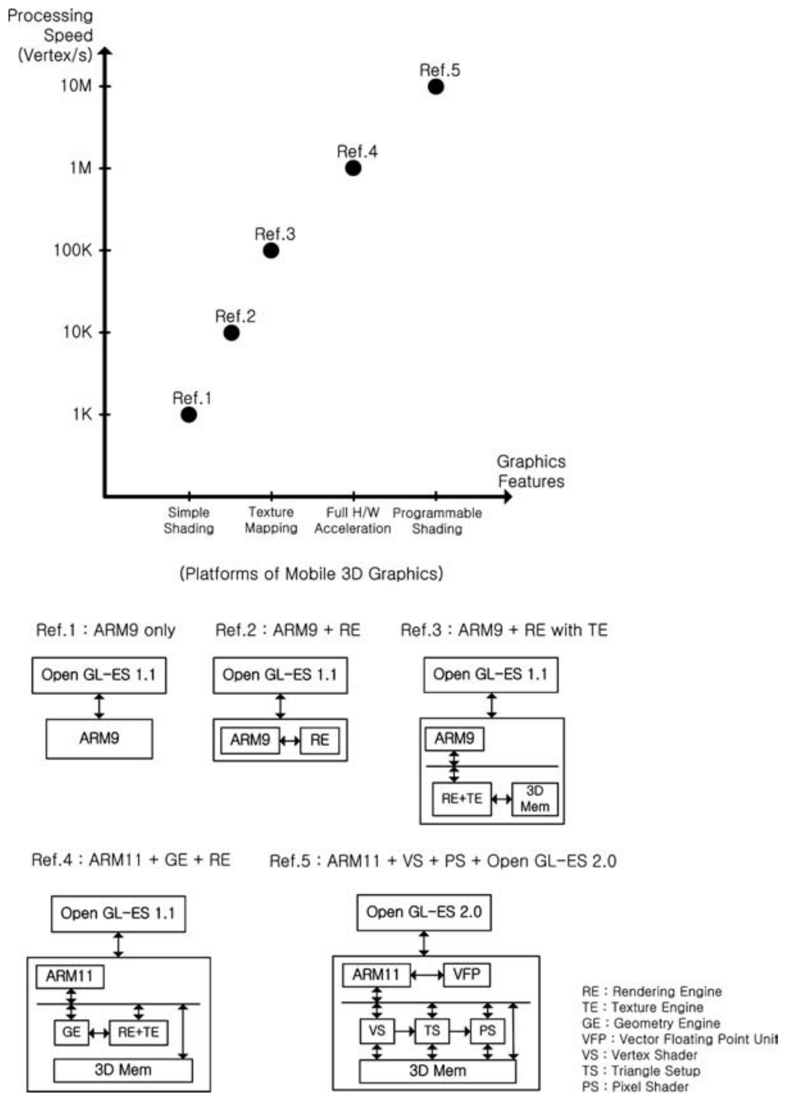


Figure 2.2 Platform of mobile 3D graphics

Figure 2.2 shows the range of graphics specifications and their reference architectures. For people wanting high-end graphics performance, programmability and full HW acceleration are necessary. In contrast, simple shading is required by some people who just want simple graphics such as the user interface of a small cellular phone. As more graphics functions are required, the processing speed must also be increased. As the target design moves towards high-end performance, more HW accelerators and programmability will be applied. The reference architecture depicted in the figure shows typical HW building blocks and the related software OpenGL-ES library. Some HW blocks such as a rendering engine (RE) and a texture engine (TE) are reused in two more of the reference architectures. The vertex shader (VS) and pixel shader (PS) are newly introduced in the reference architecture of the highest performance range. So, when designing a new mobile 3D graphics system including HW and SW, we can decide on the reference architecture by inspecting target specifications and graphics features. Then we can complete the design by reusing, updating and optimizing the additionally necessary SW and HW components based on the chosen reference architecture.

2.1.2 Modeling: Memory and Operations

2.1.2.1 Memory and Operations

How can the platform be derived from earlier design examples? Asking this question may help us to make and use the platform for our new design more efficiently.

When we design electrical components, there are certain requirements to drive signals or store information for future use. These actions are related to memory or, in other words, state variables. If an external stimulus and internal activity do not affect any part of the internal memory contents, we can say that nothing important happened. Then, after defining memories, we can consider what types of operation can be performed on them. So, we can imagine that an electrical component actually consists of the memory and the operations. In that sense, modeling can be defined as deciding on the memory architecture and its related operations for the target components. The design of an electrical *system* can be regarded as the process that defines its memories and operations by reusing and combining sub-components that are also defined as memories and operations.

Figure 2.3 outlines the modeling and design methodology. The basic elements can be defined by their memories and operations, and the memories can be called state variables. Then we have two design methods. The first is *modular design*. This means that every element can be developed independently and reused to provide multiple functions. Many interconnections – such as serial, parallel, and feedback networks – can be implemented. So, for example, one output of element A can be fed into one input of element B. The second method is *hierarchical design*. Here, elements can be organized as parent–child or tree-like structures to permit complex functions. The state variables are newly defined and the details of internal operations are encapsulated. This process can be repeated many times, step by step. Complicated designs are made possible by combining simpler elements.

Since complex designs can be divided into sub-elements, modularly and hierarchically, we can set up reference architecture for those complex designs. The reference architectures in the platform can be built by combining or selecting necessary

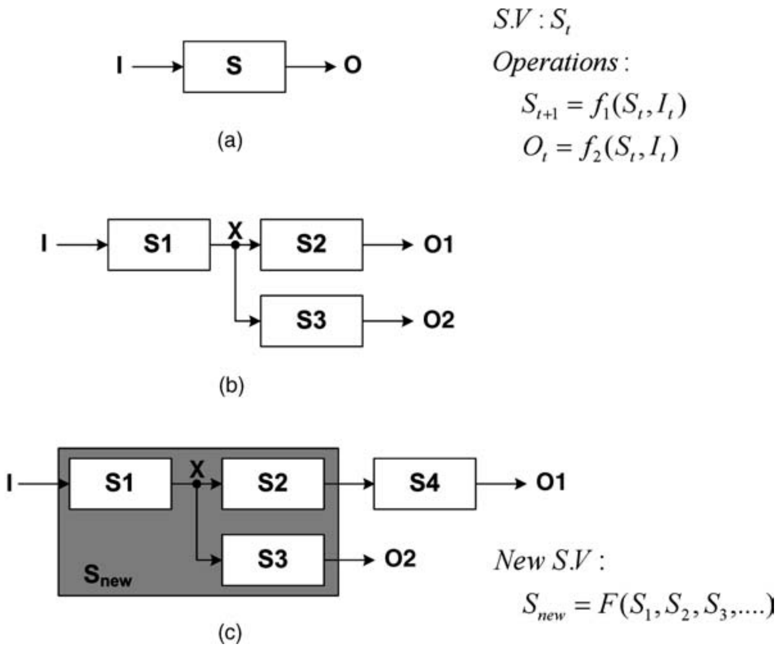


Figure 2.3 Modeling and design methodology: (a) modeling, (b) modular design, and (c) hierarchical design

sub-elements, and be reused by being combined and modified with other elements. We can update some parts of the reference design by changing the definitions of memories and operations. However, to maximize the efficiency of reuse in the reference design, we should restrict changes of input and output ports in the model of sub-elements as much as possible. This can be controlled because we know the influence of those changes by using modular and hierarchical design methods. Changing internal definitions of memories and operations does not result in changes in other parts of the whole design, and changing definitions of input and output ports can be clearly traced through the design hierarchy.

In the past, many designers have used flow charts or sequential diagrams to model their designs. However, as designs become more complex it becomes difficult to manage, update and reuse earlier designs with the flow chart method. It becomes difficult to understand the influence of changing elements. However, the sequential diagram is still useful in understating the behavior of a system when analyzing particular cases. Many design specifications are described by functional requirements, such as listening to music while viewing photographs. In that situation, the interactions of each sub-block should be clearly revealed to discover any insufficiency or bottleneck in the whole design. These interactions are triggered in the current sub-block by events in earlier blocks. The modular and hierarchical design methods, and therefore platform and set-based design, can help us not only to build the design but also to analyze particular cases of using the design, because the interfaces and internal architecture are clearly defined.

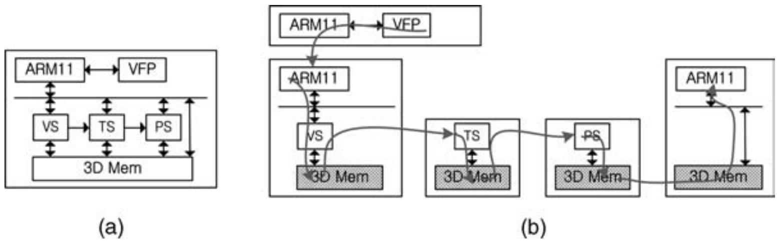


Figure 2.4 Example of use-case analysis: (a) block diagram, and (b) use-case analysis of a game

Figure 2.4 shows an example. Part (a) shows the block diagram of a mobile 3D graphics system that can perform full programmable graphics pipeline operations, including a vertex shader and a pixel shader. Part (b) illustrates a case of a game application including game logic operations and graphics operations. The game logic operations – such as game physics and artificial intelligence – are performed on the ARM11 host processor with a vector floating-point unit. Then the ARM11 commits the graphics commands into the graphics sub-system. The vertex shader is invoked first. Then a triangle setup and pixel shader follows. In this figure, note that the 3D memory block is accessed many times by multiple functional units. Finally, the ARM11 reads the final graphics results from the 3D memory. This analysis can inform us that the 3D memory block should be carefully designed for best performance.

2.1.2.2 Applications of Analog and Digital Designs

The modeling and design methodology discussed in the previous subsection can be applied to both analog and digital designs. Figure 2.5 shows examples.

In both analog and digital designs, devices manufactured with silicon materials are used – so-called semiconductor materials. The behavior of these materials is explained by physics and electromagnetic theories such as wave equations and Maxwell equations. From the viewpoint of memory and operation modeling, the electronic charges and vector fields (such as electronic and magnetic fields) are the memories. The values of those parameters represent the information carried by the materials. The governing equations define the operations performed on those memories.

In analog design, circuit elements such as field-effect transistors, resistors and capacitors are built using silicon materials. We can regard voltages and currents as new state variables, and Kirchhoff's current law (KCL) and Kirchhoff's voltage law (KVL) as new definitions of operations. Physics books describe how KCL and KVL can be deduced from Maxwell equations. Then we can build circuit blocks – such as

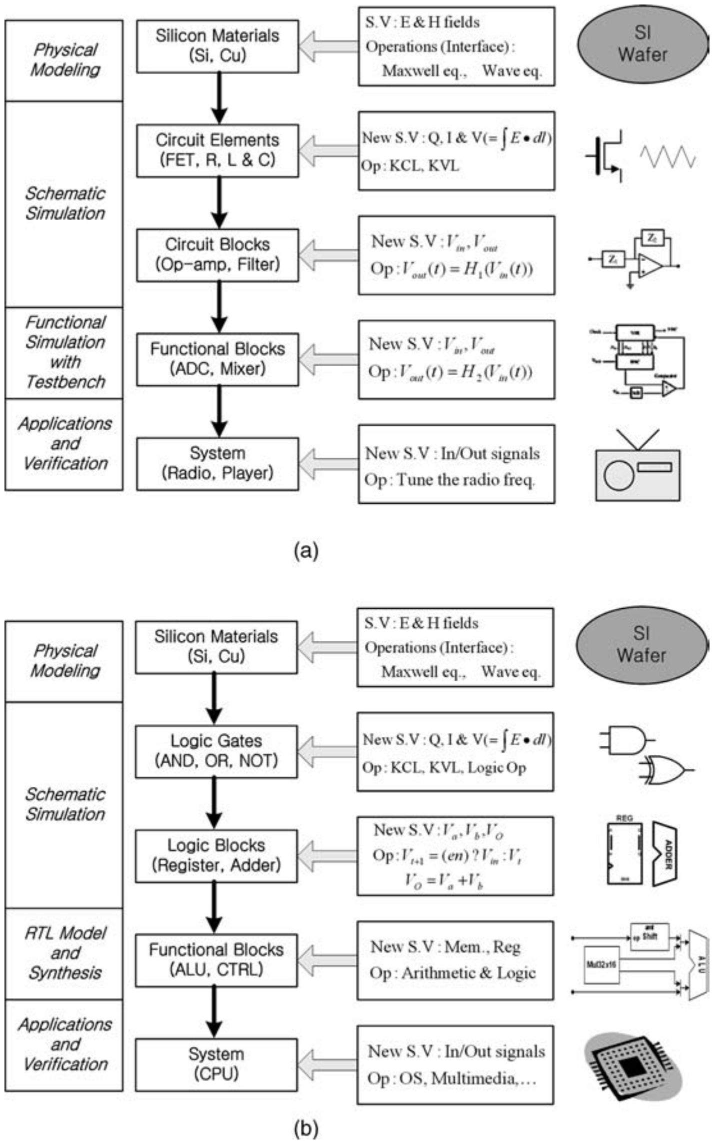

Physical Modeling
Schematic Simulation
Functional Simulation with Testbench
Applications and Verification
Silicon Materials (Si, Cu)
Circuit Elements (FET, R, L & C)
Circuit Blocks (Op-amp, Filter)
Functional Blocks (ADC, Mixer)
System (Radio, Player)
S.V : E & H fields
Operations (Interface) : Maxwell eq., Wave eq.
New S.V : Q, I & V($=\int E \bullet dl$)
Op : KCL, KVL
New S.V : V_{in}, V_{out}
Op : $V_{out}(t) = H_1(V_{in}(t))$
New S.V : V_{in}, V_{out}
Op : $V_{out}(t) = H_2(V_{in}(t))$
New S.V : In/Out signals
Op : Tune the radio freq.
SI Wafer
(a)
Physical Modeling
Schematic Simulation
RTL Model and Synthesis
Applications and Verification
Silicon Materials (Si, Cu)
Logic Gates (AND, OR, NOT)
Logic Blocks (Register, Adder)
Functional Blocks (ALU, CTRL)
System (CPU)
S.V : E & H fields
Operations (Interface) : Maxwell eq., Wave eq.
New S.V : Q, I & V($=\int E \bullet dl$)
Op : KCL, KVL, Logic Op
New S.V : V_a, V_b, V_O
Op : $V_{t+1} = (en)\,?\,V_{in} : V_t$
$V_O = V_a + V_b$
New S.V : Mem., Reg
Op : Arithmetic & Logic
New S.V : In/Out signals
Op : OS, Multimedia,...
SI Wafer
REG
ADDER
Mul32x16
ALU
(b)

Figure 2.5 Applying (a) analog and (b) digital designs

operational amplifiers and analog filters – by using circuit elements. The voltages at important nodes and currents in important circuit paths can now be introduced as new state variables. If we repeat the same process in steps, we can build functional blocks such as analog-to-digital converters, mixers and tuner, and finally an analog radio as the product. All these processes can be understood by the modular and hierarchical design methods.

Digital design also shows the same sequences. By using silicon materials and circuit elements, we can make logic gates such as AND, OR, and NOT. Then the logic blocks such as registers and adders can be developed. Again, the voltages at important nodes can be defined as the memories. By using logic blocks, we can build functional blocks such as an arithmetic and logic unit (ALU) and a control unit, and then finally the product such as a RISC processor can be released.

The above concepts in modeling, design methodology and platform should be kept in mind during all design processes. The use of a reference architecture and set-based design results in reduced design costs and permits the development of more advanced design targets. The modular and hierarchical approach based on memory and operation modeling can make it possible to divide complex problems and keep the focus on more easily handled sub-elements.

2.2 System Architecture

2.2.1 Reference Machine and API

2.2.1.1 Definition of Reference Machine

We have described the reference architecture as the standalone module that becomes the basis of the system, and the platform as a set of those standalone modules. Now we need to step inside the reference architecture.

When deciding to implement a real system by using the reference architecture, we have to consider which parts will be mapped into software and which into hardware. The software will run on general-purpose processing elements such as RISC processors or digital signal processors (DSPs). The hardware parts can be mapped into hardware accelerators or application-specific processors with their own instruction set, such as DirectX graphics shaders. However, before beginning the separation of HW and SW parts, we have to consider how programmers or applications engineers approach the target system efficiently. Programmers require function lists that cover all possible things they can do with the system. They do not need to know how the system works internally. On the other hand, hardware or system engineers need to know how each function is actually implemented in the system. They will also want to keep the feature set within a controlled range in order to ensure design feasibility. In relation to this we can introduce two concepts concerning the reference architecture.

The first concept is the *reference machine*. It is defined as a state machine that controls a set of specific (or target) functions. So, it represents all features to be implemented. Conceptually, the reference machine is composed of datapaths, local states, global states and selectors (Figure 2.6). *Datapaths* are the computing elements that represent the operations to be performed. *Local states* are the memories storing internal information for datapaths; they are not shared with other datapaths.

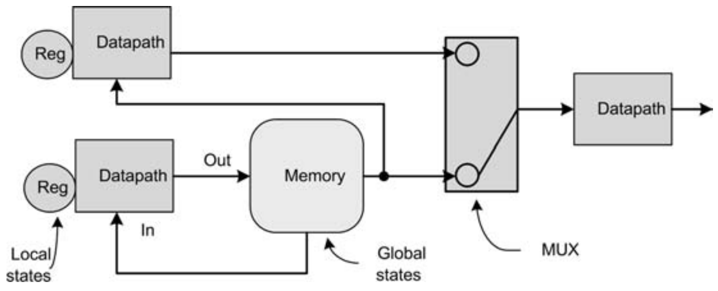


Figure 2.6 Elements of a reference machine

Global states are the memories that can be shared between datapaths. *Selectors* interconnect between datapaths. The output port of one datapath is connected to the input port of another datapath. The selector also performs operations such as multiplexing of multiple inputs, which are controlled by some parameters.

The second concept is the *application programming interface* (API). It can be defined as the programmer's interface to target the reference machine. It totally encapsulates the internal structure of the reference machine. APIs can be categorized as data processing operations, control operations and memory operations. They are related, respectively, to datapaths, selectors and state variables in the reference machine. Therefore, any application algorithm expected to use the target reference machine should be described by using only the defined APIs. Any additional functions not covered by the APIs should be implemented by using other computing elements or processors that are outside the target reference machine.

After defining the reference machine and APIs, the system architect has to decide which parts of the reference machine will be implanted as hardware and which as software, by analyzing the performance requirements (covered in a later section).

We can say that the reference architecture is composed of the implementation of the reference machine and its APIs. Of course, the APIs can be also reused between different reference architectures. The implementation level of the reference machine is decided by the target performance requirements, so we can state the following definition and simple equation:

Platform: a set of reference architecture
Reference architecture = implementation of reference machine + API.

Building the reference machine and APIs in a given design problem is not an easy task, but the simplest approach is to use memory and operation modeling. After defining the memories and operations for each algorithmic description of the design problem, the system architect can merge and rearrange the primary operations modularly and hierarchically in order to build the reference machine. In this, the definitions of local and global states are very important.

In mobile 3D graphics, there are now industry-standard APIs. OpenGL-ES is a well-defined subset of desktop OpenGL, and adopts various optimizations such as fixed-point operations and redundancy eliminations for mobile devices with low processing power. In its latest version, OpenGL-ES enables fully programmable 3D graphics such as vertex and pixel shading. Mobile 3D graphics are being improved towards even more functionality and programmability in both of hardware and software, while achieving low power consumption.

2.2.1.2 SoC Design Flow

Figure 2.7 shows the design process from target specification to manufacture, with the focus on system architecture.

The target specification defines the type of product and rough performance requirements. Suitable algorithms are chosen to meet the specification using computer programs such as a programming language, UML [6] or MATLAB.

In the system specification, the performance of the system is given but the implementation details are not determined yet. The fact that the system has to be implemented on a chip (SoC) means that the software running on the embedded CPUs should be designed concurrently. This is the big difference between SoC design and VLSI (very large scale integrated) design. Determination of which parts of the specification will be implemented in hardware and which in software is included in the design process.

Once the concept of the target system has been grasped, the set of functions to realize the system specification should be derived and divided into more affordable unit functions. Therefore, the functional specification of a system is determined as a set of functions which calculate outputs from the inputs.

As mentioned in the previous subsection, the reference machine and its APIs can be developed from the algorithm descriptions by using memory and operation modeling. Then, we go into the step of product definition. This involves the naming of the product, "is/is-not" analysis, priority analysis, and competitive analysis. "Is/is-not" analysis is the process of defining the desired level of development: unnecessary features should be identified in order to prevent wasting design resources. Priority analysis includes the schedule, costs, and power consumption. Because the development process is always controlled by a limited delivery time, some parts of the first design plan might have to be abandoned.

Next in the design flow comes the system architecture: reference architecture selection, system specification, target architecture selection, and performance analysis. With the production definitions and the developed reference machine, a suitable platform and one well-matched reference architecture in that platform can be selected. This reference architecture can be used in the remaining design steps using

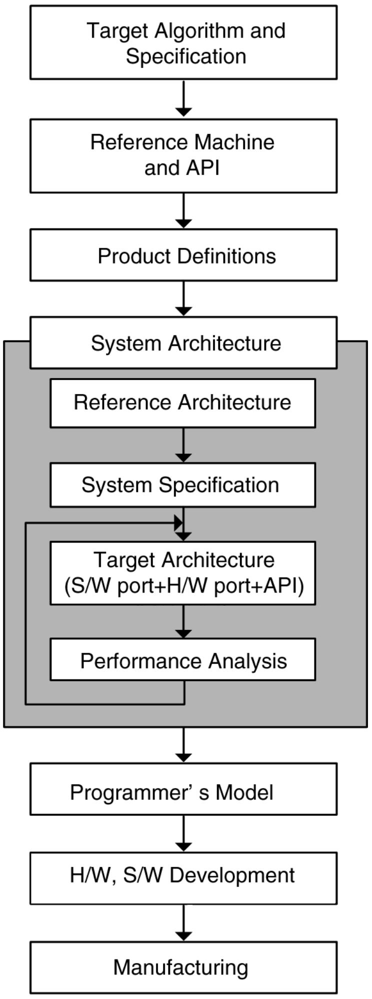


Figure 2.7 Design flow

the set-based design approach. Then, documentation of the system specification can be prepared.

The system specification can contain the following items:

1. Summary of product requirements
2. List of features
3. Top-level block diagram
4. Use case descriptions
5. Availability and status of functional blocks (As-is, New, Modified, Removed)
6. Block specification

7. Integration and communication specification
8. Power specification
9. External interface (Pin list, Pin multiplexing, Package) specification.

Although the initial specifications and performance requests are a little indistinct, the system specification describes the target product more concretely. By using the system specification and reference architecture, we can decide the internal architecture of software and hardware parts in more detail. Having decided the target architecture, we can estimate factors such as silicon area, power consumption, processing speed and memory requirements. For each use case, the performance analysis is performed to reveal bottlenecks and wastage of resources. If the performance does not meet the requirements, the target architecture should be modified. So the target architecture and performance analysis will be repeated until the performance meets the target requirements.

When the system architecture is finalized, the programmer's model for the target product should be defined. The APIs are a kind of top-level software interface. To realize each API function, there should be invocations of some hardware blocks or processing elements that have their own internal architectures. Therefore, we need descriptions of how the API developers can use each functional block in the target architecture. The programmer's model defines the behavior of each functional block. The following items can be identified:

1. Memory map and instruction sets
2. Memory format and memory interface
3. Register set
4. Exception, interrupts and reset behavior
5. External interface and debug interface
6. Timing and pipeline architecture.

At this stage we have all the descriptions of internal architecture in both the software and hardware parts. The remaining steps are SW/HW development and manufacture. Common semiconductor design processes such as register transfer level (RTL) descriptions with synthesis and custom design with circuit simulations can be employed.

2.2.2 Communication Architecture Design

2.2.2.1 Data and Command Transfers

The transfer of information between building blocks is crucial in the electrical design of, especially, multimedia applications such as 3D graphics. When we map the reference machine to a real implementation, we can observe that one component of a system is producing something that is immediately consumed by another component of the system. In fact, this features multimedia signal processing itself.

Going deeper into the real implementation, there are two kinds of information transfer: *command transfer* and *data transfer* (Figure 2.8). Command transfer uses information to control operations of building blocks such as register settings or small program codes. Mostly, command transfer is not shown in the diagram of a reference machine; it appears clearly after the step of system architecture is finished. Data transfer uses information to give intermediate results from the current processing block to the next processing block. As explained in the definition of the reference machine, there are transfers of local states and global states in these data transfers. In general, the bandwidth of data transfer is higher than for command transfer.

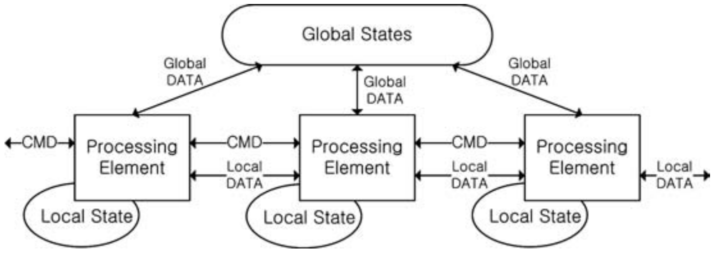


Figure 2.8 Data and command transfers

2.2.2.2 On-chip Interconnections

The demand for high performance of semiconductor devices has required increased operating frequency of the silicon chip. However, owing to the difficulty of implementations such as clock distributions, the design approach of SoC with multiple functional units has been widely adopted in multimedia and communication applications. As mentioned earlier, on-chip interconnections between the functional units influence the whole system performance in SoC design.

Generally, the host processor in the SoC has its own instruction set and memory space. Therefore, in the programmer's model, the functional unit can be attached in the memory space or in the instruction set. In a real implementation, the former adopts on-chip bus architecture and the latter adopts coprocessor architecture. The on-chip bus provides shared data wires that can be connected with data ports of multiple functional units. These data wires can be unidirectional or bidirectional. Each address port of the functional units is decoded and arbitrated by bus arbiters in the on-chip bus architecture. If the programmer accesses the address space mapped to some functional units, the bus arbiters enable the related functional unit to access the shared data bus while keeping other functional units from interrupting the data transactions. There can be multiple layers and multiple arbiters in the on-chip bus architecture for increased performance.

In the modern embedded RISC processor, the coprocessor is defined as a general mechanism for extension of the instruction set architecture. The coprocessors have their own private register set and state, and these are controlled by coprocessor instructions that mirror the host processor instructions controlling the host processor's register set. The host processor has sole responsibility for flow control, so the coprocessor instructions are concerned only with data processing and data movement. Following RISC load–store architectural principles, these categories are cleanly separated.

Figure 2.9 compares the on-chip bus and coprocessor architecture in terms of data and command transfers. Conventional bus architecture implies that an additional hardware block attached in the memory space should be connected with the data port of the main processor. This is because a modern embedded RISC processor does not have a dedicated port for memory-mapped components. Therefore, the command transfers of hardware blocks use the bus shared with main memory transactions, causing inefficient utilization of processing elements. In addition, multi-layer bus architecture requires complex interconnections including multi-port arbiters with long and wide global metal wires, leading to high power consumption. Also, concentrated data transactions may cause heavy bus arbitrations, and the main processor should always consider thread synchronizations in invoking bus-attached

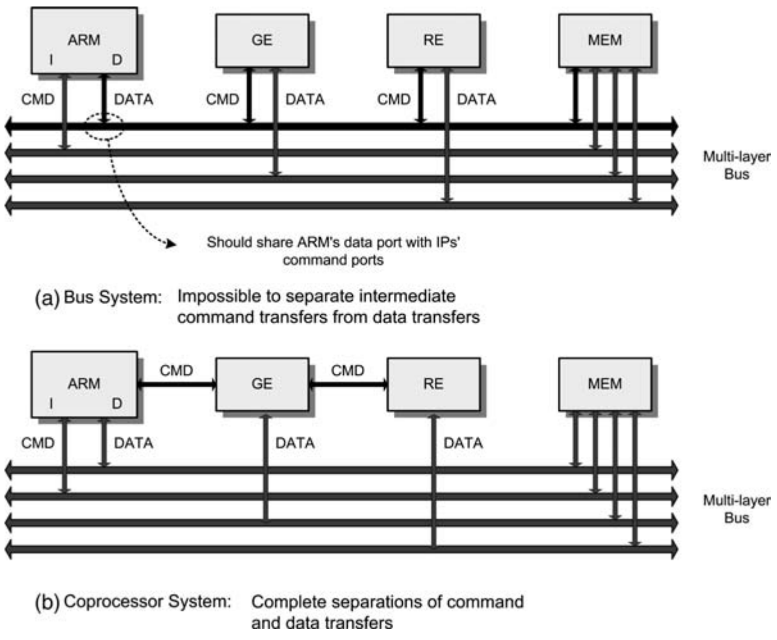

ARM
I
D
GE
RE
MEM
CMD
DATA
CMD
DATA
CMD
DATA
Multi-layer Bus
Should share ARM's data port with IPs' command ports
(a) Bus System: Impossible to separate intermediate command transfers from data transfers
ARM
I
D
CMD
GE
CMD
RE
MEM
CMD
DATA
DATA
DATA
Multi-layer Bus
(b) Coprocessor System: Complete separations of command and data transfers

Figure 2.9 Bus and coprocessor

hardware blocks. On the other hand, the coprocessor system shows the following features.

a. A direct signal path with short coprocessor interfaces provides simple interconnections. Coprocessors share a bypassed instruction port with the main processor. They do not need bus arbitrations for hardware access, unlike conventional bus-attached hardware accelerators. Therefore, the coprocessor interface can reduce unwanted delays between the main processor and hardware accelerators, and thus relevant power consumption.
b. Since the coprocessor operates in locked step with the core pipeline of the main processor, complex synchronization is avoided.
c. Since the commands of the coprocessor are regarded as extended instruction set architectures of the main processor, easy programmability can be achieved.

However, the interface coprocessor architecture is strongly dependent on the architecture of the host processor while the on-chip bus uses a typical memory interface. Therefore, the coprocessor cannot fully achieve the reusability of platform and set-based design. Which on-chip interconnections should be used will be determined by the performance requirements and availability of other functional blocks.

Recently, a new on-chip interconnection scheme has been introduced in SoC design. The network-on-a-chip (NOC) uses computer network concepts in its on-chip interconnections [7]. Instead of a circuit-switched network of conventional bus architecture, packet data transfers and fast low-voltage serializations achieve high data bandwidth while keeping the power consumption low.

2.2.3 System Analysis

One of the most important tasks of a system architect is the analysis of system performance during the development of system specifications. Generally the items to be estimated are memory bandwidth, memory capacity, processing speed, power consumption, and silicon die area.

The memory capacity and memory bandwidth can be analyzed by checking defined use cases. According to the definitions of memory mapping and the formats of memory contents such as addressing schemes, the required memory bandwidth and memory capacity can be varied. Careful assignment of memory space is very important. Use case analysis can clarify which memory accesses should be done separately. Because dynamic random access memory (DRAM) is widely used, the estimation margin should be addressed to compensate for read and write latencies. The total required memory bandwidth should be restricted to less than 60% of peak DRAM bandwidth in most cases.

Power consumption and processing speed, too, can be analyzed based on use case descriptions. The power consumptions of building blocks in earlier designs can be used as parameters for power estimations. Voltage scaling and other technology advantages can be also considered. The active power of processing can then be computed by summing the power consumptions of all functional units. If the power consumptions are known in terms of mW per MHz, the operating frequency will determine the actual power consumption, and the operating frequency is determined by use case analysis taking into account processing loads in the functional units. The leakage power can be computed by estimation of the equivalent number of basic logic elements such as 2-input NAND gates for a given functional unit. Many semiconductor manufacturers provide the leakage power consumption of these basic logic gates under various operationing conditions, such as voltage and temperature. We can make use of the following relations in power estimations:

Total power consumption = active power + leakage power
Active power of RISC sub-system = constant power + (core factor(mA/MHz) + memory factor(mA/MHz) + peripheral factor(mA/MHz))*voltage*frequency
Active power of HW accelerators = gate power factor(mA/MHz/gate)*gate count*activity level*voltage*frequency
Leakage power = logic leakage power + memory leakage power
Logic leakage power = 2-input NAND leakage power*gate count
Memory leakage power = bit-cell leakage power*capacity + 2-input NAND leakage power*memory array logic factor*capacity

Estimation of the silicon die area is important because it influences the selling price of the silicon chip. Although there is likely to be some overhead to account for internal interconnections, summation of the silicon areas of the functional units can provide meaningful information before real implementations. Of course, the shrink-down effect of semiconductor technology advances should be considered when using parameters from earlier designs. Many semiconductor manufactureres also provide the routing efficiency and integration density in terms of the number of transistors per unit area. The following simple equations can be used in die area estimations:

Logic area = gate count*gate density
Memory area = equivalent gate count*gate density or memory macro area
Sub-modulelogic area = sum of building block logic area*(1 + PnR fix-cell overhead)*(1 + clock tree overhead)*(1 + hold-time fix overhead)*(1 + large buffer overhead)
Sub-module macro = sum of macro block area*(1 + macro overhead)
Sub-module total area = sub-module logic area + sub-module macro area
Top core area = sum of total logic area + sum of total macro area*(1 + macro placement overhead)
Total chip area = (square root of top core area + 2*(IO, power ring and scribe width per side))$\hat{2}$

2.3 Low-power SoC Design

Low-power design methodologies are well developed and are actively employed in the design of SoC for cellphones [8–11]. Low power operation can be obtained at each design level. This section briefly introduces the principles.

2.3.1 CMOS Circuit-level Low-power Design

CMOS logic devices consume power when they are operating. There are two major elements to active power dissipation: dynamic switching power and short circuit power. A third element is the leakage power that results from sub-threshold current, or the current flowing through a MOSFET when $V_{gs} = 0$ V. The total power is given by the following equation:

$$P_{\text{total}} = P_{\text{switching}} + P_{\text{short-circuit}} + P_{\text{leakage}} = \alpha_{0 \to 1} C_L V_{dd}^2 f_{CLK} + I_{sc} V_{dd} + I_{\text{leakage}} V_{dd}.$$

Low-power design methods aim to decrease power dissipation by reducing the values of $\alpha_{0 \to 1}$, $C_{L,} V_{dd}^2$, and f_{CLK}. Various techniques and their effects on the terms of the power equation are summarized in Table 2.1. The node transition activity factor is a function of the Boolean logic function being implemented, the logic style, the circuit topologies, signal statistics, signal correlations, and the sequence of operations. However, most of the factors affecting the transition activity are determined by the logic synthesis EDA tools.

Table 2.1 Summary of low-power techniques for SoC

Toggle count	Logic style; transition reduction coding
Load capacitance	Wire length minimization; partial activation
Voltage scaling (VS)	Small swing
	Multi V_{dd}; dynamic VS; adaptive VS
	Multi V_{th}; variable V_{th} (substrate bias); negative V_{gs}
	Power shutoff; power gating
Frequency scaling (FS)	Clock gating; dynamic FS

2.3.2 Architecture-level Low-power Design

There are many low-power schemes above the level of register transfer level designs. The most common method is clock gating, which disables unnecessary blocks in the synchronous system. The clock is connected to the internal circuits through an AND gate which is controlled by the gate enabling signal. This scheme can be applied block by block to selectively control the power consumption.

At the architecture level, *parallelism* can be used to reduce power consumption. For example, if one puts an identical functional module in parallel with the original one,

one can double the throughput of the functional operation, and the clock frequency can be halved if the throughput is the same as for the original one. Pre-computation can remove unnecessary *toggles* too. Before the main operation of the circuit, a part of the circuit is pre-computed and the internal switching activities of the main circuit are controlled by using the pre-computed results to reduce the number of toggles.

2.3.3 System-level Low-power Design

A SoC or subsystem has one or more major functional modes, prime examples being operational mode, idle mode, sleep mode, and power-down mode. The operational mode is when the SoC operates its normal functions. In the idle mode, the clock block is ON but no signal is switching. In the sleep mode, even the clock part is OFF as well as the main blocks. When the SoC is turned off with the power supply connected, the SoC is in power-down mode. At the system level, low-power solutions are “multi-supply voltage” or “voltage scaling,” “power shut-off,” “adaptive voltage scaling,” and “dynamic voltage and frequency scaling.”

In system-level low-power schemes, the SoC is divided into multiple voltage and frequency domains, and then it adopts DVFS (*dynamic voltage–frequency scaling*), AVS (*adaptive voltage scaling*), and *power shut-off* or *power gating* to control the power dissipation in each domain.

2.4 Network-on-Chip based SoC

As chip integration evolves, current SoC designs incorporate a number of processing elements to meet performance requirements with reasonable power consumption [1–3]. This design trend makes it simpler to achieve high performance with moderate design effort because a verified processor core can be replicated. In addition, SoC design requires integration of numerous peripheral modules such as on-chip memory, an external memory controller, and I/O interfaces. As a result it is very important to provide efficient interconnections between numerous processing cores and peripheral modules within an SoC.

Traditional bus-based interconnection techniques are not suitable for current large-scale SoCs because of their inherent poor scalability, so a design paradigm based on network-on-a-chip (NoC) has been proposed as a solution for on-chip interconnection of large-scale SoCs [4, 5]. The modular structure of NoCs makes chip architecture highly scalable, and well-controlled electrical parameters of the modular block improve reliability and operation frequency.

There have been many architectural and theoretical studies of NoCs, such as design methodology, topology exploration, quality-of-service (QoS) guarantees, and low-power design. In this section, basic NoC design issues and building blocks are briefly described, and then practical NoC design considerations and case studies for real chip implementations are introduced.

2.4.1 Network-on-Chip Basics

2.4.1.1 Homogeneous and Heterogeneous SoCs

Network-on-chip architectures are emerging as a strong candidate for a highly scalable, reliable, and modular on-chip communication infrastructure platform for high-performance SoCs [12, 13].

To date, there have been two distinct types of SoC: homogeneous and heterogeneous. With *homogeneous* SoCs, suitably designed modules are replicated and placed on a single chip in a regular topology. MIT's RAW processor [16] and Intel's 80-tile processor implementation [11] are examples with two-dimensional mesh topology. These processors consist of modular tiles that include a processing core and 5-port crossbar switch. In each tile, the 5-port crossbar switch provides connections to the four neighboring tiles of a mesh topology and processing core inside the tile. For homogeneous SoCs, crossbar switches are commonly adopted instead of the conventional bus, because the non-blocking characteristic of the crossbar switch has the advantage of enabling concurrent interconnections among multiple processing cores and peripheral modules. Figure 2.10a depicts how the homogeneous SoC is organized in a regular array structure.

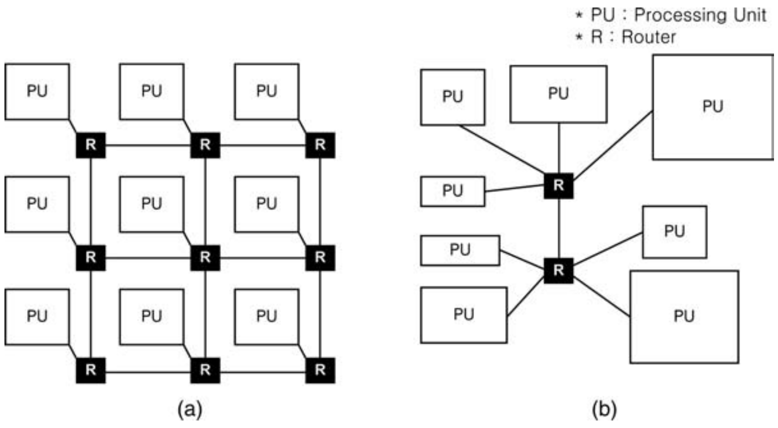


Figure 2.10 Two types of SoC: (a) homogeneous, and (b) heterogeneous

A *heterogeneous* SoC [17, 18] integrates various functional modules that are usually dedicated to accelerating specialized computations of the target application. With a heterogeneous SoC it is difficult to build up regular structures – such as a two-dimensional array of processors – because the sizes of the functional modules are different from each other. In addition, adopting regular interconnections may result in waste of wire resources owing to highly localized and fixed traffic patterns among functional modules. Therefore, in most cases, a heterogeneous SoC requires optimized interconnections that are tailored to the data traffic patterns of the target application. Figure 2.10b shows a simplified example.

Entirely homogeneous or heterogeneous SoCs have been introduced above to highlight the distinctions between the two types, but hybrid versions will be in widespread use in the near future. The homogeneous architecture is mainly beneficial for providing huge computing power, so integrating additional functional modules such as external I/O or application-specific accelerators is usually necessary, and this results in the need for a hybrid architecture. Then, scalablity and structured interconnections are essential to establish efficient interconnections for a complex SoC architecture. When compared to the conventional bus and point-to-point architecture, NoC has huge advantages. The next subsection expands on the benefits of adopting NoC.

2.4.1.2 Comparison of NoC and Buses

NoC-based SoC design uses two major concepts that are distinguishable from those of bus-based SoC architecture. There are packet transactions rather than circuit transactions, and there is a distributed network structure instead of a conventional globally shared bus or a centralized matrix. In NoC-based SoC design, each of the functional modules should be designed to be latency-insensitive, to support packet transactions. Although this makes functional module design slightly more complex, many benefits are gained from having packet transactions. It improves reliability and the speed of interconnection links, because packet transactions are intrinsically pipelined so that the physical lengths of interconnection links can be kept short. Efficient link utilization is another advantage, because only part of the end-to-end path between functional modules is occupied by the traversing packets. It is also advantageous that the electrical parameters of one NoC are not influenced by the addition of other NoC modules, owing to the structured characteristic of the NoC. This enables the building up of large-scale SoCs from smaller existing components by the addition of any required functional modules. For all these reasons, advanced bus architectures, too, are gradually considering a packet transaction concept into their protocols; examples are multiple-outstanding addressing, split transactions, and multi-threaded transactions [19, 20]. In the near future, it is expected that commercial bus architectures for a high-end SoC will take the NoC design into their specifications [21, 22].

As mentioned earlier, the NoC paradigm arose to alleviate the design complexity of a very-large-scale SoC design that could not be fabricated on a single chip. As manufacturing scale goes further down into the very deep submicron (VDSM) level, communication between integrated modules becomes more complicated and unmanageable with a conventional design methodology. Communication itself becomes a design bottleneck in relation to performance, design effort, area cost, and power consumption. The NoC paradigm tries to solve this problem based on a well-defined layered architecture that is much like the Open System Interconnection (OSI) basic reference model [23]. This methodology divides the communication mechanism into several layers that are independent of each other, so that the design procedure becomes more manageable and easily modifiable. The detailed correspondence of NoC to the OSI reference model will be described in a later subsection.

First we will briefly summarize the pros and cons of NoC-based design and bus-based design as shown in Table 2.2. For many reasons, NoC-based design is advantageous; the details are addressed in the table.

2.4.1.3 NoC Design Issues

Figure 2.11 illustrates the overall architecture of an NoC. First, an appropriate topology and protocol should be selected when NoC design begins. The NoC topology can be configured with regular topologies such as "mesh," "torus," "tree," or "star." Alternatively, an optimization can be carried out to build an application-specific topology without a regular pattern. Second, protocols including packet format, end-to-end services, and flow control should be defined and implemented in a network interface (NI) module. Packet size definition affects the buffer capacity requirement and multiplexing gain in the network.

The packet switching scheme is the next factor to be determined. There are many switching methods such as store-and-forward, wormhole switching, and cut-through switching.

- **Store-and-forward.** The entire data of a packet at the incoming link are stored in the buffer for switching and forwarding. A buffer with a large capacity is required.
- **Wormhole routing.** An incoming packet is forwarded right after the packet header is identified and the complete packet follows the header without any discontinuity. The path that the packet occupies traveling through the switch is blocked against access by other packets.
- **Virtual cut-through.** The path is determined as with wormhole routing. If the next hop is occupied by another packet, the packet tail is stored in a local buffer to await clearing of the path. Its buffer size can be smaller than the store-and-forward switch; but if the packet size is large, many local buffers are occupied to delay the switch throughput.

Choosing an appropriate switching scheme to suit the target application and silicon resource budget is necessary. In many NoC implementations a wormhole routing scheme is chosen because of its lower buffer resource requirement.

Once the basic topology, protocol, and routing method have been determined, the operation of the crossbar switch is as follows. When the input packets arrive at the input port, the crossbar switch scheduler gets the destination information from the input packets. If every packet arrives at a different input port and wants to leave from a different output port, there are no input or output conflicts. Then, the scheduler connects the cross junctions to connect the packets at the input ports to their output

Table 2.2 Comparison between NoC-based and bus-based design methodologies

	Network-on-chip			Bus
Bandwidth and speed	• Nonblocked switching guarantees multiple concurrent transactions.	+	–	
	• Pipelined links: higher throughput and clock speed.			• A transaction blocks the other transactions in a shared bus.
	• Regular repetition of similar wire segments which are easier to model as DSM interconnects.			• Every unit attached adds parasitic capacitance; therefore electrical performance degrades with growth.[a] [24]
Resource utilization	• Packet transactions share the link resources in a statistically multiplexing manner.	+	–	• A single master occupies a shared bus during its transaction.
Reliability	• Link-level and packet-basis error control enables earlier detection and gives less penalty.	+	–	• End-to-end error control imposes more penalty.
	• Shorter switch-to-switch links, more error-reliable signaling.			• Longer bus wires are prone to errors.
	• Rerouting is possible when a fault path exists. (self-repairing).			• A fault path in a bus is a system failure.
Arbitration	• Distributed arbiters are smaller, thus faster.	+	–	• All masters request to a single arbiter; thus the arbiter becomes big and slow, which degrades bus speed.
	• Distributed arbiters use only local information, not a global traffic condition	–	+	• Central arbitration may make a better decision.
Transaction energy	• Point-to-point connection consumes the minimum energy.	+	–	• A broadcast transaction needs more energy.
Modularity and complexity	• A switch/link design is re-instantiated, thus less design time.	+	–	• A bus design is specific, thus not reusable.
	• Decoupling between communicational and computational designs			
Scalability	• Aggregated bandwidth is scaled with network size.	+	–	• A shared bus becomes slower as the design bigger, thus it is less scalable.

Clocking	• Plesiochronous, mesochronous, and GALS fashion don't need a globally synchronized clock: very advantageous for high-speed clocking.	+	−	• A global clock needs to be synchronized over the whole chip bus area.
Latency	• Internal network contention cause a packet latency. • Repeated arbitration on each switch may cause cumulative latency. • Packetizing, synchronizing, interfacing cause an additional latency.	−	+	• Bus latency is wire-speed once a master has a grant from an arbiter.
Overhead	• Additional routers/switches and buffers consume area and power.	−	+	• Less area is consumed.[b] • Fewer buffers are used.[b]
Standardization	• There is no NoC-oriented global standard protocol yet; however, legacy interfaces such as OCP and AXI can be used.	−	+	• AMBA and OCP protocols are widely used and designed for many functional IPs.

[a]Recent advanced buses are using pipelined wires by inserting registers in between long wires [14, 15].
[b]Recent advanced buses use crossbar switches and buffers (register slices) in their bus structure.

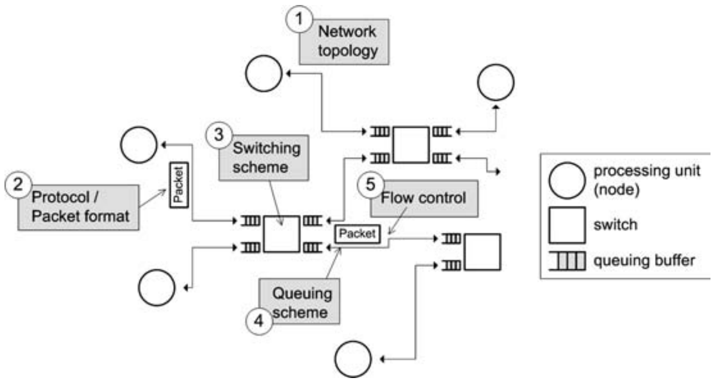


Figure 2.11 Basic design parameters of NoC

ports. When conflicts occur, the scheduler should resolve them according to a predefined algorithm. Buffers are important to store the packet data temporarily for congestion control.

There are three queuing schemes distinguished by the location of the buffers inside the router (switch): input queuing, output queuing, and virtual output queuing (Figure 2.12).

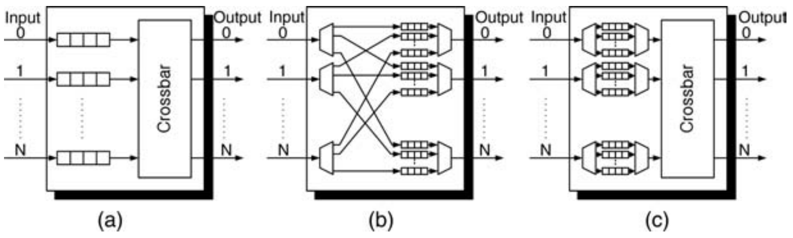


Figure 2.12 Queuing schemes: (a) input queuing, (b) output queuing, and (c) virtual output queuing

- **Input queuing.** Every incoming link has a single input queue so that N queues are necessary for an $N \times N$ Switch. Input queuing suffers from the "head-of-line" blocking problem; that is, the switch utilization is saturated at the 58.6% load.
- **Output queuing.** The queues are placed at the output port of the link, but N output queues for every outgoing link are required to resolve the output conflict, resulting in N^2 queues. Owing to the excessive number of buffers and its complex wiring, in spite of its optimal performance, output queuing is not used.
- **Virtual output queuing.** The advantages of input queuing and output queuing are combined. A separate input queue is placed at each input port for each output, requiring N^2 buffers. The "head-of-line" blocking problem is resolved by scheduling. Complicated scheduling algorithms such as iterative algorithms are needed.

The last issue in Figure 2.12 is flow control or congestion control. There are several solutions which prevent packets from output conflict and buffer overflow.

- **Packet discarding.** Once the buffer is overflowed, the packets coming next are simply dropped off.
- **Credit-based flow control.** A back-pressure scheme uses separate wires for the receiver, to notify congestion of the buffer to the transmitter to prevent packet loss, as shown in Figure 2.13a. The propagation delay time between transmitter and receiver should be considered carefully to avoid packet transmission while the wait signal is coming on the wire. In the *Window* scheme of Figure 2.13b, the receiver regularly informs the transmitter about the available buffer space.
- **Rate-based flow control.** The sender gradually adjusts the packet transmission according to the control flow messages from the receiver. For example, the error control uses Go-back-*N* algorithm and related signals. In this case, a certain amount of buffer space is guaranteed for effective flow control, but owing to its long control loop, rate-based flow control potentially suffers from instability.

In addition to the basic design parameters required to be determined for an NoC design, additional functionalities are required to enhance NoC performance and optimize the cost overhead. Quality of service is one of the most critical issues. The quality factors in an NoC should be bandwidth and latency. Guaranteed bandwidth and limited latency enable packet transactions to be punctual, thus making it possible to execute real-time applications in the NoC-based SoC.

2.4.1.4 NoC Building Blocks

The circuits of the basic building blocks will be introduced. The basic building blocks are high-speed signaling circuits, queuing buffers and memories, switches, crossbar switch schedulers, and SERDES.

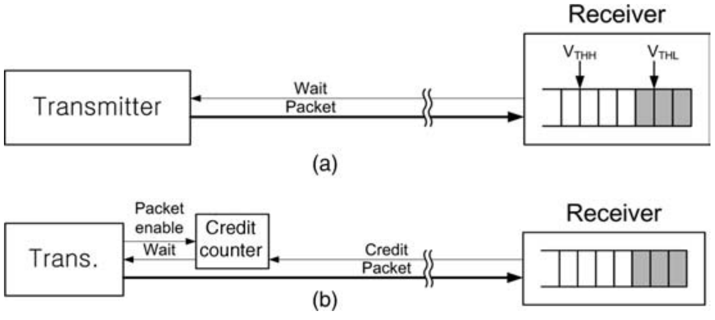


Figure 2.13 Flow control schemes: (a) back pressure, and (b) credit-based flow control

High-speed Signaling

For high-speed signaling, bit width, operating frequency, and differential signaling need to be carefully explored. If we use a wide bit width, interference among channel wires hinders high-speed operation of the channels. Differential signaling can achieve a high speed of operation with a relatively high signal to noise ratio (SNR), but it uses up twice the area of single wire signaling, and for a wide bit width it is impractical. Figure 2.14a shows voltage mode signaling and it usually uses repeaters along the wire to reduce the capacitive load of the driver circuits. Figure 2.14b shows current mode signaling circuits and it is known that current mode signaling is faster than voltage mode signaling.

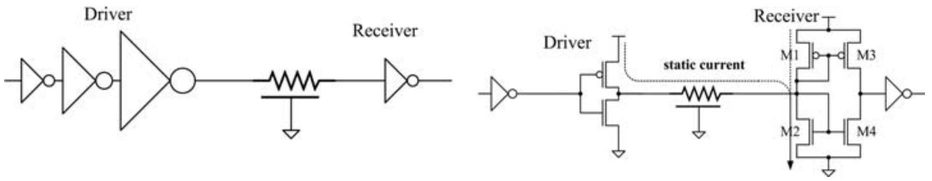


Figure 2.14 Interconnection drivers: (a) voltage mode, and (b) current mode

Queuing Buffer Design

The effective bandwidth of the data-link layer is heavily influenced by the traffic pattern and queue size. A queue buffer is located at the input or output port of the switch, and in the network interface that interconnects logic blocks to the sender or receiver. Queuing buffers consume the most area and power among building blocks in the NoC. There are two ways to implement a buffer: flip–flop-based (register) and SRAM-based. Figure 2.15 shows four different register designs: a conventional shift register, a push-in shift-out register, a push-in bus-out register, and a push-in MUX-out register. A SRAM-based design is also shown in Figure 2.9.

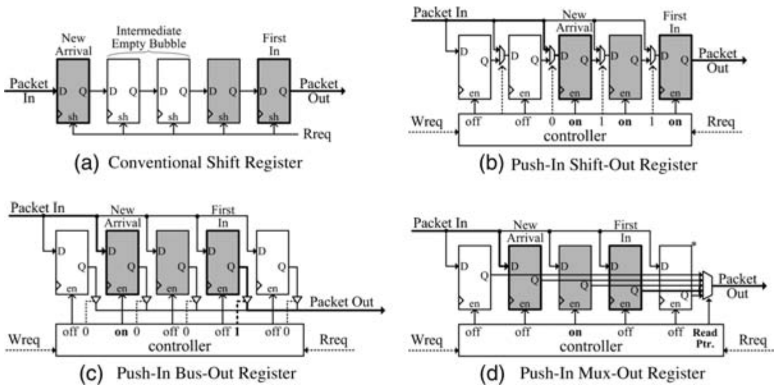


Figure 2.15 DFF-based queues

In a conventional shift-register type, bubble cells may occur when the packet input/ output rates are different. Shifting all the registers at every packet-out consumes a huge amount of power. Furthermore the minimum latency in a queue is as long as the physical queue length rather than the backlog. Although this design is the simplest, it is not suitable to implement on a chip.

To remove the intermediate empty bubble, the arrival packet can be stored at the front empty cell rather than at the tail of a queue. This input style is called as “push-in.” It can remove unnecessary latency and power consumption caused by the empty bubble. Only the occupied register cells are enabled. However, the shifting register style still consumes unnecessary power by shifting all the occupied cells when a packet is output. To avoid the shifting operation, the outputs of all registers are tied to a shared output bus line via tri-state buffers, as shown if Figure 2.15c. The register holding the first-in packet is connected to the output bus and turns on the tri-state buffers. In this design, only a cell in which a newly arrived packet is stored is enabled. As the queuing capacity increases, the capacitance of the shared bus wire increases as well because of the parasitic capacitance of tri-state buffers, and the delay and power consumption become considerable. To eliminate this effect, output multiplexers can be used as shown in Figure 2.15d.

The areas and power consumption of register-based buffers are relatively large. As the queuing capacity increases to dozens of packets, the register-based implementation is not good in both respects of area and power. The dual-port SRAM cell is used for large-capacity queuing as shown in Figure 2.16. This figure shows the circuit and layout of a unit cell and, for comparison, the layout of the D-FF. A SRAM cell occupies only a fifth of a register (flip–flop) area.

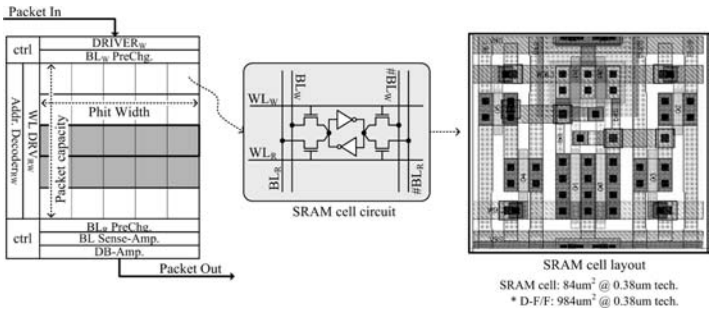


Figure 2.16 SRAM-based queue

Switch Design

The conventional switch consists of input queue (IQ), scheduler, switch fabric, and output queue (OQ), as shown in Figure 2.17a. There are two types of switch fabric

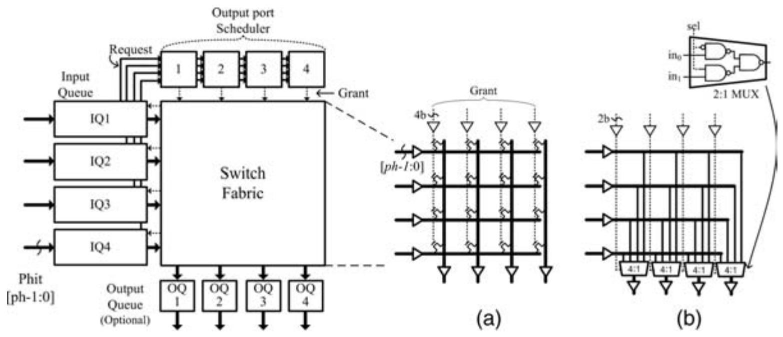


Figure 2.17 (a) Cross-point, and (b) MUX-based switch fabrics

design: cross-point and MUX-based. The cross-point switch has pass transistors at each crossing junction of the input and output wires. In this switch fabric, the capacitive loading due to the input driver is the junction capacitance of pass transistors on input and output wires and the wire capacitance itself. These parasitic capacitances and the series resistance of the wire cause RC delay, so limiting the bandwidth of the switch. The voltage swing on the output wire is reduced to $V_{DD} - V_{th\_N}$ because of the threshold voltage drop of the NMOS pass-transistor, so the power dissipation is reduced. A CMOS pass gate can be used to avoid the voltage drop, but another control wire is required which increases the area consumption. The fabric area is determined by the wiring area and not by the transistors, so that its area cost can be minimized. The MUX-based switch uses a multiplexer for each output port. The capacitive loading driven by the input driver is the input gate capacitance of the multiplexers and input wire capacitance. The parasitic capacitance is larger than with cross-point switch, but the high resistance of the pass transistor does not exist. It can operate at relatively high speeds and is synthesized by EDA tools.

Figure 2.18 shows other circuit implementations of the 256×256 crossbar switch [25]. Part (a) is a circuit based on a tree of NAND inverters (AND) and local decoders, and (b) shows a pre-programmed crossbar.

To arbitrate the output conflicts, a scheduler is used on each output port. The arbitration scheduling is required to perform fair routing, no-starvation, and maximum throughput in high-level, but it does not significantly increase the latency, power, and area of the switch design. The latency of the arbiter becomes larger than that of the switch fabric as the switch size exceeds 16×16. As shown in Figure 2.19, the scheduler occupies similar area to the switch fabric when the phit (*ph*ysical dig*it*) width of a port is 10 bits. Therefore, scheduler design is as important as switch fabric design.

Scheduler Design

Many scheduling algorithms are known, examples being round-robin, propagation basis, and maximum weight matching algorithms. However, diagonal/rectilinear

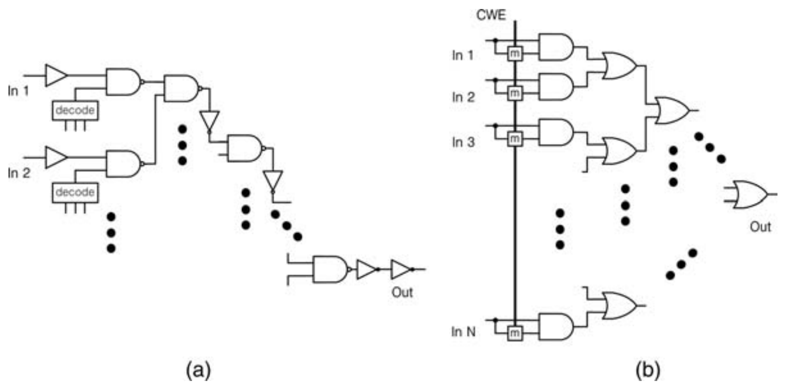


Figure 2.18 Circuits for crossbar switch: (a) tree of NAND invertors, and (b) pre-programmed

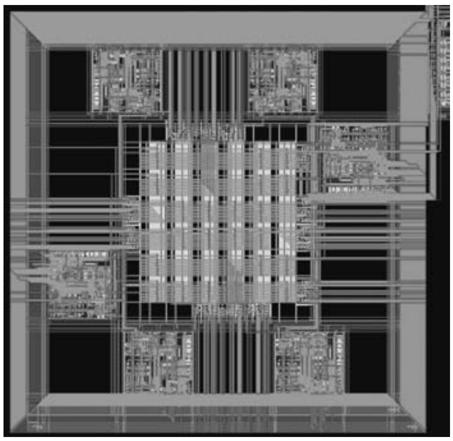


Figure 2.19 Switch layout diagram of 6 × 6 and phit width of 10 bits/link

propagation methods or weighting matching such as LQF and OQF are too complicated to lay out on a chip with small area. The round-robin scheduling algorithm is most widely used in the on-chip network because of its fairness and no-starvation properties.

In round-robin scheduling, each port in turn is given the *opportunity* to transmit data for a fixed amount of data or time. The port may not take the opportunity, and the opportunity is moved to the next port. Figure 2.20a shows an example of round-robin scheduling for a six-port system when 1, 3, 4, and 6 ports are ready to transmit a packet. The round-robin scheduler can be implemented by using two priority encoders (PEs), as shown in Figure 2.20b, or by MUX-tree-connected logic.

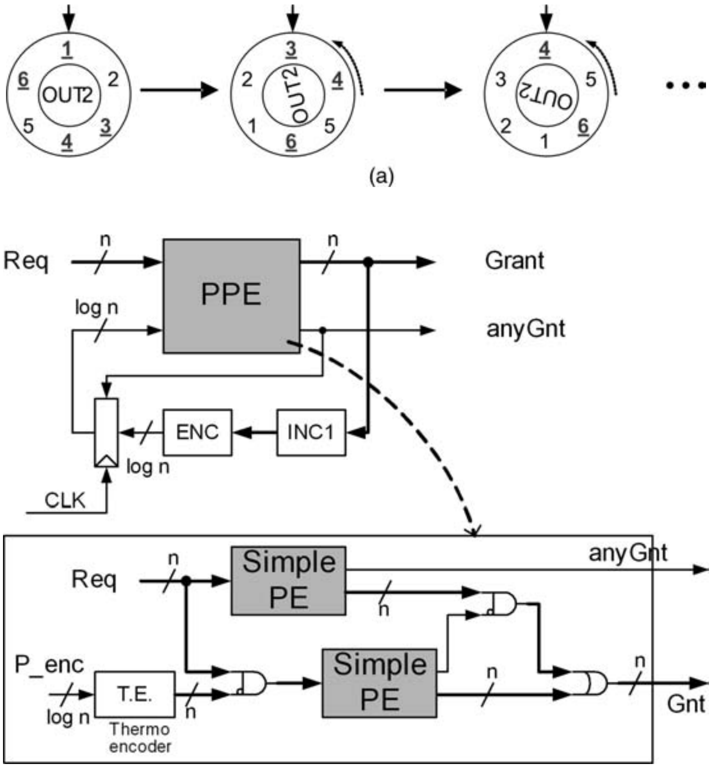


Figure 2.20 (a) Round-robin algorithm, and (b) round-robin algorithm circuits with two priority encoders

SERDES Design

As stated above, an on-chip serialization technique reduces the area of the NoC significantly, so a serializer and de-serializer (SERDES) circuit is an essential building block for practical NoC designs. There are two typical SERDES circuits: shift register and MUX-tree as shown in Figure 2.21. The shift register (SR) serializer fetches the parallel packet through 2:1 MUXs when the load signal is enabled. Then, the shift mechanism of the series F/Fs realizes high-speed serialization. The MUX-type serializer divides a packet into several parts, and then multiplexes them into the serialized link. In both the serializer types the maximum clock frequency is limited by the delay time of the D-FF, and a high-speed clock is required for the serialization speed. To overcome these limitations, a new serializer structure has been described that uses delay elements (DEs) as a timing reference instead of a clock, and uses signal propagation phenomena instead of the shifting mechanism [26]. The serialization scheme is called *wavefront train* or WAFT.

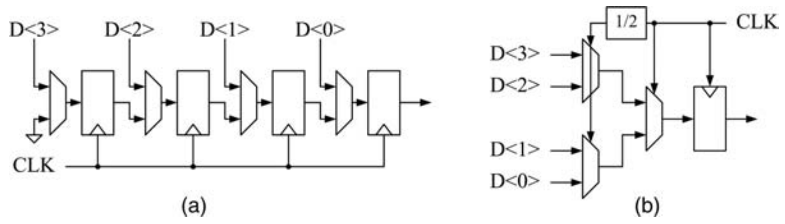


Figure 2.21 (a) Shift-register type, and (b) MUX-tree type serializers

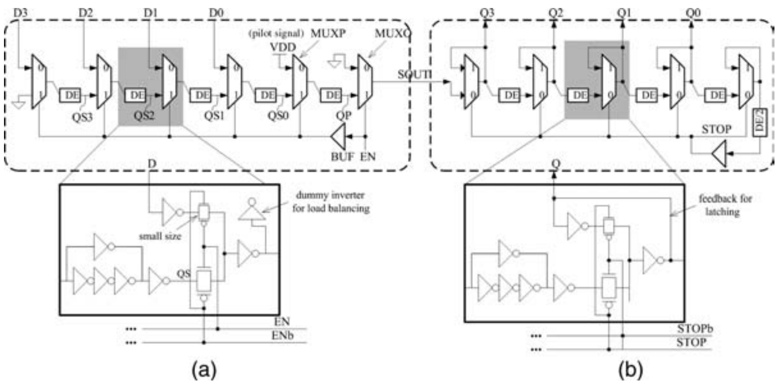


Figure 2.22 WAFT (a) serializer and (b) deserializer

Figure 2.22 shows a 4:1 WAFT serializer and deserializer schematic. When EN is low, D⟨3:0⟩ is waiting at QS⟨3:0⟩. The VDD input of MUXP, which is called a pilot signal, is also loaded to QP. The GND input of MUXO discharges the serial output (SOUT) while the serializer is disabled. If EN is asserted, QS⟨3:0⟩ and the pilot signal start to propagate through the serial link wire. Each signal forms a wavefront of the SOUT signal, and the timing distance between the wavefronts is the DE and MUX delay which we call a "unit delay." The series of wavefronts propagates to the deserializer like a train (hence the name).

When the SOUT signal arrives at the deserializer, it propagates through the deserializer until the pilot signal arrives at the end of the deserializer, or STOP node. As long as the unit delay times of the sender and the receiver are the same, D⟨3:0⟩ arrives at its exact position when the pilot signal arrives at the STOP node. When the STOP signal is asserted, the MUX feeds back its output to its input, so that the output value is latched.

2.4.2 NoC Design Considerations

NoC design strategies can be divided into two main categories: one is to determine or optimize fundamental network parameters, and the other is to propose new additional features for enhancing performance and/or reducing cost. These NoC designs are based on a top-down approach that tries to shrink the legacy network to fit it into a given silicon area. Thus, such approaches have a tendency to follow communication and network features adopted in legacy networks, but some features seem to be inadequate or unnecessarily complicated for on-chip situations.

Although the NoC borrows basic concepts from legacy network architectures, implementing the network architecture on a chip needs to consider many physical design issues. However, most work on NoC design overlooks actual implementation issues because it has not yet reached chip implementations, ignoring the on-chip characteristics. Architectural specifications like topology, routing, switching, and link bit width are determined without consideration of implementation constraints like low power consumption and small area. More physical issues like on-chip serialization and mesochronous communication also must be considered in order to implement NoC on silicon. Here, some architectural decision and physical implementation issues for the practical design of NoCs will be described from a chip designer's viewpoint.

2.4.2.1 Topology Selection

Traditionally, a mesh topology as shown in Figure 2.23a is widely used and studied for parallel computing architectures owing to its highly scalable and regular features. Most NoC implementations [27, 28] have used a mesh topology or a derivative, and wide link bit width over 128-bit. However, the mesh topology needs to be reassessed for practical NoC design in terms of area and power consumption in an SoC environment. Although a wide link has been used in an on-chip situation, the large number of metal wires complicates not only metal routing but also placement of processing elements (PEs). In an actual layout, PEs may be placed irregularly to minimize the chip area, so the regular structure of mesh topology may not be possible

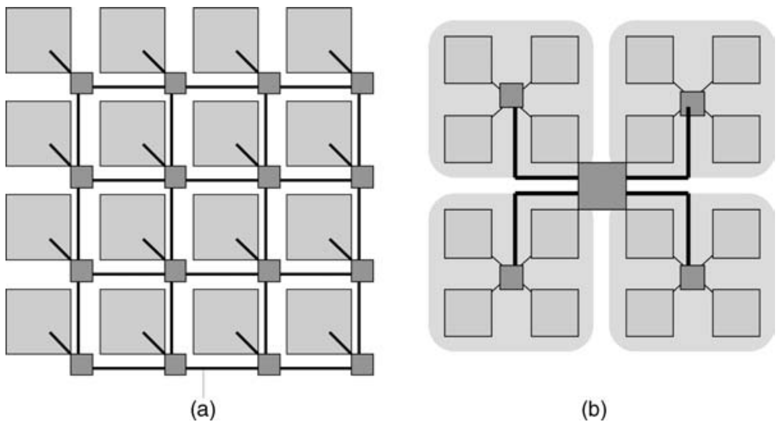


Figure 2.23 Topologies: (a) mesh, and (b) hierarchical star

in an SoC design. Furthermore, a wide link means wide switch fabrics, which increases the network area significantly. Moreover, the mesh topology results in inefficient global packet transactions because of its large hop counts. The alternative star topology has not been popular owing to its poor scalability, in spite of its higher bisection bandwidth and small hop count. However, in an on-chip situation the number of integrated PEs (*N*) is limited to a few tens. Even when *N* is larger than 100, the PEs can be interconnected hierarchically, as shown in Figure 2.23b. PEs communicating with each other intensively will be grouped as a cluster, and a local network will interconnect the PEs. The clusters will be interconnected by a global network. Then, interconnection in a cluster (as well as interconnecting the clusters) becomes the same issue as interconnecting a few tens of PEs. Therefore, the hierarchical star topology is a better candidate than the mesh topology for practical NoC design, because it shows better cost efficiency and the lowest latency [29].

2.4.2.2 Routing Scheme

An adaptive routing method is widely used for general macro-networks because it enables fault-tolerant packet transfer and hot-spot avoidance by using alternative packet routing paths, which leads to higher throughput. However, the packets may arrive out of order in adaptive routing, so huge scratch buffering resources are needed to store incoming packets temporarily, and packet re-ordering based on a packet sequence number is required (Figure 2.24). Because of such a huge area cost and unpredictable latency, adaptive routing is not suitable for NoC implementation considering the hardware overhead in relation to performance improvement.

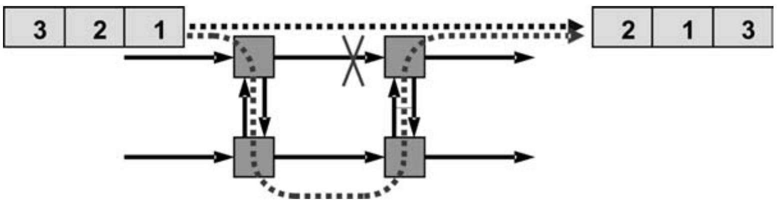


Figure 2.24 Adaptive routing method

On the other hand, *deterministic routing* such as source routing may be a good solution. In deterministic routing, a packet is transferred to a destination through a fixed routing path that is defined at a network interface. A switch does not support the adaptive routing function, and network interfaces need no packet re-ordering function. As a result, the NoC hardware implementation cost is very low compared to adaptive routing. There is still a potential problem regarding hot-spots with deterministic routing. However, even though bandwidth is limited by hot-spots, we can say that it is the maximum affordable bandwidth for the network. Moreover, in an SoC environment, the designer has a good knowledge of the traffic characteristics for specific applications and can avoid the hot-spot problem by allocating the routing paths wisely.

2.4.2.3 Switching Scheme

A packet-switching method like wormhole-based routing is widely used for NoC because of the limited buffering resources. A packet-switched network uses channel resources more efficiently than a circuit-switched network because the route from a source to a destination is pipelined. This advantage increases as the route becomes longer. However, the route length in current SoCs is insignificant, so this alone does little for a packet-switched network. Clearly, if a NoC uses a high clock frequency and small packets, the packet-switched network works well as a global interconnect architecture. However, if the NoC has a lower clock frequency for low power consumption or uses burst packet transfer, repetitive processes such as synchronization, packet queuing, and arbitration in all the intermediate switches are redundant and inefficient, which leads to large latency. To solve this problem of the packet-switching NoC, we recommend the use of an adaptive switching method in which circuit switching techniques are combined with packet switching as shown in Figure 2.25. The NoC has level 1 and level 2 switches. A level 1 switch supports both switching modes; a level 2 switch supports only packet switching. The first packet of a burst packet flow enables the level 1 switches' circuit-switching mode, so that the remaining packets bypass the level 1 switches. This mechanism effectively reduces the number of switches along the end-to-end route, and the level 2 switches still provide the advantages of a packet-switched network. When a level 1 switch is in the circuit-switching mode, a synchronizer, a FIFO buffer, and the input port's packet-parsing logic are bypassed, and the packets are routed to the prescheduled output port. This effectively reduces delay and energy consumption for a packet transfer.

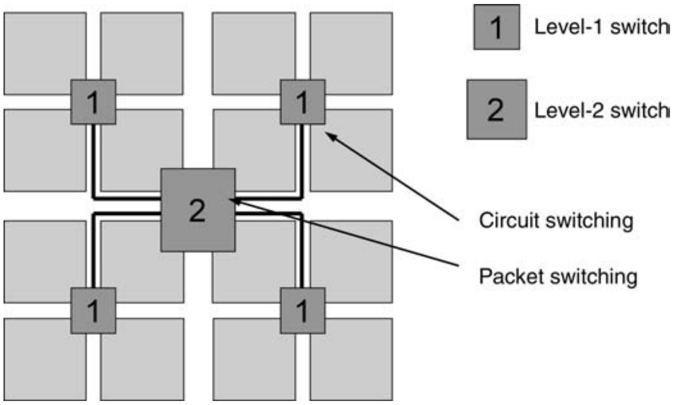


Figure 2.25 Adaptive switching method

2.4.2.4 Phit Size Determination

The physical transfer unit or "physical digit" (phit) is a unit in which a packet is divided and transmitted through the core network. The phit size is the bit-width of a link and determines the switch area. Therefore, for practical NoC design, the phit size should be carefully determined considering both the NoC cost and performance. An on-chip serialization technique can be effectively used to reduce the area and power of the NoC. The phit size determination is largely related to the on-chip serialization. If the phit size is smaller than the packet length, serialization must be performed by the factor of

$$\text{serialization ratio(SERR)} = \text{packet size/phit size.}$$

Using a large phit size, as shown in Figure 2.26a, obtains wide bandwidth easily but is inefficient in respect of the NoC cost – that is, area and power consumption. On the other hand, use of a small phit size, as shown in Figure 2.26b, reduces the number of link wires, so the spaces between link wires can be widened to decrease coupling capacitance. Also, the smaller phit reduces the switch fabric size, so again the area and power consumption of a switch can be reduced [30].

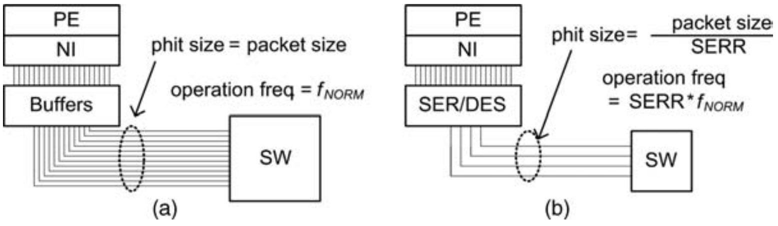

PE
NI
Buffers
phit size = packet size
operation freq = f_{NORM}
SW
(a)
PE
NI
SER/DES
phit size= $\frac{\text{packet size}}{\text{SERR}}$
operation freq = SERR*f_{NORM}
SW
(b)

Figure 2.26 NoC structure according to phit size

In more detail, switch power consumption is the sum of arbiter and switch fabric power, and the switch fabric power is the dominant part. When serialization is used, the operating frequency of the core network must be increased by the factor of the SERR in order to maintain network bandwidth. Considering additional circuitry for the frequency increment, energy consumption and area of NoC building blocks such as serialization and de-serialization (SERDES), link, synchronizer, and switch are analyzed according to the SERR. The operating frequency without serialization is set to be 200 MHz, and detailed designs of the blocks are based on the implementation results. Figure 2.27 shows analytical results of energy consumption per packet transmission and area of building blocks in the star topology NoC. In part (a), the energy consumptions in a switch and links decrease as the SERR increases as mentioned above. On the contrary, those of a SER, DES, and SYNC increase due to additional circuitry for the serialization. In most cases, an SERR of 4 minimizes the overall power consumption. As shown in part (b), serialization also reduces the overall NoC area effectively, and the SERR of 4 is optimal. An SERR of 8 remains the same

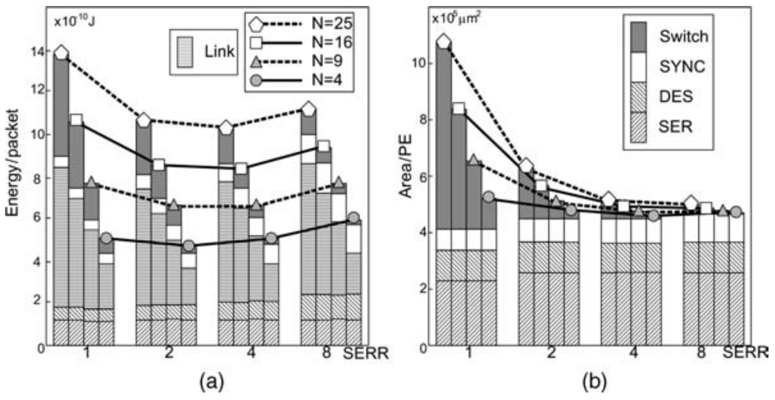


Figure 2.27 (a) Energy and (b) area of NoC according to SERR

as 4 in terms of area, but is inefficient in terms of power. In the implemented chip [31], the phit size is determined as a quarter packet length to cover design issues on a 4:1 serialized network.

2.4.2.5 Mesochronous Synchronizer

One of NoC's contributions to SoC design is to ease the burden of global synchronization by using mesochronous communication, meaning that network blocks share the same clock source but the clock phases of functional blocks may be different from each other owing to asymmetric clock tree design and difference in load capacitance of the leaf cell. Without a mechanism to compensate for the phase difference, nondeterministic operation, such as meta-stability, would impair stability or functionality of NoC-based systems. To resolve such a problem, synchronizers are required between the clock domains.

Several types of synchronizer have been used in NoC design, examples being first-in first-out (FIFO), delay-line, and simple pipeline synchronizers [31, 32]. The FIFO synchronizer is useful when the phase difference between clock domains is unknown. However, its power and latency overhead is considerable. A single-stage pipeline synchronizer provides the best performance and lowest overhead if intensive SPICE simulations under various conditions eliminate all possible synchronization failures. However, as the operating frequency and the number of network nodes increase, the full-custom solution is not practical.

To resolve these problems, a programmable delay synchronizer has been devised as shown in Figure 2.28. A variable delay (VD) is connected with a simple pipeline synchronizer, and the VD is controlled according to the network circumstances. The appropriate VD setting for a certain circumstance is obtained through a calibration process, performed as follows. In the network initialization period, the calibration master unit (CMU) sends packets to all the PEs in the network. When a synchronizer

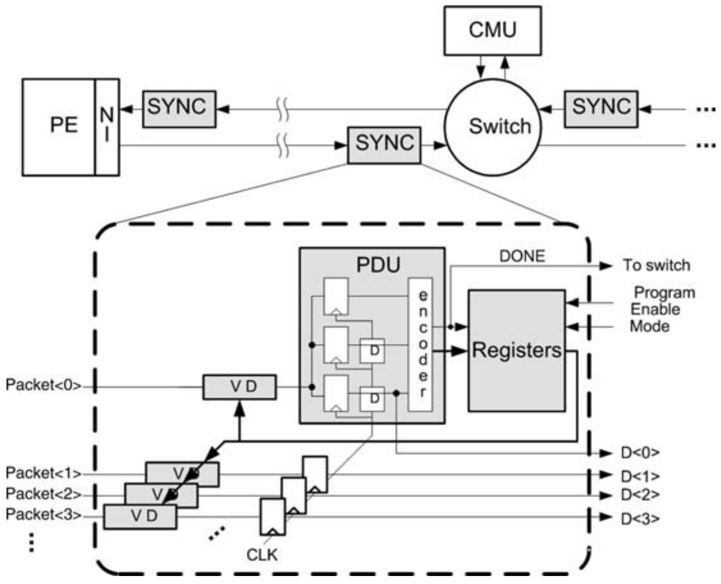


Figure 2.28 Programmable delay synchronizer

(SYNC) receives a packet, the phase detecting unit (PDU) in the SYNC finds the best timing to sample the input signal. Then, the timing information is encoded into the appropriate VD setting value, and it is programmed into a register and VDs. After the VD setting, the PDU will sample input signals correctly and thus asserts the done-signal. Then, the switch fetches the output of the SYNC, and performs packet switching. While the packets from the CMU are delivered to every PE, all the SYNCs in the CMU-to-PE paths are calibrated. The network interface is designed to return the packet when it receives a packet in the initialization period. Therefore, the SYNCs in the PE-to-CMU paths are also calibrated. This calibration process is repeated for all combinations of network circumstances like operating frequency and topology configuration.

Programmable delay synchronizer operation is measured. In Figure 2.29, the EN signal is fetched at the positive edge of the clock. After that, the EN is de-asserted by a handshaking protocol. One problem is that the EN signal is not synchronized with the clock so that the rising edge of the EN could be very close to the "rising edge minus setup time margin." In Figure 2.29a, the mode C represents a situation in which the rising edge of the EN signal is very close to the clock timing. When the mode C is enabled, fetching the EN signal is very unstable so that the EN would be fetched at the next clock cycle as shown in Figure 2.29b. Figure 2.30 shows the measured waveform when the programmable delay synchronizer is enabled. When the mode C is enabled, the delay time applied to the EN signal is reduced so that the fetching timing is effectively delayed compared to normal operation, which ensures sufficient timing

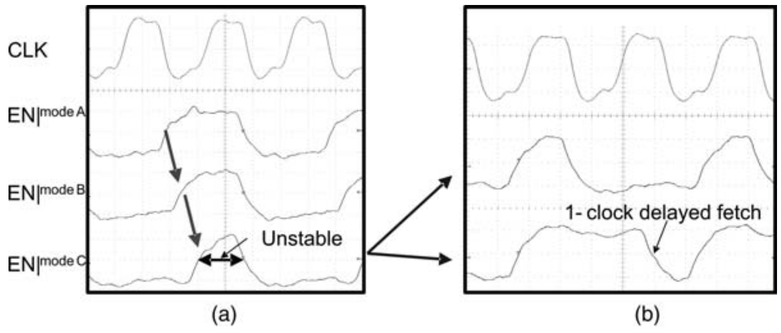


Figure 2.29 (a) EN signal variation according to the modes, and (b) unstable EN signal fetching

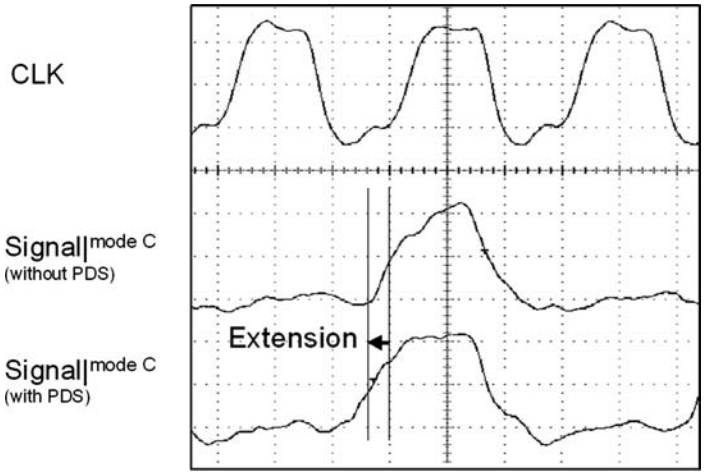


Figure 2.30 EN signal expansion using programmable delay synchronizer

margin to fetch the EN signal. As a result, the de-assertion of the EN signal is delayed as shown in Figure 5.21.

2.4.3 Case Studies of Chip Implementation

In this subsection we shall introduce silicon chip implementation trials for NoC-based SoCs. They can be grouped into two categories: academic research and industrial approaches. The academic research shows complete chip implementations and demonstrations for specific applications. The industrial approaches are mainly about the new protocol specifications, EDA tool chain, and IP library support for the NoC developers.

2.4.3.1 Intel Teraflop 80-Core NoC

The Intel Corporation launched a Tera-Scale Computing Research Program a few years ago to handle tomorrow's advanced applications that will need a thousand times more computing capability than is available in today's giga-scale devices. For example there is real-time data mining across teraflops of data, artificial intelligence (AI) for smarter cars and appliances, and virtual reality (VR) for modeling, visualization, physics simulation, and medical training [33].

The Intel tera-scale research consists of three categories: teraflop of performance, terabytes per second of memory bandwidth, and terabits per second of I/O performance [33]. Here we shall focus on the "teraflop of performance" research where NoC is developed and implemented (Figure 2.31). Eighty processing cores are interconnected through a 2D mesh, packet-switched, on-chip network. It performs up to 1-teraflops at 4 GHz clock speed and consumes less than 100 W. [11]

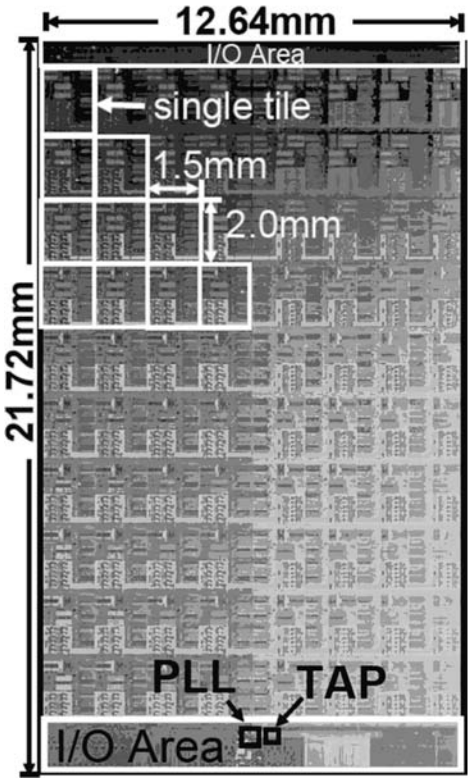


Figure 2.31 Intel's teraflop 80-core NoC chip. The Intel logo is a registered trademark of Intel Corporation

Key Enablers for Teraflop on a Chip [11]

- 80 processing engines and 160 single-precision floating-point units
- Designed for 4 GHz operation
- Fast single-cycle accumulate loop
- Sustained FPU throughput: 2 flops per cycle
- 80 GB/s router, operating at 4 GHz
- Shared and double-pumped crossbar switch
- 2D mesh topology, 256 GB/s bisection bandwidth
- A 15FO4 balanced core and router pipeline
- Robust, scalable mesochronous clock distribution
- 65 nm eight-metal CMOS.

Architecture Overview [11]

The NoC architecture contains 80 tiles arranged as a 10 × 8 two-dimensional mesh network operating at 4 GHz (Figure 2.32). Each tile consists of a processing engine connected to a 5-port router with mesochronous interfaces, which forwards packets between tiles. The 80-tile on-chip network enables a bisection bandwidth of 256 GB/s. The PE contains two independent fully-pipelined single-precision floating-point multiply–accumulator (FPMAC) units, 3 KB of single-cycle instruction memory (IMEM), and 2 KB of data memory (DMEM). A 96-bit VLIW (very long instruction word) encodes up to eight operations per cycle. With a 10-port (6-read, 4-write)

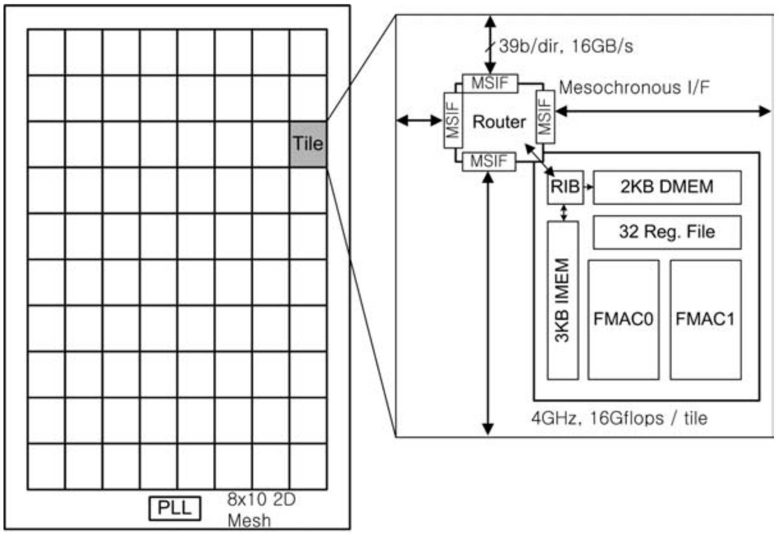


Figure 2.32 Chip architecture

register file, the architecture allows scheduling to both FPMACs, simultaneous DMEM load and stores, packet send/receive from the mesh network, program control, and dynamic sleep instructions. A router interface block (RIB) handles packet encapsulation between the PE and router. The fully symmetric architecture allows any processing engine to send (receive) instruction and data packets to (from) any other tile.

The 4 GHz 5-port wormhole-switched router uses two logical lanes – virtual channels – for dead-lock free routing, and a fully nonblocking crossbar switch with a total bandwidth of 80 GB/s. Each lane has a 16-flit (*fl*ow control un*it*) queue, arbiter, and flow control logic. The router uses a 5-stage pipeline with a two-stage round-robin arbitration scheme that first binds an input port to an output port in each lane and then selects a pending flit from one of the two lanes.

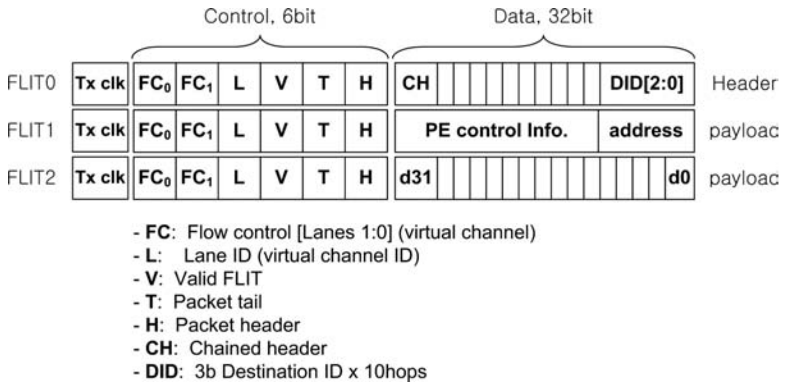


Figure 2.33 Packet format and protocols

Figure 2.33 shows a NoC packet format. Each packet is subdivided into multiple flits. It has a minimum of two flits and there is no maximum packet size limit. Each flit consists of a 6-bit control field and 32-bit data field. The control field includes two flow control bits for each lane, a valid indication bit for the flit, and packet header/tail indication bits. There are three types of flit: header, PE control, and data flits. A header flit has a 3-bit destination ID (DID) which represents the out-port direction on each switching hop. Because of the data field size limit, the maximum hop count is limited to 10. However, a chained header seems to support larger hop counts to break this limitation. The PE control flit includes an address file and PE control information like PE power management signals [11].

Double-pumped Crossbar Router and Mesochronous Interface

Each 36-bit crossbar data bus is double-pumped at the fourth pipe stage by interleaving alternate data bits using dual edge-triggered flip–flops, reducing crossbar area by 50% [34]. The double-pumped technique is the same as the 2:1 serialization within a

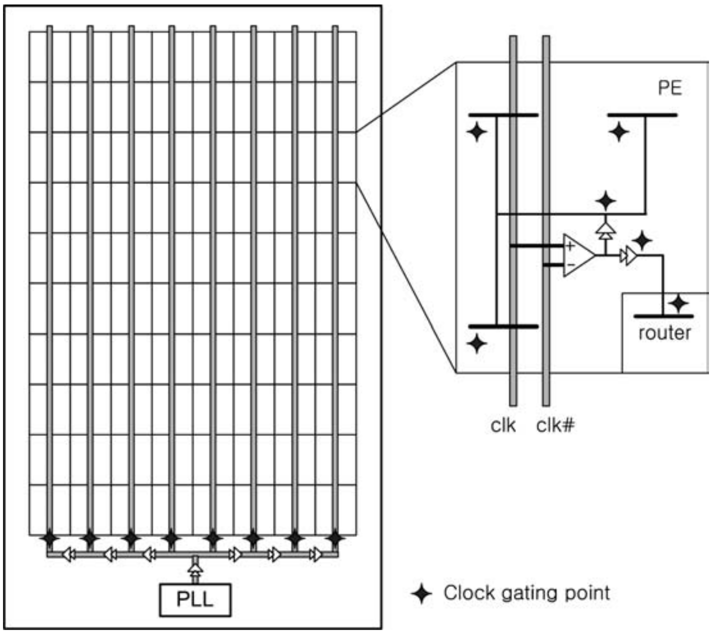


Figure 2.34 Clock distribution

switch fabric line, while it maintains the clock frequency by using double-edges like a double-data-rate SRDAM (DDR-SDRAM) interface.

The chip uses scalable global mesochronous clocking, which allows for clock-phase-insensitive communication across tiles and synchronous operation within each tile – *globally mesochronous and locally synchronous* (GMLS) – see Figure 2.34. The on-chip PLL output is routed using horizontal M8 and vertical M7 spines. Each spine consists of differential clocks for low duty-cycle variation along the worst-case clock route of 26 mm. An operational amplifier at each tile converts the differential clocks to a single-ended clock with 50% duty cycle. The worst-case simulated global duty-cycle variation is 3 ps, and local clock skew within the tile is 4 ps. The systematic clock skews inherent in the distribution help spread clock power owing to simultaneous clock switching over the entire cycle. The estimated global clock distribution power at 4 GHz, 1.2 V supply is 2.2 W.

Fine-grained Power Management

Fine-grained clock gating, sleep transistors, and body bias circuits [35] are used to reduce active and standby leakage power, and are controlled at full-chip, tile-slice, and individual tile levels based on workload. Each tile is partitioned into 21 smaller sleep regions with dynamic control of individual blocks in PE and router units, based on instruction type. The router is enabled on a per-port basis, depending on network

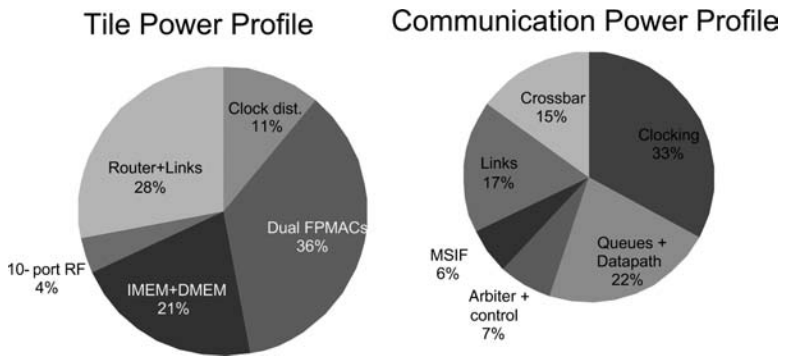


Figure 2.35 Power breakdown of a tile and NoC building blocks

traffic patterns. The design uses NMOS sleep transistors to reduce frequency penalty and area overhead. Memory arrays use an active clamped sleep transistor [36] that ensures data retention and minimizes standby leakage power. The average sleep transistor area overhead is 5.4% with a 4% frequency penalty. About 90% of FPMAC logic and 74% of each PE is sleep-enabled. Forward body bias can be applied to NMOS devices during active mode to increase the operating frequency, and reverse body bias can be applied during idle mode for further leakage savings.

Figure 2.35 shows the power breakdown of a tile and NoC building blocks. Clocking and buffering are the major power consumers. Table 2.3 shows the power and performance summary of the teraflop 80-core NoC.

Table 2.3 Power and performance summary

Frequency	Voltage	Power	Aggregated BW	Performance
3.16 GHz	0.95 V	62 W	1.62 Tb/s	1.01 Tflops
5.1 GHz	1.2 V	175 W	2.61 Tb/s	1.63 Tflops
5.7 GHz	1.35 V	256 W	2.92 Tb/s	1.81 Tflops

2.4.3.2 KAIST BONE-V (Basic On-chip Network for Vision Applications)

Introduction

This subsection describes a memory-centric NoC (MC-NoC) that facilitates flexible and traffic-insensitive mapping of tasks on a homogeneous multi-processor SoC (MP-SoC) [37]. The MC-NoC features a hierarchical star topology network and memory management scheme which supports unidirectional inter-processor communication. The MC-NoC incorporates distributed and fine-grained shared memory for simultaneous data transactions among processing elements, while the hierarchical star topology network is adopted to provide processing elements with area-efficient external memory interconnections. The MC-NoC improves feasibility and flexibility in mapping series of tasks into the homogeneous MP-SoC.

Architecture and Operation

Figure 2.36 shows the architecture and operation of the MC-NoC. It incorporates 10 RISC processors. The building blocks of the MC-NoC are dual-port SRAMs, crossbar switches, network interfaces, and channel controllers. Dual-port SRAMs are dynamically assigned to the subset of the RISC processors involved in data communication. Then, shared data is exchanged by accessing assigned dual-port SRAM. Crossbar switches provide nonblocking concurrent interconnections between dual-port SRAMs and RISC processors. The operating frequency of the crossbar switches is decided to be twice that of the other part of the MC-NoC to reduce the overhead of packet switching latency. The network interface performs packet processing and clock synchronization between a crossbar switch and other building blocks. The key building block of the MC-NoC is the channel controller. This automatically manages communication channels between RISC processors to facilitate mapping of tasks on the homogeneous SoC. The role of the channel controller is described in more below.

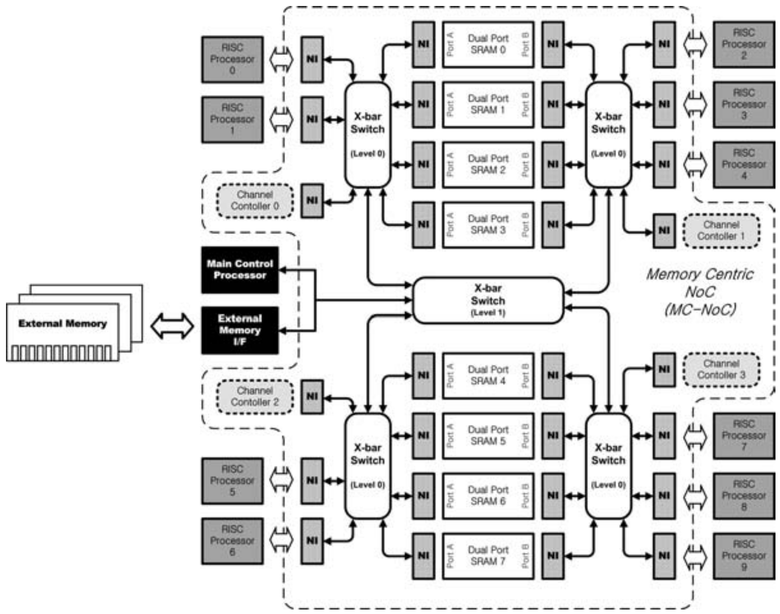


Figure 2.36 Overall architecture of the MC-NoC

Figure 2.37 outlines the important steps of MC-NoC operation. In this figure, crossbar switches are not drawn for simplicity. While the operation is explained, we will assume that RISC processor 0 wants to pass the processed results into RISC

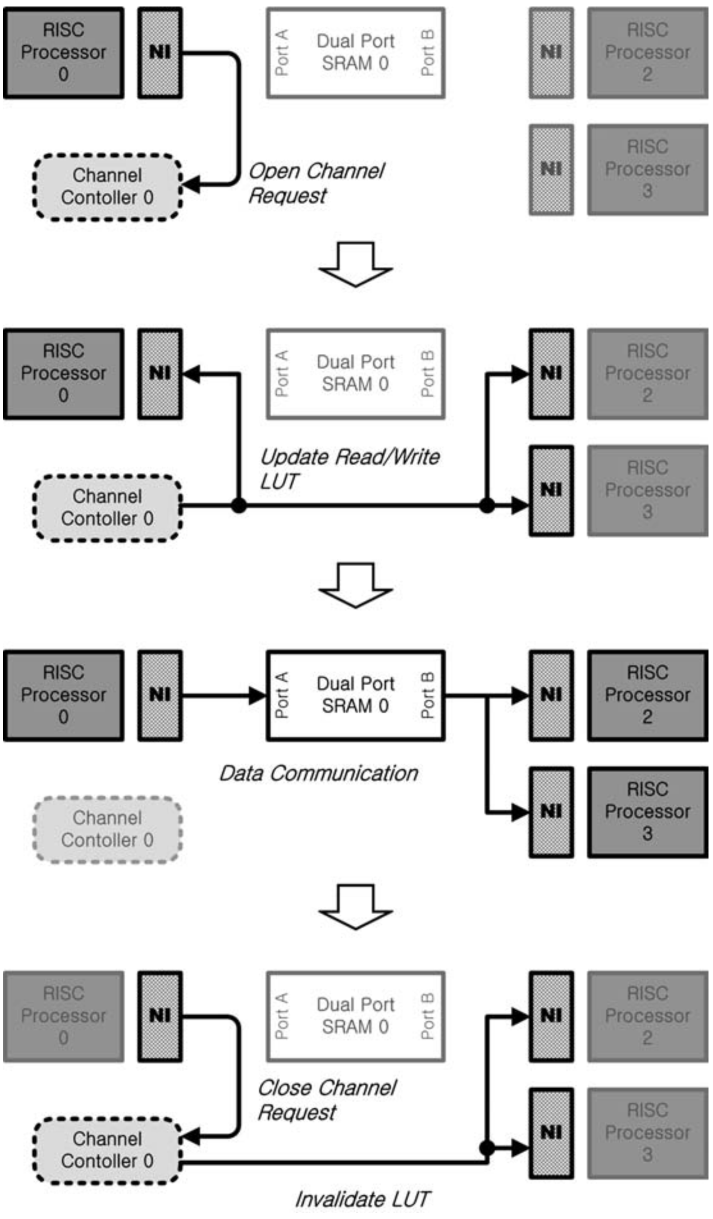


Figure 2.37 Important steps of the MC-NoC operation

processors 2 and 3. The MC-NoC operation is initiated by RISC processor 0 sending an open-channel request to the channel controller. Information about source and destination RISC processors is also included in the open-channel request. After that, the channel controller assigns one dual-port SRAM as a data communication channel if any of the SRAMs is available. By updating the routing look-up tables (LUTs) in the network interfaces of corresponding processors, SRAM assignment is completed. In this way, assigned SRAM is made to be accessible only for the RISC processors involved in data communication. At the end of data transfer through the dual-port SRAM, source RISC processors send a close-channel request to the channel controller. Then the channel controller invalidates updated LUTs after checking completion of data transfer. In the proposed MC-NoC, each processor is able to send multiple open-channel requests as required. If all the SRAMs are being used by other processors, the data transfer is stalled until one of the SRAMs becomes available. Open- and close-channel requests and LUT updates are performed by sending special packets that are not visible to any processors or memories. Controlling operations of the MC-NoC by using special packets has the advantage of eliminating additional control signal wires.

While data communication is performed through the dual-port SRAM that is assigned by the channel controller, progress of data access from destination processors may differ from each other. To improve programming feasibility of the multiple RISC processors, the MC-NoC provides a data synchronization scheme to resolve the consistency problem arising from the different data access order of destination processors. Figure 2.38 shows an example. Processor 2 reads data from address 0×0, while processor 3 accesses address $0 \times C$. Until this moment, processor 0 has written valid data only at the address 0×0. The next step is shown in Figure 2.38b. In this case, only processor 2 gets valid data from the dual-port SRAM, and processor 3 receives an invalidate signal from valid check logic inside the dual-port SRAM. After that, the network interface of processor 3 holds the processor and retries reading after specified wait cycles, as shown in Figure 2.38c. Once processor 0 writes valid data at address $0 \times C$, processor 3 also gets valid data and continues processing, as in Figure 2.38d. The retry procedure described in Figure 2.38c is transparent to the RISC processors because the NI module of the MC-NoC automatically manages the procedure. As with an open- or close-channel request, an invalid signal from valid check logic of the dual-port SRAM is also transferred as a special packet. In our implementation, the valid check logic takes 5% of the dual-port SRAM area.

Benefits of the MC-NoC

The main advantage of the MC-NoC is its flexibility of task mapping on a homogeneous SoC. Here we discuss the benefits of the MC-NoC by comparison with a conventional 2D mesh topology NoC. As a task mapping example, edge detection operation is shown in Figure 2.39a [38]. In the figure, rectangular boxes represent processors performing tasks and solid/dotted lines depict data flow between tasks. In this operation, an input image is first converted from RGB color space to HSI color space. The converted image is processed by Gaussian filters with varying coefficients (sigma) and subtractions between filtered results are calculated to detect edges in different scale. Figures 2.39b and c show mapping of the edge detection operation on a

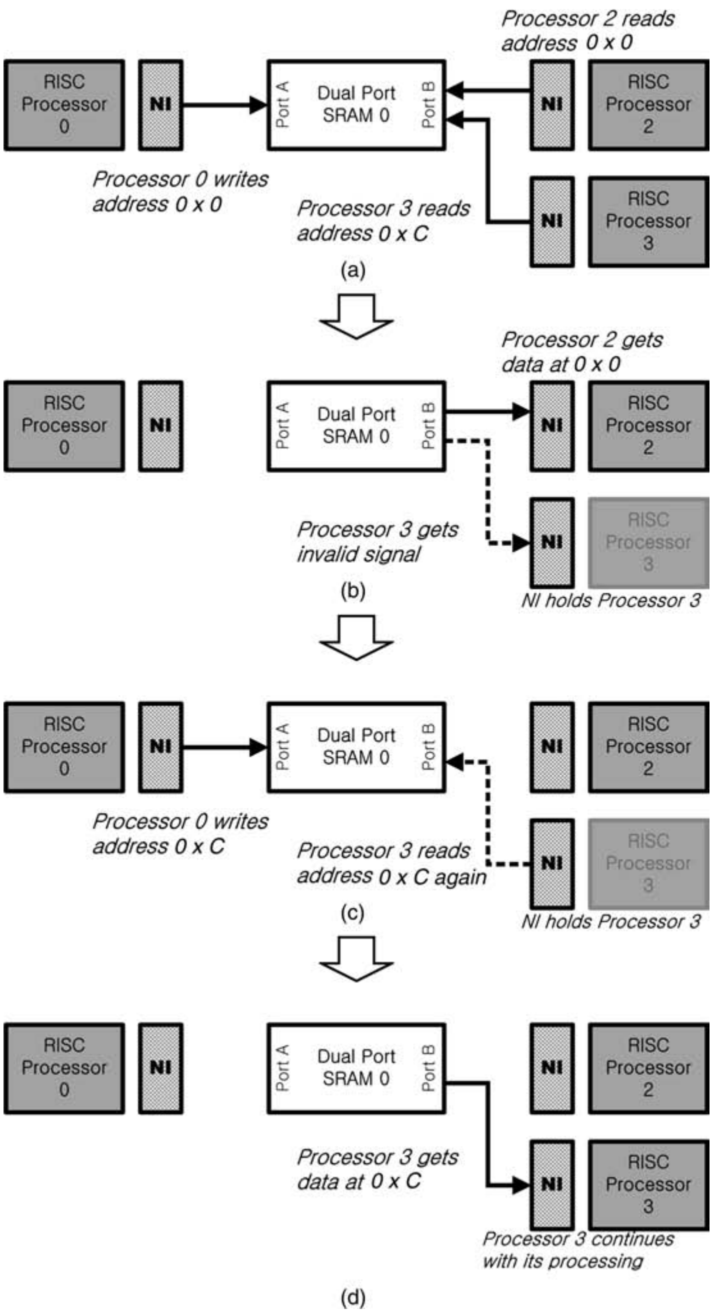

Processor 2 reads address 0 x 0
RISC Processor 0
NI
Port A
Dual Port SRAM 0
Port B
NI
RISC Processor 2
Processor 0 writes address 0 x 0
Processor 3 reads address 0 x C
NI
RISC Processor 3
(a)
Processor 2 gets data at 0 x 0
Processor 3 gets invalid signal
NI holds Processor 3
(b)
Processor 0 writes address 0 x C
Processor 3 reads address 0 x C again
NI holds Processor 3
(c)
Processor 3 gets data at 0 x C
Processor 3 continues with its processing
(d)

Figure 2.38 Data synchronization scheme of the MC-NoC

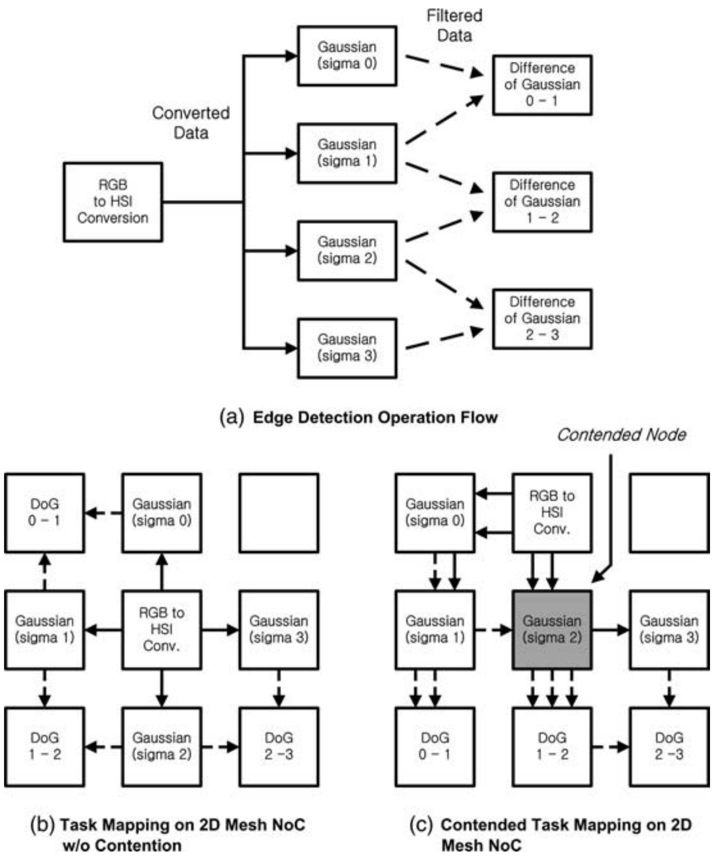

Filtered Data
Converted Data
RGB to HSI Conversion
Gaussian (sigma 0)
Gaussian (sigma 1)
Gaussian (sigma 2)
Gaussian (sigma 3)
Difference of Gaussian 0 - 1
Difference of Gaussian 1 - 2
Difference of Gaussian 2 - 3
(a) Edge Detection Operation Flow
Contended Node
DoG 0 - 1
DoG 1 - 2
DoG 2 -3
RGB to HSI Conv.
(b) Task Mapping on 2D Mesh NoC w/o Contention
(c) Contended Task Mapping on 2D Mesh NoC

Figure 2.39 Mapping of edge detection task into conventional mesh NoC

homogeneous SoC with conventional 2D mesh NoC. Because there is no contention in data flow for the task mapping of Figure 2.39b, it will outperform the task mapping of Figure 2.39c, even though all other conditions are equally given. The contention of data flow in Figure 2.39c is visualized by the number of arrows in the same locations. The drawback of the conventional 2D mesh NoC is dependency of overall SoC performance on mapping of the task. Even more, finding optimal task mapping may be very difficult for applications with complex data dependencies. Longer average hop counts and possibilities of deadlock on the way of finding bandwidth-optimized task mapping are additional drawbacks of the 2D mesh NoC.

Feasibility of task mapping on the MC-NoC can be judged from Figure 2.40. For simplicity, crossbar switches are not drawn and only a portion of the MC-NoC is depicted in this figure. In the MC-NoC, processors and dual-port SRAMs are interconnected through the crossbar switches which provide full nonblocking connections. Therefore, interchanging task mappings just inside the left side or right side

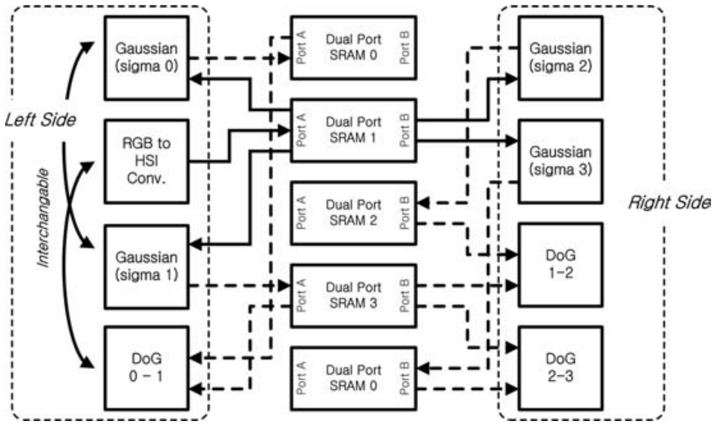


Figure 2.40 Mapping of edge detection task on MC-NoC

of the MC-NoC does not affect the data flow characteristic and resulting overall performance. For example, interchanging task mapping of *difference of Gaussian* (DoG) 0–1 and RGB to HSI conversion in Figure 2.40 has no impact on contentions in data flow. This attribute of the MC-NoC improves the flexibility of task mapping on a homogeneous SoC, because the key decision of task mapping reduces to whether the given task is mapped on the left side or the right side. Dual-port SRAMs are adopted to remove performance loss when SRAM access comes from both left and right sides.

In addition, variations on required bandwidth between co-working tasks are also successfully supported by the MC-NoC. If a large bandwidth is required for some tasks, multiple numbers of SRAMs can be dynamically assigned for the demanding task. For a small amount of data transfer, only one dual-port SRAM is assigned. With regard to traffic, the MC-NoC improves locality because most packet transactions between processors and memories are confined to a single crossbar switch to which the involved processors and SRAMs are connected.

Evaluation of the MC-NoC

To demonstrate feasibility and flexibility of task mapping on the MC-NoC, we shall briefly report experimental results showing how the overall performance is affected by different task mappings (Figure 2.41).

Although the MC-NoC is drawn as a single rectangular "black box" for simplicity, the architecture shown in Figure 2.41a is exactly applied for our performance comparison. First, RGB to HSI conversion, Gaussian filter operations, and DoG calculation tasks are mapped randomly on the given architecture (Figure 2.41a). Second, Gaussian filter operations are mapped on the upper half of the SoC, while DoG calculations are mapped on the lower half (Figure 2.41b). Similarly, the DoG and

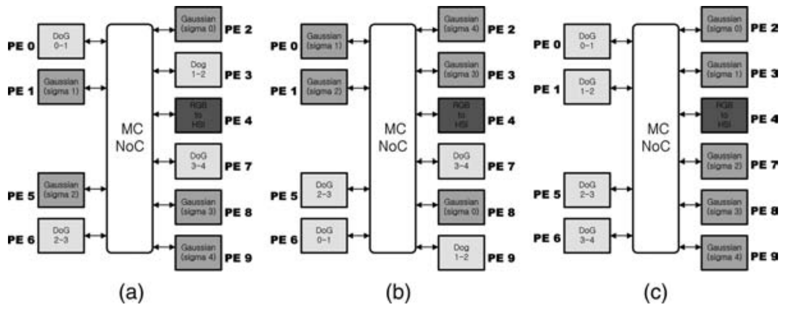


Figure 2.41 Task mapping of edge detection on the MC-NoC

Gaussian filtering tasks are separated into left and right, respectively, in the third task mapping configuration (Figure 2.41b).

The results of our performance comparison are given in Table 2.4. In this comparison, verilog HDL description is used for the MC-NoC and other parts of the SoC. Therefore, the performance comparison result derived from the simulation is cycle-accurate. In Table 2.4, numbers in the "cycle count" column refer to clock cycles required to perform edge detection for 320 $\times$ 240 pixels of image. The numbers in the extreme right column show cycle count ratio compared to the task mapping configuration in Figure 2.41a. The results in Table 2.4 prove the flexibility of task mapping on the proposed MC-NoC. In the MC-NoC based homogeneous SoC, the difference in overall performance according to the various task mapping is less than 3%. This feature of the MC-NoC also facilitates software level optimization.

Table 2.4 Performance comparisons of different task mappings on the MC-NoC

Task mapping	Cycle count	Ratio to mapping (*a*)
(a)	8,202,900	1
(b)	8,086,580	0.986
(c)	8,021,820	0.978

2.4.3.3 FAUST (Flexible Architecture of Unified System for Telecoms)

Eleven European industrial bodies, research institutes and universities launched a joint project named 4-MORE, standing for "4G Multi-carrier CDMA multiple antenna system-on-chip for radio enhancements." Recently, their architecture has been published for the application of orthogonal frequency-division multiplexing (OFDM) to multi-carrier baseband processing such as 802.11n, 802.16e, 3GPP/LTE [39, 40]. They proposed *asynchronous* network-on-chip (ANOC) with a GALS (globally asynchronous locally synchronous) paradigm. The ANOC architecture uses virtual channels to provide low latency and quality of service, and is implemented in quasi-delay-insensitive (QDI) asynchronous logic [41].

The FAUST chip integrates 20 asynchronous NoC routers, 23 synchronous units including an ARM946 core, embedded memories, various IP blocks, reconfigurable data-path engines, and one clock management unit to generate the 24 distinct unit clocks (Figure 2.42).

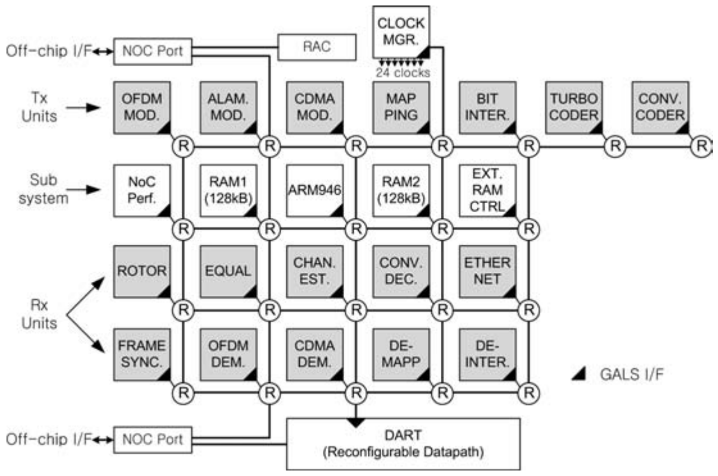


Figure 2.42 FAUST architecture

To integrate any synchronous IP within the ANOC architecture, a dedicated network interface performs two main tasks. The first is synchronization between the synchronous and asynchronous logic domains using ad-hoc decoupling FIFOs [42]. The second task provides all facilities to access the NoC communication infrastructure: network routing path programming, network data packet generation, and IP core configuration.

Table 2.5 presents the main features of the FAUST chip. It is implemented in a 6 ML, 130 nm, 1.2 V CMOS process from STMicroelectronics. The whole chip integrates more than 3 Mgates and 3.5 MB of embedded RAM, which corresponds to a core area of 72.71 mm^2 and a chip area of 79.5 mm^2. The maximum NoC throughput measured between two adjacent nodes or between an IP and its connected node is 5.12 Gb/s per link. The latency is about 6 ns per crossed node, 12 ns for the GALS IF, and 12 ns for the network interface.

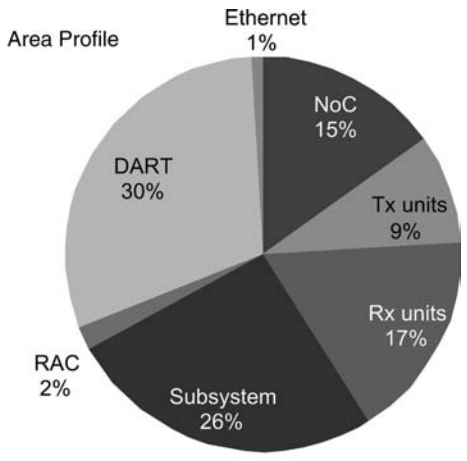
A real-time 100 Mb/s single-input single-output (SISO) OFDM transceiver needs a bandwidth of 10 Gb/s that corresponds to a 10% network load, for a complexity of 1.7 million gates at the transmitter level and 1.9 Mgates at the receiver level. The integration of each IP within the NoC costs 41 Kgates ($\sim$0.45 mm^2) for the 5×5 asynchronous node (19K), the GALS interface (12 K), and the network interface

Table 2.5 Features of the FAUST chip

NoC architecture	
Topology and size	2D mesh including 20 nodes
Switching and routing modes	Packet switching and wormhole routing
Flow control technique	Credit-based
Flit size (i.e., word size)	32 bits
QoS support	2 virtual channels
I/O	Direct external NoC accesses
Implementation	Asynchronous logic (QDI)
Power-saving technique	Dynamic frequency scaling
Computing and memory aspects	
IP count	23 units connected to the NoC
Processor core	ARM946 ES
DMA engines	3 units to manage on-chip and off-chip memories
Reconfigurable datapath	SIMD structure for channel estimation
Host computer interface	100 Mb/s full duplex Ethernet unit
Technology and complexity	
Process	130 nm CMOS 6 ML (STMicroelectronics)
Logic gate count	3.124 Mgates (excluding SRAM blocks)
On-chip clocks	24
Die size	$8.900\,\text{mm} \times 8.933\,\text{mm} = 79.50\,\text{mm}^2$
Package	BGA420 ($35\,\text{mm} \times 35\,\text{mm}$)
Signal I/O count	275
Supply voltage	1.2 V for core, 3.3 V for I/O
Measured performance	
NoC throughput	5.120 Gb/s per link
IP operating frequency	162 MHz
Chip power consumption	640 mW in TX mode, 760 mW in RX mode
NoC area	15% of the global area
NoC power consumption	6% of the global consumption

(10 K without configuration registers). The 20-node NoC represents about 15% of the overall area, and the average complexity of the 23 IP connected is close to 300 Kgates (including RAM). Using an optimized frequency scaling between 160 MHz and 250 MHz, the transceiver functions consume 640 mW in TX mode and 760 mW in RX mode. The NoC power consumption accounts for only about 6% of the overall power consumption for a typical traffic defined by the targeted applications (Figure 2.43).

The cost of the NoC in terms of area is similar to a bus-based architecture, but the properties of NoC structures are better suited to addressing the design issues associated with complex SoCs.



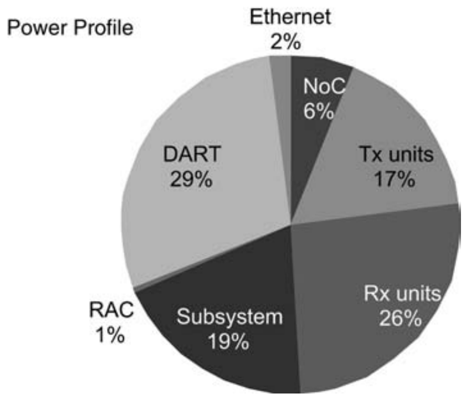


Figure 2.43 Area and power profile of the FAUST chip

References

1 Sohn, J.-H. *et al.* (2005) Low-power 3D graphics processors for mobile terminals. *IEEE Commun. Mag.*, **33** (12), 90–99.
2 Kurose, Y., Kumata, I., Okabe, M. *et al.* (2004) A 90 nm embedded DRAM single chip LSI with a 3D graphics, H.264 codec engine, and a reconfigurable processor. *Proc. of Hot Chips 16: Symposium on High Performance Chips 2004.*
3 Imagination Technology MBX graphics IP core. Available at http://www.imgtec.com/powervr/mbx.asp.
4 ARM RISC processor. Available at http://www.arm.com.
5 OpenGL-ES Khronos group. Available at http://www.khronos.org/.
6 Fowler, Martin and Scott, Kendall (2000) *UML Distilled*, 2nd edn, Addison–Wesley, Boston.
7 Yoo, H.-J. *et al.* (2007) *Low-power NoC for High-performance SoC Design*, CRC Press, New York.
8 Chndrakasan, Anantha P. and Broderson, Robert W. (1996) *Low-power Digital CMOS Design*, Kluwer Academic, Boston.
9 Chandrakasan, Anantha and Brodersen, Robert (1998) *Low-power CMOS Design*, IEEE Press, New York.
10 Gupta, Ankur and Hattori, Toshihiro (2007) Low-power CMOS design. *Asia and South Pacific Design Automation Conference* (tutorials).

11 Sriram, Vangal *et al.* (2007) On an 80-tile 1.28 Tflops network-on-chip in 65nm CMOS. *Digest of Technical Papers, IEEE International Solid State Circuits Conference*, pp. 98–589.

12 Brucek, Khalany *et al.* (2007) A programmable 512 GOPS stream processor for signal, image, and video processing. *Digest of Technical Papers, IEEE International Solid State Circuits Conference*, pp. 272–602.

13 Didier, Lattard *et al.* (2007) A telecom baseband circuit based on an asynchronous network-on-chip. *Digest of Technical Papers, IEEE International Solid State Circuits Conference*, pp. 258–601.

14 Benini, Luca and De Micheli, Giovanni (2002) Networks on chips: a new SoC paradigm. *IEEE Comput.*, **35**, 70–78.

15 Dally, W.J. and Towles, B. (2001) Route packets, not wires: on-chip interconnection networks. *Proc. of IEEE Design Automation Conference*, pp. 684–689.

16 Taylor, M.B. *et al.* (2002) The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro.*, **22** (2), 25–35.

17 Sohn, Ju.-Ho. *et al.* (2006) A 155-mW 50M vertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *IEEE J. Solid-St. Circ.*, **41** (5), 1081–1091.

18 Intel Xscale processor. Available at http://www.intel.com/design/intelxscale/.

19 AMBA AXI specification.

20 OCP 2.0 protocol specification.

21 STBus functional specifications, STMicroelectronics public web support site, http://www.stmcu.com/inchtml-pages-STBus_intro.html, April 2003.

22 Goossens, Kees *et al.* (2005) Æthereal network on chip: concepts, architectures, and implementations. *IEEE Des. Test Comput.*, **22**, 414–421.

23 Zimmermann, Hubert (1980) OSI reference model: the ISO model of architecture for open systems interconnection. *IEEE T. Commun.*, **28** (4), 425–432.

24 Wingrad, D. (2001) MicroNetwork-based integration for SOCs. *Proc. of Design Automation Conference 2001*, pp. 673–677.

25 Choi, Kyusun and Adams, William S. (1992) VLSI implementation of a 256×256 crossbar interconnection network. *Proc. of IEEE 6th International Parallel Processing Symposium*, pp. 289–293.

26 Lee, S.-J. *et al.* (2005) Adaptive network-on-chip with wave-front train serialization scheme. *Digest of Technical Papers, IEEE Symposium on VLSI Circuits 2005*, pp. 104–107.

27 Worn, F., Lenne, P., Thiran, P., and De Micheli, G. (2005) A robust self-calibrating transmission scheme for on-chip networks. *IEEE T. VLSI Syst.*, **13**, 126–139.

28 Millberg, M. *et al.* (2004) The Nostrum backbone: a communication protocol stack for network-on-chip. *Proc. of International Conference on VLSI Design*, pp. 693–696.

29 Lee, K. *et al.* (2006) Low-power network-on-chip for high-performance SoC design. *IEEE T. VLSI Syst.*, **14**, 148–160.

30 Lee, S.-J. *et al.* (2005) Packet-switched on-chip interconnection network for system-on-chip applications. *IEEE T. Circuits-II*, **52**, 308–312.

31 Lee, Se-Joong *et al.* (2003) An 800 MHz star-connected on-chip network for application to systems on a chip. *Digest of Technical Papers, IEEE International Solid-States Circuits Conference 2003*, pp. 468–469.

32 Lee, Kangmin *et al.* (2004) A 51mW 1.6GHz on-chip network for low-power heterogeneous SoC platform. *Digest of Technical Papers, IEEE International Solid-States Circuits Conference 2004*, pp. 152–518.

33 Held, J. *et al.* From a Few Cores to Many: A Tera-scale Computing Research Overview, white paper, Intel Corporation, 2006. www.intel.com.

34 Vangal, S., Borkar, N.Y., and Alvandpour, A. (2005) A six-port 57GB/s double-pumped non-blocking router core. *Digest of Symposium on VLSI Circuits*, pp. 268–269.

35 Tschanz, J., Narendra, S.G., Ye, Y. *et al.* (2003) Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE J. Solid-St. Circ.*, **1**, 1838–1845.

36 Khellah, M., Kim, N.S., Howard, J. *et al.* (2006) A 4.2GHz 0. 3mm^2 256kb dual-V$_{cc}$ SRAM building block in 65 nm CMOS. *Digest of Technical Papers, ISSCC 2006*, pp. 624–625.

37 Kim, D. *et al.* (2007) Solutions for real chip implementation issues of NoC and their application to memory-centric NoC. *Proc. of IEEE International Symposium on Networks-on-Chip (NOCS) 2007*, pp. 30–39.

38 Lowe, D.G. (2004) Distinctive image features from scale-invariant keypoints. *ACM Int. J. Comput. Vision*, **60** (2), 91–110.

39 Lattard, D. *et al.* (2007) A telecom baseband circuit based on an asynchronous network-on-chip. *Digest of Technical Papers, ISSCC 2007*, pp. 258–259.

40 Viviet, P. *et al.* (2007) FAUST: an asynchronous network-on-chip based architecture for telecom applications. *Proc. of Design, Automation and Test in Europe Conference.*
41 Beigne, E., Clermidy, F., Vivet, P. *et al.* (2005) An asynchronous NOC architecture providing low latency service and its multi-level design framework. *Proc.of ASYNC, New York, 2005*, pp. 54–63.
42 Beigne, E. and Vivet, P. (2006) Design of on-chip and off-chip interfaces for a GALS NoC architecture. *Proc. of ASYNC, Grenoble, 2006*, pp. 172–181.

3

Introduction to 3D Graphics

Three-dimensional graphics started with the display of data on hardcopy plotters and CRT screens soon after the introduction of computers themselves. It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual structures, natural phenomena, and so on.

Until the early 1980s, 3D graphics was used in specialized fields because the hardware was expensive and there were few graphics-based application programs that were easy to use and cost-effective. Since personal computers have become popular, 3D graphics is widely used for various applications, such as user interfaces and games. Today, almost all interactive programs, even those for manipulating text (e.g., word processors) and numerical data (e.g., spreadsheet programs), use graphics extensively in the user interface and for visualizing and manipulating the application-specific objects. So 3D graphics is no longer a rarity and is indispensable for visualizing objects in areas as diverse as education, science, engineering, medicine, commerce, military, advertising, and entertainment.

Fundamentally, 3D graphics simulates the physical phenomena that occur in the real world – especially dynamic mechanical and lighting effects – on 2D display devices. Thus the role of the 3D graphics pipeline is to project 3D objects on to a 2D screen with appropriate lighting effects. As shown in Figure 3.1, the 3D graphics pipeline is composed of application, geometry, and rendering stages. The application stage computes the dynamic behavior description of 3D objects; the objects are transformed and vertex information is computed in the geometry stage; and the information for each pixel is computed in the rendering stage. Recently, programmability has been introduced into the 3D graphics pipeline to support various graphics effects, including non-photorealistic effects. This approach supports programmability in the geometry and rendering stages.

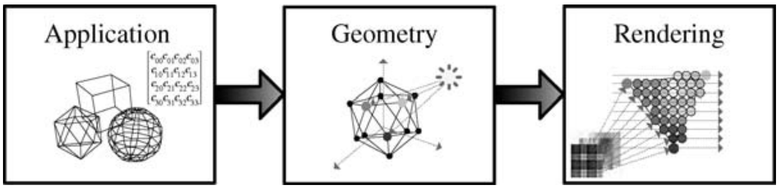


Figure 3.1 3D graphics pipeline stages

This chapter is organized as follows. Section 3.1 describes the overall graphics pipeline including the application, geometry, and rendering stages. The modified programmable version is explained in Section 3.2.

3.1 The 3D Graphics Pipeline

3.1.1 The Application Stage

The application stage starts and drives the 3D graphics pipeline by feeding 3D models to be rendered according to the information determined in the application stage. Thus it should be understood in the context of the 3D graphics pipeline although the application stage is not an actual part of the 3D graphics subsystem.

The application stage generates the movements of 3D objects based on the information gathered from the environment. The environmental information includes the user interaction from keyboard or mouse, and internally generated information in the real world. Thus the application stage also processes the artificial intelligence (AI), collision detection, and physics simulations to generate this information. Based on these, the objects' movements produce the 3D animation by moving the objects from frame to frame.

The dynamics of the 3D objects and the camera position defined in the application stage also affects the animation, in which the 3D objects are moved by frames taken at certain viewpoints. In the application stage, the 3D objects are represented as sets of polygons or triangles and their movements are specified by geometry transformation matrices. These matrices are sent to the geometry stage to be used for transformation of vertex positions.

3.1.2 The Geometry Stage

The geometry stage operates on the polygons or vertices. The major operations in this stage are, first, geometric transformation of the vertices according to the matrices determined in the application stage and, second, the lighting which determines the color intensity of each vertex according to the relationship between the properties of the vertex and the light source.

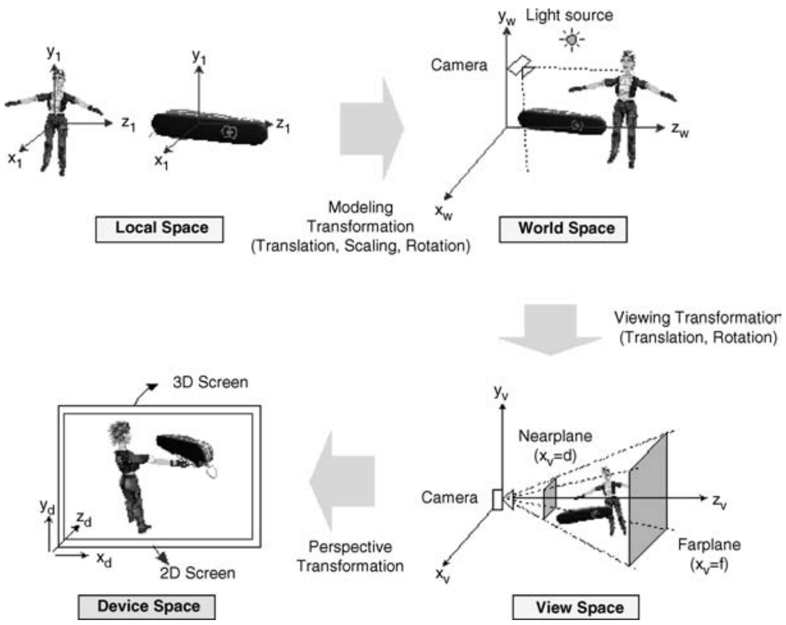
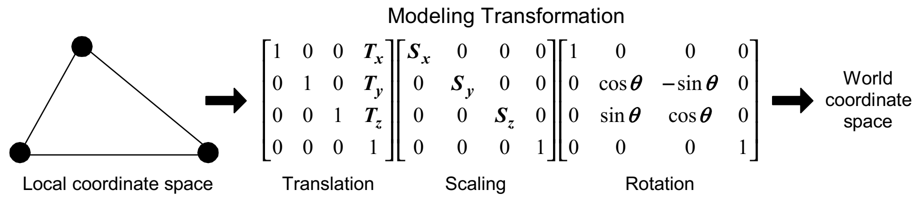


Figure 3.2 Spaces and coordinate systems in 3D graphics

The geometric transformation goes through several coordinate transformations as shown in Figure 3.2. The objects defined in local model coordinate space are transformed into world coordinate and camera coordinate spaces, and finally into the device coordinate space. Each coordinate space for the geometric transformation is explained in detail in this section.

3.1.2.1 Local Coordinate Space

The local coordinate space is the space where 3D objects are developed. For modeling convenience, the 3D objects are modeled in their local coordinate spaces and the origin is located at the center or corner of each model. These models are gathered into the world space by transforming the center of each local coordinate space to the point where the object is located in the world space. This is called *modeling transformation* and it involves shifting, rotating and scaling operations on the original 3D object. The vertex normal is also transformed into the world space for the lighting operation. Each operation is specified by matrix coefficients, and the matrices are combined into a single modeling transformation matrix by multiplying the matrices. Figure 3.3 shows the modeling transformation operations and examples of corresponding matrices.



$$\begin{bmatrix}1&0&0&T_x\\0&1&0&T_y\\0&0&1&T_z\\0&0&0&1\end{bmatrix}\begin{bmatrix}S_x&0&0&0\\0&S_y&0&0\\0&0&S_z&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&\cos\theta&-\sin\theta&0\\0&\sin\theta&\cos\theta&0\\0&0&0&1\end{bmatrix}$$

Figure 3.3 Modeling transformation example

3.1.2.2 World Coordinate Space

The 3D models are gathered in the world coordinate space to make a 3D world. In this world space, the light sources are defined and intensity calculations are performed for the objects in the 3D world. According to the shading strategy chosen, the actual lighting operation takes place in this coordinate space or later in 3D screen coordinate space. If Gouraud shading (also called intensity interpolation) is adopted [1], the intensity calculation takes place in this space for each vertex of the object using its transformed vertex coordinates and normal values.

The lighting model for the intensity calculation is composed of ambient, diffuse, and specular terms as follows:

$$I = i_{\text{amb}} + i_{\text{diff}} + i_{\text{spec}}. \tag{3.1}$$

The ambient light approximates a constant background light coming from all directions, which is represented as below, where l_{amb} is a global ambient light source parameter and m_{amb} is ambient material parameter:

$$i_{\text{amb}} = l_{\text{amb}} \times m_{\text{amb}}. \tag{3.2}$$

It does *not* depend on the geometrical relationships between the light position and the pixel position in 3D space.

The diffuse light term depends on the light direction vector (L) and the surface normal vector (N) as shown in Figure 3.4. This term is maximized when the light source is incident perpendicular to the object surface. Thus it can be described by:

$$i_{\text{diff}} = l_{\text{diff}} \times m_{\text{diff}} \times \cos\theta = l_{\text{diff}} \times m_{\text{diff}} \times (N \cdot L) \tag{3.3}$$

where l_{diff} is the diffuse parameter of the light source and m_{amb} is the diffuse color of the material.

The specular term depends on the angle (ϕ) between the light reflection vector (R) and the view vector (V). Assuming the object's surface is a shiny material, such as a mirror, we can obtain maximum intensity when the viewing vector is coincident with the reflection vector. As the viewing vector deviates from the reflection vector, the

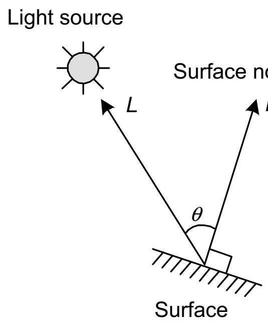


Figure 3.4 Geometry for diffuse lighting

specular term becomes smaller, which is described by:

$$i_{\text{spec}} = l_{\text{spec}} \times m_{\text{spec}} \times \cos^{m_{\text{shin}}} \phi = l_{\text{spec}} \times m_{\text{spec}} \times (R \cdot V)^{m_{\text{shin}}}. \tag{3.4}$$

Here, l_{spec} is the color of the specular contribution of the light source and m_{amb} is the specular color of the material. The exponent term, m_{shin}, in the specular intensity calculation represents the shininess of the surface. The reflection vector R can be calculated by:

$$R = 2(N \cdot L)N - L \tag{3.5}$$

where L and N are normalized vectors. A popular variant of (3.4) avoiding the computation of the reflection vector R is:

$$i_{spec} = l_{\text{spec}} \times m_{\text{spec}} \times \cos^{m_{shin}} \varphi = l_{\text{spec}} \times m_{\text{spec}} \times (N \cdot H)^{m_{\text{shin}}}. \tag{3.6}$$

The geometries of the specular lighting are illustrated in Figure 3.5.

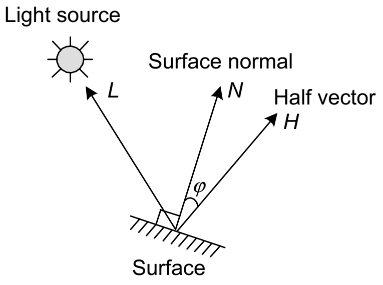
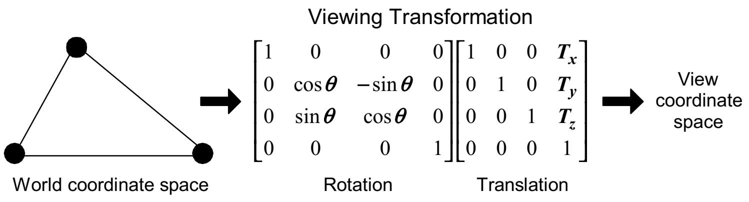


Figure 3.5 Geometry for specular lighting

If Phong shading is applied [2], the intensity is calculated per pixel and is deferred until the objects are transformed into the 3D screen coordinate space; this will be explained later. In this case, the normal vectors should be carried up to the 3D screen coordinate space for the lighting operation.

In the world space, a camera is set up through which we can observe the 3D world from a certain position. By moving the position and angle of the camera in the world space, we can get scenes navigating the 3D world. For convenience of computations in the following stages, the world coordinate space is transformed into a view coordinate space with the camera located at the origin. This transformation is called the *viewing transformation.*

Figure 3.6 illustrates the viewing transformation which first performs coordinate translation of the camera and then the rotation.



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.6 Viewing transformation example

3.1.2.3 Viewing Coordinate Space

After the viewing transformation, all the objects are spaced with respect to the camera position at the origin of the view space. In this view space, *culling* and *clipping* operations are carried out in preparation for later rendering stage operations.

When only the front-facing polygons of a 3D object are visible to the camera, a culling operation, also called "back-face culling," can remove polygons that will be invisible on the 2D screen. Thus the number of polygons to be processed in the later stages is reduced. This is done by rejecting back-facing polygons when seen from the camera position, based on the following strategy:

$$\text{Visibility} = N_p \cdot N. \tag{3.7}$$

Based on this equation, we can determine whether the polygon is back-facing or not by testing the sign of the inner product of two vectors: the polygon normal vector (N_p) and the line of sight vector (N). Therefore, a large amount of processing in later stages can be avoided if the visibility of a polygon is determined and culled out at this stage.

In the view space, the *view frustum* is defined to determine the objects to be considered for a scene. Figure 3.7 shows a view frustum defined with six clipping planes, including the near and far clip planes. The objects are transformed into the clipping coordinate space by perspective transformation shown in Figure 3.8, which is defined in terms of the view frustum definition.

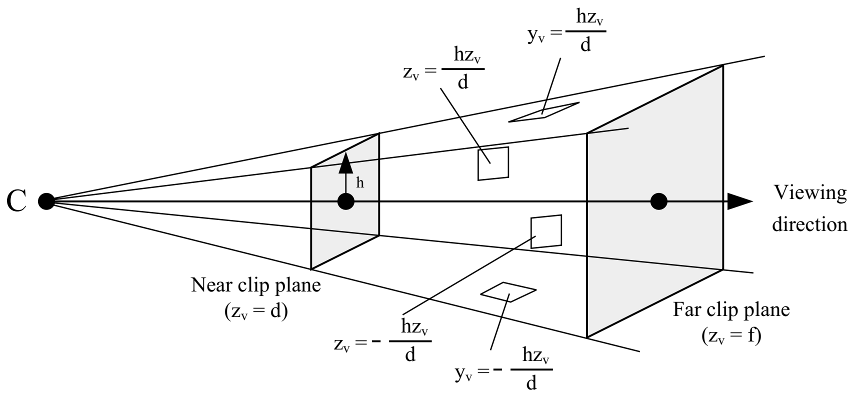
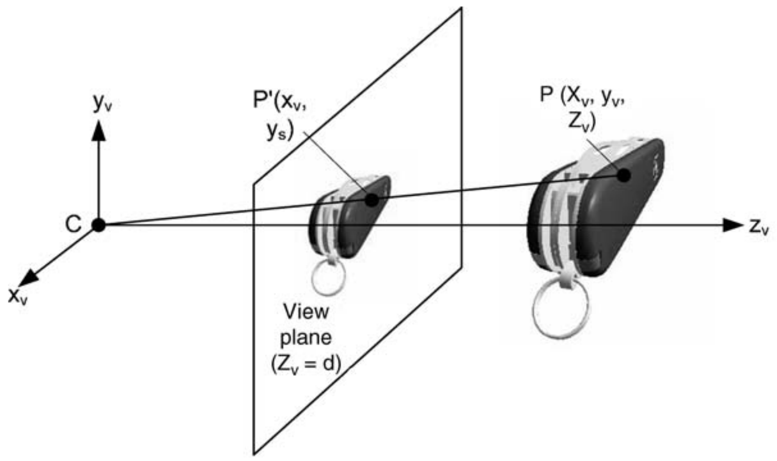


Figure 3.7 View frustum



$$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix} = \begin{bmatrix} d/h & 0 & 0 & 0 \\ 0 & d/h & 0 & 0 \\ 0 & 0 & f/(f-d) & -df/(f-d) \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

$$x_s = d\frac{x_e}{hz_v} \qquad y_s = d\frac{y_e}{hz_v} \qquad z_s = \frac{f(1-d/z_v)}{(f-d)}$$

$$\Rightarrow \quad x_s = \frac{X}{w} \qquad y_s = \frac{Y}{w} \qquad z_s = \frac{Z}{w}$$

, where

$$X = \frac{d}{h}x_v \qquad Y = \frac{d}{h}y_v \qquad Z = \frac{fz_v}{f-d} - \frac{df}{f-d} \qquad w = z_v$$

Figure 3.8 Perspective transformation and its matrix

3.1.2.4 Clipping Coordinate Space

Although polygon clipping can be done in the view coordinate space against the view frustum with six planes, the clipping occurs in this clipping space to avoid solving plane equations. Polygon clipping against a square volume in this space is easier than in the view space, since simple limit comparisons with *w* component value as follows are sufficient for the clip tests:

$$\begin{aligned} -w \leq x \leq w \\ -w \leq y \leq w \\ -w \leq z \leq w. \end{aligned} \tag{3.8}$$

The polygons are tested according to (3.8) and the results fall into one of three categories: completely outside, completely inside, or straddling. The “completely outside” polygons are simply rejected. The “completely inside” polygons are processed as normal. The “straddling” polygons are clipped against the six clipping planes, and those inside are processed as normal.

After clipping, the polygons in the clipping space are divided by their *w* component, which converts the homogeneous coordinate system into a normalized device coordinate (NDC) space, as described below.

3.1.2.5 Normalized Device Coordinate Space

The range of polygon coordinates in the normalized device coordinate (NDC) space is $[-1, 1]$, as shown in Figure 3.9. The polygons in this space are transformed into the device coordinate space using the *viewport transformation*, which determines how a scene is mapped on to the device screen. The viewport defines the size and shape of the device screen area on to which the scene is mapped. Therefore, in this transformation, the polygons in NDC are enlarged, shrunk or distorted according to the *aspect ratio* of the viewport.

3.1.2.6 Device Coordinate Space

After viewport transformation, the polygons are in the device coordinate space as shown in Figure 3.10. In this space, all the pixel-level operations, such as shading, Z testing, texture mapping, and blending are performed. Up to the viewport transformation is called the geometry stage, and the later stages are called the rendering stage, where each pixel value is evaluated to fill the polygons.

3.1.3 The Rendering Stage

In the rendering stage, pixel-level operations take place in the device coordinate space. Various pixel-level operations are performed, such as pixel rendering by Gouraud or

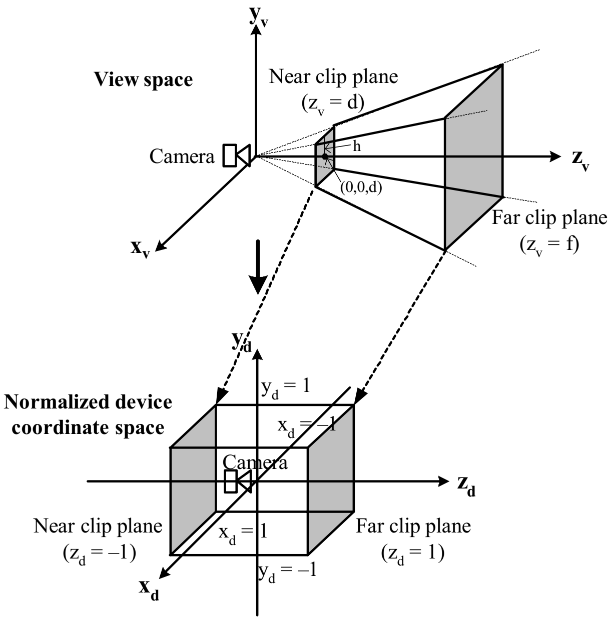


Figure 3.9 Normalized device coordinate space

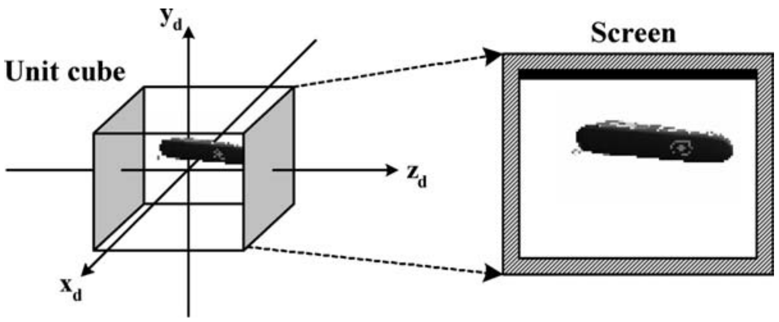


Figure 3.10 Device coordinate space

Phong shading, depth testing, texture mapping, and several extra effects such as alpha blending and anti-aliasing.

3.1.3.1 Triangle Setup

Using the vertices from the geometry stage, a triangle is made up before the rasterization can start. Information on the triangle attributes is calculated, such as the attribute deltas between triangle vertices and edge slops for the rasterization. This is called the *triangle setup* operation. These values are given to the rasterizer, where the triangle attributes – such as color and texture coordinates – are interpolated for the pixels inside a triangle by incrementing the delta value as we move one pixel at a time.

3.1.3.2 Shading

The first thing to be determined in the rendering stage is what shading algorithm is to be used. There are two commonly used shading algorithms: Gouraud and Phong.

Gouraud shading is a *per-vertex algorithm* that computes the intensity of each vertex, as discussed already. The rendering stage just linearly interpolates the color of each vertex determined in the geometry stage to determine the intensities of pixels inside the polygon. In contrast, Phong shading is a *per-pixel algorithm* in which the intensity of every pixel in a given polygon is computed rather than just interpolating the vertex color. Thus, Phong shading requires high computational power to do complex lighting operations according to the lighting model explained earlier. However, it can generate very sharp specular lighting effects even if the light source is located just above the center of a polygon, in which case the pixel color changes rapidly. This is illustrated in Figure 3.11.

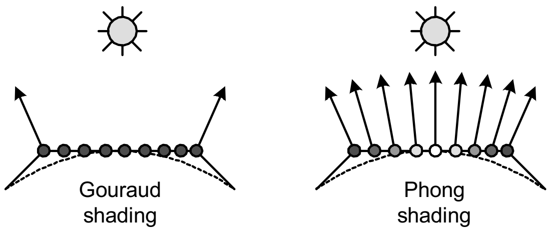


Figure 3.11 Shading schemes

3.1.3.3 Texture Mapping

Texture mapping enhances the reality of 3D graphics scenes significantly with relatively simple computations. This operation wraps a 3D object with 2D texture image obtained by taking a picture of the real surface of a 3D object. Thus, texture mapping can easily represent surface details such as color and roughness without requiring complex computations of lighting or geometric transformation. On the other hand, it requires large memory bandwidth to fetch texture image data, called *texels*, to be used for the mapping. It also requires filtering of the texels, called *texture filtering*, in order to reduce the aliasing artifacts of the textured image. There are various texture filtering algorithms: point sampling, bilinear interpolation, mip-mapping, and so on. The texture mapping operation is illustrated in Figure 3.12, and the difference between using and not using texture mapping is shown in Figure 3.13.

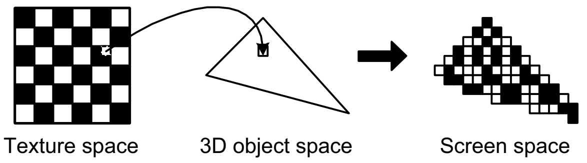


Figure 3.12 Texture mapping

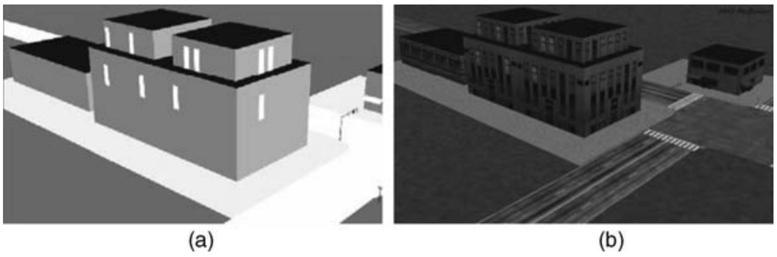


Figure 3.13 Texture mapping effects: (a) flat shaded image, and (b) texture-mapped image

3.1.3.4 Depth Testing

Depth testing (or Z-testing) is used to remove hidden surface pixels. For this scheme, a depth buffer of the same size as the 2D screen resolution is required. The depth buffer stores the depth values of the pixels drawn on the screen, and every depth value of the pixel being drawn is compared with the value at the same position in the depth buffer to determine its visibility. The pixel is drawn on the screen if it passes the test. In this case, the depth and color values (R, G, B) of the pixel are updated into the depth and frame buffer. Otherwise, the depth and color values are discarded because the pixel resides further from the viewer than the one currently stored at the given position in the frame buffer.

3.1.3.5 Blending, Fog, and Anti-aliasing

There are several extra effects that can enhance the final scene. "Alpha blending" is used to give a translucent effect to the polygon being drawn. This scheme blends the color value of the pixel being processed with the one from the frame buffer at the same position. This blending is done according to the alpha value associated with the pixel. The alpha value represents the opacity of the given pixel. After blending, the result color value is updated to the frame buffer.

Meanwhile, the final graphics image can seem unrealistic because it is too sharp for an outdoor scene. The scene can be made more realistic by adopting the "fog effect," which is a simple atmospheric effect that makes objects fade away as they are located further from the viewpoint.

"Anti-aliasing" is applied to reduce the jagged look of the final 3D graphics scene. The jagged look is due to the approximation of ideal lines with digitized pixels on a screen. To remove these artifacts, an anti-aliasing algorithm calculates the coverage value – what fraction of the pixel covers the line.

Although there are other ways to classify the 3D graphics pipeline, and various graphics algorithms in each pipeline stage, those described above represent the fundamental structure of a 3D graphics pipeline.

3.2 Programmable 3D Graphics

The functions of the 3D graphics pipeline explained in the previous section are fixed. However, a fixed function pipeline can support only predetermined graphics effects. Recently, new 3D graphics standards like OpenGL [3] and DirectX [4] have introduced programmable 3D graphics [5] to support various effects by programming the graphics pipeline to adopt a certain graphics effect [6].

3.2.1 Programmable Graphics Pipeline

The modern programmable graphics pipeline adopts two major programmable stages: vertex shading and pixel shading. In this configuration, the vertex shader and pixel shader replace, respectively, the transformation and lighting operations of the geometry stage and the texture mapping operation of the rendering stage. This is illustrated in Figure 3.14.

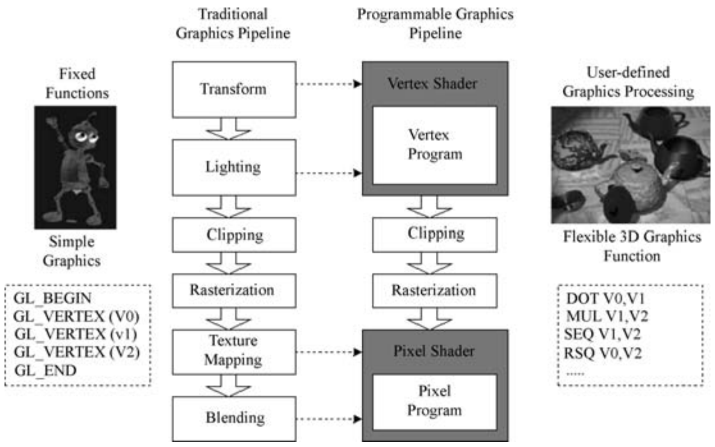


Figure 3.14 Programmable 3D graphics pipeline

The vertex shader works on vertex attributes such as the vertex position, normal, color, and texture coordinates. The pixel shader works on the position of each pixel and carries out the procedural texture mapping by accessing the texture sampler with its texture instructions. Detailed descriptions of these two programmable shaders follow.

3.2.1.1 Vertex Shader

The vertex shader is defined as a process that accepts input vertex streams and produces new vertex streams as the output. A vertex is composed of several attributes which include vertex position, color, texture coordinates, fog values, and some user-defined attribute values. The vertex shader updates these vertex attributes, so that a new vertex position is located in the clipping coordinate space.

The role of the vertex shader is to transform the vertex coordinates from the local coordinate space into the clipping space and to compute the intensity of vertex color. The other fixed-function stages of the pipeline – such as culling, clipping, perspective division, and viewport mapping – are not replaced by the vertex shader.

The general execution model of the vertex shader is shown in Figure 3.15. The vertex shader has several operand register files and write-only output register files. The source operand register files include read-only input register files, constant memory, and read–write temporary register file. These register files are made up of entries with floating-point four-component vectors.

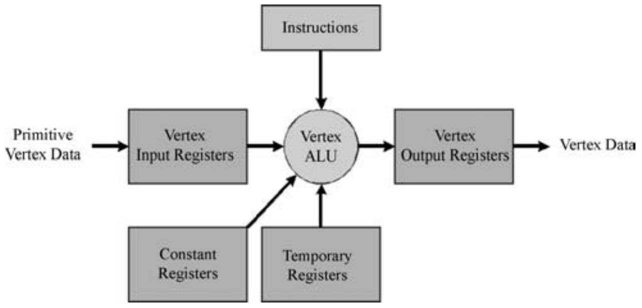


Figure 3.15 Vertex shader model

The vertex shader accesses the source registers to get vertex attributes and these registers contain two types of input data, changing *per vertex* or *per frame*. The input registers contain the changing per-vertex data such as the vertex position and normal, while the constant memory contains the changing per-frame data such as the transformation matrix, light position and material. Integer registers, which are not directly accessible from the shader programs, are also provided only for indexed array addressing.

After vertex shading, the result is written to the write-only output register file. The output registers have the transformed vertex position in the homogeneous clipping coordinate space, and lit vertex colors. The output vertex from the vertex shader goes through fixed-function stages and the pixel shader, which will be discussed in the following section.

3.2.1.2 Pixel Shader

A pixel (also known as a fragment shader) is defined by the point associated with its position in the device coordinate space, color, depth, and texture coordinates. The attributes of each pixel are interpolated in the rasterizer and these are used as the inputs for the pixel shader.

The pixel shader executes per pixel in the device coordinate space during rasterization. Its major operations include texture mapping and color blending according to a programmed routine. The pixel shader includes the texture sampler stage to give more flexibility in texture mapping. With the texture sampler, a dependent texture read – where the result from a texture read modifies the texture coordinates of a later texture access – becomes possible, enabling more advanced rendering effects. The other rendering parts – such as depth test, alpha blending, and anti-aliasing operations – remain as fixed functions separated from the pixel shader.

The general model of the pixel shader is shown in Figure 3.16, which is similar to that of the vertex shader. It also has a read-only input register file, constant memory, and read–write temporary register file. These register files are also of four-component floating-point vectors. The input register file contains interpolated pixel attributes such

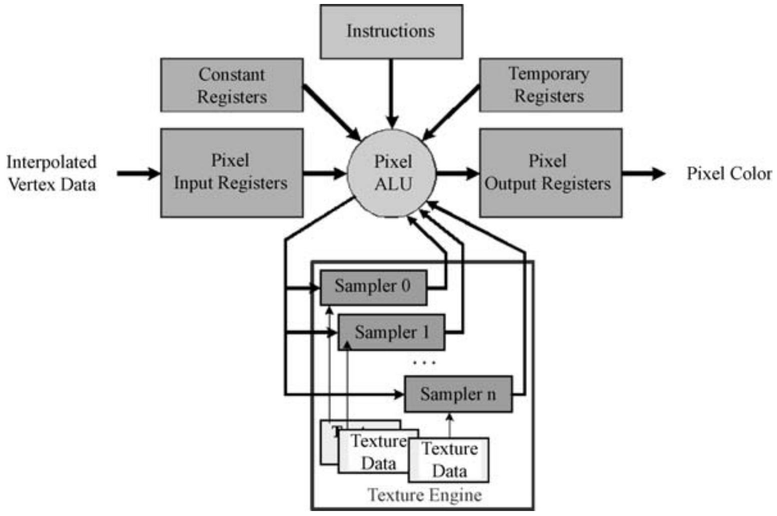


Figure 3.16 Pixel shader model

as color and texture coordinates. With the attributes from the input register and the texture read values, the pixel shader determines the resultant color that will be displayed and stores it in the output register file. The depth value is also determined to be used in the later depth test stage.

3.2.2 Shader Models

This subsection explains shader models based on the generic shader architectures described in the previous subsection. The shaders are basically numeric vector processors and they provide CPU-like assembly languages even though their instruction sets are mainly defined for four-component floating-point vectors and limited to 3D graphics processing only.

3.2.2.1 Shader Model 1.0

Shader model 1.0 was the first major programmable shader model [7]. The vertex shader of this model replaced the transform and lighting stages of the fixed-function pipeline. The pixel shader replaced the texture mapping and blending stages. However, its instruction usage was quite restrictive, and the number of registers was quite limited for use with complex shading algorithms.

Based on the generic architecture introduced in Figure 3.15, the vertex shader architecture in this version has four-way floating-point registers including 16 input registers, 12 temporary registers, 96-entry constant memory, 13 output registers, and a single address register. The address register is also of the four-way floating-point type and only the x component is rounded to the integer and used for indexing. The output registers have fixed names and meanings and are used for passing the vertex shader results on to the next stages of the graphics pipeline.

The vertex shader has two types of instruction: *general* and *macro*. General instructions take one instruction slot whereas macro instructions are expanded into several general instruction slots. A single vertex shader routine can use up to a maximum of 128 instruction slots. The operand modifiers are embedded into the instructions to support primitive unary operations on the operands without additional cycle costs. The operand modifiers supported in this model include source negation, swizzle, and destination mask. "Source negation" allows the source operand to be negated before it is used in the instruction. "Source swizzle" allows swapping or replication of the components in an operand register before it is used. The "destination mask" controls the updates to certain components in the destination register.

The pixel shader of this model is primitive and restricted. It has two main types of instruction: *arithmetic* and *texture addressing*. In this version, these instructions cannot be used in a mix and the texture addressing instructions should come before any arithmetic instructions. Therefore, the texture addressing instructions provide matrix transformation functionality for the texture coordinates to avoid the mixed use of arithmetic and texture addressing instructions.

The arithmetic instructions are used to blend the interpolated pixel colors passed from the rasterizer and the texels fetched from the texture addressing instructions. The blended pixel color and the depth value make up the final output of the pixel shader.

The programming model is limited in that a couple of useful instructions are missing, and instructions can be programmed in only a quite restricted manner. Several architectural advances have been made in later model specifications.

3.2.2.2 Shader Model 2.0

The vertex shader 2.0 version has an increased constant memory of 256 entries [8]. It also contains new Boolean registers for conditional execution and new integer registers used for counters in a loop. The maximum number of instructions is also increased to 256.

In this version, several new arithmetic instructions are defined. These include several vector and transcendental functions such as vector normalize, cross-product, power, and sincos functions. These can be considered as macro instructions since they are multi-cycle instructions and can be emulated by multiple instructions.

The most significant change in this version is *flow control instructions*, of which there are two types: *static* and *dynamic*. These flow control instructions can reduce code size by enabling loops to avoid repeated block copies. For example, the lighting routine for multiple light sources can avoid repeated copies of the lighting routine for each light source if static flow control instructions are supported.

In this model, there are significant improvements in the pixel shader. A maximum 64 instructions and 32 texture instructions are supported. With this model, all of the arithmetic instructions in the vertex shader are also supported in the pixel shader.

The texture coordinates and texture samplers are separated into two different register banks. They provide 8 texture coordinate registers and 16 sampler registers. There are only three texture lookup instructions: *texld* is used for regular texture sampling, *texldp* is for projective texture sampling, and *texldb* is for MIPMAP texture sampling.

3.2.2.3 Shader Model 3.0

Shader model 3.0 provides a more unified structure of the vertex and pixel shaders in order to make the shader codes more consistent [9]. Thus there are various common features between these two shaders. The most significant improvement is the *dynamic flow control mechanisms*. The dynamic flow control instructions allow if-statements, loops, subroutine calls, and breaks that are executed according to a condition value determined during program running. They also provide a predication scheme to better support fine-grained dynamic flow controls. This is preferred in cases of very short branching sequences to the dynamic branch instructions. This shader model supports nested flow controls: four nesting levels are supported for static flow control and 24 levels for dynamic flow control.

The arbitrary swizzling in this model allows the source components to be arranged in any order and eliminates the move of registers to align the components to a specific order. This arbitrary swizzling is also compatible with the texture instructions.

The vertex shader in this model incorporates 32 temporary registers and 12 output registers. Moreover, it provides a loop counter register, which allows indexing of constant memory within a loop, now to be used for the relative addressing of input and output registers. The maximum number of instructions is also increased to at least 512. Supporting longer instructions can reduce the state change of the vertex shader and thus improve its performance.

The most prominent improvement is to the vertex texturing. The vertex texturing instruction is quite similar to that of the pixel shader, except that only the *texldl* instruction is supported in the vertex shader since the rate of change is not available and the level of detail (LOD) should be calculated and provided to the texture sampling instruction.

The pixel shader 3.0 model supports 10 input registers, 32 temporary registers, and 256 constant registers. The predication register p0 is provided, and flow control is controlled by the loop counter register aL. The aL is also used to index input registers.

The minimum instruction count of this model has increased to 512. With the increased instruction count and the static and dynamic flow control schemes, several pixel shader routines can be combined into a single one, thereby reducing the shader state change and increasing the performance. The gradient instructions, *dsx* and *dsy*, are newly introduced to the pixel shader 3.0. These are used to calculate the rate of change of a pixel attribute across pixels in horizontal and vertical directions. This pixel shader model supports unlimited texture read operations.

3.2.2.4 Shader Model 4.0

Shader model 4.0 consists of three programmable shaders – vertex shader, pixel shader, and geometry shader – and defines a single common shader core virtual machine as the base for each of these programmable shaders [10]. This shader model provides expanded register files of 4096 temporary registers and 16 banks of 4096 constant memories. The instruction set also provides 32-bit integer instructions including integer arithmetic, bitwise, and conversion instructions, and supports infinite instruction slots. Its texture sampler state is decoupled from the texture unit. To this base model, each of the shaders adds some stage-specific functionality to make a complete shader.

The most significant improvement of this shader model is the *geometry shader*. The geometry shader can generate vertices on a GPU taking the output vertices from the vertex shader. It can add or remove vertices from a polygon and thus can produce more detailed geometry out of existing rather plain polygon meshes. The geometry shader can be used when a large geometry amplification such as tessellation is required.

References

1 Gouraud, H.(June 1971) Computer display of curved surfaces. Technical Report UTEC-CSc-71-113, Dept. of Computer Science, University of Utah.
2 Tuong-Phong, Bui(July 1973) Illumination for computer-generated images. Technical Report UTEC-CSc-73-129, Dept. of Computer Science, University of Utah.
3 OpenGL ARB (1999) *OpenGL Programming Guide*, 3rd edn, Addison–Wesley.
4 Microsoft Corporation. Available at http://msdn.microsoft.com/directx.
5 Lindholm, E., Kilgard, M.J., and Moreton, H. (2003) A user-programmable vertex engine. *Proc. of SIGGRAPH 2001*, pp. 149–158.
6 Gooch, A., Gooch, B., Shirley, P., and Cohen, E. (1998) A non-photorealistic lighting model for automatic technical illustration. *Proc. of SIGGRAPH 1998*, pp. 447–452.
7 Microsoft Corporation (2001) Microsoft DirectX 8.1 Programmer's Reference, DirectX 8.1 SDK.
8 Microsoft Corporation (2002) Microsoft DirectX 9.0 Programmer's Reference, DirectX 9.0 SDK.
9 Fernando, R. (2004) Shader Model 3.0 Unleashed, NVIDIA Developer Technology Group.
10 Blythe, D. (2006) The Direct3D 10 System. *Proc. of SIGGRAPH 2006*, pp. 724–734.

4

Mobile 3D Graphics

Advancements in very-large-scale integration (VLSI) technologies have enabled the integration of several system components on to a single chip, leading to so-called systems-on-a-chip. SoC technologies now allow high-speed low-power electronic devices to be manufactured in a compact size so as to eliminate slow and power-consuming off-chip communications between modules as well as area-consuming off-chip interconnections. As a consequence, mobile electronic devices such as smartphones and personal digital assistants (PDAs) have become major players in the consumer electronics market and their popularity increases every year.

However, mobile electronic devices have been hampered by their limited resources, such as small screen size, user input interfaces, computing capability, and battery lifetime. Thus their applications have also been restricted to relatively simple operations like text processing. Nevertheless, mobile electronic devices have evolved from being text-based into many kinds of multimedia applications such as MP3, H.264, and even to realtime 3D computer graphics. Figure 4.1 shows an example of a 3G system that consists of an RF frontend, a baseband modem, an application processor, and peripherals. The power consumption of each component in these types of system can be found elsewhere [1, 2].

Realization of 3D graphics is a challenging issue because the huge computing power and memory bandwidth inherently required for realtime processung has to be achieved within a limited power budget. In this chapter, design principles will be presented, and a few examples of mobile 3D graphics will be introduced.

4.1 Principles of Mobile 3D Graphics

Since the computing power and power budgets of mobile devices are quite limited, mobile 3D graphics is not intended for cinematic images requiring huge computing resources; it is targeted at mid-quality graphics. However, it should still be capable of presenting

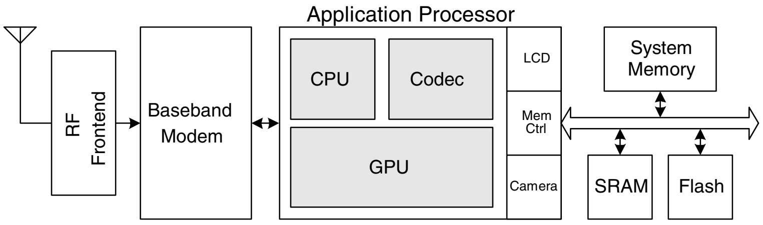


Figure 4.1 Architecture of modern cellphone systems

sufficiently elaborate 3D graphics scenes to be used for practical applications such as advertisements, the user interface, avatars, and gaming. There should be an optimal design tradeoff point between quality and hardware complexity to be within system power budgets. In this section, this tradeoff point will be explored for modern systems.

4.1.1 Application Challenges

There are several design challenges because of the limited computing power, memory bandwidth, power budgets, and footprint of mobile devices. We have analyzed the performance of 3D graphics pipelines on mobile systems with an in-house mobile graphics library, Mobile-GL, which will be covered in the following section. This library is optimized for fixed-point arithmetic on embedded CPUs used in mobile devices in which floating-point arithmetic units are not incorporated.

Figure 4.2a shows the pixel fill rate when the 3D graphics pipeline is implemented with the library running on the 400 MHz PXA-255 application processor. For QVGA (320×240) size screen, the required pixel fill rate is more than 23 Mpixels/s for 60 frames per second, but the performance level shown in the graph is far below that. So the rendering stage should be accelerated in hardware because it becomes the major performance bottleneck.

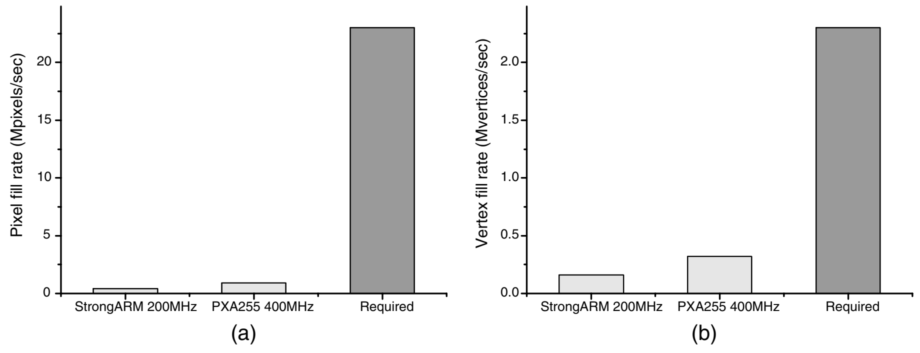


Figure 4.2 Performance graphs: (a) pixel fill rate, and (b) vertex fill rate

After the rendering stage performance issues are overcome, the geometry stage computation becomes another performance bottleneck. Since the average pixel count in a triangle is about 10 in QVGA screen resolution, the vertex fill rate should be more than 2.3 Mvertices/s. However, it is still under the required performance level as shown in Figure 4.2b. So hardware acceleration is required for both the geometry and rendering stages in mobile 3D graphics systems.

Three-dimensional graphics rendering requires large memory bandwidth support for rendering operations such as texture image mapping, depth test, and alpha blending. Currently, mobile DDR SDRAM can provide a maximum memory bandwidth up to 1 GB/s. However, the widely adopted texture mapping operation with trilinear MIPMAP filtering requires 1.1 GB/s for a 50 pixels/s pixel fill rate. Moreover, the whole memory bandwidth cannot be occupied by the rendering operations only; it has to be shared with, for example, geometry operations and other application hardware IPs operating concurrently. Therefore, memory bandwidth reduction is desirable in designing mobile 3D graphics systems.

In addition to these performance requirements, the most critical design issue is the power budget. A modern Li–ion battery can supply 2 Wh, so the system power budget for LCD, CPU, memory, and other peripherals is only about 800 mW for 2–3 hours of operation, and then only 200–300 mW can be allocated for 3D graphics processing.

Chip size has to be taken into account in hardware accelerator design. The limited footprint and cost value of mobile devices limits the die area and thereby reduces the cost per die.

There are various solutions to reducing power consumption and obtaining sufficient memory bandwidth, such as adopting finer process technology or incorporating embedded DRAM technology. However, what is required is a more cost-effective way to address the design challenges of mobile 3D graphics applications. Several design principles will be covered here.

4.1.2 Design Principles

Basically, the design of mobile 3D graphics systems is based on exploitation of the limitations of the human visual system [3–9]. This tolerates a reasonable amount of computational errors during graphics processing without any apparent image artifacts, and thereby reduces significantly the required arithmetic hardware resources significantly. Since 3D graphics applications require intensive arithmetic operations, the efficient design of arithmetic units aims to reduce their area and power consumption.

RAMP (RAM processor) architecture has been proposed in this line of approach from KAIST [4–9]. Its focus is on low-power designs to exploit the reduced complexities of arithmetic hardware and external memory bandwidth requirements. Despite its limited dynamic range, the fixed-point (FXP) number system uses simple integer arithmetic circuits that operate at higher clock frequency and lower power consumption compared with floating-point units counterpart. Therefore, FXP arithmetic units are used for rendering engine implementations, since the rendering operations are performed in screen coordinate space which has limited dynamic range. Thus, the word length and the precision are optimized for each operation in the rendering engine.

The shaders in the programmable graphics pipeline incorporate a 4-way SIMD multiply–accumulate (MAC) unit and a special function unit for reciprocal (RCP) and reciprocal square-root (RSQ) operations. Therefore, programmable precision control is required for the FXP units to be used for the different precision requirements in each shader program. The precision of the FXP units is made to be programmable from Q32.0 through Q1.31, as shown in Figure 4.3, so that the precision can be optimized for each shader program and result in low-power implementation of the programmable graphics pipeline. This FXP unit implementation for the programmable shaders showed 30% higher operating frequency and 17% less power consumption compared with the floating-point implementation.

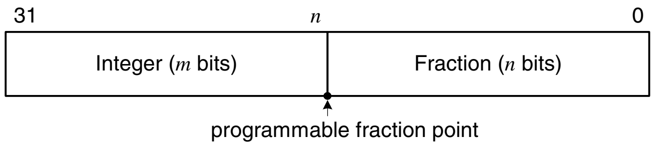


Figure 4.3 Number format of fixed-point arithmetic with programmable precision

In order to further improve the power and area efficiency, logarithmic arithmetic is adopted in the RAMP architecture. Even though the logarithmic number system (LNS) carries a certain amount of computational error, it can reduce arithmetic operation complexities significantly: multiplication, division, square-root, and power functions can be reduced to addition, subtraction, right shift, and multiplication in the LNS, respectively, as shown in Table 4.1. However, addition and subtraction become more complicated, requiring nonlinear function evaluation as shown in Table 4.1. Therefore, we adopted the hybrid approach of combining the strong points of FXP and LNS, so that additions and subtractions are performed in FXP while all the other operations are converted into LNS.

Table 4.1 Operations in ordinary and logarithmic arithmetic

Operation	Ordinary arithmetic	Logarithmic arithmetic
Multiplication	$x \times y$	$X+Y$
Division	$x \div y$	$X-Y$
Square-root	$\sqrt{x}$	$X \gg 1$
Power	x^y	$2^{y \times X}$
Addition	$x+y$	$X+\log_2(1+2^{Y-X})$
Subtraction	$x-y$	$X+\log_2(1-2^{Y-X})$

Based on this hybrid (called FXP–HNS), a power- and area-efficient multifunction unit is proposed [10]. It unifies various vector and elementary functions, including vector multiplication, multiply-and-add, division, division-by-square-root, dot-product, power, logarithm, and several trigonometric functions including sin, cos, and arctan, into a 4-way single arithmetic unit. It achieves a single-cycle throughput for all the supported operations with maximum 4-cycle latency. It shows about 50% less power consumption for the vector multiply-and-add operation than the FXP approach.

The newly defined mobile 3D graphics API, OpenGL-ES 2.0, requires floating-point operation for the vertex shaders [11]. Thus, the FXP–HNS is extended into the floating-point version, that is, a combination of FLP and LNS, which makes FLP–HNS. In this approach, the additions and subtractions are carried out in FLP while all other operations are done in LNS.

In addition to the vector and elementary functions, the multifunction unit in this FLP–HNS also unifies matrix–vector multiplication, which is used for 3D geometry transformation, with vector and elementary functions into a 4-way single arithmetic unit. It achieves a single-cycle throughput with maximum five-cycle latency for all supported operations except for the matrix-vector multiplication, which takes two cycles per result and six-cycle latency.

The memory bandwidth reduction mechanism is another major feature of the RAMP architecture. Embedded DRAM technology was employed in the RAMP series [4, 5] to exploit the higher memory bandwidth and lower power consumption by integrating the logic and DRAM on a single die. Using this technology, each DRAM macro for the frame buffer and texture memory is customized to its functionality. Although the merits for this technology are still promising, its high fabrication cost for the integration of two different technologies prevented it from being widely adopted for mobile electronic devices. Therefore, a pure DRAM technology is exploited as a cost-effective alternative in the succeeding RAMP series [6]. In this approach, logic modules are implemented using peripheral transistors of DRAM technology, which can degrade logic speed and increase the area because of the larger minimum gate length of the peripheral transistors. This approach only requires an additional single metal layer for the global signal routing in the logic modules, with a modest cost increase (Figure 4.4).

The graphics cache system is adopted as an alternative approach to reduce memory bandwidth requirements in 3D graphic systems. This approach is based on conventional pure logic process technology and it incorporates embedded SRAM to be used for the cache memory. The cache system contains the frame, depth, and texture caches for the frame buffer, depth buffer, and texture memory in the external system memory, respectively.

In addition to reduction of arithmetic complexity and memory bandwidth, the RAMP series also incorporate several power management schemes. As shown in Figure 4.5, the clock gating is fully employed throughout the RAMP series to reduce dynamic switching power. In its graphics pipeline, the depth test is moved to the

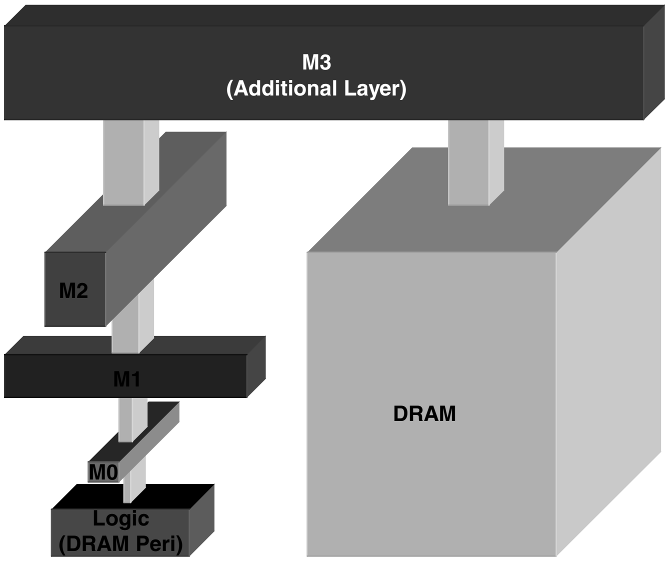


Figure 4.4 Embedded DRAM vs. DRAM-based SoC process technology

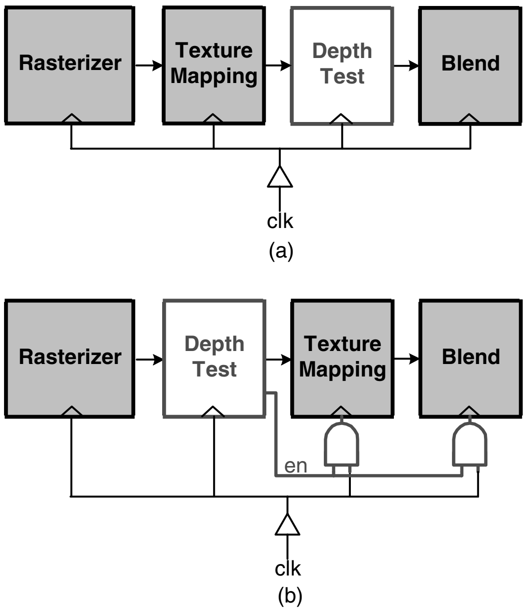


Figure 4.5 Depth reordering: (a) conventional pipeline, and (b) proposed pipeline

earlier stage of the pipeline so that the later pipeline stages can be disabled for the invisible pixels from the viewers. Therefore, the texture mapping and blending operations can be disabled for the invisible pixels, so that overall power consumption and memory bandwidth requirements are reduced.

Dynamic frequency scaling (DFS) has been adopted to optimize the power consumption for a given performance level [6, 7]. Three frequency levels are defined for fast, normal, and slow modes, as shown in Figure 4.6, and the frequency can be changed adaptively according to the required performance level.

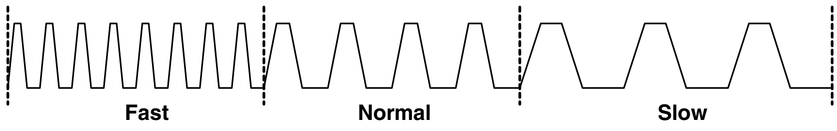


Figure 4.6 Clock frequency scaling scheme according to the mode

In addition to frequency scaling, dynamic voltage scaling has been exploited [8], which makes dynamic voltage and frequency scaling (DVFS), since the dynamic power consumption is quadratically proportional to the supply voltage. As the workloads for the application, geometry, and rendering stages are completely different in a GPU pipeline, the DVFS scheme is adopted for the multiple power domains. The GPU is divided into triple individual power domains, in which the clock frequency and supply voltage are controlled separately according to the given workload conditions.

4.2 Mobile 3D Graphics APIs

There have been efforts to establish realtime mobile 3D graphics APIs (application program interfaces) for performance-limited microprocessors. Since mobile 3D graphics requires a low-power design and is usually targeted at relatively limited applications like 3D gaming and user interfaces, there is usually a tradeoff between computational complexities and generated scene quality. Recently, several mobile 3D graphics APIs including MobileGL [12], OpenGL-ES [11], and Direct3D-Mobile [13], have been proposed in this context and will be discussed in this section.

4.2.1 KAIST MobileGL

MobileGL is the first 3D graphics API introduced to the mobile 3D graphics community. It is the best-fit to the embedded processors such as ARM cores which have limited cache memory and lack floating-point units.

MobileGL supports graphics primitives such as triangles, quads, fans, and strips in modeling. Its geometry pipeline supports culling and clipping as well as full model-view transformation and projection matrices. Ambient, diffuse, and specular lighting

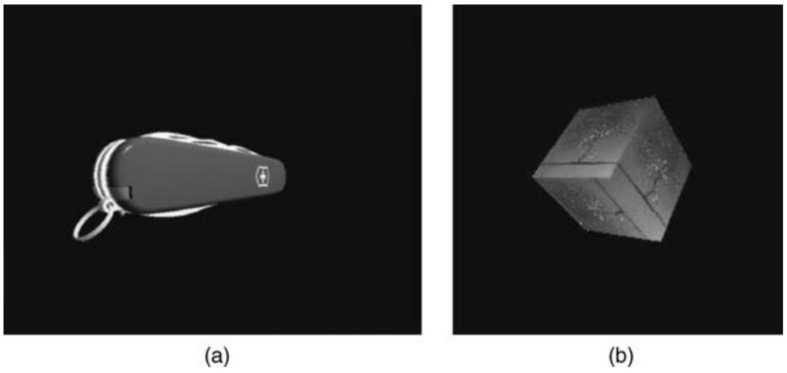


Figure 4.7 Scenes from MobileGL: (a) lighting, and (b) texture mapping

are supported to give more realistic images. The lighting can be turned on or off based on the performance conditions. Runtime switching between fixed-point and floating-point for these geometry operations is supported.

The rendering pipeline supports several per-fragment operations including texture image mapping, depth test, alpha blending, and anti-aliasing. Perspective texturing is also supported and can be turned off. The texture filtering algorithm can also be selected among the nearest neighborhood, bilinear, MIPMAP, and trilinear sampling methods.

The MobilGL implementation achieves 67 Kpolygons/s on an embedded microprocessor running at 200 MHz. It takes 483 KB for the library size and requires runtime memory of 1.14 MB. The test scenes for lighting and texture mapping operation from MobilGL are shown in Figure 4.7. The MobileGL specification is summarized in Table 4.2, and a comparison between MobileGL and the industry standard OpenGL-ES is summarized in Table 4.3.

Table 4.2 MobileGL specification

Data types	Floating-point; integer; fixed-point
Performance	67 Kpolygons/s at Xscale 200 MHz
	Full 3D graphics pipeline
Library size	483 KB
Runtime memory size	1.14 MB (including depth buffer)
Resolution	QVGA (320 × 240)
Texture type	2D
Texture image size	256 × 256
Coordinate limit (x, y)	32 bits
Number of light sources	8 independent sources
	Diffuse and specular highlighting

Table 4.3 Comparison between MobileGL and OpenGL-ES

	MobileGL	OpenGL-ES
Basic GL Operations		
Begin/end paradigm	○	○
Vertex specification	○	○
Coordinate transformation	○	○
Colors and coloring	○	○
Clipping	○	○
Rasterization		
Texturing	○	○
Anti-aliasing	○	○
Polygon	○	○
Alpha, depth	○	○
Scissor, stencil, fog	×	○
Core additions and extensions to MobileGL		
Runtime format conversion	○	×
Functions for fixed-point	○	×
Precision for fixed-point format	Variable	Q16.16

4.2.2 Khronos OpenGL-ES

OpenGL-ES is an industry-standard embedded graphics API defined by the Khronos Group. It is a subset of the OpenGL API for desktops. It is a low-level rendering API that provides a basic library of functions for specifying graphics primitives, attributes, geometric transformations, lighting, and many other rendering operations. It is a hardware-independent API and operates as a low-level interface between the software and graphics hardware devices. It includes both fixed-point and floating-point profiles.

The graphics pipeline supports both the fixed function pipeline in version 1.x and programmable pipeline in version 2.x. The first version, OpenGL-ES 1.0, was developed to provide an extremely compact API in mobile devices. It focused on enabling software-only execution of 3D graphics. Therefore, it could fully implement 3D graphics without hardware acceleration. A later version, OpenGL-ES 1.1, added more features amenable to first-generation mobile 3D graphics hardware acceleration. The OpenGL-ES 1.x versions were developed for a fixed function pipeline. Although they support floating-point arithmetic, they initially focused on fixed-point. But the latest version, OpenGL-ES 2.0, has been developed for a programmable pipeline. For that, more than 24-bit floating-point precision is required for the shader model. The shading language is also defined in version 2.0 to support high-level language programming. Basically, its syntax looks much like the C programming language. However, it has several built-in data types and operations optimized for 3D graphics processing, such as vectors, matrices, and their associated operations like matrix multiplication, dot product, normalize, and many other operations.

The primitives such as points, lines, and triangles are used for modeling geometric shapes in OpenGL-ES. The models are translated, rotated, or scaled using geometric transformation matrices and projected in an orthographic or perspective way. The ambient, diffuse, and specular lightings are supported and at least eight light sources can be used in a scene.

Based on the lighting result, color for each pixel is interpolated and blended with the texture mapping result in the rendering stages. Fragment operations like depth test, alpha blending, and anti-aliasing are carried out at the final stage of rendering.

4.2.2.1 OpenGL-ES 1.x

In creating OpenGL-ES 1.0, much functionality has been removed from the original OpenGL API and a little bit added. Two big differences between OpenGL-ES and OpenGL are the removal of the glBegin–glEnd calling semantics for primitive rendering and the adoptation of fixed-point data types for vertex coordinates and attributes. Since the embedded processors mainly consist of fixed-point datapaths, the graphics API also adopted fixed-point to enhance the computational abilities. Many other areas of functionality have been removed in version 1.0 to produce a compact API under 50 KB code size. OpenGL-ES 1.1 was developed based on OpenGL-1.5 and it had new features, to provide enhanced functionality, to improve image quality, and to optimize for increasing performance while reducing memory bandwidth usage. The features of OpenGL-ES 1.1 are listed below:

- Buffer objects – to enable efficient caching of geometry data in graphics memory;
- Enhanced texture processing – including a minimum of two multi-textures and texture combiner functionality for effects such as bump-mapping and per-pixel lighting;
- Vertex skinning functionality – to smoothly animate complex figures and geometries;
- User clip plane – for efficient early culling of hidden polygons, increasing performance and saving power;
- Enhanced point sprites – for efficient and realistic particle effects;
- Dynamic state queries – enabling OpenGL ES to be used in a layered software environment;
- Draw texture – for efficient sprites, bitmapped fonts and 2D framing elements in games.

4.2.2.2 OpenGL-ES 2.x

Vertex Processing

OpenGL-ES 2.0 supports programmable vertex and fragment shaders instead of the fixed-function pipeline to allow applications to describe operations on the vertex and fragment data. Therefore, it requires a shader compiler for the programmable vertex/fragment shader and provides APIs to compile the shader source codes or to load directly the pre-compiled shader binaries.

In comparison with OpenGL 2.0, the fixed-function vertex and fragment pipeline is no longer supported in OpenGL-ES 2.0. Replacing the fixed-function transformation pipeline, the application can calculate the necessary matrices for the model view transformation and projection and load them in the vertex shader. However, the viewport transformation and clipping are still supported as a fixed function since they are not replaced by the vertex shader. The fixed-function lighting model is no longer supported either. Instead, a user-defined lighting model can be applied by writing appropriate vertex and fragment shaders in OpenGL-ES 2.0.

Fragment Processing

OpenGL-ES 2.0 supports the programmable fragment shader in its fragment processing pipeline and replaces several fixed-function fragment processes such as texture mapping, color blend, and fog. A fragment shader is defined to be a kernel that operates on each fragment which results from the rasterization.

OpenGL-ES 2.0 adopts multi-sampling for its anti-aliasing scheme in its rasterization pipeline. In texture mapping, it supports 2D textures and cubemaps. 1D textures and depth textures are not supported, and 3D textures can be optionally supported. It supports non-power-of-two 2D textures and cubemaps as well as power-of-two textures. The cubemaps can provide functionalities icluding reflections, per-pixel lighting, and so on. The use of 3D textures is limited to only a few applications, so it is supported optionally. Copying from the framebuffer is supported for many shading algorithms. Texture compression is also supported to reduce memory space and bandwidth requirements.

Frame Buffer Operations

Among the per-fragment operations, the alpha test is not directly supported in OpenGL-ES 2.0 and it can be implemented using a fragment shader. The OpenGL-ES 2.0 requires the depth buffer since it is essential for many 3D applications. Blending is supported for implementing transparency and color sums and dithering is useful for displays in low resolution. Supporting the reading to the frame buffer is useful for some applications and testing scheme.

4.2.3 Microsoft's Direct3D-Mobile

Microsoft has developed this mobile 3D graphics API to support applications on Windows CE platforms. It is based on DirectX, the desktop 3D graphics API of the Windows operating system. Compared to OpenGL-ES, it is not portable to other platforms as it works only with the Windows platform.

The graphics pipeline of Direct3D-Mobile is similar to the OpenGL pipeline and consists of geometry transformation, lighting, rasterization, texture mapping, and fragment operations like depth test. The geometric primitives include points, lines, and triangles (triangle strip and fan), to describe 3D objects.

The geometry transformation matrices can be represented in Q16.16 fixed-point format as well as in IEEE-754 single-precision floating-point format. Three types of transformation are defined in this API: "world transformation," "view transformation," and "projection transformation."

For the lighting, Direct3D-Mobile supports diffuse and specular lightings and an unlimited number of light sources in a scene. Four types of light source are defined in this API: point, directional, spot, and ambient.

4.3 Summary and Future Directions

In this chapter we have explored the concept, challenges, and design principles of mobile 3D graphics. We have defined the basic concept and discussed design challenges presented by limited resources. Several mobile 3D graphics APIs for software solutions have been reviewed. The future of mobile 3D graphics is expected to include more programmability, such as the inclusion of geometry shaders in their pipeline in a similar way that the PC graphics pipeline has evolved. More enhanced texturing capability is expected for each programmable shader to give more realistic 3D scenes. Therefore, there would be increasing requirements for obtaining higher memory bandwidth and the optimization of the floating-point arithmetic circuits for the efficient implementation of the texture mapping units as well as the datapaths of programmable shader cores.

References

1 Viredaz, M.A. and Wallach, D.A. (2003) Power evaluation of a handheld computer. *IEEE Micro.*, **23**, 66–74.

2 Simunic, T., Benini, L., Glynn, P., and De Micheli, G. (2001) Event-driven power management. *IEEE Trans. Comput. Aided Design*, **20** (7), 840–857.

3 Crisu, D., Vassiliadis, S., Cotofana, S., and Liuha, P. (2004) Low-cost and latency embedded 3D graphics reciprocation. *Proc. of IEEE International Symposium on Circuits and Systems 2004.*

4 Park, Y.-H., Han, S.-H., Kim, J.-S. *et al.* (2000) A 7.1-GB/s low-power 3D rendering engine in 2D array embedded memory logic CMOS. *Digest Technical Papers of IEEE International Solid-State Circuits Conference 2000.*

5 Yoon, C.-W., Woo, R., Kook, J. *et al.* (2001) 80/20-MHz 160-mW multimedia processor integrated with embedded DRAM MPEG-4 accelerator 3D rendering engine for mobile applications. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2001.*

6 Woo, R., Choi, S., Sohn, J.-H. *et al.* (2003) A 210-mW graphics LSI implementing full 3D pipeline with 264-Mtexels/s texturing for mobile multimedia applications. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2003.*

7 Sohn, J.-H., Woo, J.-H., Lee, M.-W. *et al.* (2005) A 50-Mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2005.*

8 Nam, B.-G., Lee, J., Kim, K. *et al.* (2007) A 52.4-mW 3D graphics processor with 141-Mvertices/s vertex shader and three power domains of dynamic voltage and frequency scaling. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2007*.

9 Woo, J.-H., Sohn, J.-H., Kim, H. *et al.* (2007) A 152-mW mobile multimedia SoC with fully programmable 3D graphics and MEPG4/H.264/JPEG. *Digest of Technical Papers of Symposium on VLSI Circuits 2007*.

10 Nam, B.-G., Kim, H., and Yoo, H.-J. (2006) A low-power unified arithmetic unit for programmable handheld 3-D graphics systems. *Proc. of IEEE Custom Integrated Circuits Conference 2006*.

11 Khronos Group (July 2005) OpenGL 2.0. Available at http://www.khronos.org.

12 Semiconductor System Laboratory at KAIST, *MobileGL: the Standard for Embedded 3D Graphics*. Available at http://ssl.kast.ac.kr/ramp/.

13 Microsoft Corporation, *Microsoft Direct3D Mobile*. Available at http://msdn.microsoft.com/en-us/library/aa452478.aspx.

5

Mobile 3D Graphics SoC

A 3D graphics pipeline has geometry operations for computing the attributes of vertices of polygons, and rendering operations for filling colors inside the polygons. The geometry stage generates vertex data from primitive input vertex attributes using transformation, lighting, and perspective projection. The rendering stage draws pixels from the vertex data computed in the geometry stage. First, vertex data are transformed into sets of 2D triangles using interpolation to calculate edge coordinates for each polygon. Then every pixel is rendered by shading and texture mapping. Alpha blending for translucent objects and depth comparison for hidden surface removal are performed during the rendering stage.

The geometry operation is computation-intensive, and the rendering operation is data-intensive as well computation-intensive. To relieve bottlenecks, 3D graphics hardware has been evolving using fast, parallel datapaths such as multicore vector processors with 3D graphics-optimized memory systems [1, 2].

Mobile devices have limitations imposed by their power supply, their computational power, physical dimensions, and input devices. The fundamental problem is that mobile devices are powered by batteries.

With regard to the hardware, the key issue to achieve good graphics performance and low power consumption is to reduce the internal data transactions between graphics-processing unit (GPU) and memory. Many researchers have focused on how to reduce or how to compress data transactions. As a result, various mobile graphics hardware approaches have been reported, including tile-based rendering [3], specialized embedded DRAM [4], and compression algorithms. They have employed various low-power techniques such as clock gating [5], power-down schemes, and voltage/frequency scaling [5].

With regard to the software, standard application programming interfaces (APIs) for embedded systems have been released: OpenGL-ES [6] and Microsoft Direct3D-Mobile [7]. The first is a subset of desktop OpenGL and adopts optimizations and redundancy eliminations for mobile devices with low processing power, while supporting programmable 3D graphics such as vertex and pixel shading.

The goal of this chapter is to impart an understanding of various low-power techniques for mobile applications. We begin by discussing low-power architectures related to a real-time rendering system. Then, low-power techniques for vertex shaders and pixel shaders are described. The chapter concludes with an overview of a mobile unified shader, focusing on the differences from PC or console systems.

5.1 Low-power Rendering Processor

This section will present low-power architectures for a real-time rendering system. Figure 5.1 shows a block diagram of a simple rendering pipeline for PCs or console devices. The first step in the rendering system is *triangle setup*, which computes various deltas and slopes from the vertex information. In this step, three vertices form

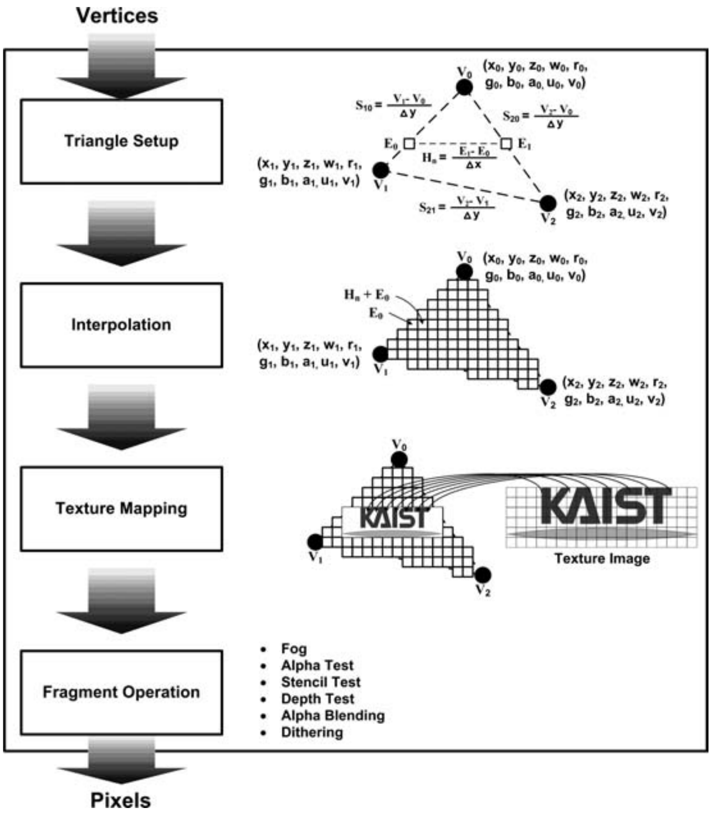

Vertices
Triangle Setup
Interpolation
Texture Mapping
Fragment Operation
Pixels
V_0 $(x_0, y_0, z_0, w_0, r_0, g_0, b_0, a_0, u_0, v_0)$
V_1 $(x_1, y_1, z_1, w_1, r_1, g_1, b_1, a_1, u_1, v_1)$
V_2 $(x_2, y_2, z_2, w_2, r_2, g_2, b_2, a_2, u_2, v_2)$
$S_{10} = \frac{V_1 - V_0}{\Delta y}$
$S_{20} = \frac{V_2 - V_0}{\Delta y}$
$S_{21} = \frac{V_2 - V_1}{\Delta y}$
$H_n = \frac{E_1 - E_0}{\Delta x}$
E_0
E_1
$H_n + E_0$
KAIST
Texture Image
Fog
Alpha Test
Stencil Test
Depth Test
Alpha Blending
Dithering

Figure 5.1 Simple rendering pipeline

a triangle and various deltas and slopes of vertex attributes such as coordinate, color, and texture coordinate are computed. The next step is *interpolation*. Using the slopes computed in the triangle setup, a fragment – which is simply data for a pixel covered by a primitive – is generated. If there is a texture associated with the primitive, then texture is applied. After that, the final *fragment operation* is performed – fog, alpha test, stencil test, depth test, alpha blending, dithering, and so on – with fragment information such as depth, alpha value, color, and so on.

5.1.1 Early Depth Test

In general, 3D graphics processors for PCs or console devices follow the standard OpenGL machine [8]. The depth test is performed in the *fragment operation* stage as shown in Figure 5.1, and the visibility of the pixels is determined at the end. This means that the processor will compute some invisible pixels, causing unnecessary computations and power consumption. To reduce this drawback, the early depth test is widely used [4]. The idea is to put the depth test into the middle of the rendering pipeline as shown Figure 5.2. The depth test is performed immediately after interpolation, and this can prevent unnecessary operations such as shading and texturing if the fragment is an invisible pixel. It also eliminates unnecessary requests to the corresponding memories. Using the early depth test result, the clock gating scheme can be applied to the texture

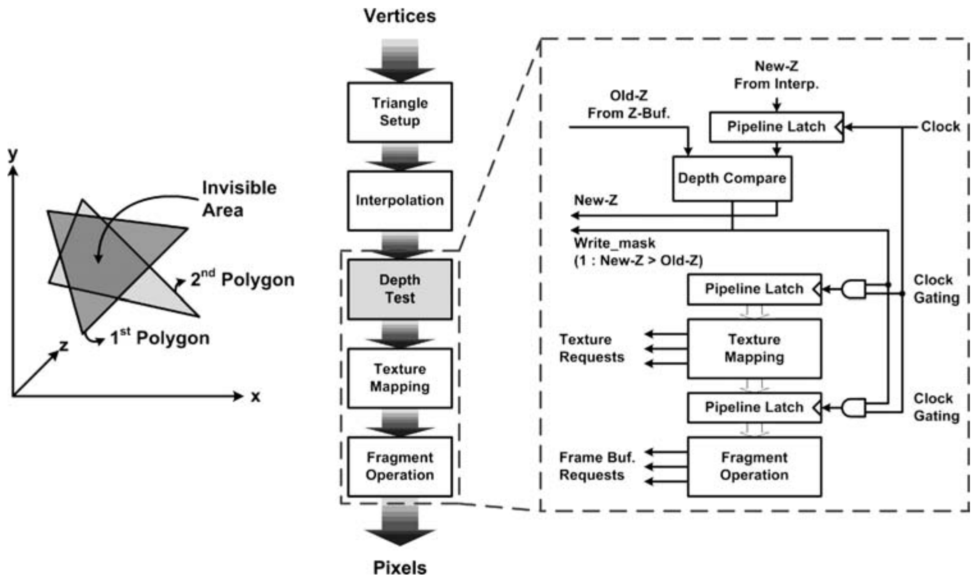


Figure 5.2 Early depth test

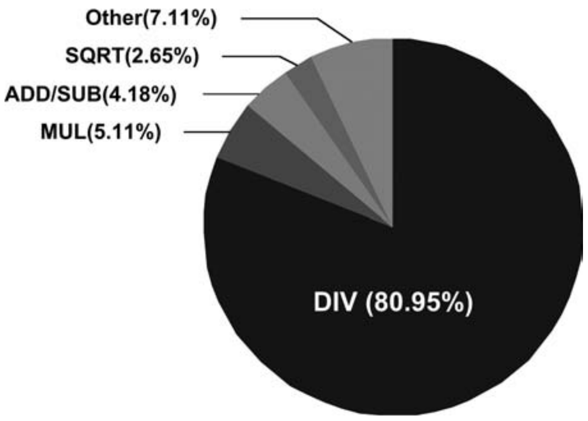


Figure 5.3 Percentage of processing time of the 3D graphics rendering pipeline

unit and fragment operation block as shown in Figure 5.3, and this reduces dynamic power consumption of the rendering processor. For typical graphics applications that have depth complexity of two, pixel-level clock gating of the rendering engine shows an average 25% power reduction.

5.1.2 Logarithmic Datapaths

Processors for 3D graphics perform more arithmetic calculations such as division, reciprocal, square-root, square, and powering operations than do general processors (Figure 5.3). It has been estimated that 83% of the processing time of a 3D graphics rendering pipeline relates to those arithmetic functions [9]. These functions consume most of the computing power because they use most of the clock cycles in realtime systems; so, to lower the power consumption, the clock cycles of these complex functions should be reduced as much as possible.

The *logarithmic* number system simplifies arithmetic computations, as shown in Table 5.1, so many researchers have tried to apply it to both generic and

Table 5.1 Operations in the logarithmic number system

Operation	Normal arithmetic	Logarithmic arithmetic (where $X = \log_2 x$ and $Y = \log_2 y$)
Multiplication	$Z = x \cdot y$	$X + Y$
Division	$Z = x/y$	$X - Y$
Reciprocal	$Z = 1/y$	$-X$
Square-root	$Z = \checkmark x$	$X \gg 2$
Reciprocal square-root	$Z = 1/\checkmark x$	$-X \gg 2$
Square	$Z = x^2$	$X \ll 2$
Powering	$Z = x^y$	$Y + \log_2 X$

application-specific processors. It is also good for mobile 3D graphics processors, especially for low power consumption, high computation speed, and small gate counts.

However, as many researchers have pointed out, one critical issue with logarithmic datapaths is errors in data conversion from ordinary to logarithmic, and vice versa. Since Mitchell introduced the binary logarithmic converting algorithm [10], linear approximation algorithms for logarithmic arithmetic have been studied to minimize errors in hardware implementation [10–12].

Since 3D graphics systems are more sensitive to conversion errors compared with conventional DSP applications, a logarithmic arithmetic unit with precise log- and antilog-conversion algorithms has been developed [13] (Figure 5.4). First the ordinary input signals are converted into logarithmic values. Then the arithmetic operations mentioned in Table 5.1 are performed. Finally the logarithmic results are converted into ordinary numbers in anti-log converters. To minimize conversion errors, an 8-region piecewise-linear interpolation approximation algorithm is used in the log- and anti-log convertors.

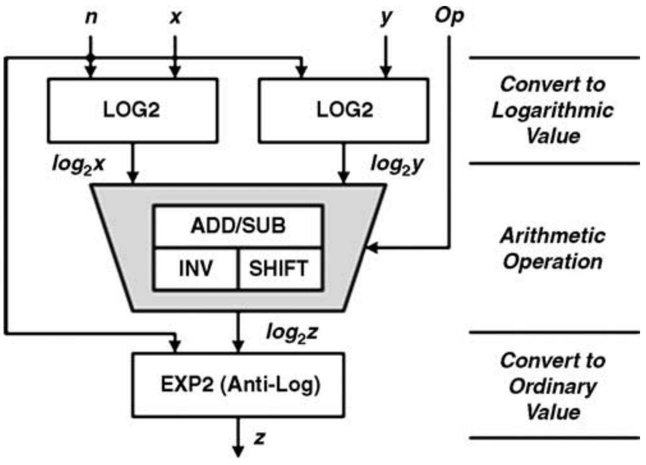


Figure 5.4 Logarithmic datapath

Figure 5.5 shows a 3D image rendered by a fixed-point number library and the logarithmic calculation. In logarithmic calculations, all operations including vertex transformation, lighting, rasterization and texture mapping are performed except for addition and subtraction. In the figure there is barely any difference between the two images discernable by the naked eye.

As mentioned earlier, the logarithmic approach is good for lower computation complexity, high computation speed, and small gate counts. Compared with a

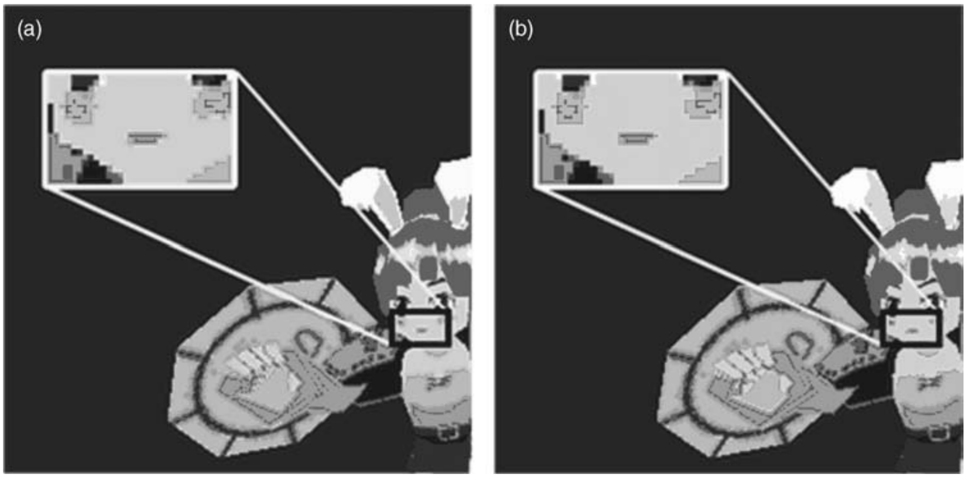


Figure 5.5 Comparison of 3D graphics results: (a) normal fixed-point calculation, and (b) logarithmic number calculation

conventional RADIX4 datapath [14], the logarithmic datapath reduces the silicon area to a quarter and the power consumption to a half, as shown in Table 5.2. Since the realtime rendering system includes huge division operations – triangle setup, rasterization, and texture address generation – the divider is a critical bottleneck. The logarithmic datapath can be a good solution for power consumption and performance for low-end rendering systems.

Table 5.2 Comparison of LAU and RADIX4 in 0.18 μm CMOS technology

	RADIX4	LAU
Gate count	44K	9K
Latency/throughput	10-cycle/8-cycle	2-cycle/1-cycle
Maximum operating frequency	60 MHz	231 MHz
Power consumption	4.29 mW	2.18 mW

5.1.3 Low-power Texture Unit

In realtime 3D graphics applications the GPU requires large memory bandwidth. To achieve this, the embedded memory system needs a wide data bus or a fast clock frequency. Fetching various data directly from embedded memory with wide I/O may consume a large amount of power owing to the concurrent data transitions in many capacitive I/Os and the activation power of memories themselves. Therefore, reducing the number of memory requests is desirable for low power consumption. One strong

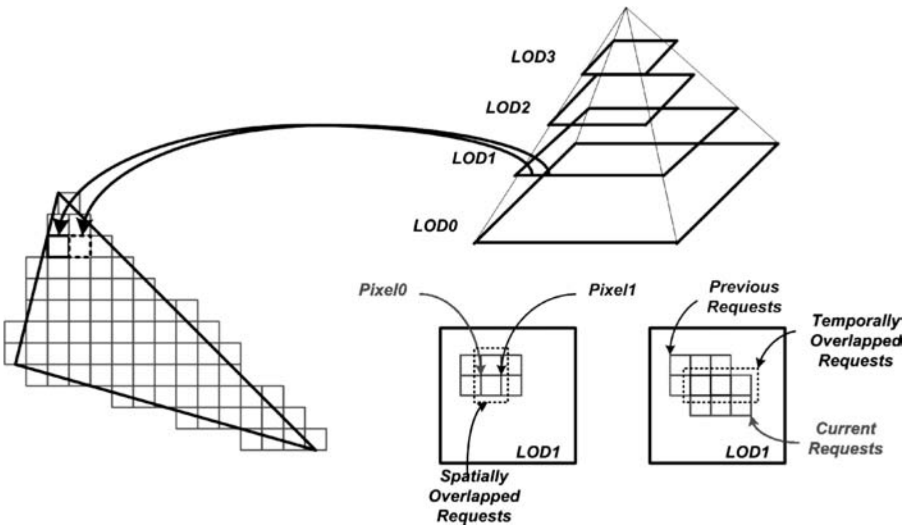
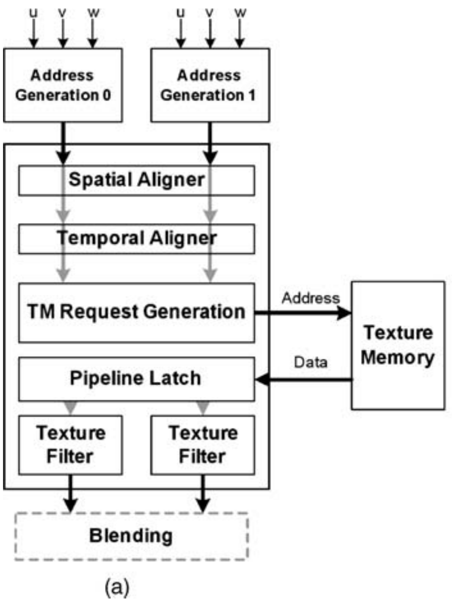


Figure 5.6 Texture requests

candidate is to exploit localities – spatial locality and temporal locality. For texture data, *address alignment logic* (AAL), which exploits the access pattern of the texture space, was introduced [4].

In bilinear MIPMAP filtering [15], a pixel uses four *texels* on a nearest level-of-detail (LOD) plane, as shown in Figure 5.6. Since two texture units generate four texture requests each, there are eight texture requests at every cycle. However, there are several requests that are overlapped because their footprints are separated by approximately one texel distance. Figure 5.7a represents the block diagram of the AAL. The spatial aligner finds and eliminates these overlapped requests, reducing the total number of requests. Then the temporal aligner compares the current texture address with previous ones and leaves only the different addresses. It stores recently used texels working with pipeline latches and comparators. The temporal aligner is basically similar to the 8-entry texture cache [16]. However, texels are simply stored in the pipeline latches instead of power-consuming SRAM. Also, the caching concept is extended to dual pixel processors. After the spatial and temporal overlapping of texels are removed, the average number of remaining requests is reduced to less than 2.3.

This AAL reduces the energy required to draw a scene, as summarized in Figure 5.7b. The average number of cycles in the four texture memoriess (TMs) with AAL is slightly increased to 1.1 owing to the memory conflict. The power consumption of the TM is proportional to the number of texture memories to be activated per cycle, as shown in Figure 5.7b. With the help of AAL, the number is reduced to 2.3 while doubling the performance compared with four TMs with 1-PP architecture. Therefore,



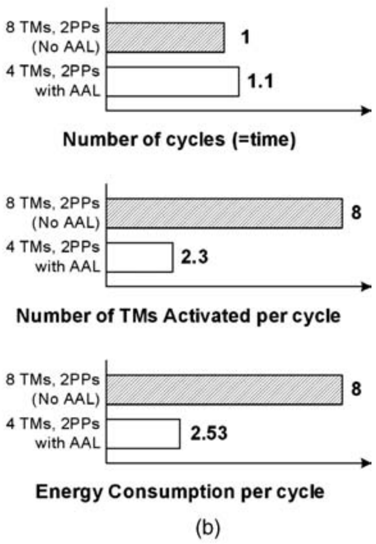


Figure 5.7 Address alignment logic: (a) block diagram, and (b) energy efficiency

the energy consumption (multiplication of time by power) required to access the texture memory can be reduced by 68% on average.

5.1.4 Tile-based Rendering

Tile-based rendering (TBR) was invented for the PC platform. PowerVR KYRO, KYRO-II, and Microsoft Talismanprocessors employed TBR, but they failed in the PC market. A few years later, however, TBR came to the forefront as a mobile GPU architecture. In 2003, PowerVR introduced MBX, a low-power mobile GPU with tile-based rendering. The conventional GPUs rendered a scene in a single pass – brute force rendering. In contrast, the basic principle of TBR is to decompose a scene into several small *tiles* and to render each one separately (Figure 5.8).

One reason for the success of TBR in mobile devices is power reduction. Since, as shown in Figure 5.9, TBR operates on a small tile of the screen, tile-based GPUs require only data for a tile, not for the whole screen. Thus tile-based GPUs reduce data transactions. Also, tile-based GPUs require a smaller and faster on-chip memory, not a complex cache system or a complex compression algorithm, to store the depth and color values for the tile being rendered, and this helps to reduce design complexity.

Another advantage of TBR is *hidden-surface removal* (HSR). Conventional GPUs first fill all the polygons without considering which ones will be hidden later on. In contrast, PowerVR's TBR technology first verifies whether it is necessary to fill

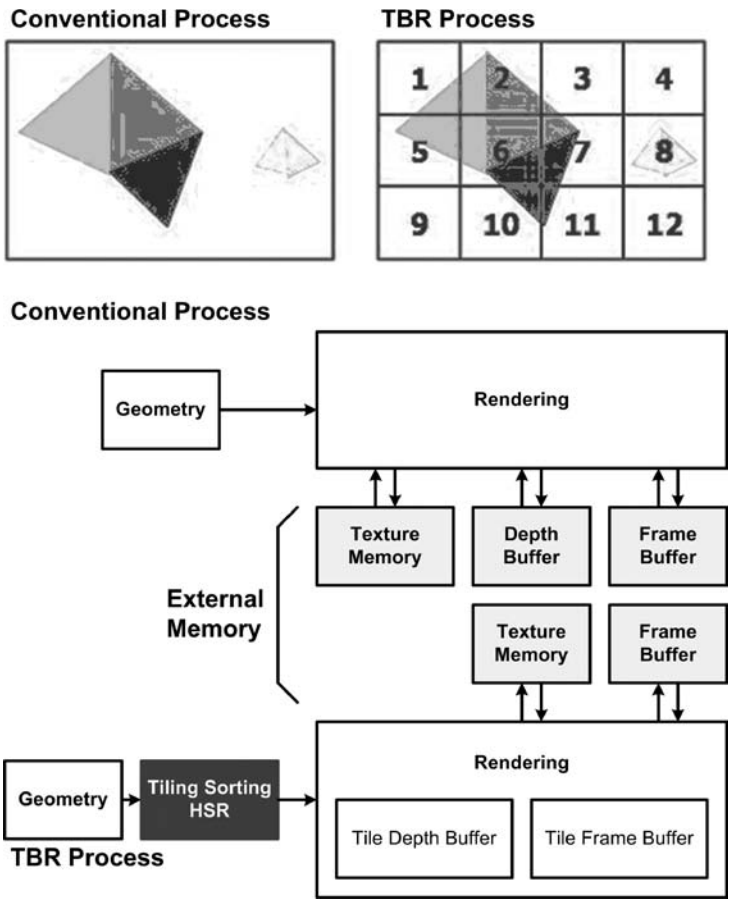


Figure 5.8 Tile-based rendering

all the polygons. Generally, only objects in the foreground require this process. Classic 3D games need to access memory constantly for the *z*-buffer and to load textures, thereby eating up valuable memory bandwidth and degrading graphics performance. PowerVR's rendering pipeline is significantly different from the classic approach. The *z*-buffer has basically been done away with, and textures are loaded only if a visible object requires them. The whole procedure is skipped for hidden objects (see Figure 5.9).

5.1.5 Texture Compression

Texture compression has been proposed to reduce memory bandwidth. The basic idea is to use compression on the texture images. The aim is to save memory bandwidth without degrading the generated image quality too much.

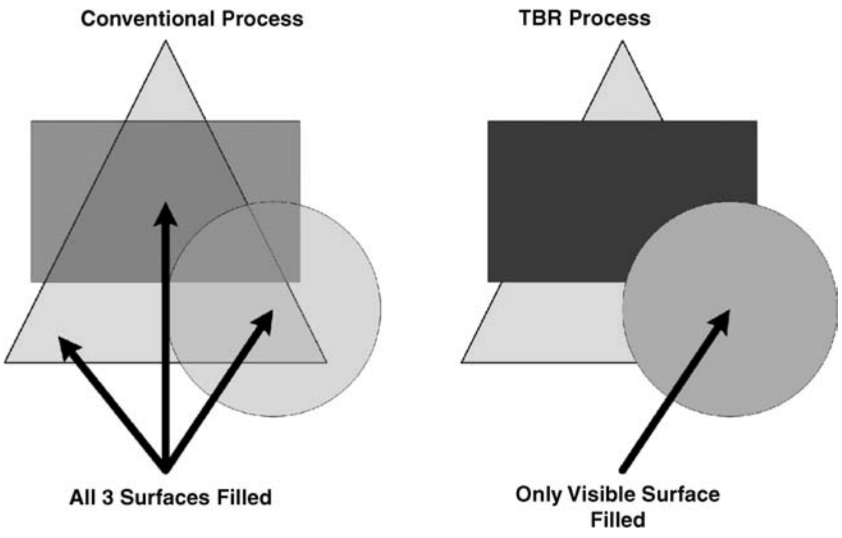
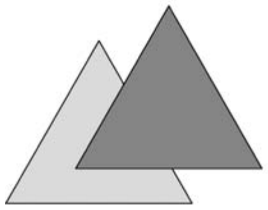


Figure 5.9 Tile-based rendering (HSR)

There are a few requirements for efficient texture compression in mobile devices. A fixed compression rate is required for straightforward address computations, and the number of indirect look-ups for the compression should be limited. Then, decompression should be fast and easy to implement in hardware to keep the pipeline latency short.



- Draw two triangles with detph test
- Each triangle is 100pixels
- 50pixels overlapped
- Each triangle has a single texture
- Color, Depth, Texture are 32bit data

	External Bandwidth Usage	
	Conventional	TBR
Texture Reads	150 x 4bytes	150 x 4bytes
Depth Reads	200 x 4bytes	0bytes
Depth Writes	150 x 4bytes	0bytes
Color Writes	150 x 4bytes	0bytes
Total	2600bytes	600bytes

Figure 5.10 Comparison of external bandwidth usage

There are currently two major texture compression schemes. The palettized scheme [17, 18] is adopted in OpenGL-ES 1.x for mobile devices. It uses a color table containing high-precision color values, and an index is used for each texel to access the corresponding color in the table. The iPackman scheme [19] exploits the fact that the human eye is more sensitive to luminance than to chrominance, so the scheme changes the luminance for every texel while varying the chrominance for multiple texels. The palettized scheme is good for images with a limited number of colors, while the iPackman scheme is good for images that have their variations mainly in luminance, with relatively uniform chrominance.

5.1.6 Texture Filtering and Anti-aliasing

"Bilinear–average mipmapping" has been proposed as a low-cost texture filtering mechanism to save memory bandwidth [17, 20]. In contrast to the trilinear mipmapping in high-end graphics systems (which requires access to the 2×2 neighborhood in two levels of the mipmap, it requires the four texels from the higher resolution mipmap level, as shown in Figure 5.11. It computes a bilinear filtering for level n, and a nearest-neighbor filtering for level $n + 1$, and then a linear filtering between them.

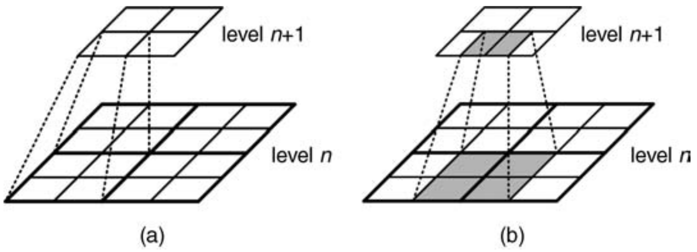


Figure 5.11 Mipmap pyramid: (a) conventional approach, and (b) bilinear–average mipmap

A low-cost anti-aliasing mechanism called FlipQuad has been proposed [21]. The scheme is based on a rotated-grid super-sampling (RGSS) pattern with the sample locations at the pixel borders, which is flipped after every other pattern (Figure 5.12).

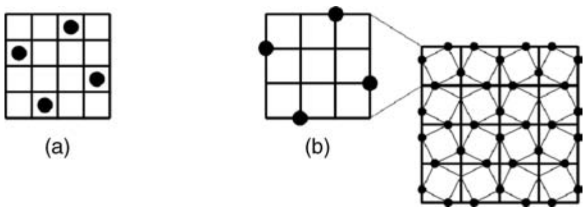


Figure 5.12 FlipQuad sampling: (a) RGSS pattern, and (b) FlipQuad pattern

This simplifies sample sharing between neighboring pixels. This sampling pattern requires only two samples per pixel.

5.2 Low-power Shader

5.2.1 Vertex Cache

Bandwidth sharing between external memories is an important design issue, because various IPs share the limited bandwidth resource. Among the IPs in a mobile multimedia SoC, the 3D graphics processor is one of the biggest bandwidth consumers owing to the large data requirements for vertex, texture, color and depth. To reduce the bandwidth demand, various techniques are employed [4, 5, 13].

A vertex cache is widely employed to reduce the bandwidth demand for input vertex attributes. The pre-TnL ("transformation and lighting") vertex cache is used to reduce data transfer between GPU and host, and a post-TnL vertex cache is used to reduce the redundant vertex processing [5, 22]. Although the concept of having pre- and post-TnL vertex caches was introduced for high-end 3D graphics accelerators on the PC platform [23, 24], it can be more effective in embedded systems for reducing power consumption.

5.2.1.1 Pre-TnL Cache

Since the primitive vertex attributes come from the external system memory through a shared bus, the pre-TnL cache is used to reduce the amount of data coming from external memory. Pre-fetching of the vertex data can be used to take advantage of the burst transfer mode of DRAM which is usually more beneficial than separate accesses [25]. In addition, the order of indexes in the index buffer (the same as the processing sequence) is not the order of vertices in the vertex buffer, as shown in Figure 5.13a. Vertices can be reused by caching the previous data. Figure 5.13b illustrates the distribution of the Δindex: according to the result in Figure 5.7b, about 81% of vertices can be reused if the pre-TnL vertex cache has 32 entries [25].

5.2.1.2 Post-TnL Cache

A series of vertices such as triangle strips or triangle fans might be rendered as shown in Figure 5.14, so that several vertices are processed repeatedly. For example, vertex 4 is processed three times. The post-TnL cache aims to improve performance by reducing such repeated operations. The cache reserves vertices results temporarily and caches the results before a new vertex operation occurs. Figure 5.15 represents the overall architecture with vertex caches. Only when both pre-TnL cache and post-TnL cache are missed does an external data fetch occur. Therefore, up to 65% of data bandwidth

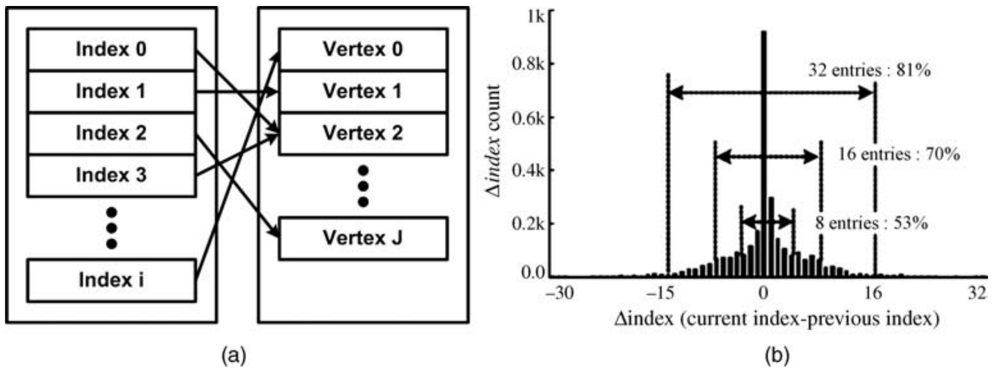


Figure 5.13 Characteristics of the indexed vertex: (a) vertex ordering, and (b) coverage of the Δindex

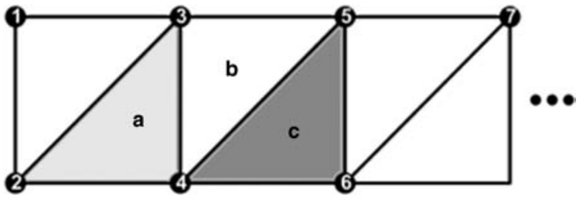


Figure 5.14 Triangle strip alignment

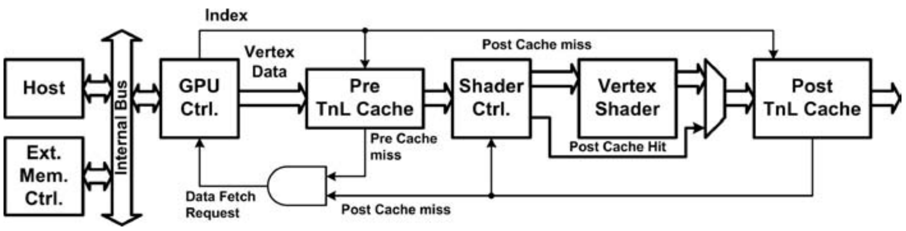


Figure 5.15 Overall architecture with vertex cache

between the host and the graphics processor can be reduced when 32 entries in pre-TnL vertex cache and 8 entries in post-TnL vertex cache are used.

5.2.2 Low-power Register File

The pixel and vertex shader uses various registers for streaming graphics processing, as explained in Chapter 3. The input register is used to hold vertex attributes such as position and normal vector, and pixel attributes such as position, color, and texture

Table 5.3 Required registers for shader model 2.0

	Vertex shader	Pixel shader
Input register	4×32 bits, 16 entries	4×32 bits, 13 entries
Output register	4×32 bits, 13 entries	4×32 bits, 5 entries
Temporary register	4×32 bits, 16 entries	4×32 bits, 16 entries
Constant register	4×32 bits, 256 entries	4×32 bits, 32 entries
Texture sample register	N/A	4×32 bits, 16 entries

coordinates. In order to reduce data fetch time, the input register consists of two register banks for double buffering. The *constant* register stores the coefficients for 3D graphics operations, and the *temporary* register is used to store temporary computational results during vertex program and pixel program execution. The modified vertex and modified pixel information are stored in the output register. Table 5.3 summarizes the required registers for vertex shader and pixel shader according to shader model 2.0. In general, since the shaders use 4-way SIMD type data, the power consumption of the register file is an important design issue. To reduce the power consumption of the register files, selective channel activation and partial activation schemes are widely used [21].

Although the register file also consists of four channels for 4-way SIMD datapaths, depending on instructions, not all channels are activated all the time:

```
MAD  O1,      I1,      T1      4- channel activation
ADD  O1, xy,  I2.xz,   T3.yw   2- channel activation
RCP  O1.x     C4.y             1- channel activation.
```

A selective channel activation scheme can reduce the dynamic power consumption of unused channels. The activated channel is selected by instruction decoding and write-mask, and an unselected channel is gated to prevent unnecessary signal transitions.

Since the constant register has more than 200 entries, generally it is designed using power-consuming SRAM. Therefore, the constant register is the most power-consuming block among the register file of the shaders. To reduce the power consumption of the constant register, a partial activation scheme can be the best solution. However, although a partial activation scheme can reduce dynamic power consumption, it increases the silicon area owing to the submodules.

Table 5.4 summarizes the characteristics of the embedded SRAM in a 0.13 µm CMOS logic process. In using a partial activation scheme, the designer should consider the area/power-reduction tradeoff.

Let us assume that the constant register consists of four 32-bit (256) entries and it is partitioned into two submodules as shown in Figure 5.16. Then, only one submodule is activated while the other modules are in standby mode. The power consumption of the constant register is reduced by 40%.

Table 5.4 SRAM characteristics in 0.13 μm CMOS processor (200 MHz, 1.2 V)

Capacity	Size (μm^2)	AC current (mA)[a]	Peak current (mA)	Standby current (mA)
32 × 16 (512 b)	316 × 124	12.5	154.1	0.002
64 × 16 (1 Kb)	559 × 124	14.9	144.8	0.002
128 × 16 (2 Kb)	559 × 134	14.9	144.8	0.002
256 × 16 (4 Kb)	563 × 154	15.0	144.8	0.002
512 × 16 (1 KB)	568 × 195	15.0	145.0	0.002
1024 × 16 (2 KB)	753 × 125	13.0	183.4	0.003
32 × 32 (1 Kb)	559 × 124	20.8	271.5	0.003
64 × 32 (2 Kb)	1044 × 124	25.5	277.0	0.003
128 × 32 (4 Kb)	1044 × 134	25.6	277.1	0.002
256 × 32 (1 KB)	1044 × 154	25.6	277.2	0.002
512 × 32 (2 KB)	1053 × 195	25.6	277.4	0.002
1024 × 32 (4 KB)	1451 × 125	24.8	354.5	0.006

[a] AC current is measured at 25% read, 25% write, 50% idle state.

5.2.3 Mobile Unified Shader

As shaders have developed, vertex shaders and pixel shaders have similar instruction set architecture and register files, except for some unique instructions such as texture sampling. Therefore a unified shader architecture, which can compute vertex shading and pixel shading with the same hardware, has been developed to reduce

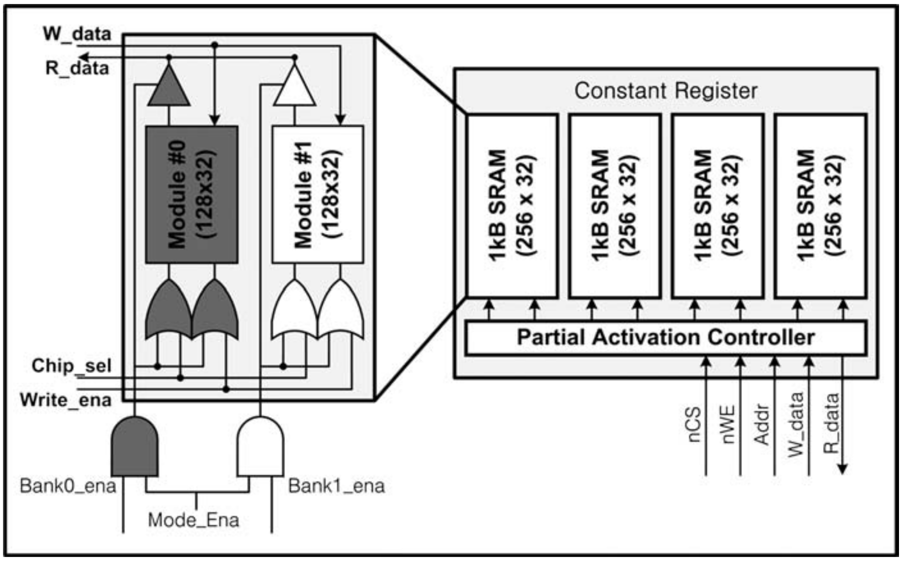


Figure 5.16 Partial activation

design complexity and turnaround time. The first unified shader was implemented in Xenos by ATI for X-Box 360 [26]. In PC and console devices, a few tens of unified shaders are integrated and controlled with multi-thread control. Therefore, programmable graphics operations can be mapped to those unified shaders dynamically in real time, and the 3D graphics processor with unified shader can utilize the hardware resources more efficiently than conventional architecture with vertex shader and pixel shader.

In the mobile environment, a fully programmable 3D graphics pipeline is required. Owing to the need for low power consumption and small area, the conventional architecture with separate vertex shader and pixel shader is hard to implement. Since a unified shader can compute vertex shading and pixel shading in a single hardware, it is a good solution for programmable 3D graphics [21]. Figure 5.17a shows the block diagram of the 3D graphics processor, in which the unified shader performs vertex and pixel shading and other blocks perform other operations of the 3D graphics pipeline such as clipping, rasterization, and blending.

The unified shader is a 4-way SIMD processor. It consists of 128 b, 4 $\times$ 32-bit SIMD datapath, a special functional unit, a texture engine, a low-power lighting engine, register files, and control logic, as shown in Figure 5.17b. The SIMD datapath is responsible for vector and matrix arithmetic operations such as addition (ADD), multiplication (MUL), and inner product (DOT), and the special functional unit is dedicated to special functional scalar operations such as logarithm (LOG), exponent

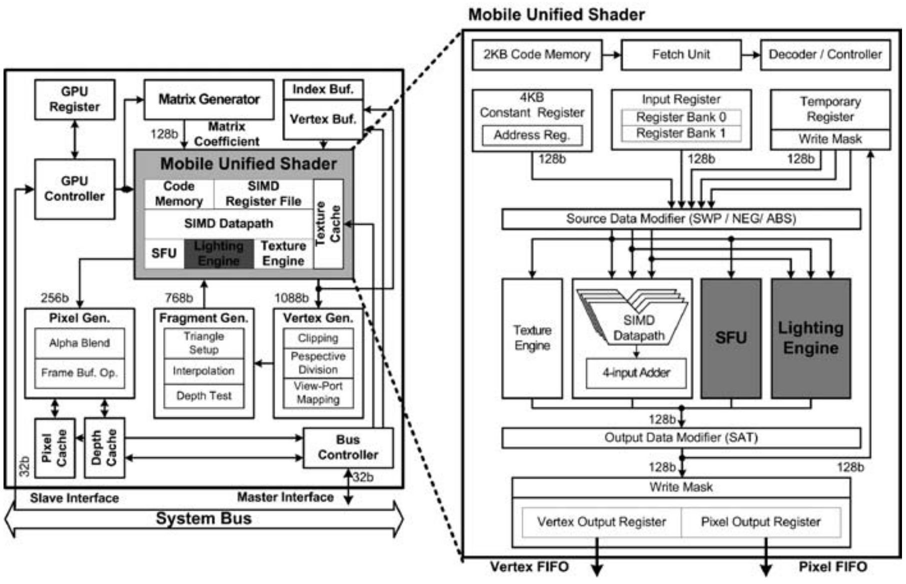


Figure 5.17 Mobile unified shader: (a) 3D graphics processor, and (b) mobile unified shader

(EXP), reciprocal (RCP), and reciprocal square-root (RSQ). The texture engine performs texture address generation, texture fetching, and texture filtering.

Since the single shader performs both vertex shading and pixel shading, task scheduling is crucial. Figure 5.18a shows a data flow diagram of the programmable 3D graphics pipeline. In conventional architecture, the primitive vertices are computed in the vertex shader for per-vertex operations such as transformation and lighting. After per-vertex operations, vertex generator and fragment generator perform clipping and

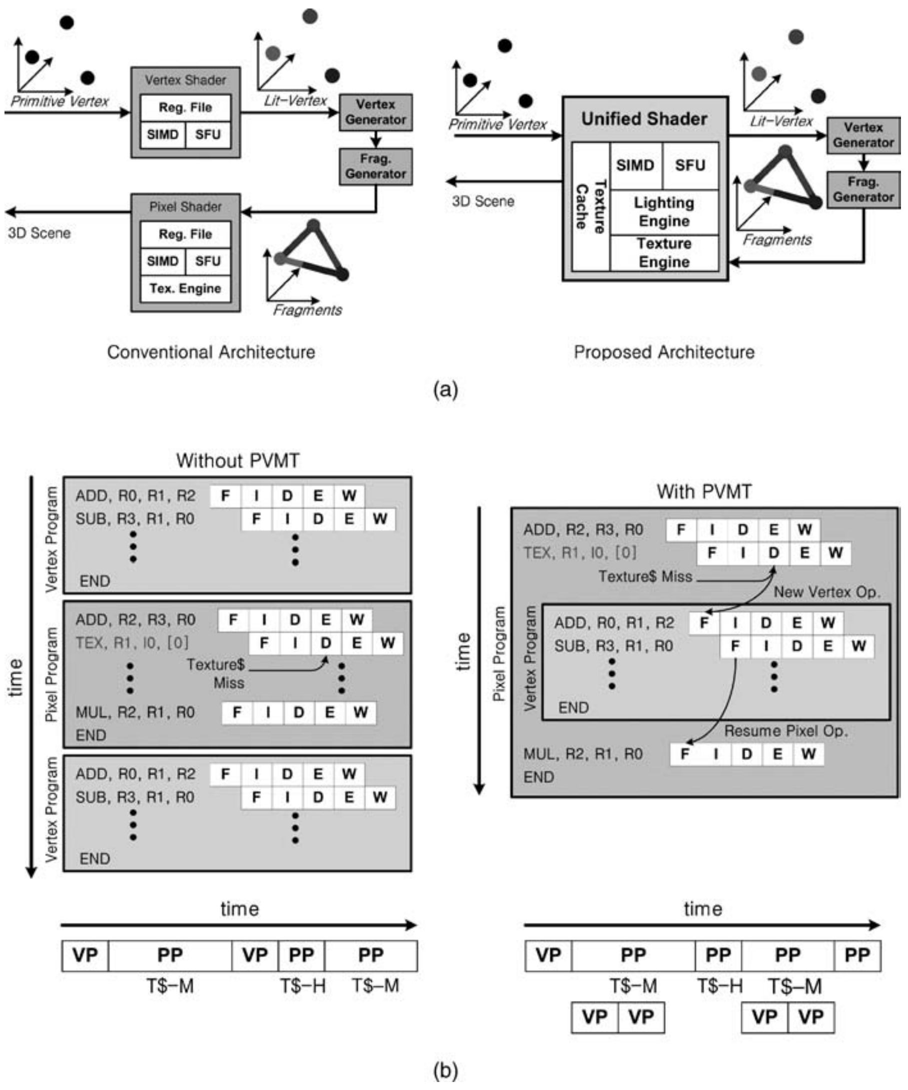


Figure 5.18 Pixel–vertex multi-threading: (a) data flow diagram, and (b) pixel–vertex multi-threading

rasterization and they generate interpolated pixels (fragments). After that, the fragments are modified in the pixel shader using per-pixel effects and blending operations generate final pixel data. In contrast to conventional architecture, the graphics data traverse the mobile unified shader twice in a single 3D graphics pipeline, as shown in Figure 5.18a, so performance is limited to less than half of its peak performance owing to task switching. To use the mobile unified shader more efficiently, pixel-vertex multithreading (PVMT) was introduced. This uses the datapaths of the mobile unified shader in parallel. Since the texture engine performs texture fetching and filtering independently with the SIMD datapath and special functional block, those datapaths are idle during the texture operations. Therefore, PVMT enables the SIMD datapath and special functional unit to compute per-vertex operations during the texture cache miss, as shown in Figure 5.18b. When the texture cache miss occurs during per-pixel operations and there are vertices to compute, PVMT issues the next vertices and the SIMD datapath and SFU perform per-vertex operations. If the texture cache filling is finished during the per-vertex operations, the mobile unified shader moves back to per-pixel operation and finalizes the remaining pixel operations. Otherwise, the PVMT issues the next vertices and performs per-vertex operations continuously.

Figure 5.19 shows the efficiency of the PVMT in a mobile multimedia SoC versus the number of vertex buffers and the average number of pixels in a polygon. While the effects of the PVMT vary with the vertex/pixel ratio, the PVMT reduces the cycles of vertex operations by at least 60%, or 94% at the most. Although the PVMT reduces the

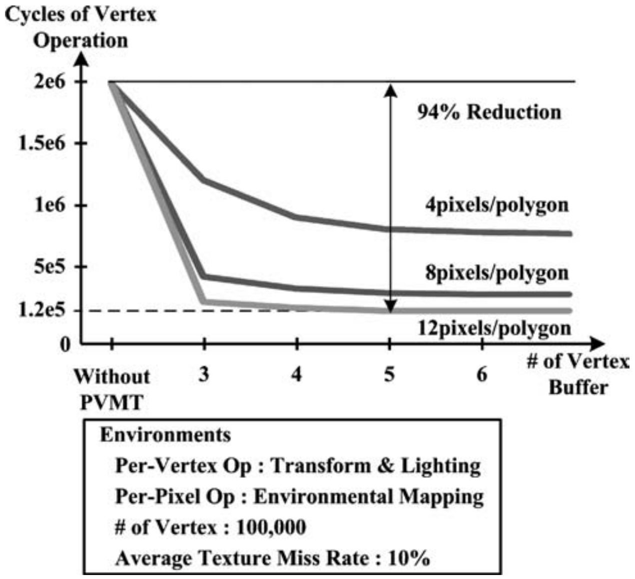


Figure 5.19 Efficiency of PVMT

cycles of vertex operations, the effect of the PVMT is bounded when the vertex buffer has five entries because the PVMT uses cycles wasted by the texture cache miss and the cycle counts are limited to around 100 cycles on average. Therefore, the 3D graphics processor that has been developed has five entries of the vertex buffer as a tradeoff between hardware cost and desired performance. By employing a PVMT with 5-entry vertex buffer, about 90% of the vertex operations are interleaved into the pixel operations and the cycle time of the vertex operations can be removed from the graphics pipeline.

References

1 Montry, J. and Moreton, H. (2005) The GeForce 6800. *IEEE Micro.*, **25** (1), 41–51.
2 Morein, Steve. (2000) ATI Radeon HyperZ Technology, ACM SIGGRAPH/EuroGraphics Graphics Hardware Workshop.
3 Imagination Technology MBX/SGX engine. Available at http://www.imgtec.com/powervr/powervr-graphics.asp.
4 Woo, R. *et al.* (2004) A 210-mW graphics LSI implementing full 3D pipeline with 264-Mtexels/s texturing for mobile multimedia applications. *IEEE J. Solid-St. Circ.*, **39** (2), 358–367.
5 Nam, B.-G. *et al.* (2007) A 52.4-mW 3D graphics processor with 141-Mvertices/s vertex shader and three power domains of dynamic voltage and frequency scaling. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2007.*
6 Khronos Group (July 2005) *OpenGL 2.0.* Available at http://www.khronos.org.
7 Microsoft Corporation, *Microsoft Direct3D Mobile.* Available at http://msdn.microsoft.com/en-us/library/aa452478.aspx.
8 OpenGL (2003) [Online]. Available at http://www.opengl.org.
9 Yosida, K., Sakamoto, T., and Hase, T. (1998) A 3D graphics library for 32-bit mocroprocessors for embedded systems. *IEEE Trans. Consum. Electron.*, **44** (4), 1107–1114.
10 Mitchell, J.N. Jr. (1962) Computer multiplication and division using binary logarithms. *IRE Trans. Electron. Comput.*, **11**, 512–517.
11 SanGregory, S.L., Siferd, R.E., Brother, C., and Gallagher, D. (1999) A fast, low-power logarithm approximation with CMOS VLSI implementation. *Proc. of IEEE Midwest Symposium on Circuits and Systems 1999*, pp. 388–391.
12 Combet, M., Zonneveld, H., and Verbeek, L. (1965) Computation of the base-two logarithm of binary numbers. *IEEE Trans. Electron. Comput.*, **14**, 863–867.
13 Kim, H. *et al.* (2006) A 231-MHz, 2.18-mW 32-bit logarithmic arithmetic unit for fixed-point 3D graphics system. *IEEE J. Solid-St. Circ.*, **41** (11), 2373–2381.
14 Sohn, J.-H. *et al.* (2005) A 50-Mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2005.*
15 Williams, L. (1983) Pyramidal parametrics. *Proc. of SIGGRAPH*, pp. 1–11.
16 Hakura, Z.S. and Gupta, A. (1997) The design and analysis of a cache architecture for texture mapping. *Proc. of 24th International Symposium on Computer Architecture*, pp. 108–120.
17 Park, Y.-H., Han, S.-H., Kim, J.-S. *et al.* (2000) A 7.1-GB/s low -power 3D rendering engine in 2D array embedded memory logic CMOS. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2000.*
18 Knittel, G., Schilling, A., Kugler, A., and Straβer, W. (1996) Hardware for superior texture performance. *Comput. Graph.*, **20**, 475–481.
19 Ström, J. and Akenine-Möller, T. (2005) iPACKMAN: high-quality, low-complexity texture compression for mobile phones. *HWWS'05: Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 63–70.
20 Akenine-Möller, T. and Ström, J. (2003) Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Trans. Graph*, **22** (3), 792–800.
21 Woo, J.-H. *et al.* (2008) A 195/152-mW mobile multimedia SoC with fully programmable 3D graphics and MPEG4/H.264/JPEG. *IEEE J. Solid-St. Circ.*, **43** (9), 2047–2056.

22 Yu, C.H. *et al.* (2007) An energy-efficient mobile vertex processor with multithread expanded VLIW architecture and vertex caches. *IEEE J. Solid-St. Circ.*, **42** (10), 2257–2269.
23 Hoppe, H. (1999) Optimization of mesh locality for transparent vertex caching. *Proc. of SIGGRAPH 1999*, pp. 269–276.
24 Bogomjakov, A. and Gotsman, C. (2001) Universal rendering sequences for transparent vertex caching of progressive meshes. *Proc. of Graphics Interface Conference 2001*, pp. 81–90.
25 ARM Corporation, *AMBA 2.0 Specification*, revision 2.0.
26 Doggett, M. (2005) Xenos: XBOX360 GPU, ACM Eurographics.

6

Real Chip Implementations

PC and console graphics hardware has been evolving rapidly from simple shading accelerator to the unified shader architecture, and mobile graphics hardware has followed that trend. However, mobile systems cannot supply unlimited power or system resources to graphics hardware, so the hardware is evolving with its own architectures for low hardware cost and low power consumption. In this chapter we will explain the design concepts and architecture of several mobile graphics hardwares. First we review the RAMP architecture developed by KAIST. We then go on to introduce commercial graphics hardwares developed by industry, looking at their key features.

6.1 KAIST RAMP Architecture

As explained in earlier chapters, rendering operations such as rasterization and texture mapping dominate the 3D graphics pipeline, and require high memory bandwidth [1]. Solving the bandwidth bottleneck with traditional approaches such as high-speed crossbars and off-chip DDR-SDRAMs can result in increased power consumption. However, the limited screen resolutions in mobile terminals (e.g., QVGA) imply that a reasonable amount of integrated memory, from a few tens of kilobytes to a few megabytes, is sufficient for graphics memories, depth buffer, frame buffer, and texture memory. In addition, by embedding all the required memory with the logic on a single die, external memory accesses are dramatically reduced, so we can develop more efficient architectures in terms of performance and power consumption.

The RAMP (RAM processor: Figure 6.1) is designed based on the design philosophy that memory is no longer a *passive* device, nor a *sub*system. The RAMP architecture uses embedded DRAM (RAMP-I [2], RAMP-II [3], and RAMP-IV) for 3D rendering in a very efficient manner that avoids connecting the memory with a large number of wires and corresponding high-speed crossbar switch. The characteristics of

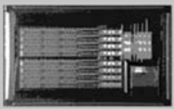
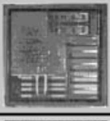
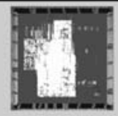
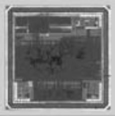
	Architecture	Process Technology	Power Consumption	Chip Area Embedded Memory	Features
Fixed 3D Graphics Pipeline	RAMP-I 	0.35μm Embedded Memory Logic	560mW	56mm^2 512kb eDRAM	Gouraud-Shading Alpha Blending Depth Comparison
	RAMP-II	0.18μm Embedded Memory Logic	160mW	84mm^2 6Mb eDRAM	
	RAMP-IV 	0.16μm Pure DRAM	210mW	121mm^2 29Mb eDRAM	+ Bilinear MIPMAP Textureing Fogging / Motion Blur Antialiasing
Programmable 3D Graphics Pipeline	RAMP-V 	0.18μm CMOS Logic	155mW	36mm^2 96kb eSRAM	Fixed-Point Vertex Shader Dual-Operations
	RAMP-VI 	0.18μm CMOS Logic	153mW Peak 52.4mW Avg. (60fps)	25mm^2 29kb eSRAM	Logarithmic Number Vertex Shader DVFS for 3D Graphics Pipeline
	RAMP-VII	0.13μm CMOS Logic	195mW	41mm^2 128kB eSRAM	Mobile Unified Shader Pixel-Vertex Multi-Threading Logarithmic Lighting Engine

Figure 6.1 Summary of RAMP architectures

3D rendering operations are exploited to distribute the memory accesses in time, hence reducing the power consumption by activating one or some of the memories locally. The design of the embedded DRAM is specified in terms of the latency, throughput, number of buses, and commands of each DRAM block. The logic pipelines are tuned to the modified timing and functions of the DRAMs. Various low-power techniques such as clock gating are used extensively inside the memory and logic.

6.1.1 RAMP-IV

In 2002, KAIST released RAMP-IV [4]. This was the first graphics processor to implement texture mapping in mobile devices. It focuses more on realtime 3D gaming applications, drawing bilinear MIPMAP texture-mapped pixels with special rendering effects such as fogging and motion blur at 66 Mpixels/s and 264 Mtexels/s, as well as supporting the shading operations. Figure 6.2 illustrates the block diagram of the SlimShader, the rendering processor in RAMP-IV.

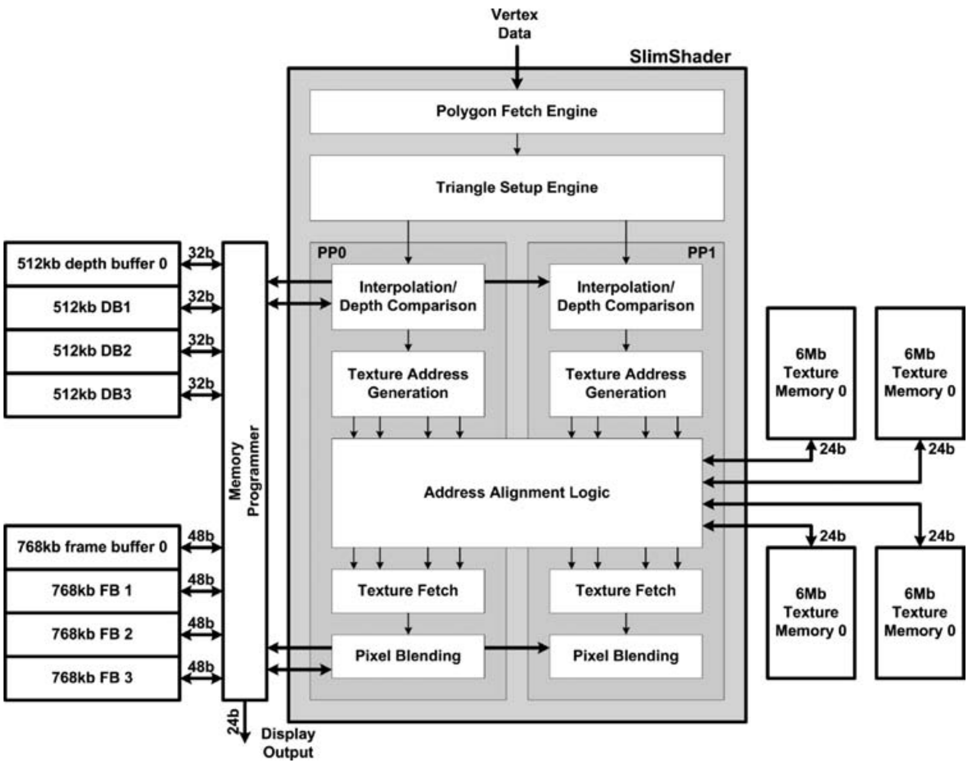


Figure 6.2 SlimShader architecture

It consists of a hardwired triangle setup engine, an edge-processing (EP) block, two pixel processors (PPs), and 29 Mb of embedded DRAM. To create realistic 3D images in real time, the SlimShader contains two energy-efficient texture engines (TEs), which are in charge of texture address generation, texture fetching, and texture filtering. One of the difficulties in implementing texture mapping in mobile devices is power consumption. In bilinear filtering [5], two pixels to texel space require eight texture memory requests every cycle, and those cause huge power consumption. To reduce the power consumption, TEs employ address alignment logic (AAL), introduced in Chapter 5, which uses temporal and spatial localities of texture addresses in MIPMAP filtering to reduce total memory requests, thus yielding power savings.

For realtime special effects such as fog, anti-aliasing, and cartoon shading, a memory programmer is implemented in SlimShader and post-processes the rendered pixels of the frame buffer by using a dedicated instruction set and its SIMD datapath.

6.1.1.1 Embedded Memory

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 20 ns, while commodity SDRAMs work at 65 ns 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 more 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 the PP pipeline, the following stages and corresponding memories can be gated off according to the comparison result.

6.1.1.2 Implementation

The SlimShader architecture was integrated into a RAMP-IV chip (Figure 6.3) together with an ARM9-compatible RISC processor with enhanced multiply-and-accumulate (MAC), 29 Mb of embedded DRAM, and a power management unit. The chip was fabricated using a 0.16 μm Hynix 256 Mb SDRAM process. Its area and power consumption were 121 mm^2 and 210 mW. The RAMP-IV chip utilizes a pure DRAM process to reduce the fabrication cost. Although the pure DRAM process has

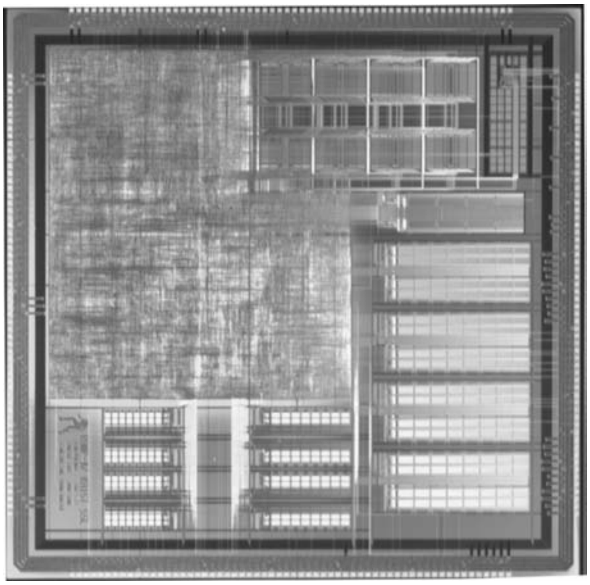


Figure 6.3 RAMP-IV chip micro-photo

slower logic transistor speed and fewer metal layers, 133 MHz speed can be achieved in the chip's RISC processor. The negligible sub-threshold leakage current of the DRAM process also reduces standby current, which is a critical issue for battery-driven devices. Since modern SoC design prefers the standard CMOS logic process to a DRAM-based CMOS process, the application of SlimShader architecture to a CMOS logic process is also being completed for future integration into next-generation graphics processors, while improving the scalability of memory capacity.

6.1.2 RAMP-V

As mobile 3D graphics becomes more popular, the trend in research is moving towards development of high-quality flexible graphics architectures capable of providing more realistic images for handheld devices, using advanced graphics algorithms. To satisfy this requirement, the RAMP-V was developed in 2005 [6–8]. RAMP-V, shown in Figure 6.4, was the first mobile graphics processor with a programmable vertex shader and low-power rendering processor in a single chip.

RAMP-V consists of an ARM10-compatible 32-bit RISC processor with 16 kB I/D caches, a 128-bit programmable fixed-point SIMD vertex shader, a low-power

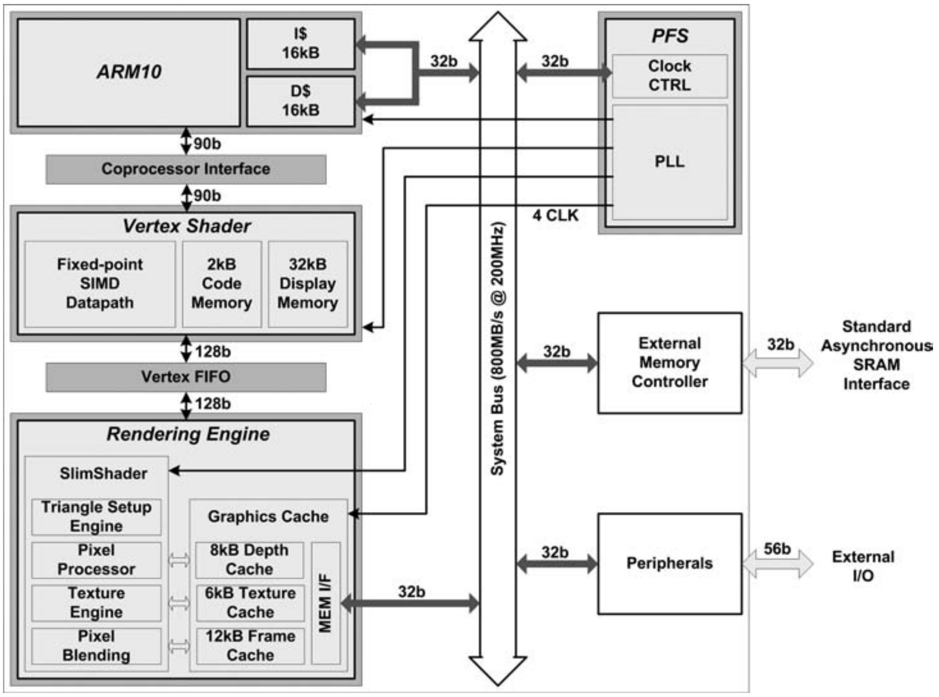


Figure 6.4 RAMP-V architecture

rendering engine – using the same architecture as RAMP-IV – and a programmable frequency synthesizer (PFS). RAMP-V features a fixed-point programmable vertex shading architecture called SATINE. The vertex shader is implemented as an ARM10 *coprocessor* and processes all per-vertex operations such as geometry transformations and lighting calculations. Primitive assembly methods such as clipping and culling are also performed by the vertex shader in collaboration with the ARM10 processor.

6.1.2.1 SATINE Architecture

Figure 6.5 shows the internal architecture of the SATINE vertex shader. The vertex program control unit (VPCTRL) issues the graphics instructions independently of the ARM10 processor for vertex shading. SATINE can also execute general-purpose integer and fixed-point SIMD instructions controlled via the coprocessor interface in order to implement various multimedia operations beyond 3D graphics, such as MPEG4 video decoding. A 32 kB display buffer integrated into the SATINE vertex shader decreases the system-bus bandwidth requirements when used in vertex array implementation and indexed primitive drawing. It stores vertex model data as well as graphics parameters such as matrix and light coefficients.

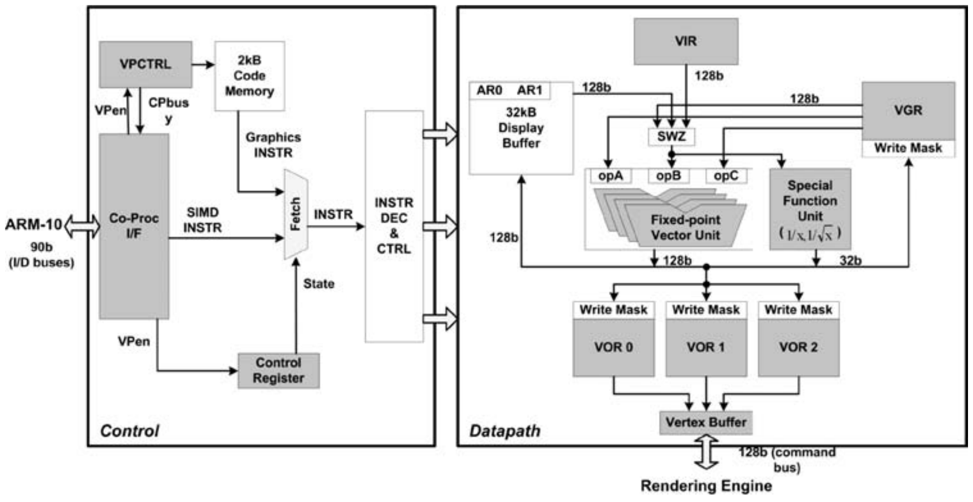


Figure 6.5 SATINE architecture

6.1.2.2 Dual Operations

SATINE implements dual operations (Figure 6.6). Unlike a conventional ARM coprocessor architecture, the SATINE vertex shader has dual operating states to allow it to be better adapted to the parallelism inherent in graphics processing.

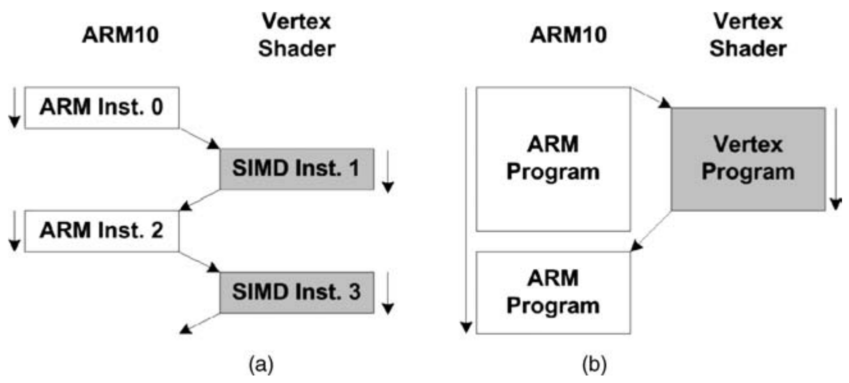


Figure 6.6 Dual operations: (a) TCC state, and (b) PP state

1. **Tightly coupled coprocessor (TCC) state.** In this state, SATINE is a normal ARM10 coprocessor. The instructions of the coprocessor are issued in the instruction stream of the main processor as extended coprocessor instructions, and they are executed in lock step with the pipeline of the main processor. The TCC state implements integer and fixed-point SIMD data-processing instructions, and all instructions can be executed conditionally like conventional ARM instructions.
2. **Parallel processor (PP) state.** In this state, SATINE is an independent processor and can operate independently of the ARM10 processor. The PP state has a graphics instruction set that is separate from the general SIMD instructions of the TCC state. SATINE executes the independent vertex program while the ARM10 processor performs the main application program or enters a cache miss state. In the programmer's model, the graphics instructions set is a subset of the general SIMD instructions with graphics extensions, such as source swizzling and writemasks. In the PP state, there are more register file sets that can be used as input operands of instructions. SATINE maintains the communication protocol of the ARM10 coprocessor interface by driving the coprocessor busy signal to the ARM10 processor, allowing the next coprocessor instruction from the ARM10 processor to be blocked for synchronization.

The two operating states share all the hardware blocks except the instruction fetch units. Dual operations enable a single hardware resource to perform various multimedia operations. In addition, RAMP-V can process streaming graphics data more efficiently because various user-defined vertex processing can be performed for the current vertex input during the next vertex fetch of the ARM10 processor.

6.1.2.3 Fixed-point SIMD Processing

Most mobile 3D graphics applications require real number representation to support various graphics algorithms. For this, fixed-point number representation shown in Figure 6.7 is used instead of floating-point number format.

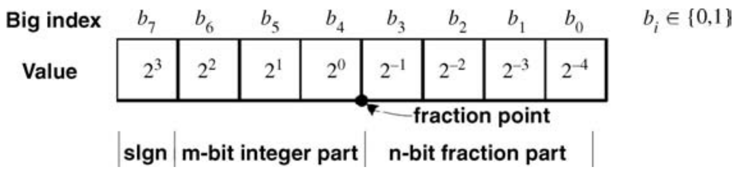


Figure 6.7 Fixed-point representation

The simple integer datapath of a fixed-point unit can achieve higher clock frequency while consuming less power than a floating-point unit, yielding total energy reduction. For typical 3D matrix transformation, gate-level simulation of a four-stage pipelined 32-bit fixed-point multiplier showed 30% higher maximum operating frequency than a six-stage pipelined single-precision floating-point multiplier. In addition, the fixed-point multiplier consumed only 83% of the power of the floating-point multiplier at the same operating frequency. Consequently, when fixed-point arithmetic is applied to graphics applications, 36% of total energy consumption can be saved on average (Figure 6.8).

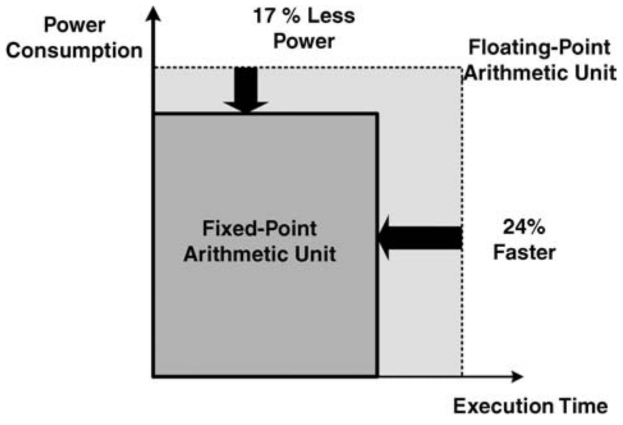


Figure 6.8 Energy saving by fixed-point arithmetic, measured during transformation operations

To evaluate the accuracy of fixed-point arithmetic in 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 [9], where m is the number of bits representing the integer part and n is the number of bits representing the fractional part.

For transformation:

$$n_\mathrm{f} = n_\mathrm{a} + 3 + \log_2\left[1 + \frac{\text{distance of far plane from eye}}{\text{distance of scene vertex to eye}}\right]; \qquad (6.1)$$

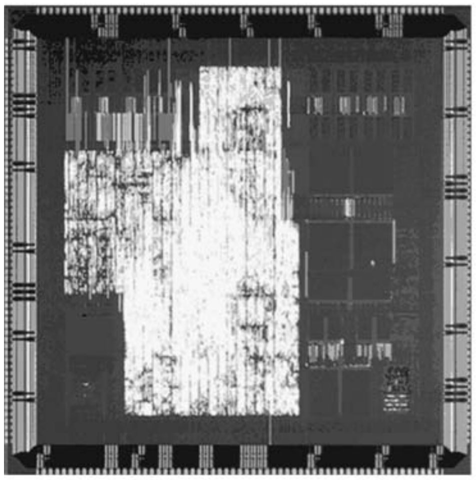
and *for lighting*:

$$n_f = n_a + 8 \quad \text{or} \quad n_a + 9; \tag{6.2}$$

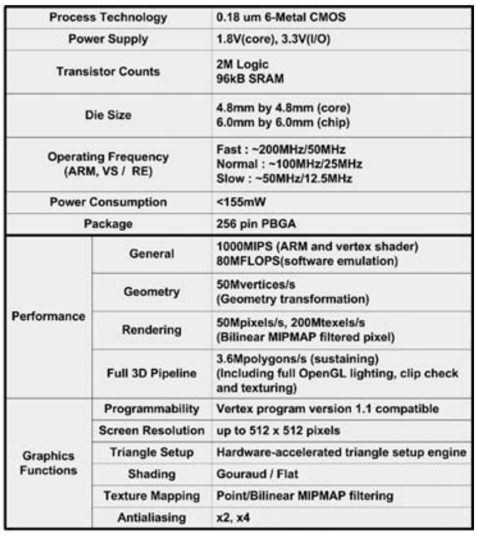
where n_a is the number of bits required for securing the accuracy in transformation and lighting calculations.

6.1.2.4 Implementation

RAMP-V is fabricated in a 0.18 μm six-metal standard CMOS logic process. The chip size is 36 mm including 2 M logic transistors and 96 kB of SRAM. Figure 6.9 shows the die photograph and summarizes its features. By using this chip, various 3D graphics algorithms and other multimedia functions can be processed with 50 Mvertices/s peak graphics performance, and 24-bit true-colored and texture-mapped graphics images can be drawn at the speed of 50 Mpixels/s and 200 Mtexels/s.



(a)



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

(b)

Figure 6.9 RAMP-V implementation results

6.1.3 RAMP-VI

The overall architecture of RAMP-VI is shown in Figure 6.10 [10, 11]. It consists of a RISC processor, a vertex shader, and a rendering engine. The RISC processor controls the entire GPU system and runs artificial intelligence and collision detection for 3D gaming. Logarithmic arithmetic [12] is exploited in this design for reduced arithmetic complexity. The GPU is divided into three independent power domains to optimize power consumption for a given performance level.

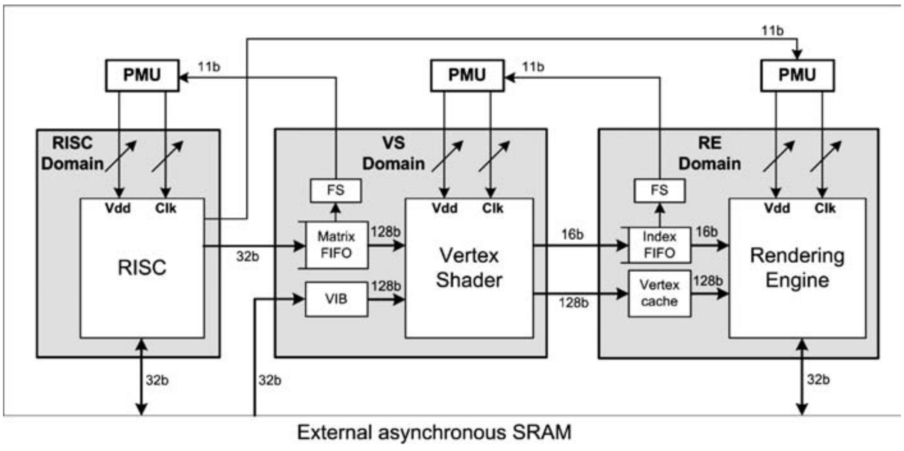


Figure 6.10 Proposed handheld GPU

6.1.3.1 Vertex Shader

The vertex shader is described in Figure 6.11. It includes vector register files, multifunction unit, and vertex cache. The vertex cache is included to reuse the previously processed vertices. The multifunction unit unifies the matrix, vector, and elementary functions in a single four-way arithmetic unit shown in Figure 6.12.

The logarithm converter (LOGC) and antilogarithm converter (ALOGC) use piecewise-linear interpolation for approximation of their nonlinear function evaluations. These number converters are used for conversion between floating-point and logarithmic numbers. This unit includes a programmable multiplier (PMUL) that can be programmed into a Booth multiplier, LOGCs, or ALOGCs for the target operations to be implemented. This is done by just adding 15-entry LOG and 8-entry ALOG lookup tables (LUTs) to the Booth multiplier and sharing the common adder tree. It also includes a programmable adder (PADD) that can be programmed into a single 5-input adder tree or 4-way 2-input SIMD adders for target operations. Using these components, various operations are implemented in this multifunction unit as follows.

6.1.3.2 Matrix–Vector Multiplication

The 3D geometry transformation is computed by multiplication of a 4×4 matrix and a 4-element vector, which requires sixteen multiplications and twelve additions. This can be converted into an operation requiring twenty LOGCs, sixteen adders, sixteen ALOGCs, and twelve FLP adders by exploiting the logarithmic arithmetic as follows:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} c_{00} \\ c_{10} \\ c_{20} \\ c_{30} \end{bmatrix} x_0 + \begin{bmatrix} c_{01} \\ c_{11} \\ c_{21} \\ c_{31} \end{bmatrix} x_1 + \begin{bmatrix} c_{02} \\ c_{12} \\ c_{22} \\ c_{32} \end{bmatrix} x_2 + \begin{bmatrix} c_{03} \\ c_{13} \\ c_{23} \\ c_{33} \end{bmatrix} x_3 =$$

$$\underbrace{\underbrace{2^{\left(\begin{bmatrix} \log_2 c_{00} \\ \log_2 c_{10} \\ \log_2 c_{20} \\ \log_2 c_{30} \end{bmatrix} + \log_2 x_0\right)} + 2^{\left(\begin{bmatrix} \log_2 c_{01} \\ \log_2 c_{11} \\ \log_2 c_{21} \\ \log_2 c_{31} \end{bmatrix} + \log_2 x_1\right)}}_{\textbf{MAT 1st phase}} + 2^{\left(\begin{bmatrix} \log_2 c_{02} \\ \log_2 c_{12} \\ \log_2 c_{22} \\ \log_2 c_{32} \end{bmatrix} + \log_2 x_2\right)} + 2^{\left(\begin{bmatrix} \log_2 c_{03} \\ \log_2 c_{13} \\ \log_2 c_{23} \\ \log_2 c_{33} \end{bmatrix} + \log_2 x_3\right)}}_{\textbf{MAT 2nd phase}} \tag{6.3}$$

Taking note that the coefficients of a geometry transformation matrix are fixed during processing of a 3D object, these can be pre-converted into the logarithmic domain and be regarded as constants during the processing. Therefore, the required number of LOGCs for matrix–vector multiplication (MAT) can be reduced from twenty

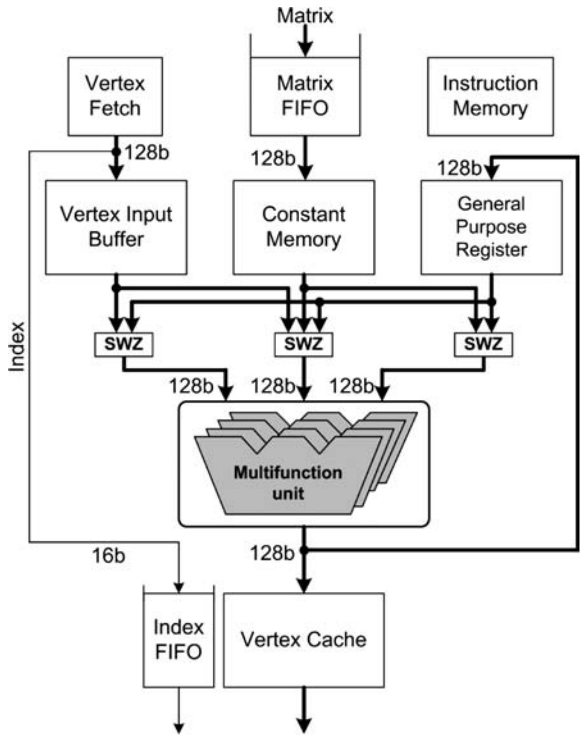


Figure 6.11 Vertex shader

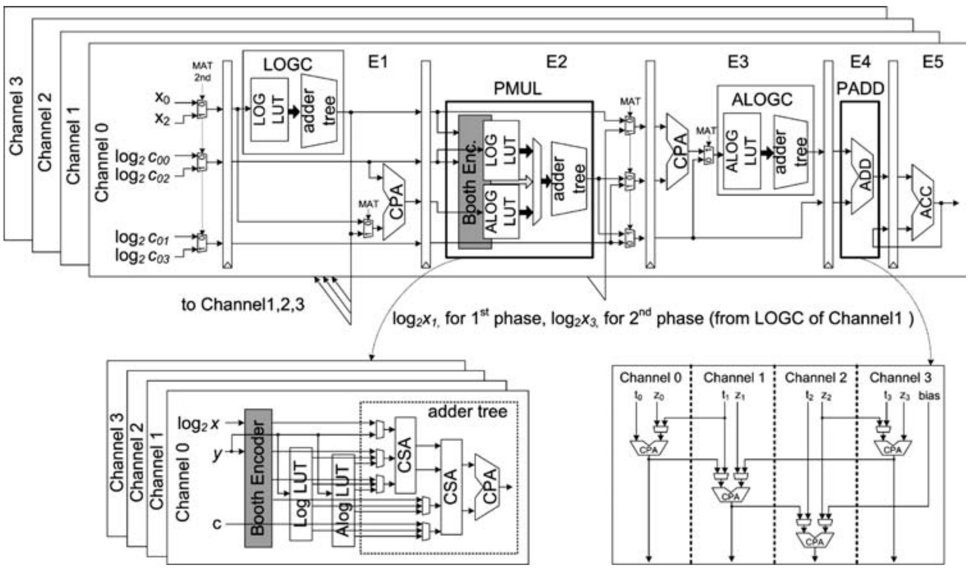


Figure 6.12 Multifunction unit

to four. This can be implemented in two phases on the proposed 4-way arithmetic unit as illustrated in Figure 6.12, requiring eight adders and eight ALOGCs per phase. The eight ALOGCs are obtained by programming the PMUL into four ALOGCs together with four ALOGCs in E3. The carry propagate adders (CPAs) in E1 and E3 stages are used for the required eight adders. The four multiplication results from the ALOGCs in E2 stage and the other four from the E3 stage are added by programming the PADD into a 4-way SIMD FLP adder to get the first phase result. With the same process repeated, accumulation with the first phase result in E5 completes the MAT. Thus, the MAT produces one result every two cycles on this 4-way arithmetic unit, whereas it would have taken four cycles per result in the conventional way [4].

6.1.3.3 Vector Operations

The vector SIMD operations such as vector multiplication, division, square-root, and multiply–add (MAD) can be represented as a single generic operation, which can be converted into one using logarithmic arithmetic as follows:

$$\begin{gathered}\left(x_i \otimes y_i^s \oplus z_i\right)_{i\in\{0,1,2,3\}} = \left(2^{(\log_2 x_i)\oplus(s\times\log_2 y_i)} \oplus z_i\right)_{i\in\{0,1,2,3\}} \\ \text{where } \otimes \in \{\times,\div\}, \oplus \in \{+,-\}, s \in \{0.5, 1\}.\end{gathered} \tag{6.4}$$

Since we require two LOGCs per channel – that is, eight LOGCs for four channels – the PMUL is programmed into four LOGCs to make the eight LOGCs together with the four LOGCs in the E1 stage. Both the vector MAD and dot product (DOT) require vector element multiplication and final summation. Thus, the PADD is programmed into a 4-way 2-input SIMD adder for the vector MAD and a single 5-input adder tree for the DOT.

6.1.3.4 Elementary Functions

In general, elementary functions can be represented as a power series, such as the Taylor series expansion. For the first five terms of the Taylor series expansion, a new generic operation is defined and converted into one exploiting logarithmic arithmetic:

$$\begin{aligned}
&c_0x^{k_0} \oplus c_1x^{k_1} \oplus c_2x^{k_2} \oplus c_3x^{k_3} \oplus c_4x^{k_4}\\
&= c_0x^{k_0} \oplus 2^{\log_2 c_1 + k_1 \times \log_2 x} \oplus 2^{\log_2 c_2 + k_2 \times \log_2 x} \oplus 2^{\log_2 c_3 + k_3 \times \log_2 x} \oplus 2^{\log_2 c_4 + k_4 \times \log_2 x}\\
&\text{where } \oplus \in \{+,-\}, c_i, \log_2 c_i, k_i \text{ are coefficients.}
\end{aligned} \tag{6.5}$$

Since the power function is converted into multiplication in the logarithmic domain, and k_i is a small integer, the power functions can be implemented with a 4-way 32-bit $\times$ 6-bit multiplier in the logarithmic domain. The first term does not need to be converted into the logarithmic domain since the first term of the Taylor series is usually a constant or the input x that can be fed directly into the "bias" port. Therefore, the generic power series can be implemented by programming the PMUL into 4-way 32-bit $\times$ 6-bit BMUL and the PADD into a single 5-input adder tree.

6.1.3.5 Power Management

In this GPU, triple power domains are tuned with separate frequencies and supply voltage by tracking the workloads. Since the objects in a scene are composed of a number of triangles and these are again composed of a number of pixels, the workloads of the RISC, VS, and RE – which operate on objects, triangle, and pixels, respectively – can be different completely. Therefore, the RISC, VS, and RE are divided into different power domains and their frequencies and supply voltages are controlled separately according to their workloads. The workload for each domain is measured from the occupation level of the FIFO and the level is compared with the reference level so that the PMU determines target clock frequency and supply voltage. Figure 6.13 illustrates this power management scheme.

6.1.3.6 Chip Implementation

RamP-VI is fabricated into a chip using 0.18 μm six-metal CMOS technology [10]. Figure 6.14 shows the chip micrograph. It integrates the RISC, VS, and RE into a small area of 17.2 mm^2 containing 1.57 M transistors and 29 kB of SRAM. It shows 141 Mvertices/s of peak performance by exploiting logarithmic arithmetic. This chip dissipates only 52.4 mW when the scenes are drawn at 60 frames per second by using the triple-domain power management scheme.

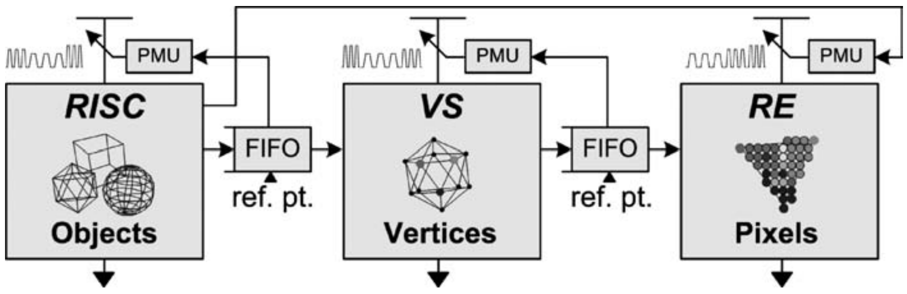


Figure 6.13 Triple-domain DVFS scheme

6.1.4 RAMP-VII

RAMP-VII is a low-power processor that is fully programmable for vertex shading and pixel shading, including a mobile unified shader architecture. In PC and console platforms, 3D images can be generated using programmable vertex shading and programmable pixel shading, but the most recent employ a unified shader architecture [13]

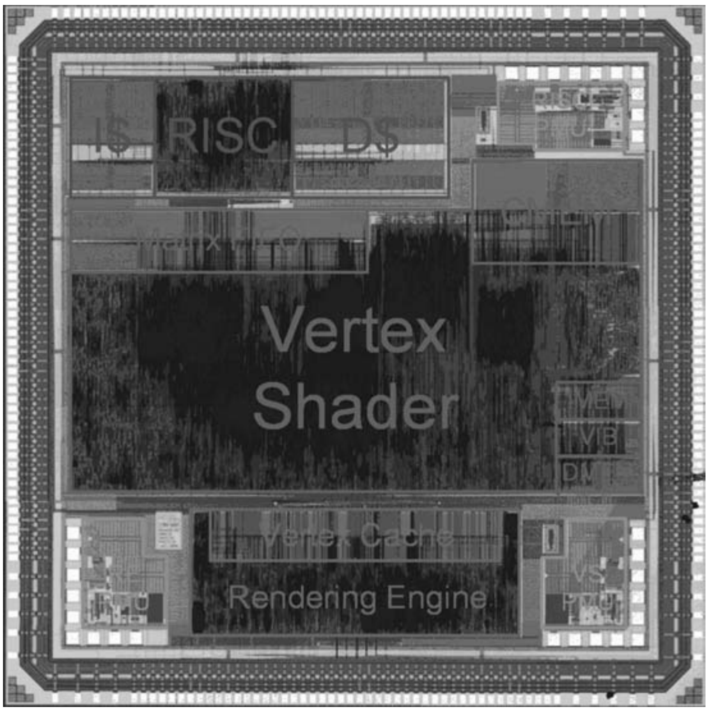


Figure 6.14 RAMP-VI chip micrograph

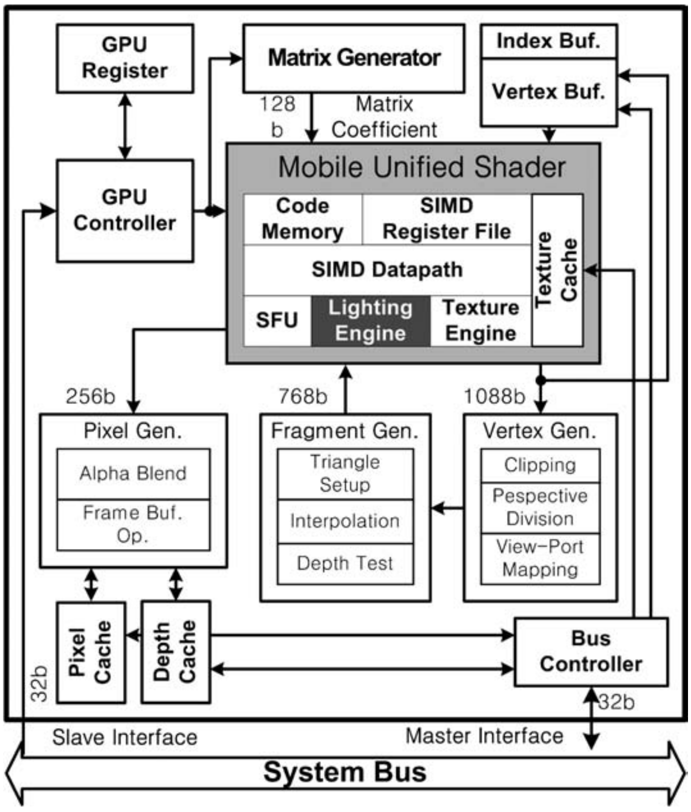


Figure 6.15 Block diagram of RAMP-VII

to enhance the graphics performance and hardware utilization. To meet the specific characteristics of mobile systems, RAMP-VII proposed a mobile unified shader architecture (Figure 6.15). It consists of the unified shader, matrix/quaternion vector generator, vector generator, fragment generator, pixel generator, and graphics caches.

6.1.4.1 Mobile Unified Shader

The mobile unified shader is a single-instruction multiple-data (SIMD) processor. It contains a 128-bit (4×32) SIMD datapath, special functional unit, texture engine, specialized lighting engine, SIMD register files, and control logic, as shown in Figure 6.16.

The SIMD datapath is responsible for vector arithmetic operations such as addition (ADD), multiplication (MUL), and inner product (DOT). The SFU is dedicated to special functional scalar operations such as logarithm (LOG), exponent (EXP), reciprocal (RCP), and reciprocal square-root (RSQ). The texture engine performs

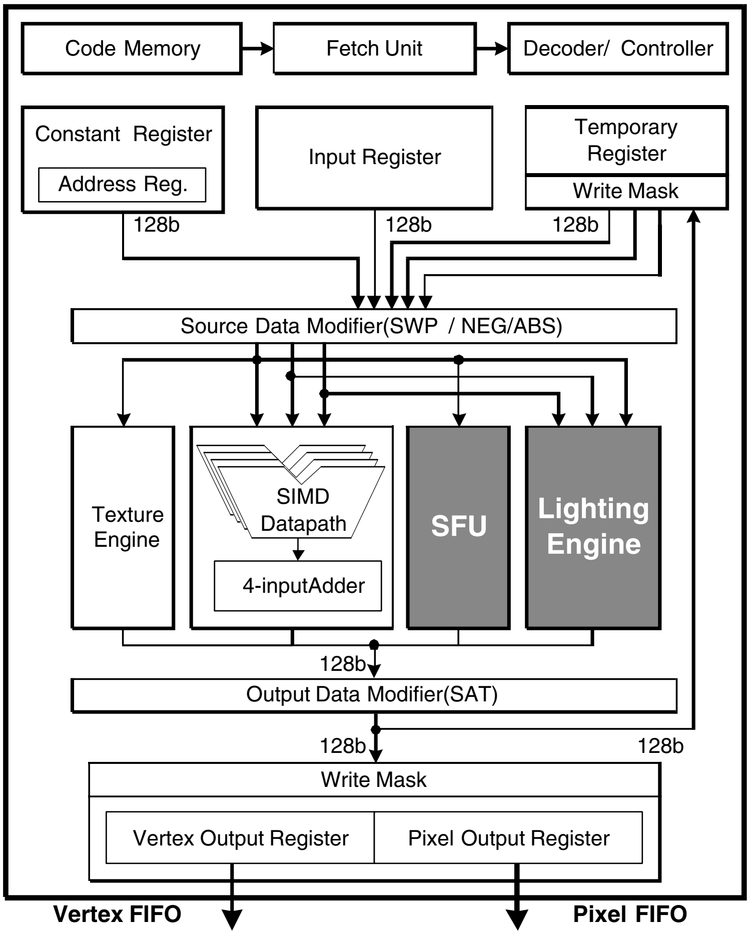


Figure 6.16 Mobile unified shader

texture-related operations such as perspective-correct texture address generation and texture filtering.

In contrast to a conventional unified shader in PCs or consoles, the mobile unified shader contains a dedicated lighting engine. The lighting equation is the most complex operation during the vertex operation owing to the power (POW) operation, so it consumes lots of time in a mobile system. In RAMP-VII, to accelerate the lighting equation with low power consumption, the lighting engine employs a logarithmic number datapath [12]. The OpenGL lighting equation – which includes an ambient light, a diffuse light, and a specular light – is described below in the ordinary number system and the log number system. The complex POW operation can be simply converted into a multiplication:

$$\text{Color\_ORD} = C_{\text{amb}} + \{(N' \times H) \times C_{\text{diff}}\} + \{(N' \times H)^{c_{\text{pow}}} \times C_{\text{spec}}\}; \tag{6.6}$$

$$\text{Color\_LNS} = C_{\text{amb}} + \{(N' \times H) \times C_{\text{diff}}\} + 2^{\{\log_2(N' \times H) \times C_{\text{pow}} + \log_2 C_{\text{spec}}\}}. \tag{6.7}$$

Therefore, the lighting engine consists of the logarithmic number datapath for the power (POW) operation of the specular light and the ordinary datapath for the ambient and the diffuse light calculations, as shown in Figure 6.17. In addition, since the data dependency in the lighting calculation degrades graphics performance (Figure 6.18), the specialized lighting instruction, TLT, is proposed to reduce data dependency. The TLT instruction combines the light coefficient calculation with the multiplication of coefficients and materials together and it generate a lit-vertex in every two cycles with the lighting engine.

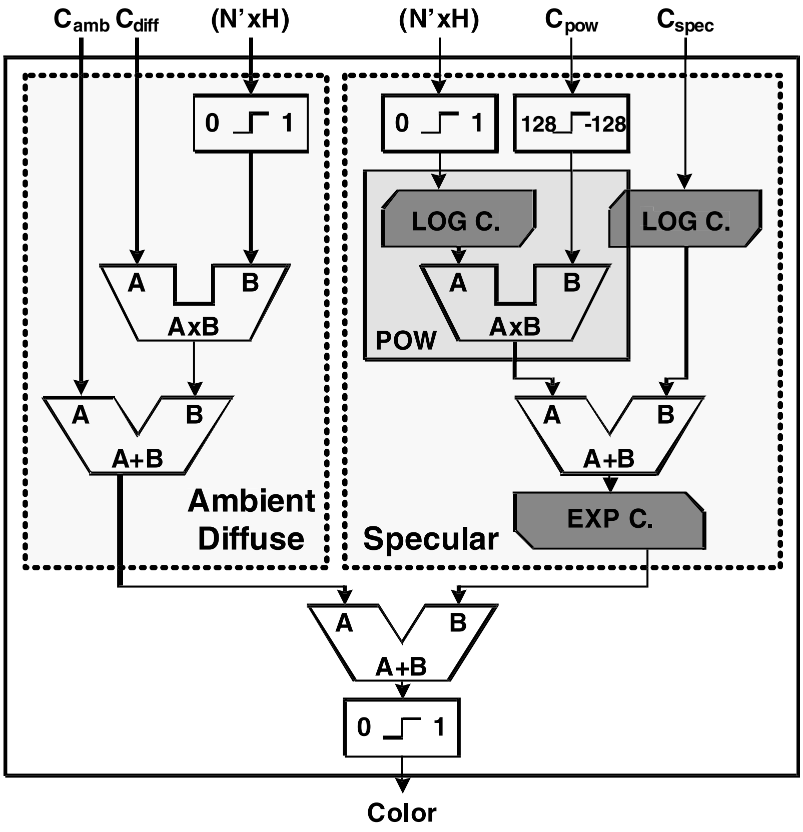


Figure 6.17 Lighting engine

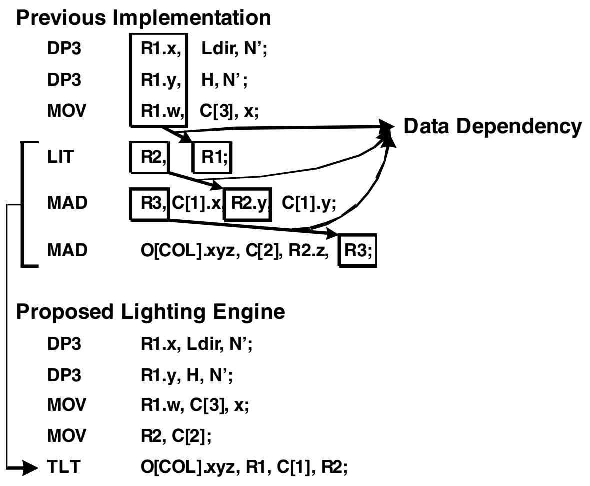


Figure 6.18 Lighting instruction

6.1.4.2 Pixel–Vertex Multi-threading

Figure 6.19 shows a data flow diagram of the programmable 3D graphics pipeline. In conventional architecture, the primitive vertices are computed in the vertex shader for per-vertex operations such as transformation and lighting. After per-vertex operations, the vertex generator and fragment generator perform clipping and rasterization and they generate interpolated pixels (fragments). After that, the fragments are modified in the pixel shader using per-pixel effects, and blending operations generate final pixel data. In contrast with conventional architecture, the mobile unified shader is responsible for both per-vertex operations and per-pixel operations in a single hardware. Therefore, the graphics data traverse the mobile unified shader twice in a single 3D graphics pipeline as shown in Figure 6.19, so the 3D graphics performance is degraded less than half of its peak performance due to task switching. Moreover, during the pixel operation, texture cache miss halts the 3D graphics processor until the cache is filled up, so the graphics performance is degraded further. To improve the 3D graphics performance, RAMP-VII adopts pixel–vertex multi-threading (PVMT), which utilizes datapaths of the mobile unified shader in parallel.

Since the texture engine performs texture fetching and filtering independent of the SIMD datapath and special functional block, those datapaths are idle during the texture operations. Therefore, the PVMT enables the SIMD datapath and special functional unit to compute per-vertex operations during the texture cache miss, as shown in Figure 6.20. When the texture cache miss occurs during per-pixel operations and there are vertices to compute, the PVMT issues the next vertices and the SIMD datapath and SFU perform per-vertex operations. If the texture cache filling is finished during the

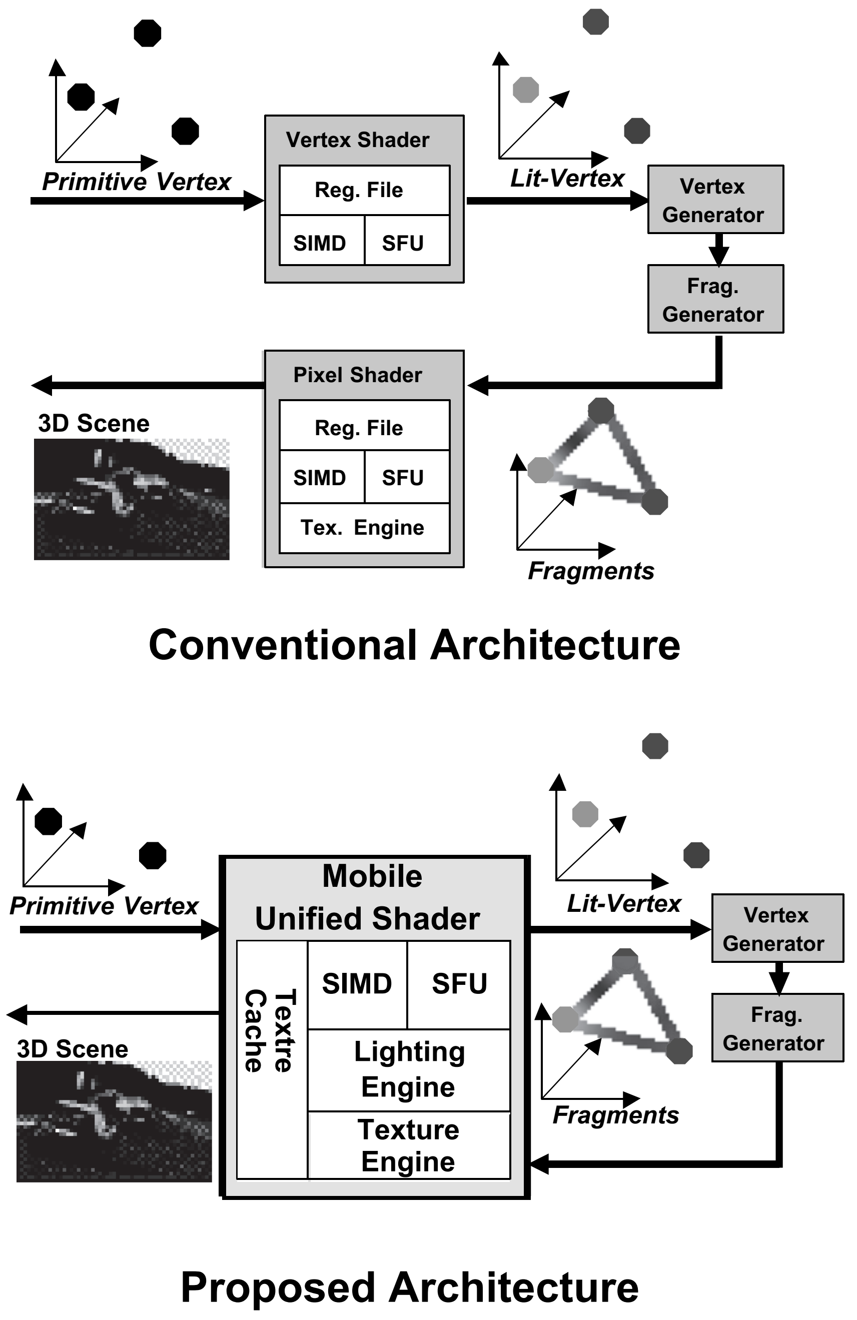


Figure 6.19 Data flow diagram in programmable 3D graphics pipeline

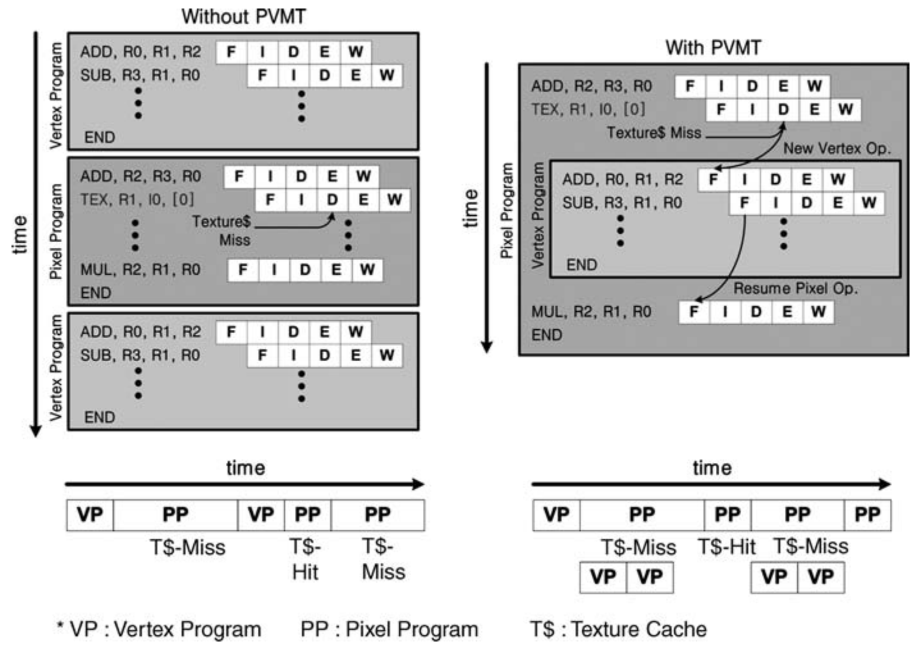


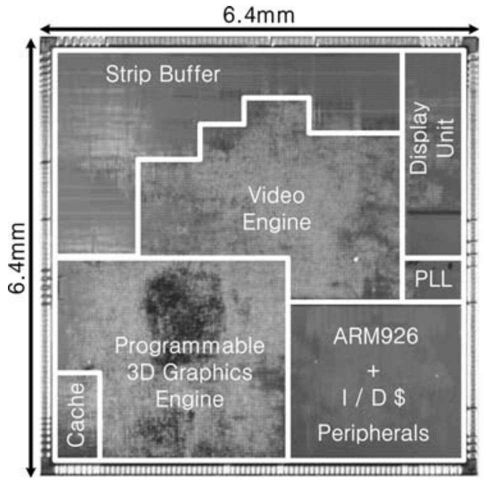
Figure 6.20 Pixel–vertex multi-threading

per-vertex operations, the mobile unified shader moves back to per-pixel operation and finalizes the remaining pixel operations. Otherwise, the PVMT issues the next vertices and performs per-vertex operations continuously.

Since the PVMT utilizes the wasted cycle of the texture cache miss, contents characteristics such as texture cache miss rate and vertex/pixel ratio strongly influence the effect of the PVMT. In our environment, the texture cache miss occurs with about 10% probability on average, and a texture cache miss consumes 100 cycles on average (as short as 64 cycles, or 148 cycles at most). Since the simple per-vertex operation including transform and lighting consumes about 20 cycles, the 3D graphics processor with PVMT can compute more than three vertices during the texture cache miss if the vertex buffer is enough. While the effect of the PVMT varies with the vertex/pixel ratio and the number of vertex buffer entries, the PVMT reduces the cycles of the vertex operation at least 60%, or 94% at the most. When the output vertex queue has five entries and an average polygon consists of 12 pixels, 94% of vertex operations are interleaved with pixel operations. By employing PVMT with a 5-entry vertex buffer, 94% of the vertex operations are interleaved into the pixel operations with only 6% performance degradation, and the single mobile unified shader can compute both per-vertex operations and per-pixel operations in a single hardware.

6.1.4.3 Implementation

RAMP-VII has been fabricated using a 0.13 μm seven-metal CMOS logic process and it was integrated into an ARM9-based mobile multimedia SoC [14]. It contains 1.3 M logic gates and graphics cache in an area 3.3 mm × 3.0 mm. Figure 6.21 shows the chip micro-photograph and feature summary.



(a)

Implementation Results	Technology	0.13mm 7M CMOS
	Power Supply	1.2V
	Transistor Counts	1.3M logic gates
	Die Size	3.3mm x 3.0mm
	Frequency	100MHz
	Power Consumption	**195mW @ Full 3D Graphics Processing** ARM-9 : 48mW System Peripherals : 22mW Mobile Unified Shader : 68mW Vertex / Pixel / FragmentGenerator : 57mW
Graphics Performance	Performance	**Vertex Fill Rate :** 9.1Mvertices/s (w/ TnL) **Raster Fill Rate :** 100Mpixels/s **Texel Fill Rate :** 400Mtexels/s **Pixel Fill Rate :** 7.4Mpixels/s (with Vertex Shading, Pixel Shading)
	Programmability	**Programmable Vertex Shader** **Programmable Pixel Shader**
	Standard API	OpenGL|ES ver. 1.1 OpenGL|ES ver. 2.0
	Screen Resolution	Up to 1024 x 1024 pixels

(b)

Figure 6.21 Implementation results of RAMP-VII

6.2 Industry Architecture

6.2.1 nVidia Mobile GPU – SC10 and Tegra

Since 2003, nVidia has been developing mobile 3D graphics processors using their knowledge base on GPUs for PCs. The first mobile GPU, the SC10, was introduced in 2004. This is an application processor targeted at handheld devices [15]. It works as a companion chip to the host processor and accelerates image, video, 2D, and 3D graphics processing. Its block diagram is shown in Figure 6.22 and its layout figure is shown in Figure 6.24. The 3D graphics processing engine, AR10 architecture, in this chip works with the external host processor and system memory. The chip interfaces with the host processor using scalable I/O from 8-bit to full 32-bit. The full duplex

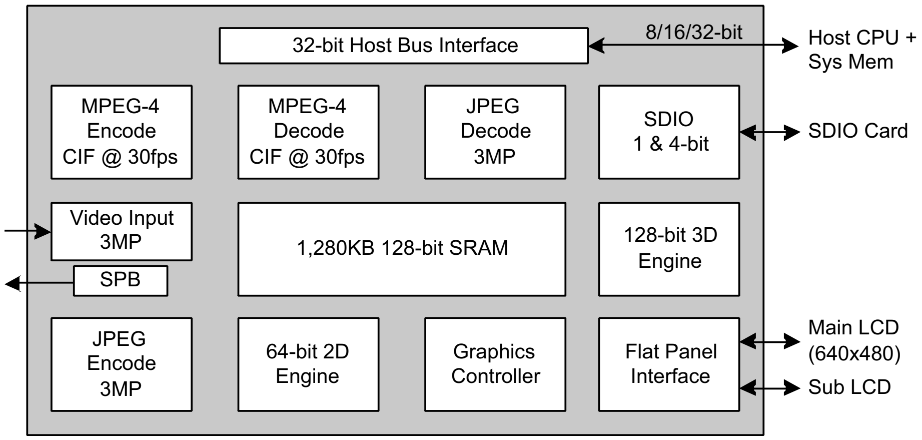


Figure 6.22 Block diagram of the nVidia SC10

hardware MPEG4 codec, serial peripheral bus for camera control, and LCD interface enables various multimedia solutions on a single chip. The SC10 distinguishes itself from other architectures by implementing pixel-level programmability in a 3D graphics accelerator for blending and combing operations to give various graphics effects on handheld devices. The embedded 1280 kB of SRAM (variable according to product line-up) with 128-bit interface provides a large vertex cache for reducing external memory accesses.

The SC10 supports several APIs for 3D graphics, such as OpenGL-ES and D3D Mobile. Its graphics pipeline consists of setup, raster, gatekeeper, data fetch, arithmetic units, and data write stages, as described below.

The setup stage provides a simple packet-based host interface and prepares the triangles for rasterization. It incorporates a software-controlled vertex cache and performs simple transformations, clipping, and back-face culling. The raster unit inter polates pixel parameters and generates pixel packets used by the rest of the pipeline. The gatekeeper unit performs the scoreboard operation of the pipeline and controls pixel packet flow. It keeps the pipeline as full as possible by tracking the recirculated (x, y) position value. The data fetch unit reads color, depth, and texture data. It performs the texture filtering and depth test. The texture filtering takes one clock cycle for nearest and bilinear filtering, and two cycles for trilinear filtering. There are four ALU units in the SC10 core. The ALUs perform blending and texture combining operations and operate on four 20-bit variables. The data write unit writes back the color and depth values. There are write merge buffers for the color and depth data, which avoids writing back the killed pixels from the depth test. It indicates retired writes to the gatekeeper for scoreboarding. The SC10 3D graphics pipeline is shown in Figure 6.23.

Low-power schemes such as low leakage process technology, embedded memory, and clock gating techniques are adopted in this processor. The processor is fabricated

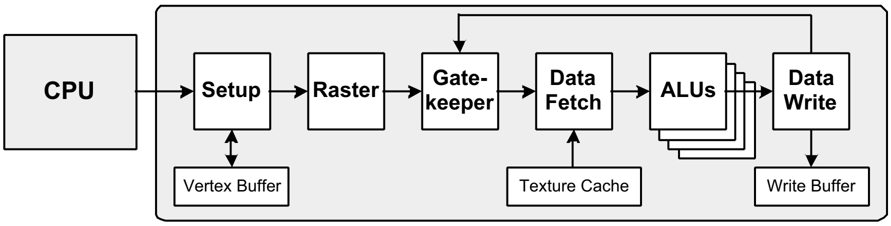


Figure 6.23 nVidia SC10 3D graphics engine

into a chip using 3.7 M transistors in 0.15 μm technology. Its pipeline shows a throughput of 1-pixel/clock and achieves 1 Mvertices/s at 72 MHz. It shows low power consumption of 50–75 mW at 30 frames per second. Although the setup unit relieves the burden on the host processor by doing simple transformations, clipping, and culling operations, the lack of a dedicated geometry engine and slow off-chip host interface limit its performance. The SC10 was the first programmable 3D graphics processor for mobile devices, but it did not achieve a big success in the mobile market owing to its limited graphics performance.

In early 2008, nVidia introduced a powerful mobile multimedia processor, the APX2500. After that, nVidia added Tegra 600 and 650 processors. Like the SC10, the Tegra APX2500 is an application processor that includes video, image signal processing, and 3D graphics. Figure 6.25 summarized the features of the APX2500 and Figure 6.26 shows the block diagram of APX2500.

For advanced 3D graphics with low power consumption, nVidia developed the ultra-low-power Geforce (ULP Geforce), which it claims is the lowest power 3D hardware solution available. The ULP Geforce supports programmable vertex lighting and

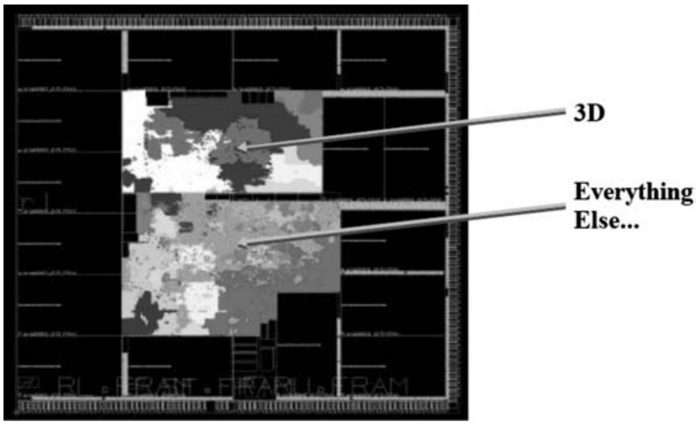
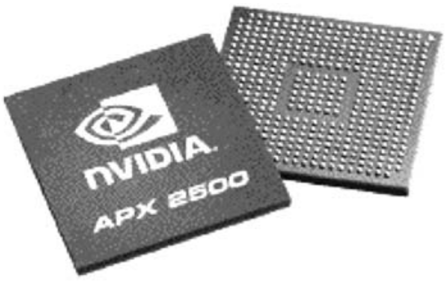


Figure 6.24 Layout figure of nVidia SC10. ©2009 NVIDIA Corp. NVIDIA, the NVIDIA logo, SC10, and Tegra are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. All rights reserved



Processor & Memory	ARM11 MP Core 16/32b LP-DDR NOR / NAND Flash
High Definition Audio/ Video Processor	720p H.264 / MPEG-4 Decode 720p H.264 / MPEG-4 Encode Multi Standard Audio Standards - AAC, AMR, WMA, MP3 JPEG Encode / Decode
Ultra Low Power Geforce	OpenGL ES 2.0 / D3D Mobile Programmable Pixel Shading Programmable Vertex and Lighting CSAA Supporting Advanced 2D Graphics 40Mtriangle /s 600Mpixels /s 240Mpixels /s when texturing
Imaging	Up to 12Mpixel camera sensor support Integrated ISP (Image Signal Processor) Advanced imaging features
Display	Dual display Supports 720p (1280 x 720) HDMI 1.2 support SXGA(1280 x 1024) LCD/CRT support

Figure 6.25 nVidia APX2500. ©2009 NVIDIA Corp. NVIDIA, the NVIDIA logo, SC10, and Tegra are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. All rights reserved

programmable pixel lighting and it is also compatible with OpenGL-ES 2.0 and Direct 3D Mobile, especially it targeted at the DirectX-9 model. In DirectX-9 the programmable 3D graphics pipeline, the vertex, and state traffic have been increased compared with the DirectX-7 model, so the tile-based rendering cannot achieve good graphics performance with low power consumption. Therefore, the ULP Geforce employs the traditional 3D graphics pipeline with early depth test and fragment caching, which are helpful to reduce required bandwidth of graphics data. For the same reason, it does not employ a unified shader [13], which is more efficient in the DirectX-10 model. And one notable thing is the advanced 2D graphics. Since vector graphics are widely used in mobile devices for user interfaces (e.g the Apple iPod), or flash playback, the ULP Geforce is designed to accelerate those 2D vector graphics. The ULP Geforce achieves

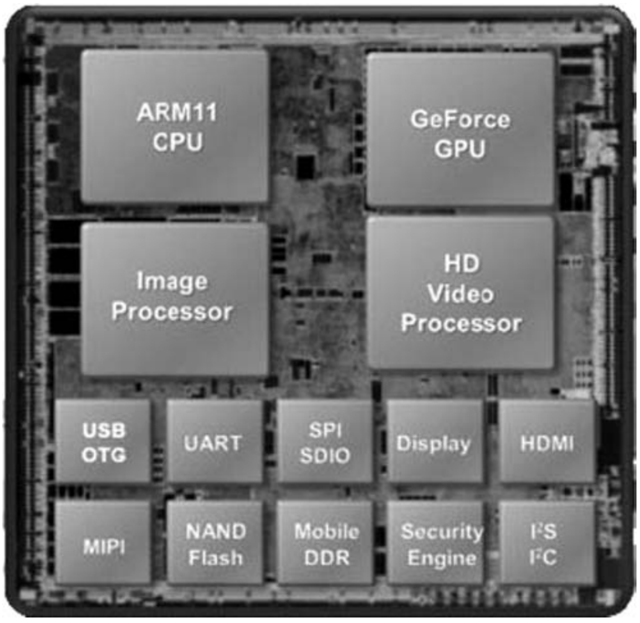


Figure 6.26 Block diagram of the Tegra family. ©2009 NVIDIA Corp. NVIDIA, the NVIDIA logo, SC10, and Tegra are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. All rights reserved

40 Mtriangles/s for geometry performance, and it shows 600 Mpixels/s without texturing and 240 Mpixels/s with texturing for rendering performance.

6.2.2 Sony PSP

In 2004, Sony released a handheld video gaming system called PlayStation Portable (PSP™) for realtime 3D gaming and multimedia applications like MPEG video and MP3 audio playbacks on a battery-powered handheld device. The graphics processor for PSP™ supports 3D computer graphics gaming on a relatively wide screen of 4.3 inches for a handheld system [16]. PSP™ includes the MIPS R4000 CPU core with vector floating-point unit (FPU) and a surface engine and a rendering engine for the game processing, as well as the H.264 codec for media processing and a reconfigurable processor for audio/video codec. It also features 4 Mbyte of embedded DRAM for enhanced support of memory bandwidth, and read–modify–write operations for 3D graphics processing.

The graphics module in PSP™ consists of surface engine and rendering engine. The surface engine of the gaming core supports the tessellation of Bezier and Spline surfaces to reduce the memory bandwidth for model transference. It also supports

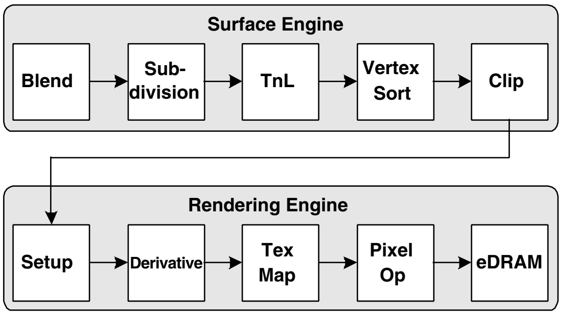


Figure 6.27 Sony PSP™ 3D graphics pipeline

geometry processing such as 3D geometry transformations, lighting operations and geometry blending for skinning and morphing. The rendering engine supports several lighting effects including directional, point, and spot lighting effects. It also supports several texture mapping schemes such as environmental mapping, projection mapping, and per-fragment operations like alpha blending, depth test, stencil test, and dithering.

Figure 6.27 shows the PSP™ 3D graphics pipeline. The media processing unit in PSP™ incorporates the hardwired H.264 codec and a real-time reconfigurable processor for audio/video codec implementation. Even though it adopts a quite conventional bus protocol for its interface with the host system, the direct eDRAM controller attached to the rendering engine significantly reduces the memory bandwidth requirements for the external main memory. The eDRAM controller also allows the host system to access the video memory directly for flexible memory operations.

The processor operates at 166 MHz with 4 Mbytes of graphics memory. It shows graphics performance of 58 Kpatches/s for surface tessellation, 35 Mpolygons/s for geometry processing, 35 Mpolygons/s for rendering setup, and 664 Mpixels/s for pixel processing. It adopts several low-power design techniques such as voltage and clock frequency control and clock and power gating techniques to each IP, so that it achieves less than 500 mW power consumption. However, even with its several low-power design techniques, it still shows relatively high power dissipation for devices like cellphones.

6.2.3 Imagination Technology MBX/SGX

In 2002, Imagination Technology (previous known as PowerVR) developed a low-power rendering processor, MBX, based on tile-based rendering [17, 18]. MBX was designed to support the OpenGL-ES 1.x, fixed-function 3D graphics pipeline. Figure 6.28 shows the architecture. It consists of tile accelerator, rendering core, texture unit with texture cache, pixel blend unit, and control unit.

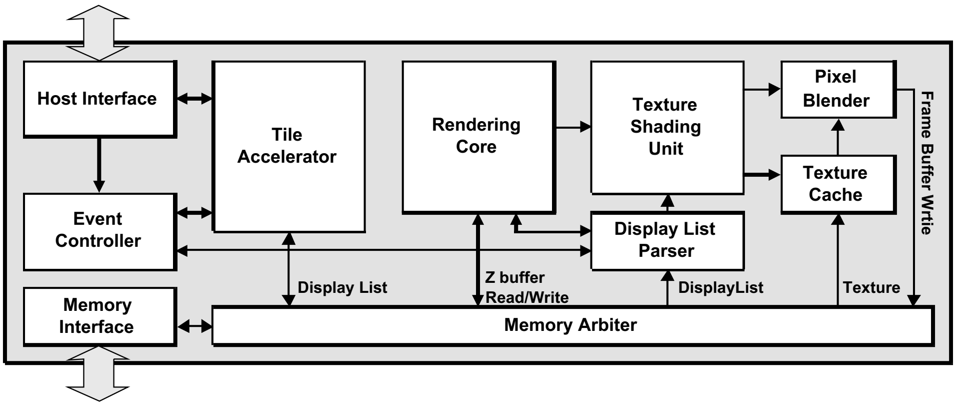


Figure 6.28 MBX architecture

MBX divides the screen into 32 × 32-pixel tiles and it operates based on that. To increase the pixel throughput, MBX has a parallel 32 × 1-pixel tile accelerator inside the rendering core. The pixel tile accelerator performs the parallel triangle setup equation and depth test per cycle. This means that it consumes 32 cycles to compute one triangle. Therefore, the peak performance is limited to 7 Mtriangles/s at 200 MHz operating frequency. To reduce memory utilization, MBX also employs texture compression, which is an algorithm unique to Imagination Technology; the texture data is stored with only 2 bits or 4 bits. One problem with adopting tile-based rendering is the geometry overhead. Since the rendering operation is done on a tile basis, it requires additional computational power. Therefore there is "tile acceleration," a kind of programmable SIMD coprocessor, inside MBX.

In 2005, Imagination Technology announced a new mobile 3D GPU, called SGX. This is targeted to the OpenGL-ES 2.x, programmable 3D graphics pipeline. Figure 6.29 shows the architecture. The company claims that the PowerVR SGX core – formerly known by its codename "Eurasia" – provides programmable shader support to beyond OpenGL 2.0 and DirectX-9 shader model 3 standard. As the figure shows, SGXs shader system employs the unified scalable shader engine (USSE), which merges vertex and pixel shading into a single pipeline capable of processing either kind of image element. Its instruction set architecture and features are optimized for three types of task: vertex shading, pixel shading, and video/imaging processing.

The coarse-grain scheduler (CGS) is the main controller for SGX. It consists of two stages, the data master selector (DMS) and program data sequencer (PDS). The DMS processes requests from the data masters and determines which tasks can be executed given the resource requirements. Then, the PDS manages the workload and processing of data on the USSE. For data parsing to the USSE, there are three data masters in the SGX core: general-purpose data master (GPD), pixel data master (PDM), and vertex data master (VDM).

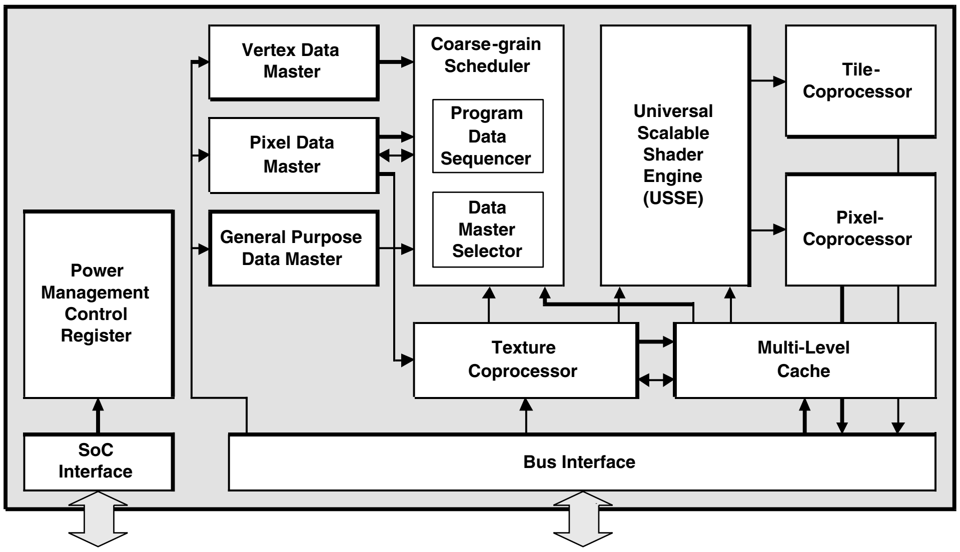


Figure 6.29 SGX architecture

The VDM initiates transform and lighting processing. The VDM reads an input control stream, which contains triangle index data, and state data. Using state data, the VDM decides the PDS program, size of the vertices, and the amount of USSE output buffer resource.

The PDM is the initiator of rasterization processing. Each pixel pipeline processes pixels for a different half of a given tile, which allows for optimum efficiency within each pipe due to locality of the data. For each task it determines the amount of resource required within the USSE. It merges this with the state address and issues a request for execution on the USSE to the DMS.

The GDM responds to events within the system (such as end of a pass of triangles from the ISP, end of a tile from the ISP, end of render, or parameter stream breakpoint event). Each event causes either an interrupt to the host or synchronized execution of a program on the PDS. The program may, or may not, cause a subsequent task to be executed on the USSE.

The multilevel cache is a two-level cache consisting of two modules, the cache core and the mux/arbiter/demux/decompression (MADD) unit. The MADD is a wrapper around the cache core designed to manage data format between cache core and other blocks such as texture coprocessor and USSE, as well as providing level 0 caching for texture and USSE requests. The MADD can accept requests from the PDS, USSE, and texture address generator modules. Arbitration is performed between the three data streams. Since the texture data are compressed, texture decompression is also performed as necessary.

The texturing coprocessor performs texture address generation and formatting of texture data. It receives requests from either the iterators or USSE modules and translates these into requests into the multilevel cache. Data returned from the cache are then formatted according to the texture format selected and sent to the USSE for pixel-shading operations. As in MBX, it also contains a tiling coprocessor to process pixels in a tiled manner. The tiling coprocessor performs dividing screen into tiles, arranging tiles as groups. A native advantage of tiling architecture is that a large amount of vertex data can be rejected at this stage, thus reducing both the memory storage requirements and the amount of pixel processing to be performed. The pixel coprocessor is the final stage of the pixel-processing pipeline and controls the format of the final pixel data sent to the memory. It supplies the USSE with an address into the output buffer, and the USSE returns the relevant pixel data. The address order is determined by the frame buffer mode. The pixel coprocessor contains a dithering and packing function.

The SGX architecture decouples geometry processing and rendering processing to minimize pipeline stalling, along with on-chip support for multiple render targets (MRTs). A large proportion of PowerVR SGX's peak throughput is achievable in real applications, and the architecture has the lowest bandwidth requirements as PowerVR SGX's deferred pixel shading gives 2–3 times fill rate compared with other solutions at the same bandwidth. Across the SGX family (510, 520, and 530), the effective fill-rate performance is from 200 M to 1200 Mpixels/s at 200 MHz (with even higher *Z* and stencil rates), and polygon throughput is from 2 M to 13.5 Mpolygons/s at 200 MHz. The SGX's USSE supports advanced geometry and pixel processing capabilities such as procedural geometry and textures, advanced per-pixel and vertex lighting effects. Further, the highly flexible nature of the USSE architecture allows this programmability to be applied to many other tasks such as other multimedia-related activities (e.g., physical modeling), flexible video and image processing, and many more. This unified approach to processing also has the benefit of requiring a single unified programming model with one compiler, reducing hardware and software qualification time.

A notable benefit of the SGX tiling architecture is the on-chip multiple render targets (MRTs). This technology is unique to Imagination Technology tile-based rendering and cannot be replicated by an immediate mode renderer (IMR) in a cost-effective manner. IMRs use conventional external MRTs with huge external memory storage and bandwidth. PowerVR's on-chip MRTs result in low geometry processing with no additional external memory cost and no additional memory bandwidth cost.

References

1 Sohn, J.-H. *et al.* (2005) A 50-Mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *Proc. of IEEE International Solid-State Circuits Conference 2005*, pp. 192–193.

2 Park, Y.-H. *et al.* (2001) A 7.1-GB/s low-power rendering engine in 2D array embedded memory logic CMOS for portable multimedia system. *IEEE J. Solid-St. Circ.*, **36** (6), 944–955.

3 Yoon, C.-W. *et al.* (2001) A 80/20-MHz 160-mW multimedia processor integrated with embedded DRAM, MPEG4 and 3D rendering engine for mobile applications. *IEEE J. Solid-St. Circ.*, **36** (11), 1758–1767.

4 Woo, R. *et al.* (2004) A 210-mW graphics LSI implementing full 3D pipeline with 264-Mtexels/s texturing for mobile multimedia applications. *IEEE J. Solid-St. Circ.*, **39** (2), 358–367.

5 Williams, L. (1983) Pyramidal parametrics. *Proc. of SIGGRAPH 1983*, pp. 1–11.

6 Sohn, J.-H. *et al.* (2005) A 50-Mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2005.*

7 Sohn, J.-H. *et al.* (2005) Low-power 3D graphics processors for mobile terminals. *IEEE Commun. Mag.*, **43** (12) 90–99.

8 Sohn, J.-H. *et al.* (2004) A programmable vertex shader with fixed-point SIMD datapath for low power wireless applications. *Proc. of ACM Graphics Hardware Workshop*, pp. 107–114.

9 Hao, Xuejun *et al.* (2001) Variable-precision rendering. *Proc. of Symposium on Interactive 3D Graphics*, pp. 149–158.

10 Nam, B.-G. *et al.* (2007) A 52.4-mW 3D graphics processor with 141-Mvertices/s vertex shader and three power domains of dynamic voltage and frequency scaling. *Digest of Technical Papers of IEEE International Solid-State Circuits Conference 2007.*

11 Nam, B.G. *et al.* (2005) Development of a 3-D graphics rendering engine with lighting acceleration for handheld multimedia systems. *IEEE Trans. Consumer Elec.*, **51** (3), 1020–1027.

12 Kim, H. *et al.* (2006) A 231-MHz 2.18-mW 32-bit logarithmic arithmetic unit for fixed-point 3D graphics system. *IEEE J. Solid-St. Circ.*, **41** (11), 2373–2381.

13 Doggett, M. (2005) Xenos: XBOX360 GPU, ACM Eurographics.

14 Woo, J.-H. *et al.* (2008) A 195/152-mW mobile multimedia SoC with fully programmable 3D graphics and MPEG4/H.264/JPEG. *IEEE J. Solid-St. Circ.*, **43** (9), 2047–2056.

15 Hutchins, E. (2004) SC10: A video processor and pixel-shading GPU for handheld devices. *Proc. of Hot Chips 16: Symposium on High Performance Chips.*

16 Kurose, Y., Kumata, I., Okabe, M. *et al.* (2004) A 90-nm embedded-DRAM single-chip LSI with a 3D graphics, H.264 codec engine, and a reconfigurable processor. *Proc. of Hot Chips 16: Symposium on High Performance Chips.*

17 MBX. Available at http://www.imgtec.com/powervr/mbx.asp.

18 SGX. Available at http://www.imgtec.com/powervr/sgx.asp.

7

Low-power Rasterizer Design

This chapter attempts to show how to design a large-scale system – from specification to real implementation – by means of an example of a low-power rasterizer. First, high-level information about the rasterizer is described: its target system architecture, performance and feature summaries, and instruction set architecture. We explain the details of the rasterizer from top module to tiny single register. We also include the verilog source codes. The source code and simulation environments are included on the CD accompanying this book.

7.1 Target System Architecture

When you start to design a certain functional unit, the most important thing is to define the target system. You need to define the boundary conditions such as power, memory bandwidth, and job partitioning from the system's view, otherwise the unit you design may not provide the desired performance in the system. In this chapter we assume that the rasterizer is connected to the system using memory-mapped I/O. The application software and geometry operations are computed in the host processor, and the rasterizer performs the remaining rendering operation, shading and texturing. Figure 7.1 shows the target system architecture including the rasterizer.

The host processor performs geometry operations using primitive data stored in system memory. The vertex data and rendering commands are transferred to the rendering processor through the system bus. The rendering processor generates texture-mapped pixels which are displayed on a liquid-crystal display of QVGA screen resolution.

In this system, the rasterizer is a slave component. The host processor controls the operation of the rasterizer and it controls the data transfer. Communication between the host processor and the rasterizer uses a chip status register (CSR). When a rendering

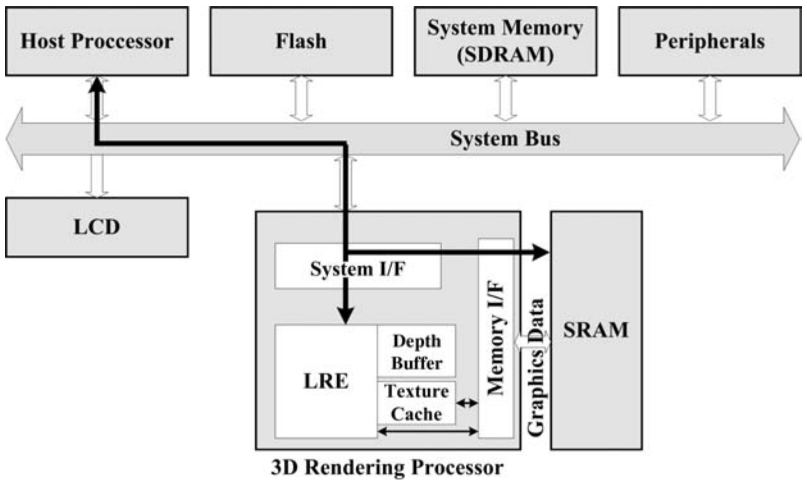


Figure 7.1 Target system architecture

operation is finished, the “Operation Done” flag of the CSR turns high and the processor checks the register to recognize that the operation is done.

7.2 Summary of Performance and Features

Table 7.1 summarizes the performance and features of the rasterizer. It is designed for low-end mobile devices, so it targets 20 Mpixels/s fill rate at 10 MHz operating frequency. It supports basic rendering functions: Gouraud shading, perspective correct bilinear texture filtering, and alpha blending.

7.3 Block Diagram of the Rasterizer

Figure 7.2 shows the block diagram. It consists of a low-power rendering engine (LRE), depth buffer, texture cache, and interface logics. The LRE contains a triangle setup engine, two pixel processors, two texture units, and two pixel blending engines. The system interface logic decodes memory addresses and decides the operating mode – between graphic mode and direct memory access mode. The memory interface logic controls the SRAM, and arbitrates memory requests from the texture cache and blending unit. The LRE performs shading and texturing operations with two pixel processors including texture units. In order to achieve low-power rendering, it uses pixel-level clock gating and address alignment logic, as explained in Chapter 5. Pixel-level clock gating prevents unnecessary operation of pipeline datapaths according to the results of depth comparisons. Pixel-level clock gating shows an average 25% power

Table 7.1 Performance and features summary

	Low-power rasterization unit (RAMP-GR)
Screen resolution	320×240
Color depth	16-bit (Red: Green: Blue = 5b: 6b: 5b)
Rendering performances	20 Mpixels/s pixel fill rate @ 10 MHz
	80 Mtexels/s texel fill rate
	Two pixel processors
Shading feature	Gouraud shading
	Pixel alpha blending
	Texture blending (decal/modulate)
Z-buffer	16-bit embedded Z-buffer
Texture mapping	Perspective correct texture address generation
	Texture sampling
	Point sampling
	Bilinear filtering
	Maximum texture size: 256×256 pixels
	Maximum number of texture = 4
	Software controllable texture address calculation
	Efficient texture fetch through cache alignment
	Logic texture filtering
Operation frequency	10 MHz
External SRAM capacity	1 MB (256×32 bits)

reduction for typical graphics applications. Bilinear MIPMAP texture filtering requires as many as eight texture memory requests at every cycle. The address alignment logic reduces memory requests from eight to an average of 2.5 by using temporal and spatial data locality.

7.4 Instruction Set Architecture (ISA)

The rasterizer has a 128-bit instruction format as shown in Figure 7.3. It consists of 32-bit command field and 96-bit data field (Figures 7.4 and 7.5). The command field has MODE, TYPE, OP1, OP2, and EXTRA fields.

The MODE field defines the rasterizer mode: normal, debug, memory test, and so on. In this example the rasterizer operates in normal mode, so that the mode field is set to 4-bit 1111.

The TYPE field classifies OP code types. In this example, the rasterizer has three types of OP code as summarized in Table 7.2. OP1 and OP2 fields carry various meanings according to the OP code.

The DATA filed contains one-vertex information. For rasterization, one vertex has to contain screen coordinates, texture coordinates, and color information. The detailed field definition is shown in Table 7.2. For rendering data transfer, the DATA field is used for vertex data, and it is used for texture data for texture setup.

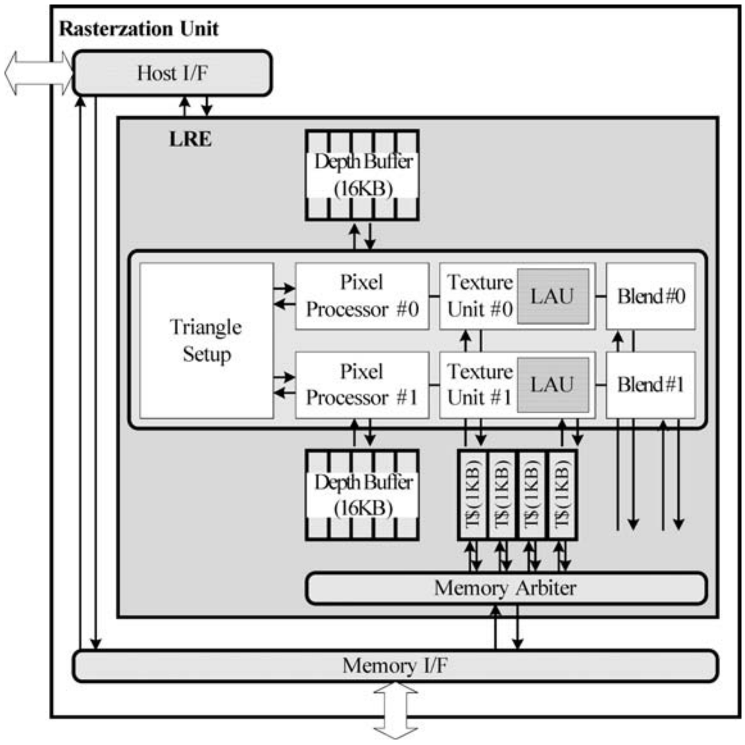


Figure 7.2 Block diagram

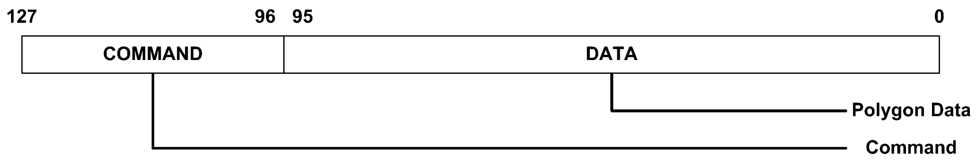


Figure 7.3 Instruction format

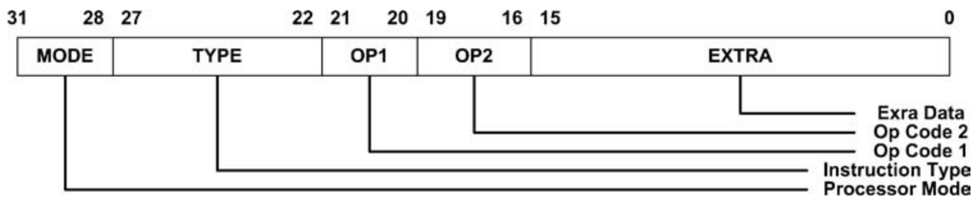


Figure 7.4 Command (CMD) format

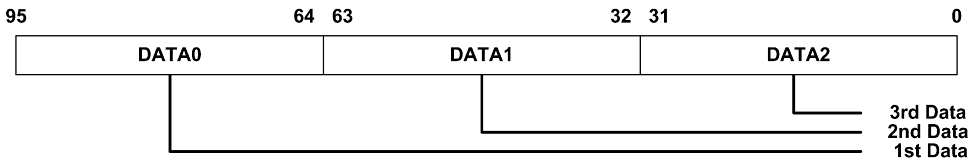


Figure 7.5 Data field format

In this example the rasterizer has four instructions: RDAT, TMOD, ASTR, and RDON. RDAT is designed for basic rasterization operations. In a typical case (not STRIP or FAN), one polygon consists of three vertices, and when the TRI field represents "End Vertex" the rendering for the polygon is started. The TMOD instruction sets the texture mode of the rasterizer. In a typical case, the TMOD instruction appears as the first line of the code. The ASTR instruction is used to clear or pre-set frame and depth buffers. Using the ASTR instruction, the user can change the background color or depth bias. This can generate some smart 3D graphics effects such as a user-defined surface. Last, the RDON instruction is designed to communicate with the host processor. When the RDON command runs the rasterizer pipeline to the last, the chip status register is changed to show the rendering operation done. Then the host processor reads the register and it transfers the next frame to the rasterizer. An example of code is shown below:

```
F44200001100010000000000000000000    //TMOD
F21000000000000000000000000000000    //ASTR ZB
F2200000000012C000000000000000000    //ASTR FB
F80400FF19190000FF00008000000000     //RDAT 1st Vertex
F80200FF19320000FF00008000000000     //RDAT 2nd Vertex
F81100FF32320000FF00008000000000     //RDAT 3rd Vertex
F8C00000000000000000000000000000     //RDON
```

Number Format and Range

```
OW, U, V [16b]          : [unsigned int [8b] : fractional int [8b]]
R, G, B                 : [5b : 6b : 5b] : unsigned int
X [9b]                  : unsigned int
Y [8b]                  : unsigned int
Z [15b]                 : unsigned int
A [5b]                  : 5 - level active
                            1_XXXX : 100% (opaque)
                            0_1XXX : 50%
                            0_01XX : 25%
                            0_001X : 12.5%
                            0_0001 : 0%
Texture ID [8b]         : unsigned int If (tid == 0) No texture
    Texture Size [9b]       : unsigned int (max. 256)
    Texture Addr [20b]      : unsigned int
    LOD Bias [3b]           : unsigned int
    FZB Address [16b]       : unsigned int
```

Screen coordinates are depicted in Figure 7.6.

Table 7.2 Instruction set

Type	Mnemonic	Op code	Description
Rendering	RDAT	MODE = 1111	Fetch vertex data
	TRI	TYPE = 1000 00	TRI: strip support
	POS	OP1 = TRI	00 = intermediate vertex
	W U V	OP2 = POS	01 = end vertex
	X Y Z	EXTRA = W	POS: reduce BW
	A R G B	DATA	0100: 1st vertex
			0010: 2nd vertex
			0001: 3rd vertex
			W[16b] = 1/W
			DATA0 [16b:16b] = u: v
			DATA1 [9b:8b:15b] = X: Y: Z
			DATA2 [8b:8b:8b:8b] = A: R: G: B
			(R,G,B: used upper 5,6,5-bit)
			(A is valid only if TRI = 01)
	RDON	MODE = 1111	Check rendering done
		TYPE = 1000 11	
Texture	TMOD	MODE = 1111	Set texture mode
	ADDR	TYPE = 0100 01	ADDR [22b]: base address
	BLND FILT ID	{OP1:OP2: EXTRA} = ADDR	BLND [4b]: blending mode
	SIZE	DATA0 = {BLND: FILT: ID: LOD: SIZE}	0001: decal
			0010: modulate
			FILT [4b]: filtering
			0001: point sampling
			0010: bilinear filtering
			0100: trilinear filtering
			ID [8b]: texture ID
			LOD [4b]: LOD bias
			0xxx: normal mode
			LOD must be 0000
			SIZE [12b]: texture size
Auxiliary	ASTR	MODE = 1111	Store data to front buffer
	FZ	TYPE = 0010 00	Frame buffer/depth buffer
	ADDR	OP1 = FZ	10 = FB only
	R G B	EXTRA = ADDR	01 = ZB only
		DATA 0	11 = FB and ZB
			ADDR = FB/ZB address
			DATA 0 [8b, 8b, 8b] = R: G: B
			(Used lower 24b of DATA0)

7.5 Detailed Design with Register Transfer Level Code

7.5.1 Rasterization Top Block

The diagram of the top block is given in Figure 7.7. Below is listed the RTL code.

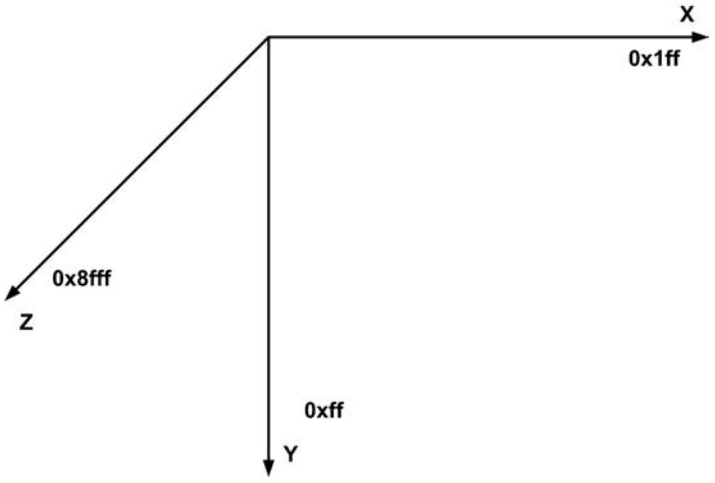


Figure 7.6 Screen coordinates

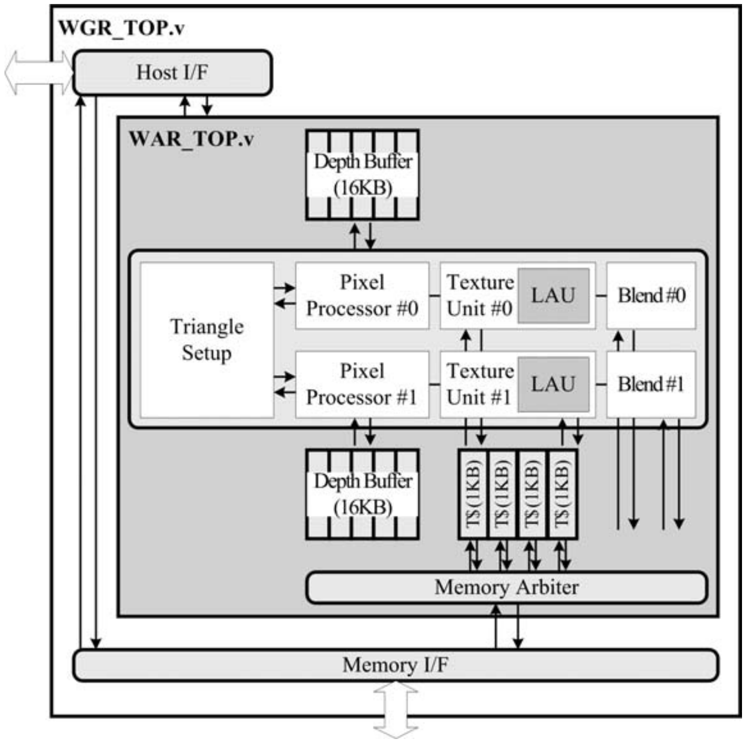


Figure 7.7 Module block diagram

RTL Code

```
/*
* Project : RAMP-GR : Low-power Rasterazation Unit
* WGR_TOP : Rasterization Top Module
* By Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/
module WGR(REclk, memclk1, memclk2, REreset,
           HnCS, HnWE, HnOE, HBSEL, HADDR, HRDATA, HWDATA, HOEN,
           EnCS, EnWE, EnOE, EBSEL, EADDR, ERDATA, EWDATA, EOEN,
           Dbg_out);

     // Memory-mapped I/O interfacel logic
     // The BEQ stands for bandwidth equalizer.
     // Since the system bus has 32-bit interface and the rasterizer
     // command uses 128-bit width, interface logic to adjust
     // two different bit widths is required.
     // And the BEQ has 6 command queues to reduce communication
     // latency between host processor and rasterization unit.

     Wbeq_re          GR_BEQ();

     // Rasteration unit core logic
     // Rasterization pipeline implemented in WAR block
     WAR              GR_CORE()

     // Memory address decoder logic
     Wmem_decode      GR_MDEC();

     //Memory request muxing logic
     Wmem_mux         GR_MMUX();
endmodule
```

7.5.2 *Pipeline Architecture*

The pipeline architecture is shown in Figure 7.8.

7.5.3 *Main Controller Design*

The block diagram is shown in Figure 7.9, and the signal descriptions are detailed in Tables 7.3 and 7.4.

The main controller of the rasterizer has two control signals, nHld and nWait. By using two signals the controller can manage multi-cycle operation. An nHld signal comes from the previous pipeline stage, otherwise an nWait signal comes from the next pipeline stage. When nHld or nWait goes high at one stage, the pipeline stage does not operate until the signal turns off. For example, if one line has 10 pixels, the pixel interpolater unit requires 10 cycles to finish the line. During those 10 cycles, earlier

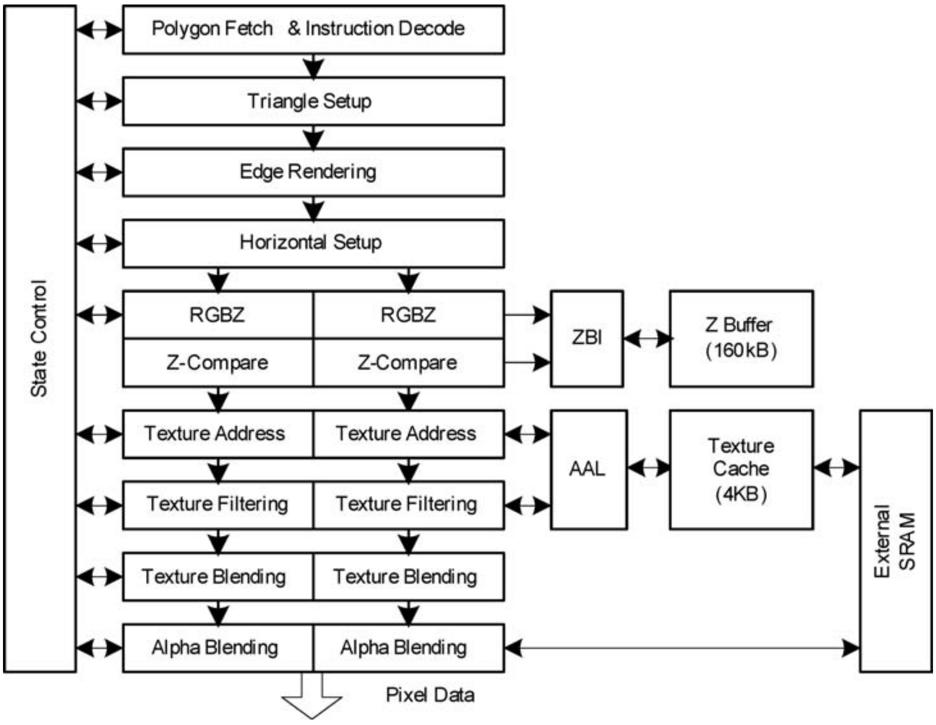


Figure 7.8 Pipeline architecture

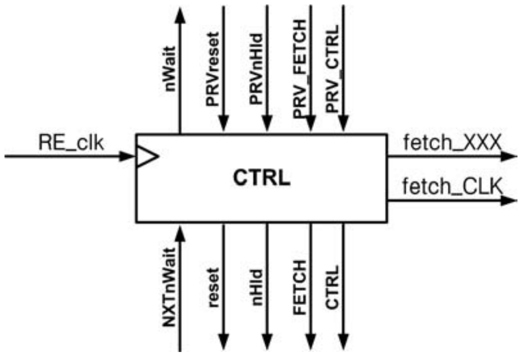


Figure 7.9 Main controller

pipeline stages – fetch, decode, triangle setup, edge processor, and horizontal setup – have to stop processing. In this case the pixel interpolation generates nWait signals to the earlier stages. In the same way, when one pipeline stage needs to stop the following pipeline stages – texture cache miss or memory arbitration – it generates nHld signals

Table 7.3 External signals of the controller

Signal	Description
RE_clk	Main rendering clock
nWait (@p)	Wait signal to previous pipe
	Overrides NXTnWait with nWait_gen
	FIR with PRVreset (nWait = 1 when PRVreset ==0)
PRVnHld (@p)	Hold signal from previous pipe
	NOP if nHld == 0
PRV_FETCH (@p)	Fetch signal from previous pipe
PRV_CTRL (@p)	Control signals from previous pipe
PRVreset (@p)	Reset signal from previous pipe
	Overrides nWait
	Force to fetch
NXTnWait (@p)	Wait signal from next pipe
nHld (@p)	Hold signal to next pipe
	Overrides PRVnHld with nHld_gen
FETCH (@n)	Fetch signal to next pipe
CTRL (@p)	Control signal to next pipe
Reset (@p)	Reset signal to next pipe
fetch_XXX	REclk & PRV_FETCH
fetch_CLK	nWait & PRVnHld

Table 7.4 Internal signals of the controller

Signal	Description
nHld_gen	Generate NOP to next pipe
nWait_gen	Generate or erase wait to previous pipe
FETCH_int (@p)	Fetch signals
nHld_int (@p)	Hold signals

to the following pipelines. In a certain pipeline stage, the enable signals of the stage are generated with nHld and nWait signals. The nWait signals of the stage enable control and data latches as shown in Figure 7.10.

7.5.4 Rasterization Core Unit

The complete block diagram of the core unit is shown in Figure 7.11.

7.5.4.1 IF: Instruction Fetch Unit

In the IF stage (Figure 7.12), the input data are latched when the rasterizer is enabled.

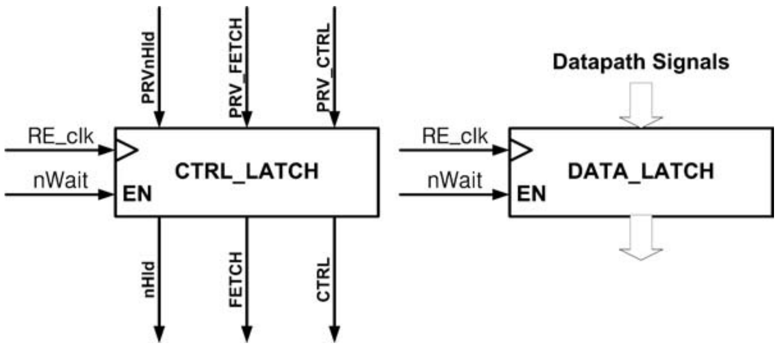


Figure 7.10 Latch control

Internal Input

- RE_nRESET: external reset signal
- InWAIT: handshaking signal
- ID1nwait: wait for the next stage

Pipeline Output

- IFnHld: hold the next stage
- IFreset: reset

Functions

- Instruction fetch control
- Instruction fetch with BEQ (bandwidth equalizer).

Signal Descriptions

Control signal	Description
REreset	0: Forced to Reset (Synchronous Reset) 1: Run
IFreset	Reset
IDnwait	0: Wait 1: Ready
IFnHld (@n)	~REreset @ ǀ IDnwait 0: Hold 1: Force to Run

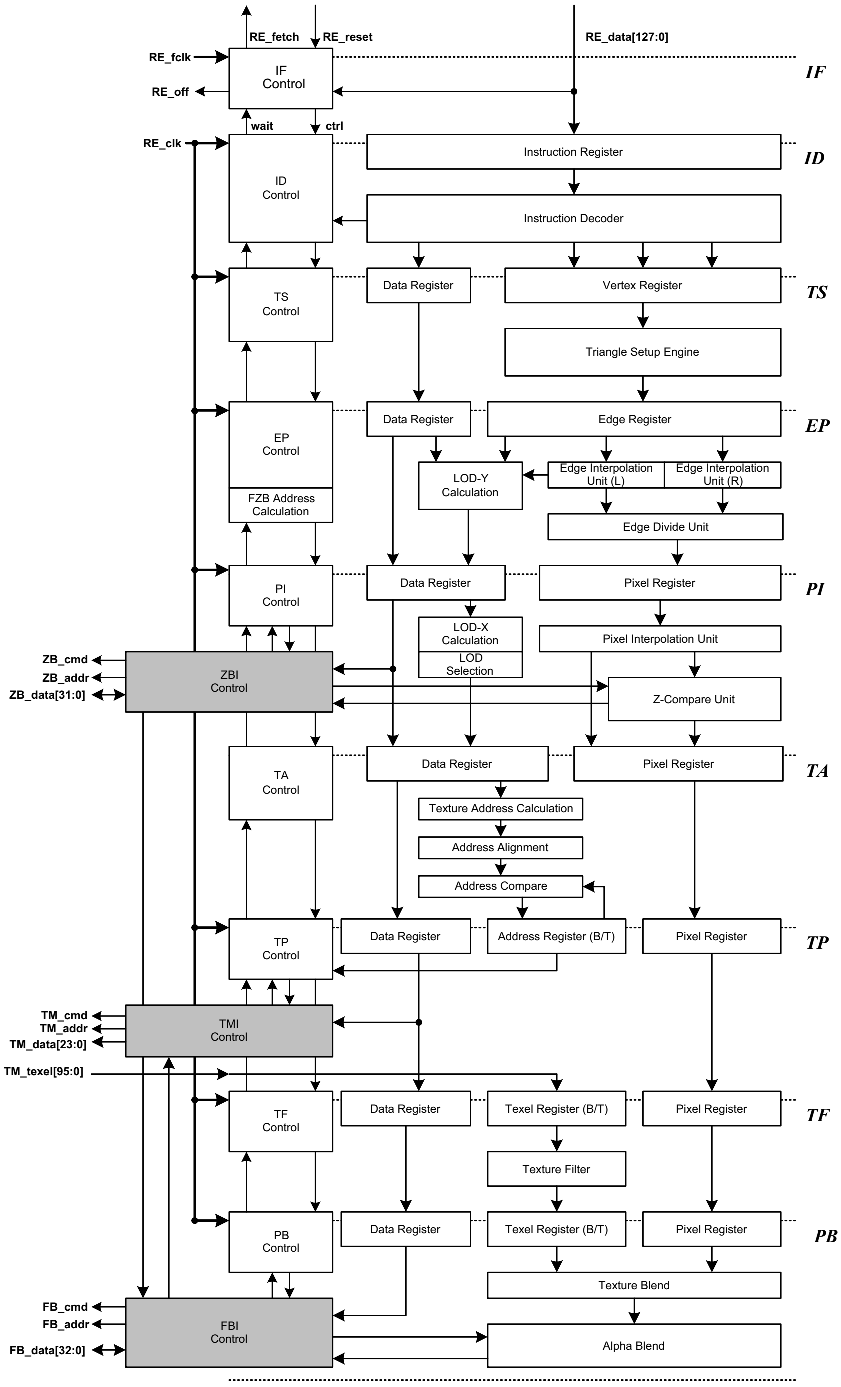


Figure 7.11 Detailed block diagram of the rasterizer

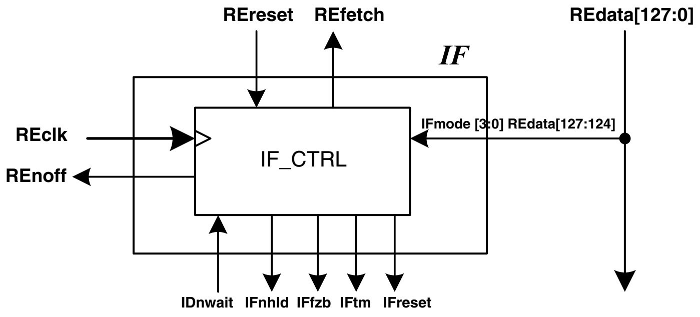


Figure 7.12 IF stage

RTL Code

```
/*
* RAMP-GR
* IF: Instruction Fetch
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/
module Wif(REclk, REreset, ID1nwait, InWAIT, IFmode,
                InREQ, IFnhld, IFreset);
           //List of ports omitted

           //InREQ is request signal for input command
           assign #1    InREQ =    ~(ID1nwait | ~IFreset);

           // If the input data is not ready, the rest of the pipeline should be stalled
           assign #1...IFnhld = InWAIT;

           always @(posedge REclk)
           begin
                IFreset <= REreset;
           end
endmodule
```

7.5.4.2 ID1: Instruction Decode Stage #1

In the ID1 stage (Figure 7.13), almost signals decoded from the latched data from IF.

Pipeline Input

- REclk: rendering engine clock
- REdata[123:0]: rendering engine data
- IFnhld: pipeline hold
- IFreset: reset
- ID2nwait: wait for the next stage

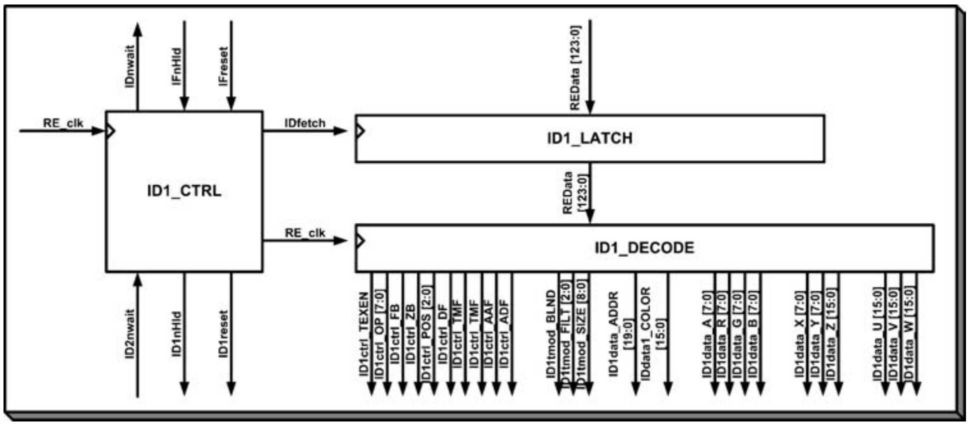


Figure 7.13 ID1 stage

Pipeline Output

- ID1nwait: wait signal
- ID1reset: reset
- ID1nhld: hold the next stage
- ID1ctrl_OP [7:0]: OP code
- ID1ctrl_FB: frame buffer flag
- ID1ctrl_ZB: depth buffer (Z buffer) flag
- ID1ctrl_POS [2:0]: pixel fetch position (input)
- ID1ctrl_DF: pixel fetch position (output)
- ID1ctrl_TMF: texture mode fetch
- ID1ctrl_AAF: auxiliary address fetch
- ID1ctrl_ADF: auxiliary data fetch
- ID1ctrl_TEXEN: texture enable status
- ID1data_ADDR [19:0]: auxiliary address
- ID1data_COLOR [15:0]: auxiliary colors
- ID1data_A [4:0]: alpha (5b)
- ID1data_R [7:0: red (5b)
- ID1data_G [7:0]: green (6b)
- ID1data_B [7:0]: blue (5b)
- ID1data_X [8:0]: coordinate X
- ID1data_Y [7:0]: coordinate Y
- ID1data_Z [14:0]: coordinate Z
- ID1data_U [15:0]: texture coordinate U
- ID1data_V [15:0]: texture coordinate V
- ID1data_W [15:0]: extra data

- ID1tmod_BLND: blend mode
- ID1tmod_FILT [2:0]: filtering mode
- ID1tmod_SIZE [8:0]: texture size

Functions

- Instruction decode
- Control signal generation

Signal Descriptions

Control signal	Description
IDreset	IFreset
ID1nwait (@n)	Rule @ Verilog
ID1nhld (@n)	Rule @ Verilog
IDfetch	REclk & IDnhld
IDctrl	Decode
IDdata	Decode
IFtmod	Decode

RTL Code

```
/*
*RAMP-GR
*ID1 : Instruction Decode #1
*by Jeong-Ho Woo ( denber@eeinfo.kaist.ac.kr)
*Semiconductor System Laboratory, KAIST
*All rights reserved
*/

module Wid1(REclk, REdata, IFreset, ID2nwait, IFnhld, ID1nwait, ID1nhld, ID1reset,
             ID1ctrl_TEXEN, ID1ctrl_OP, ID1ctrl_FB, ID1ctrl_ZB, ID1ctrl_pPOS,
             ID1ctrl_tPOS, ID1ctrl_DF, ID1ctrl_TMF, ID1ctrl_AF, ID1ctrl_FMB,
             ID1ctrl_TMB, ID1ctrl_SCR, ID1tmod_BLND, ID1tmod_FILT, ID1tmod_SIZE,
             ID1tmod_BIAS, ID1data_ADDR, ID1data_A, ID1data_R, ID1data_G,
             ID1data_B, ID1data_X, ID1data_Y, ID1data_Z, ID1data_U, ID1data_V,
             ID1data_W, RDON_clr);
             // List of port L is omitted

//Pipleine latch
Wid1_latch  s  sid1_latch();
```

```
//ID1 latch
module   Wid1_latch (clk, en, datain, dataout);
// List of port L is omitted
      always @(posedge clk)
      begin
            if(en)
            begin
                  dataout <= datain;
            end
      end
endmodule
```

// Pipeline controller
Wid1_ctrl   ssid1_ctrl();

```
// ID1 Controller
module   Wid1_ctrl (REclk, IFreset, IFnhld, ID2nwait, ID1nhld, ID1nwait,
ID1reset, ID1fetch_CLK);
      // List of port L is omitted

      //Pipeline control signals generation
      //nwait
      assign #1 ID1nwait = ID2nwait | ~IFreset;
      //Fetch enable signal. When ID1 is activated, data latch enable signal is
turned on
      assign #1 ID1fetch_CLK = ID1nwait & IFnhld;

      //Pipeline control signal latch
      always @(posedge REclk)
      begin
            //State control
            if (ID1nwait)
            begin
                  ID1nhld <= IFnhld;
                  ID1reset <= IFreset;
            end
      end
endmodule
```

// Instruction decode unit
Wid1_decode ssid1_decode( );

```
module   Wid1_decode (REclk, ID1data, ID1tir, ID1tid, ID1ctrl_TEXEN, ID1ctrl_OP,
               ID1ctrl_FB, ID1ctrl_ZB, ID1ctrl_pPOS, ID1ctrl_tPOS, ID1ctrl_DF,
               ID1ctrl_TMF, ID1ctrl_AF, ID1ctrl_FMB, ID1ctrl_TMB, ID1ctrl_SCR,
               ID1tmod_BLND, ID1tmod_FILT, ID1tmod_BIAS,
               ID1tmod_SIZE, ID1data_ADDR, ID1data_A, ID1data_R,
               ID1data_G, ID1data_B, ID1data_X, ID1data_Y, ID1data_Z,
               ID1data_U, ID1data_V, ID1data_W, RDON_clr);

   //Instruction field decode
   assign #1 ID1data_ADDR = (ID1data[123:118] == 'WISA_MBAS) ? ID1data[81:64] :
18'b0;
   assign #1 ID1data_W = (ID1data[123:118] == 'WISA_RTEX) ? ID1data[96:64] :
32'b0;
   assign #1 ID1data_U = (ID1data[123:118] == 'WISA_RTEX) ? ID1data[63:32] :
32'b0;
   assign #1  ID1data_V = (ID1data[123:118] == 'WISA_RTEX) ? ID1data[31:0] : 32'b0;
   assign #1  ID1data_X = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[95:87] : 9'b0;
   assign #1 ID1data_Y = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[86:79] : 8'b0;
   assign #1 ID1data_Z = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[111:96] :
16'b0;
   assign #1 ID1data_R = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[63:56] : 8'b0;
   assign #1 ID1data_G = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[55:48] : 8'b0;
   assign #1 ID1data_B = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[47:40] : 8'b0;
   assign #1 ID1data_A = (ID1data[123:118] == 'WISA_RSHA) ? ID1data[39:32] : 8'b0;

   //Texture mode
   assign #1 ID1tmod_BLND = (ID1data[123:118] == 'WISA_TMOD) ? ID1data[93] : 1'b0;
   assign #1 ID1tmod_FILT = (ID1data[123:118] == 'WISA_TMOD) ? ID1data[90:88]:
3'b0;
   assign #1 ID1tmod_BIAS = (ID1data[123:118] == 'WISA_TMOD) ? ID1data[79:76]:
4'b0;
   assign #1 ID1tmod_SIZE = (ID1data[123:118] == 'WISA_TMOD) ? ID1data[72:64]:
9'b0;

   //Control
   assign #1  ID1ctrl_FB = ({ID1data[121], ID1data[118]} == 2'b11) ? ID1data[117] :
1'b0;
   assign #1  ID1ctrl_ZB = ({ID1data[121], ID1data[118]} == 2'b11) ? ID1data[116] :
1'b0;
   assign #1 ID1ctrl_pPOS_int = ({ID1data[123], ID1data[119:118]} == 3'b100) ?
ID1data[114:112] : 3'b000; //RSHA
   assign #1 ID1ctrl_tPOS_int = ({ID1data[123], ID1data[119:118]} == 3'b101) ?
ID1data[114:112] : 3'b000; //RTEX
   assign #1 ID1ctrl_DF_int = ({ID1data[123], ID1data[119:118]} == 3'b100) ?
ID1data[116] : 1'b0; //RSHA
   assign #1 ID1ctrl_TMF_int = ID1data[122] & ID1data[118];
   assign #1 ID1ctrl_AF_int = ID1data[121] & ~ID1data[118];
   assign #1 ID1ctrl_FMB_int = ({ID1data[121], ID1data[118]} == 2'b10) ?
ID1data[116] : 1'b0;
   assign #1 ID1ctrl_TMB_int = ({ID1data[121], ID1data[118]} == 2'b10) ?
   ID1data[117] : 1'b0;
   assign #1 RDON_clr = (ID1ctrl_OP[3]) ? 1'b0 : 1'b1;

   //Screen color depth
   always @(ID1data)
   begin
         if(ID1data[123:118] == 'WISA_TMOD) ID1ctrl_SCR <= ID1data[115:112];
         else                               ID1ctrl_SCR <= 4'b0;
   end

   //Command decoder
   always @(ID1data)
   begin
          case(ID1data[123:118])
                'WISA_RSHA : ID1ctrl_OP <= 'WCTRL_OP_RSHA;
                'WISA_RTEX : ID1ctrl_OP <= 'WCTRL_OP_RTEX;
                'WISA_RDON : ID1ctrl_OP <= 'WCTRL_OP_RDON;
                'WISA_TMOD : ID1ctrl_OP <= 'WCTRL_OP_TMOD;
                'WISA_MBAS : ID1ctrl_OP <= 'WCTRL_OP_MBAS;
                'WISA_RCLR : ID1ctrl_OP <= 'WCTRL_OP_RCLR;
                default    : ID1ctrl_OP <= 6'b0;
```

```
            endcase
    end

    //Texture enable
    always @(posedge REclk)
    begin
            if (ID1tir == ‘WISA_TMOD) ID1ctrl_TEXEN_int <= |ID1tid;
    end

    //Fetch signals
    always @(negedge REclk)
    begin
            ID1ctrl_DF       <= ID1ctrl_DF_int;
            ID1ctrl_TMF      <= ID1ctrl_TMF_int;
            ID1ctrl_AF       <= ID1ctrl_AF_int;
            ID1ctrl_FMB      <= ID1ctrl_FMB_int;
            ID1ctrl_TMB      <= ID1ctrl_TMB_int;
            ID1ctrl_TEXEN    <= ID1ctrl_TEXEN_int;
            ID1ctrl_pPOS     <= ID1ctrl_pPOS_int;
            ID1ctrl_tPOS     <= ID1ctrl_tPOS_int;
    end
endmodule
```

endmodule

7.5.4.3 ID2: Instruction Decode Stage #2

In the ID2 stage (Figure 7.14), the vertex data from ID1 are stored in order. According to the vertex alignment mode – STRIP, FAN, NORMAL – three vertices are gathered in the ID2 stage. After gathering three vertices, then the rendering operation is started.

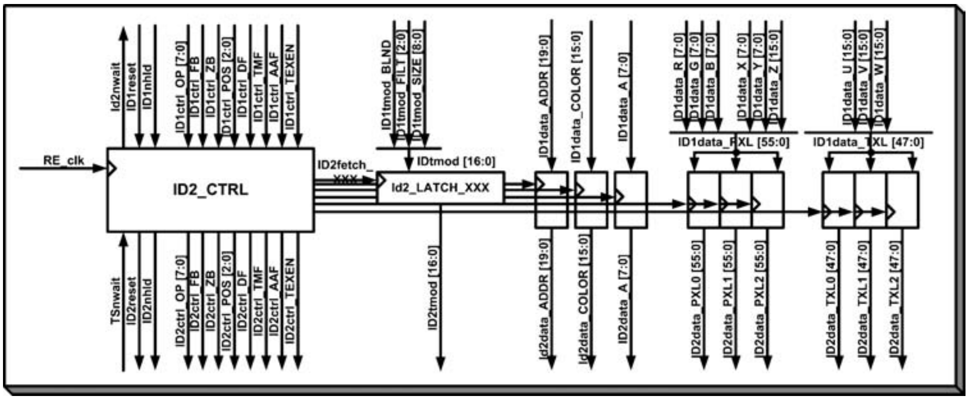


Figure 7.14 ID2 stage

Pipeline Input

- REclk: rendering engine clock
- ID1reset: reset
- ID1hld: hold
- TSnwait: wait for the next stage
- ID1ctrl_OP [7:0]: OP code
- ID1ctrl_FB: FB flag
- ID1ctrl_ZB: ZB flag
- ID1ctrl_POS [2:0]: pixel fetch position
- ID1ctrl_DF: pixel output control
- ID1ctrl_TMF: texture mode fetch
- ID1ctrl_AAF: auxiliary address fetch
- ID1ctrl_ADF: auxiliary data fetch
- ID1cttl_TEXEN: texture enable
- ID1tmod [16:0] {BLND, FILT, BIAS, SIZE}
- ID1data_ADDR [19:0]: auxiliary address
- ID1data_COLOR [15:0]: auxiliary color
- ID1data_A [7:0]: alpha (5b)
- ID1data_PXL [55:0]: {R, G, B, X, Y, Z}
- ID1data_TXL [47:0]: {U, V, W}

Pipeline Output

- ID2nwait: wait
- ID2reset: reset
- ID2nhld: hold
- ID2ctrl_OP [7:0]: OP code
- ID2ctrl_FB: FB flag
- ID2ctrl_ZB: ZB flag
- ID2ctrl_DF: pixel fetch
- ID2ctrl_TMF: texture mode fetch
- ID2ctrl_AAF: auxiliary address fetch
- ID2ctrl_ADF: auxiliary data fetch
- ID2ctrl_TEXEN: texture enable
- ID2tmod [16:0]: texture mode
- ID2data_ADDR [19:0]: auxiliary address
- ID2data_COLOR [15:0]: auxiliary colors
- ID2data_PXL0 [55:0]: pixel data 0 {X, Y, Z, R, G, B}
- ID2data_PXL1 [55:0]: pixel data 1
- ID2data_PXL2 [55:0]: pixel data 2
- ID2data_TXL0 [47:0]: texel data 0

- ID2data_TXL0 [47:0]: texel data 1
- ID2data_TXL0 [47:0]: texel data 2

Functions

- Data fetch into corresponding latch
- Isolation between TS and BEQ

Signal Descriptions

Signals	Description
ID2nwait	ID2nwait_gen = (∼ID2ctrl_DF & ID1nhld & ID2OP[7])
	ID2nwait = ID2nwait_gen | TSnwait
ID2nhld	Rule @ Verilog
ID2reset	Fetch @ Normal
	Cannot be activated with "RDAT"
ID2ctrl_TMF	ID1ctrl_TMF & ID1ctrl_TEXEN
ID2fetch_CLK	ID1nhld & REclk
ID2fetch_TMOD	ID1ctrl_TMF & ID1ctrl_TEXEN
ID2fetch_COLOR	ID1ctrl_ADF
ID2fetch_ALPHA	ID1ctrl_DF
ID2fetch_PXL [#]	ID1ctrl_POS [#]
	#: Vertex position
ID2fetch_TXL [#]	ID1ctrl_POX [#] & ID1ctrl_TEXEN

RTL Code

```
/*
* RAMP-GR
* RAMP-GR ID2 Module : Instruction DECODER #2
* by Jeong-Ho Hoo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wid2 (REclk, ID1reset, ID1nhld, TSnwait, ID1ctrl_OP, ID1ctrl_FB, ID1ctrl_ZB,
           ID1ctrl_pPOS, ID1ctrl_tPOS, ID1ctrl_DF, ID1ctrl_TMF, ID1ctrl_AF,
           ID1ctrl_TEXEN, ID1ctrl_FMB, ID1ctrl_TMB, ID1ctrl_SCR, ID1tmod_BLND,
           ID1tmod_FILT, ID1tmod_BIAS, ID1tmod_SIZE, ID1data_ADDR, ID1data_A,
           ID1data_R, ID1data_G, ID1data_B, ID1data_X, ID1data_Y, ID1data_Z,
           ID1data_U, ID1data_V, ID1data_W, ID2nwait, ID2reset, ID2nhld,
           ID2ctrl_OP, ID2ctrl_FB, ID2ctrl_ZB, ID2ctrl_pPOS, ID2ctrl_tPOS,
ID2ctrl_DF,
           ID2ctrl_TMF, ID2ctrl_AF, ID2ctrl_TEXEN, ID2ctrl_FMB,
           ID2ctrl_TMB,ID2ctrl_SCR, ID2tmod, ID2data_ADDR, ID2data_A,
           ID2dataPXL0, ID2dataPXL1, ID2dataPXL2, ID2dataTXL0, ID2dataTXL1,
           ID2dataTXL2);
```

```
        //List of Ports is omitted
        // Gathering
        assign #1 ID1tmod = {ID1tmod_BLND , ID1tmod_FILT, ID1tmod_BIAS,
ID1tmod_SIZE};
        assign #1 ID1dataPXL = {ID1data_R, ID1data_G, ID1data_B, ID1data_X,
ID1data_Y, ID1data_Z};
        assign #1 ID1dataTXL = {ID1data_U, ID1data_V, ID1data_W};
```

// Main controller
Wid2_ctrl   ssid2_ctrl();

```
// Main controller
module    Wid2_ctrl( REclk, ID1reset, ID1nhld, ID1ctrl_OP,
          ID1ctrl_FB, ID1ctrl_ZB, ID1ctrl_pPOS,
          ID1ctrl_tPOS, ID1ctrl_DF, ID1ctrl_TMF, ID1ctrl_AF, ID1ctrl_TEXEN,
          ID1ctrl_FMB, ID1ctrl_TMB, ID1ctrl_SCR, TSnwait, ID2nwait, ID2reset,
          ID2nhld, ID2ctrl_OP, ID2ctrl_FB, ID2ctrl_ZB, ID2ctrl_DF, ID2ctrl_TMF,
          ID2ctrl_AF, ID2ctrl_TEXEN, ID2ctrl_FMB, ID2ctrl_TMB, ID2ctrl_SCR,
          ID2fetch_CLK, ID2fetch_TMOD, ID2fetch_ADDR, ID2fetch_ALPHA,
          ID2fetch_PXL0, ID2fetch_PXL1, ID2fetch_PXL2, ID2fetch_TXL0,
          ID2fetch_TXL1, ID2fetch_TXL2);

        //Pipeline control signal
        //nwait
        assign #1 ID2nwait_gen = ID2nhld_int & (ID2ctrl_OP[5] | ID2ctrl_OP[4]) &
~ID2ctrl_DF_int;
        assign #1 ID2nwait = ID2nwait_gen | TSnwait | ~ID1reset;
        //nHld ( NOP for DF == 0)
        assign #1 ID2nhld_gen = ~(ID2nhld_int & (ID2ctrl_OP [5] | ID2ctrl_OP[4]) &
~ID2ctrl_DF_int);
        assign #1 ID2nhld = ID2nhld_int & ID2nhld_gen;

        // ID2 latch fetch enable signal generation
        assign #1  ID2fetch_CLK = ID2nwait & ID1nhld;
        assign  ID2fetch_TMOD = ID1ctrl_TMF & ID1ctrl_TEXEN;
        assign  ID2fetch_ADDR = ID1ctrl_AF;
        assign  ID2fetch_ALPHA = ID1ctrl_DF;
        assign  ID2fetch_PXL0 = ID1ctrl_pPOS[2];
        assign  ID2fetch_PXL1 = ID1ctrl_pPOS[1];
        assign  ID2fetch_PXL2 = ID1ctrl_pPOS[0];
        assign  ID2fetch_TXL0 = ID1ctrl_tPOS[2] & ID1ctrl_TEXEN;
        assign  ID2fetch_TXL1 = ID1ctrl_tPOS[1] & ID1ctrl_TEXEN;
        assign  ID2fetch_TXL2 = ID1ctrl_tPOS[0] & ID1ctrl_TEXEN;

        always @(posedge REclk)
        begin
             //Control signals
             if(ID2nwait)
             begin
             //State controls
                  ID2nhld_int <= ID1nhld;
                  ID2reset <= ID1reset;
                  //Fetch signals
```

```
                ID2ctrl_DF_int <= ID1ctrl_DF;
                ID2ctrl_TMF_int <= ID1ctrl_TMF & ID1ctrl_TEXEN;
                ID2ctrl_AF_int <= ID1ctrl_AF;
                ID2ctrl_TEXEN_int <= ID1ctrl_TEXEN;
                ID2ctrl_FMB_int <= ID1ctrl_FMB;
                ID2ctrl_TMB_int <= ID1ctrl_TMB;
                ID2ctrl_OP <= ID1ctrl_OP;
                ID2ctrl_FB <= ID1ctrl_FB;
                ID2ctrl_ZB <= ID1ctrl_ZB;
                ID2ctrl_SCR<= ID1ctrl_SCR;
            end
        end

        always @(negedge REclk)
        begin
            ID2ctrl_DF <= ID2ctrl_DF_int & ID2nhld;
            ID2ctrl_TMF <= ID2ctrl_TMF_int & ID2nhld;
            ID2ctrl_AF <= ID2ctrl_AF_int & ID2nhld;
            ID2ctrl_TEXEN <= ID2ctrl_TEXEN_int & ID2nhld;
            ID2ctrl_FMB <= ID2ctrl_FMB_int & ID2nhld;
            ID2ctrl_TMB <= ID2ctrl_TMB_int & ID2nhld;
        end
endmodule
```

```
// Latch
Wid2_latch   ssid2_latch();
```

```
// Data latches
module    Wid2_latch(clk, REclk, ID2fetch_TMOD, ID2fetch_ADDR, ID2fetch_ALPHA,
                ID2fetch_PXL0, ID2fetch_PXL1, ID2fetch_PXL2,
                ID2fetch_TXL0, ID2fetch_TXL1, ID2fetch_TXL2,
                ID1tmod, ID1data_ADDR, ID1data_A, ID1dataPXL, ID1dataTXL,
                ID2tmod, ID2data_ADDR, ID2data_A,
                ID2dataPXL0, ID2dataPXL1, ID2dataPXL2,
                ID2dataTXL0, ID2dataTXL1, ID2dataTXL2);

        // ID2_LATCH_TMOD
        always @(posedge REclk)
        begin
            if(clk & ID2fetch_TMOD)
            begin
                ID2tmod <= ID1tmod;
            end
        end

        // ID2_LATCH_ADDR
        always @(posedge REclk)
            if(clk & ID2fetch_ADDR) ID2data_ADDR <= ID1data_ADDR;

        // ID2_LATCH_ALPHA
        always @(posedge REclk)
            if(clk & ID2fetch_ALPHA)      ID2data_A <= ID1data_A;
```

```
          // ID2_LATCH_PXL#
          always @(posedge REclk)
               if(clk & ID2fetch_PXL0) ID2dataPXL0 <= ID1dataPXL;
          always @(posedge REclk)
               if(clk & ID2fetch_PXL1) ID2dataPXL1 <= ID1dataPXL;

          always @(posedge REclk)
               if(clk & ID2fetch_PXL2) ID2dataPXL2 <= ID1dataPXL;

          // ID2_LATCH_TXL#
          always @(posedge REclk)
               if(clk & ID2fetch_TXL0) ID2dataTXL0 <= ID1dataTXL;
          always @(posedge REclk)
               if(clk & ID2fetch_TXL1) ID2dataTXL1 <= ID1dataTXL;
          always @(posedge REclk)
               if(clk & ID2fetch_TXL2) ID2dataTXL2 <= ID1dataTXL;
 endmodule
```

```
endmodule
```

7.5.4.4 TS: Triangle Setup

In the TS stage (Figure 7.15), the order of vertices is decided according to the *y* coordinate. From the top pixel, the other two pixels are lined up, then the shape of the triangle is decided.

Pipeline Input

- REclk: rendering engine clock
- Control signals: same as ID2
- Auxiliary signals: same as ID2
- Pixel/texel data: same as ID2

Pipeline Output

- Control signals: same as ID2
- Auxiliary signals: same as ID2
- TSaddrY [7:0]: top Y address for interpolation
- TSline [7:0]: number of lines needed for EP iteration
- TSedge: edge flag needed for left/right decision
- TScount0_1_2: edge count in vertical direction
- TSePXL0_1_2 [92:0]: dP {dX, dZ, dR, dG, dB}
- TSeTXL0_1_2 [74:0]: dZ, dU, dV
- TSvPXL0_1_2 [55:0]: pixel {X, Z, R, G, B}
- TSvTXL0_1_2 [47:0]: texel {U, V, W}

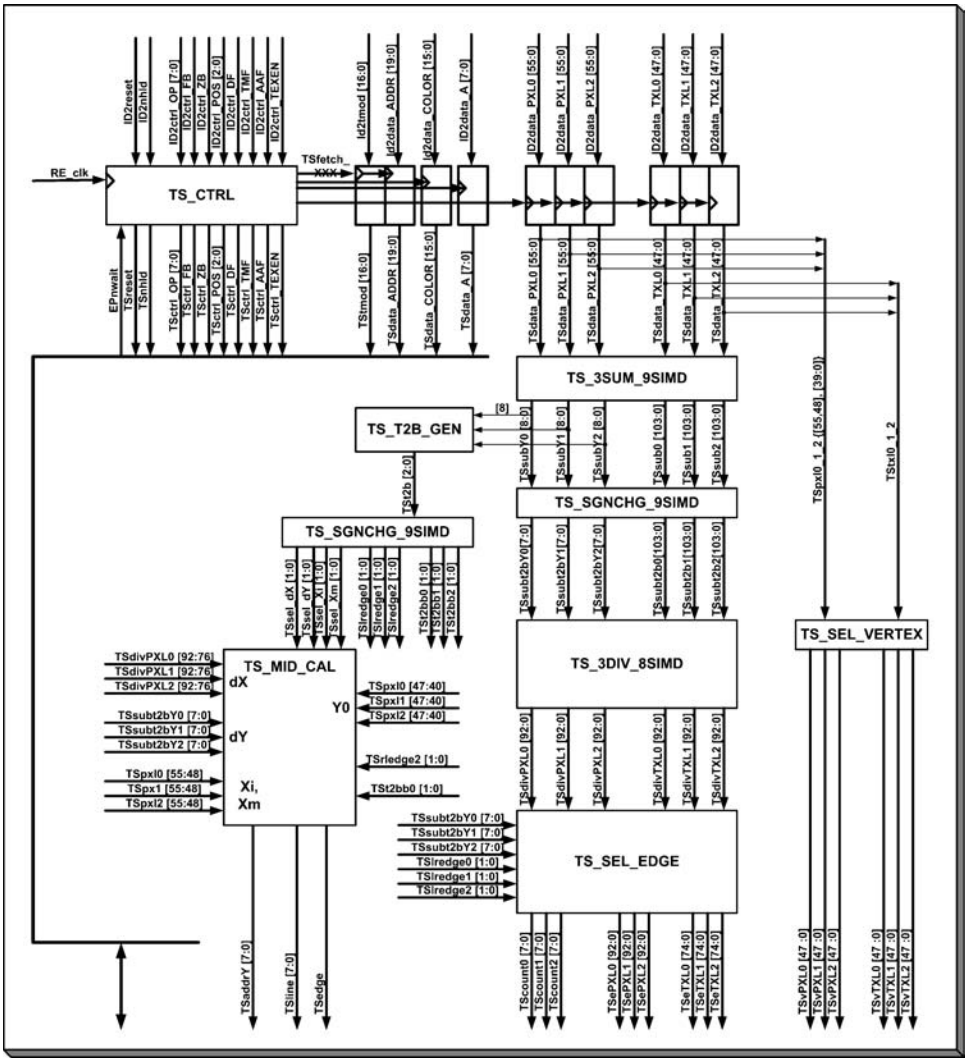


Figure 7.15 Triangle setup

Functions

- Triangle setup
- EP control signal generation

Signal Descriptions

Control signals	Description
All	Rule @ Verilog Following ID2 & ID

Datapath signals	Description
TSpxl #	{X[55:47], Y[46:39], Z[38:15], R[24:16], G[15:8], B[7:0]}
TXtxl #	{U[46:32], V[31:16], W[15:0]}

TSsubY#	Y: 9-bit signed data
	Input: {0, data} – {0, data}
	Output: {Sign, data}

TSsub#	{X[95:86], Z[85:70], R[69:64], G[63:57], B[56:51]
	U[50:34], V[33:17], W[16:0]}
	X:10-bit signed data
	R(9), G(9), B(9): signed data {0, data}-{0,data} = {s, data}
	Z: 16-bit signed data
	U, V, W: 17-bit Signed data {0, data}-{0,data} = {s, data}

TSsubt2bY#	Y: 8-bit positive data
TSsubt2b#	Same as TSsub#

TSdivPXL#	{X[92:76], Z[75:51], R[50:34], G[33:17], B[32:0]}
	X, R, G, B: 9-bit signed data {sign, data, fraction}
	Z: 25-bit signed data {sign, data, fraction}

TSdivTXL#	{U[74:50], V[49:25], W[24:0]}
	U, V, W: 25sbit signed data {sign, data, fraction}

Datapath #3: TS_MID_CAL

This is the datapath for mid point computation.

RTL Code

```
/*
* RAMP-GR
* RAMP-GR TS Module    : Triangle Setup Module
* by Jeong-Ho Woo ( denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wts ( REclk, ID2reset, ID2nhld, ID2ctrl_OP, ID2ctrl_FB, ID2ctrl_ZB,
                ID2ctrl_DF, ID2ctrl_TMF, ID2ctrl_AF, ID2ctrl_TEXEN,
                ID2ctrl_FMB, ID2ctrl_TMB, ID2ctrl_SCR, ID2tmod, ID2data_ADDR,
                ID2data_A, ID2dataPXL0, ID2dataPXL1, ID2dataPXL2,
                ID2dataTXL0,  ID2dataTXL1, ID2dataTXL2,
                EPnwait, TSreset,
                TSnhld, TSnwait,  TScOP, TScFB, TScZB, TScDF, TScTMF, TScAF,
                TScTEXEN, TScFMB,  TScTMB, TScSCR, TStmod,
                TSdADDR, TSdA, TSaddrY, TSline, TSedge,
                TScount0, TScount1, TScount2,
                TSePXL0, TSePXL1, TSePXL2, TSeTXL0, TSeTXL1, TSeTXL2,
                TSvPXL0, TSvPXL1, TSvPXL2, TSvTXL0, TSvTXL1, TSvTXL2);
```

```
//Main controller
Wts_ctrl        ssts_ctrl();
```

```
//Main controller
module     Wts_ctrl ( REclk, ID2reset, ID2nhld, ID2ctrl_OP, ID2ctrl_FB,
           ID2ctrl_ZB, ID2ctrl_DF, ID2ctrl_TMF, ID2ctrl_AF, ID2ctrl_TEXEN,
           ID2ctrl_FMB, ID2ctrl_TMB, ID2ctrl_SCR,
           EPnwait, TSreset,
           TSnhld, TSnwait, TScOP, TScFB, TScZB, TScDF, TScTMF,
           TScAF, TScTEXEN, TScFMB, TScTMB, TScSCR,
           TSfetch_CLK, TSfetch_TMOD, TSfetch_ADDR,
           TSfetch_ALPHA, TSfetch_PXL, TSfetch_TXL);

     //Pipeline control signals
     //nWait
     assign #1   TSnwait = EPnwait | ~ID2reset;
     //nHld
     assign #1   TSnhld = TSnhld_int;
     //Fetch enable
     assign #1   TSfetch_CLK = TSnwait & ID2nhld;
     assign    TSfetch_TMOD = ID2ctrl_TMF;
     assign    TSfetch_ADDR = ID2ctrl_AF;
     assign    TSfetch_ALPHA = ID2ctrl_DF;
     assign    TSfetch_PXL = ID2ctrl_DF;
     assign    TSfetch_TXL = ID2ctrl_DF & ID2ctrl_TEXEN;

     always @(posedge REclk)
     begin
           //Pipeline control signals
           if(TSnwait)
           begin
                //State control
                TSnhld_int <= ID2nhld;
                TSreset <= ID2reset;
                //Fetch signals
                TScDF_int <= ID2ctrl_DF;
                TScTMF_int <= ID2ctrl_TMF;
                TScAF_int <= ID2ctrl_AF;
                TScTEXEN_int <= ID2ctrl_TEXEN;
                TScFMB_int <= ID2ctrl_FMB;
                TScTMB_int <= ID2ctrl_TMB;
                TScOP <= ID2ctrl_OP;
                TScFB <= ID2ctrl_FB;
                TScZB <= ID2ctrl_ZB;
                TScSCR <= ID2ctrl_SCR;
           end
     end

     always @(negedge REclk)
     begin
           TScDF <= TScDF_int & TSnhld;
           TScTMF <= TScTMF_int & TSnhld;
           TScAF <= TScAF_int & TSnhld;
```

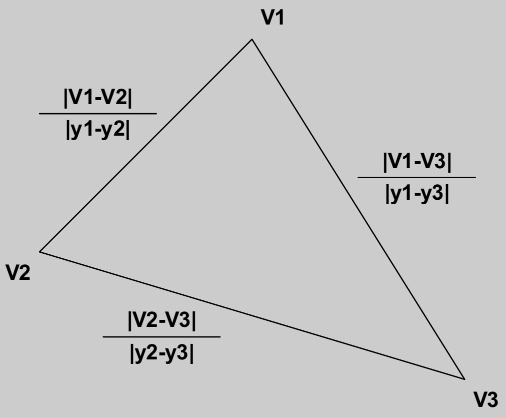
```
            TScTEXEN <= TScTEXEN_int & TSnhld;
            TScFMB <= TScFMB_int & TSnhld;
            TScTMB <= TScTMB_int & TSnhld;
     end
endmodule
```

```
//Input latches
Wts_latch      ssts_latch();
```

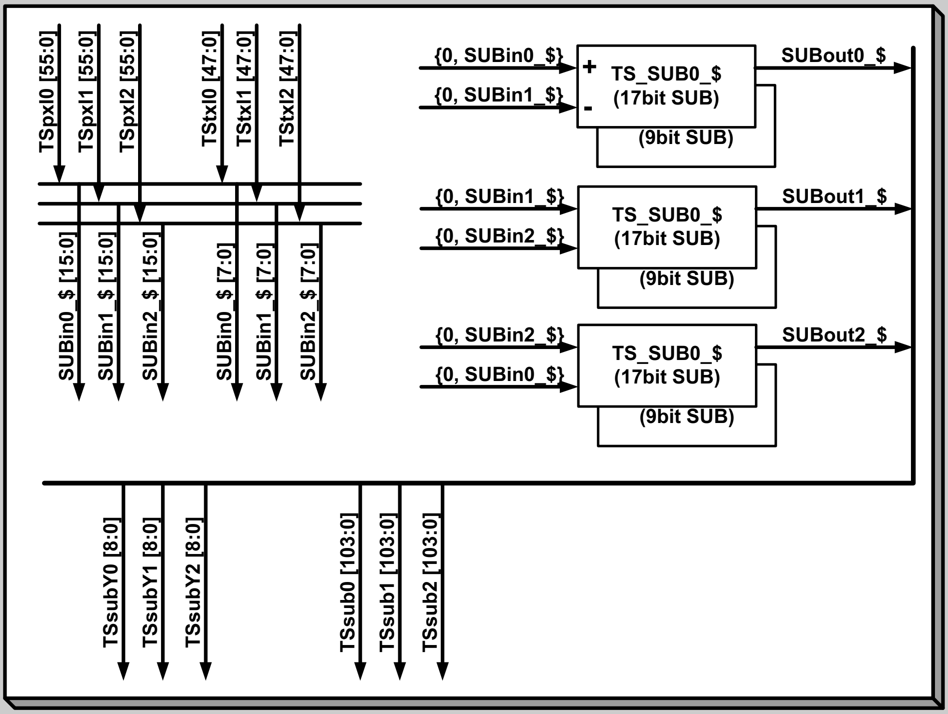
```
//Input Latches
module      Wts_latch ( clk, REclk, TSfetch_TMOD, TSfetch_ADDR, TSfetch_ALPHA,
                 TSfetch_PXL, TSfetch_TXL, ID2tmod, ID2data_ADDR,
                 ID2data_A, ID2dataPXL0, ID2dataPXL1, ID2dataPXL2,
                 ID2dataTXL0, ID2dataTXL1, ID2dataTXL2,
                 TStmod, TSdADDR, TSdA,
                 TSpxl0, TSpxl1, TSpxl2, TStxl0, TStxl1, TStxl2);

     //TS_LATCH_TMOD
     always @(posedge REclk)
            if(clk & TSfetch_TMOD) TStmod <= ID2tmod;

     //TS_LATCH_ADDR
     always @(posedge REclk)
            if(clk & TSfetch_ADDR) TSdADDR <= ID2data_ADDR;

     //TS_LATCH_ALPHA
     always @(posedge REclk)
            if(clk & TSfetch_ALPHA) TSdA <= ID2data_A;

     //TS_LATCH_PXL {R:G:B:X:Y:Z => X:Y:Z:R:G:B}
     always @(posedge REclk)
     begin
            if(clk & TSfetch_PXL)
            begin
                 TSpxl0 <= {ID2dataPXL0[32:0], ID2dataPXL0[56:33]};
                 TSpxl1 <= {ID2dataPXL1[32:0], ID2dataPXL1[56:33]};
                 TSpxl2 <= {ID2dataPXL2[32:0], ID2dataPXL2[56:33]};
            end
     end

     //TS_LATCH_TXL
     always @(posedge REclk)
     begin
            if(clk & TSfetch_TXL)
            begin
                 TStxl0 <= ID2dataTXL0;
                 TStxl1 <= ID2dataTXL1;
                 TStxl2 <= ID2dataTXL2;
            end
     end
endmodule
```

//Datapath interconnection

```
//Datapath interconnection
//These three datapaths (Wts_3sub_9simd, Wts_sgnchg_9simd, Wts_3div_8simd)
//compute the slope of all possible edges.
//Let us assume the three vertices are V1, V2, and V3
// V1{x1, y1, z1, w1, r1, g1, b1, u1, v1}
// V2{x2, y2, z2, w2, r2, g2, b2, u2, v2}
// V3{x3, y3, z3, w3, r3, g3, b3, u3, v3}
// The results of these three datapath are shown in diagram A.
```



```
//The sign bit of the results of substraction of y decides
//the order of verticies according to y position.

     //3 x 9-way SIMD SUB
     Wts_3sub_9simd  ssts_3sub_9simd();
     //3SUB 9SIMD

module     Wts_3sub_9simd(  TSpxl0, TSpxl1, TSpxl2, TStxl0, TStxl1, TStxl2,
                 TSsubY0, TSsubY1, TSsubY2, TSsub0, TSsub1, TSsub2);

     /* Module interconnection */
     //TS_SUB0 : V0- V1
     DW01_sub #(10)    sub0_X(.A  ({1'b0, subin0_X}),
                       .B  ({1'b0, subin1_X}),
                       .CI  (1'b0),
                       .DIFF  (subout0_X),
                       .CO  (c0X));
     DW01_sub #(9)     sub0_Y(.A  ({1'b0, subin0_Y}),
                       .B  ({1'b0, subin1_Y}),
                       .CI  (1'b0),
                       .DIFF  (subout0_Y),
                       .CO  (c0Y));
     DW01_sub #(17)    sub0_Z(.A  ({1'b0, subin0_Z}),
                       .B  ({1'b0, subin1_Z}),
                       .CI  (1'b0),
                       .DIFF  (subout0_Z),
                       .CO  (c0Z));
     DW01_sub #(9)     sub0_R(.A  ({1'b0, subin0_R}),
                       .B  ({1'b0, subin1_R}),
                       .CI  (1'b0),
                       .DIFF  (subout0_R),
                       .CO  (c0R));
     DW01_sub #(9)     sub0_G(.A  ({1'b0, subin0_G}),
                       .B  ({1'b0, subin1_G}),
                       .CI  (1'b0),
                       .DIFF  (subout0_G),
                       .CO  (c0G));
     DW01_sub #(9)     sub0_B(.A  ({1'b0, subin0_B}),
                       .B  ({1'b0, subin1_B}),
                       .CI  (1'b0),
                       .DIFF  (subout0_B),
                       .CO  (c0B));
     DW01_sub #(33)    sub0_U(.A  ({1'b0, subin0_U}),
                       .B  ({1'b0, subin1_U}),
                       .CI  (1'b0),
                       .DIFF  (subout0_U),
                       .CO  (c0U));
     DW01_sub #(33)    sub0_V(.A  ({1'b0, subin0_V}),
                       .B  ({1'b0, subin1_V}),
                       .CI  (1'b0),
                       .DIFF  (subout0_V),
                       .CO  (c0V));
     DW01_sub #(33)    sub0_W(.A  ({1'b0, subin0_W}),
                       .B  ({1'b0, subin1_W}),
                       .CI  (1'b0),
                       .DIFF  (subout0_W),
                       .CO  (c0W));

     //TS_SUB1 : V1- V2
     // The same connection with TS_SUB0

     //TS_SUB2 : V2- V3
     // The same connection with TS_SUB0

     /* Output grouping */
     assign     TSsubY0 = subout0_Y;
     assign     TSsubY1 = subout1_Y;
     assign     TSsubY2 = subout2_Y;
     assign     TSsub0  = {subout0_X, subout0_Z, subout0_R, subout0_G, sub-
out0_B,
                subout0_U,subout0_V, subout0_W};
     assign     TSsub1  = {subout1_X, subout1_Z, subout1_R, subout1_G, sub-
out1_B,
                subout1_U, subout1_V, subout1_W};
     assign     TSsub2  = {subout2_X, subout2_Z, subout2_R, subout2_G, sub-
out2_B,
                subout2_U, subout2_V, subout2_W};

/* See diagram B */
```



```
endmodule
```

```
//3 x 9-way SIMD Sign-Change
Wts_sgnchg_9simd  ssts_sgnchg_9simd();
```

```
module    Wts_sgnchg_9simd(  TSsubY0, TSsubY1, TSsubY2, TSsub0, TSsub1, TSsub2,
                   TSsubt2bY0, TSsubt2bY1, TSsubt2bY2,
                   TSsubt2b0, TSsubt2b1, TSsubt2b2);

     /* Input ungrouping XORing with YSIGN */
     assign    in0_Y = TSsubY0[7:0] ^ {8{TSsubY0[8]}};
     assign    in1_Y = TSsubY1[7:0] ^ {8{TSsubY1[8]}};
     assign    in2_Y = TSsubY2[7:0] ^ {8{TSsubY2[8]}};

     assign    {in0_X, in0_Z, in0_R, in0_G, in0_B, in0_U, in0_V, in0_W}
                 = TSsub0 ^ {153{TSsubY0[8]}};
     assign    {in1_X, in1_Z, in1_R, in1_G, in1_B, in1_U, in1_V, in1_W}
                 = TSsub1 ^ {153{TSsubY1[8]}};
     assign    {in2_X, in2_Z, in2_R, in2_G, in2_B, in2_U, in2_V, in2_W}
                 = TSsub2 ^ {153{TSsubY2[8]}};

     /* Module interconnection */
     // SIGNCHG 0
     DW01_add #(10)    add0_X(    .A    (10'b0),
                                      .B  (in0_X),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_X),
                                      .CO  (c0X));
      DW01_add #(8)      add0_Y(     .A    (8'b0),
                                      .B  (in0_Y),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (TSsubt2bY0),
                                      .CO  (c0Y));
      DW01_add #(17)     add0_Z(     .A    (17'b0),
                                      .B  (in0_Z),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_Z),
                                      .CO  (c0Z));
      DW01_add #(9)      add0_R(     .A    (9'b0),
                                      .B  (in0_R),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_R),
                                      .CO  (c0R));
      DW01_add #(9)      add0_G(     .A    (9'b0),
                                      .B  (in0_G),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_G),
                                      .CO  (c0G));
      DW01_add #(9)      add0_B(     .A    (9'b0),
                                      .B  (in0_B),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_B),
                                      .CO  (c0B));
      DW01_add #(33)     add0_U(     .A    (33'b0),
                                      .B  (in0_U),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_U),
                                      .CO  (c0U));
      DW01_add #(33)     add0_V(     .A    (33'b0),
                                      .B  (in0_V),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_V),
                                      .CO  (c0V));
      DW01_add #(33)     add0_W(     .A    (33'b0),
                                      .B  (in0_W),
                                      .CI  (TSsubY0[8]),
                                      .SUM  (out0_W),
                                      .CO  (c0W));


      // SIGNCHG 1 :
      // The same with SIGNCHG0

      // SIGNCHG 2
      // The same with SIGNCHG0

      /* Output grouping */
      //TSsubt2b is irrevelant to T2B variable in C model.
```

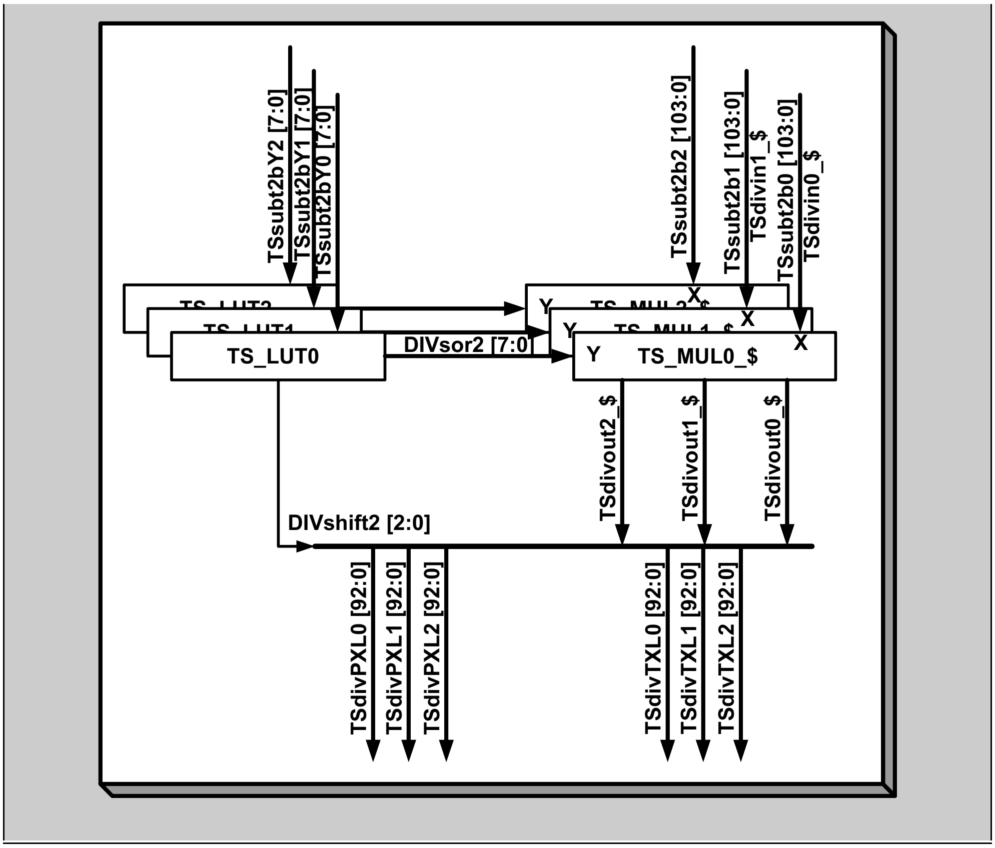
```
     assign TSsubt2b0 = {out0_X, out0_Z, out0_R, out0_G, out0_B, out0_U, out0_V,
out0_W};
     assign TSsubt2b1 = {out1_X, out1_Z, out1_R, out1_G, out1_B, out1_U, out1_V,
out1_W};
     assign TSsubt2b2 = {out2_X, out2_Z, out2_R, out2_G, out2_B, out2_U, out2_V,
out2_W};

endmodule
```

// 3 x 8-way SIMD Divider

Wts_3div_8simd ssts_3div_8simd();

```
     //3DIV 8SIMD
     module Wts_3div_8simd(  TSsubt2bY0, TSsubt2bY1, TSsubt2bY2, TSsubt2b0,
               TSsubt2b1, TSsubt2b2, TSdivPXL0, TSdivPXL1, TSdivPXL2,
               TSdivTXL0, TSdivTXL1, TSdivTXL2);

     /* Input ungrouping */
     assign  {TSdivin0_X, TSdivin0_Z, TSdivin0_R, TSdivin0_G, TSdivin0_B,
TSdivin0_U, TSdivin0_V, TSdivin0_W} = TSsubt2b0;
     assign  {TSdivin1_X, TSdivin1_Z, TSdivin1_R, TSdivin1_G, TSdivin1_B,
TSdivin1_U, TSdivin1_V, TSdivin1_W} = TSsubt2b1;
     assign  {TSdivin2_X, TSdivin2_Z, TSdivin2_R, TSdivin2_G, TSdivin2_B,
TSdivin2_U, TSdivin2_V, TSdivin2_W} = TSsubt2b2;

     /* Divide look-up table */
     // x-> LUT -> 1/X ( Mantisa , exponent)
     WDIVLUT8        lut0(  .tbin       (TSsubt2bY0),
                            .tbout      (DIVsel0),
                            .tbshift    (DIVshift0));
     WDIVLUT8        lut1(  .tbin       (TSsubt2bY1),
                            .tbout      (DIVsel1),
                            .tbshift    (DIVshift1));
     WDIVLUT8        lut2(  .tbin       (TSsubt2bY2),
                            .tbout      (DIVsel2),
                            .tbshift    (DIVshift2));


     /* Bunch of multipliers */
     //MUL0
     WMUL10x8      mul0_X(  .x  (TSdivin0_X),
                            .y  (DIVsel0),       //mantisa 8b
                            .z  (TSdivout0_X));  // 17bit
     WMUL17x8      mul0_Z(  .x  (TSdivin0_Z),
                            .y  (DIVsel0),       //mantisa 8b
                            .z  (TSdivout0_Z));  //exponent 3b
     WMUL9x8       mul0_R(  .x  (TSdivin0_R),
                            .y  (DIVsel0),       //mantisa 8b
                            .z  (TSdivout0_R));  //exponent 3b
     WMUL9x8       mul0_G(  .x  (TSdivin0_G),
                            .y  (DIVsel0),       //mantisa 8b
                            .z  (TSdivout0_G));  //exponent 3b
     WMUL9x8        mul0_B(  .x  (TSdivin0_B),
                             .y  (DIVsel0),       //mantisa 8b
                             .z  (TSdivout0_B));  //exponent 3b
     WMUL33x8       mul0_U(  .x  (TSdivin0_U),
                             .y  (DIVsel0),       //mantisa 8b
                             .z  (TSdivout0_U));  //exponent 3b
     WMUL33x8       mul0_V(  .x  (TSdivin0_V),
                             .y  (DIVsel0),       //mantisa 8b
                             .z  (TSdivout0_V));  //exponent 3b
     WMUL33x8       mul0_W(  .x  (TSdivin0_W),
                             .y  (DIVsel0),       //mantisa 8b
                             .z  (TSdivout0_W));  //exponent 3b


     //MUL 1
     WMUL10x8   mul1_X();
     WMUL17x8   mul1_Z();
     WMUL9x8    mul1_R();
     WMUL9x8    mul1_G();
     WMUL9x8    mul1_B();
     WMUL33x8   mul1_U();
     WMUL33x8   mul1_V();
     WMUL33x8   mul1_W();


     //MUL 2
     WMUL10x8   mul2_X();
     WMUL17x8   mul2_Z();
     WMUL9x8    mul2_R();
     WMUL9x8    mul2_G();
     WMUL9x8    mul2_B();
     WMUL33x8   mul2_U();
     WMUL33x8   mul2_V();
     WMUL33x8   mul2_W();


     //Shift 0
     Wts_shift_18   shift0_X(.din (TSdivout0_X), .dout (TSdivout0_Xr), .shift
(DIVshift0));
     Wts_shift_25   shift0_Z(.din (TSdivout0_Z), .dout (TSdivout0_Zr), .shift
(DIVshift0));
     Wts_shift_17   shift0_R(.din (TSdivout0_R), .dout (TSdivout0_Rr), .shift
(DIVshift0));
     Wts_shift_17   shift0_G(.din (TSdivout0_G), .dout (TSdivout0_Gr), .shift
(DIVshift0));
     Wts_shift_17   shift0_B(.din (TSdivout0_B), .dout (TSdivout0_Br), .shift
(DIVshift0));
     Wts_shift_41   shift0_U(.din (TSdivout0_U), .dout (TSdivout0_Ur), .shift
(DIVshift0));
     Wts_shift_41   shift0_V(.din (TSdivout0_V), .dout (TSdivout0_Vr), .shift
(DIVshift0));
     Wts_shift_41   shift0_W(.din (TSdivout0_W), .dout (TSdivout0_Wr), .shift
(DIVshift0));
```

```
     //Shift 1
     Wts_shift_18   shift1_X(.din (TSdivout1_X), .dout (TSdivout1_Xr), .shift
(DIVshift1));
     Wts_shift_25   shift1_Z(.din (TSdivout1_Z), .dout (TSdivout1_Zr), .shift
(DIVshift1));
     Wts_shift_17   shift1_R(.din (TSdivout1_R), .dout (TSdivout1_Rr), .shift
(DIVshift1));
     Wts_shift_17   shift1_G(.din (TSdivout1_G), .dout (TSdivout1_Gr), .shift
(DIVshift1));
     Wts_shift_17   shift1_B(.din (TSdivout1_B), .dout (TSdivout1_Br), .shift
(DIVshift1));
     Wts_shift_41   shift1_U(.din (TSdivout1_U), .dout (TSdivout1_Ur), .shift
(DIVshift1));
     Wts_shift_41   shift1_V(.din (TSdivout1_V), .dout (TSdivout1_Vr), .shift
(DIVshift1));
     Wts_shift_41   shift1_W(.din (TSdivout1_W), .dout (TSdivout1_Wr), .shift
(DIVshift1));


     //Shift 2
     Wts_shift_18   shift2_X(.din (TSdivout2_X), .dout (TSdivout2_Xr), .shift
(DIVshift2));
     Wts_shift_25   shift2_Z(.din (TSdivout2_Z), .dout (TSdivout2_Zr), .shift
(DIVshift2));
     Wts_shift_17   shift2_R(.din (TSdivout2_R), .dout (TSdivout2_Rr), .shift
(DIVshift2));
     Wts_shift_17   shift2_G(.din (TSdivout2_G), .dout (TSdivout2_Gr), .shift
(DIVshift2));
     Wts_shift_17   shift2_B(.din (TSdivout2_B), .dout (TSdivout2_Br), .shift
(DIVshift2));
     Wts_shift_41   shift2_U(.din (TSdivout2_U), .dout (TSdivout2_Ur), .shift
(DIVshift2));
     Wts_shift_41   shift2_V(.din (TSdivout2_V), .dout (TSdivout2_Vr), .shift
(DIVshift2));
     Wts_shift_41   shift2_W(.din (TSdivout2_W), .dout (TSdivout2_Wr), .shift
(DIVshift2));


     /* Output grouping */
     //Pixel
     assign   TSdivPXL0 = {TSdivout0_Xr, TSdivout0_Zr,
                 TSdivout0_Rr, TSdivout0_Gr, TSdivout0_Br};
     assign   TSdivPXL1 = {TSdivout1_Xr, TSdivout1_Zr,
                 TSdivout1_Rr, TSdivout1_Gr, TSdivout1_Br};
     assign   TSdivPXL2 = {TSdivout2_Xr, TSdivout2_Zr,
                 TSdivout2_Rr, TSdivout2_Gr, TSdivout2_Br};


     //Texel
     assign   TSdivTXL0  = {TSdivout0_Ur, TSdivout0_Vr, TSdivout0_Wr};
     assign   TSdivTXL1  = {TSdivout1_Ur, TSdivout1_Vr, TSdivout1_Wr};
     assign   TSdivTXL2  = {TSdivout2_Ur, TSdivout2_Vr, TSdivout2_Wr};
     //See diagram C.
     endmodule
```



```
//Using the results of the substraction of the y position,
//the datapaths T2B_GEN, T2B_SELGEN, SEL_EDGE, and SEL_VERTEX
//decide the order of vertices and slopes according to the y position.
```

// TS_T2B_GEN
Wts_t2b_gen ssts_t2b_gen()

```
//T2B_GEN
module   Wts_t2b_gen(Y0, Y1, Y2, TSt2b);
      input         Y0; // Sign of each substraction of |y0-y1|, |y1-y2|, |y0-y2|
      input         Y1;
      input         Y2;
      output [2:0]  TSt2b;

      /* Register */
      reg              [2:0]     TSt2b;

      always @(Y0 or Y1 or Y2)
      begin
      casex({Y0, Y1, Y2})
            //Means the order of vertices according to y
            //Sort vertices by y position from top to bottom
            3'b00X:  TSt2b <= `WT2B_210; // V2, V1, V0
            3'b010:  TSt2b <= `WT2B_102; // V1, V0, V2
            3'b011:  TSt2b <= `WT2B_120; // V1, V2, V0
            3'b100:  TSt2b <= `WT2B_021; // V0, V2, V1
            3'b101:  TSt2b <= `WT2B_201; // V2, V0, V1
            3'b11X:  TSt2b <= `WT2B_012; // V0, V1, V2
            default: TSt2b <= 3'bX;
      endcase
      end
endmodule
```

// TS_T2B_SELGEN : Vertex and slope slection signal generation
Wts_t2b_selgen ssts_t2b_selgen();

```
//T2B SELGEN
module   Wts_t2b_selgen(  TSt2b, TSsel_dX, TSsel_dY, TSsel_Xi,
               TSsel_Xm, TSlredge0, TSlredge1, TSlredge2,
               TSt2bb0, TSt2bb1, TSt2bb2);

      always @(TSt2b)
      begin
           //TSsel_dX
           casex(TSt2b)
                 `WT2B_210  : TSsel_dX <= 2'd2;
                 `WT2B_102  : TSsel_dX <= 2'd1;
                 `WT2B_120  : TSsel_dX <= 2'd0;
                 `WT2B_021  : TSsel_dX <= 2'd0;
                 `WT2B_201  : TSsel_dX <= 2'd1;
                 `WT2B_012  : TSsel_dX <= 2'd2;
                 default    : TSsel_dX <= 2'bX;
           endcase
           //TSsel_dY
           casex(TSt2b)
                 `WT2B_210  : TSsel_dY <= 2'd1;
                 `WT2B_102  : TSsel_dY <= 2'd0;
                 `WT2B_120  : TSsel_dY <= 2'd1;
                 `WT2B_021  : TSsel_dY <= 2'd2;
                 `WT2B_201  : TSsel_dY <= 2'd2;
                 `WT2B_012  : TSsel_dY <= 2'd0;
                 default    : TSsel_dY <= 2'bX;
           endcase
           //TSsel_Xi
           casex(TSt2b)
                 `WT2B_210  : TSsel_Xi <= 2'd2;
                 `WT2B_102  : TSsel_Xi <= 2'd1;
                 `WT2B_120  : TSsel_Xi <= 2'd1;
                 `WT2B_021  : TSsel_Xi <= 2'd0;
                 `WT2B_201  : TSsel_Xi <= 2'd2;
                 `WT2B_012  : TSsel_Xi <= 2'd0;
                 default    : TSsel_Xi <= 2'bX;
           endcase
           //TSsel_Xm
           casex(TSt2b)
                 `WT2B_210  : TSsel_Xm <= 2'd1;
                 `WT2B_102  : TSsel_Xm <= 2'd0;
                 `WT2B_120  : TSsel_Xm <= 2'd2;
```

```
            'WT2B_021  : TSsel_Xm <= 2'd2;
            'WT2B_201  : TSsel_Xm <= 2'd0;
            'WT2B_012  : TSsel_Xm <= 2'd1;
            default    : TSsel_Xm <= 2'bX;
      endcase
      //TSlredge0
      casex(TSt2b)
            'WT2B_210  : TSlredge0 <= 2'd1;
            'WT2B_102  : TSlredge0 <= 2'd0;
            'WT2B_120  : TSlredge0 <= 2'd1;
            'WT2B_021  : TSlredge0 <= 2'd2;
            'WT2B_201  : TSlredge0 <= 2'd2;
            'WT2B_012  : TSlredge0 <= 2'd0;
            default    : TSlredge0 <= 2'bX;
      endcase
      //TSlredge1
      casex(TSt2b)
            'WT2B_210  : TSlredge1 <= 2'd0;
            'WT2B_102  : TSlredge1 <= 2'd2;
            'WT2B_120  : TSlredge1 <= 2'd2;
            'WT2B_021  : TSlredge1 <= 2'd1;
            'WT2B_201  : TSlredge1 <= 2'd0;
            'WT2B_012  : TSlredge1 <= 2'd1;
            default    : TSlredge1 <= 2'bX;
      endcase

      //TSlredge2
      casex(TSt2b)
            'WT2B_210  : TSlredge2 <= 2'd2;
            'WT2B_102  : TSlredge2 <= 2'd1;
            'WT2B_120  : TSlredge2 <= 2'd0;
            'WT2B_021  : TSlredge2 <= 2'd0;
            'WT2B_201  : TSlredge2 <= 2'd1;
            'WT2B_012  : TSlredge2 <= 2'd2;
            default    : TSlredge2 <= 2'bX;
      endcase
      //TSt2bb0
      casex(TSt2b)
            'WT2B_210  : TSt2bb0 <= 2'd2;
            'WT2B_102  : TSt2bb0 <= 2'd1;
            'WT2B_120  : TSt2bb0 <= 2'd1;
            'WT2B_021  : TSt2bb0 <= 2'd0;
            'WT2B_201  : TSt2bb0 <= 2'd2;
            'WT2B_012  : TSt2bb0 <= 2'd0;
            default    : TSt2bb0 <= 2'bX;
      endcase
      //TSt2bb1
      casex(TSt2b)
            'WT2B_210  : TSt2bb1 <= 2'd1;
            'WT2B_102  : TSt2bb1 <= 2'd0;
            'WT2B_120  : TSt2bb1 <= 2'd2;
            'WT2B_021  : TSt2bb1 <= 2'd2;
            'WT2B_201  : TSt2bb1 <= 2'd0;
            'WT2B_012  : TSt2bb1 <= 2'd1;
            default    : TSt2bb1 <= 2'bX;
```

```
          endcase
          //TSt2bb2
          casex(TSt2b)
               `WT2B_210  : TSt2bb2 <= 2'd0;
               `WT2B_102  : TSt2bb2 <= 2'd2;
               `WT2B_120  : TSt2bb2 <= 2'd0;
               `WT2B_021  : TSt2bb2 <= 2'd1;
               `WT2B_201  : TSt2bb2 <= 2'd1;
               `WT2B_012  : TSt2bb2 <= 2'd2;
               default    : TSt2bb2 <= 2'bX;
          endcase
     end
//Results of the T2B_SELGEN are shown in the chart.
endmodule
```

T2B	dX	dY	Xi	Xm		
T2B_210	2	1	2	1		
T2B_102	1	0	1	0		
T2B_120	0	1	1	2		
T2B_021	0	2	0	2		
T2B_201	1	2	2	0		
T2B_012	2	0	0	1		
T2B	LREDGE0	LREDGE1	LREDGE2	T2BB0	T2BB1	T2BB2
T2B_210	1	0	2	2	1	0
T2B_102	0	2	1	1	0	2
T2B_120	1	2	0	1	2	0
T2B_021	2	1	0	0	2	1
T2B_201	2	0	1	2	0	1
T2B_012	0	1	2	0	1	2

//TS_SEL_EDGE
Wts_sel_edge   ssts_sel_edge();

```
//SEL EDGE
module   Wts_sel_edge(TSlredge0, TSlredge1, TSlredge2, TSsubt2bY0,
               TSsubt2bY1, TSsubt2bY2, TSdivPXL0,
               TSdivPXL1, TSdivPXL2, TSdivTXL0,
               TSdivTXL1, TSdivTXL2,
               TScount0, TScount1, TScount2,
               TSePXL0, TSePXL1, TSePXL2,
               TSeTXL0, TSeTXL1, TSeTXL2);

     //Y count selection
     //This Y count value controls multi-cycle operation
     always @(TSlredge0 or TSlredge1 or TSlredge2 or TSsubt2bY0 or TSsubt2bY1
                    or TSsubt2bY2)
     begin
          //Count0
          casex(TSlredge0)
               2'd0       : TScount0 <= TSsubt2bY0;
               2'd1       : TScount0 <= TSsubt2bY1;
```

```
                2'd2        : TScount0 <= TSsubt2bY2;
                default     : TScount0 <= 8'bX;
           endcase
           //Count1
           casex(TSlredge1)
                2'd0        : TScount1 <= TSsubt2bY0;
                2'd1        : TScount1 <= TSsubt2bY1;
                2'd2        : TScount1 <= TSsubt2bY2;
                default     : TScount1 <= 8'bX;
           endcase
           //Count2
           casex(TSlredge2)
                2'd0        : TScount2 <= TSsubt2bY0;
                2'd1        : TScount2 <= TSsubt2bY1;
                2'd2        : TScount2 <= TSsubt2bY2;
                default     : TScount2 <= 8'bX;
           endcase
      end

      //PXL
      always @(TSlredge0 or TSlredge1 or TSlredge2 or TSdivPXL0 or TSdivPXL1 or
TSdivPXL2)
      begin
           //PXL 0
           casex(TSlredge0)
                2'd0        : TSePXL0 <= TSdivPXL0;
                2'd1        : TSePXL0 <= TSdivPXL1;
                2'd2        : TSePXL0 <= TSdivPXL2;
                default     : TSePXL0 <= 94'bX;
           endcase
           //PXL 1
           casex(TSlredge1)
                2'd0        : TSePXL1 <= TSdivPXL0;
                2'd1        : TSePXL1 <= TSdivPXL1;
                2'd2        : TSePXL1 <= TSdivPXL2;
                default     : TSePXL1 <= 94'bX;
           endcase
           //PXL 2
           casex(TSlredge2)
                2'd0        : TSePXL2 <= TSdivPXL0;
                2'd1        : TSePXL2 <= TSdivPXL1;
                2'd2        : TSePXL2 <= TSdivPXL2;
                default     : TSePXL2 <= 94'bX;
           endcase
      end

      //TXL
      always @(TSlredge0 or TSlredge1 or TSlredge2 or TSdivTXL0 or TSdivTXL1 or
TSdivTXL2)
      begin
           //PXL 0
           casex(TSlredge0)
                2'd0        : TSeTXL0 <= TSdivTXL0;
                2'd1        : TSeTXL0 <= TSdivTXL1;
                2'd2        : TSeTXL0 <= TSdivTXL2;
                default    : TSeTXL0 <= 123'bX;
            endcase
            //PXL 1
            casex(TSlredge1)
                2'd0       : TSeTXL1 <= TSdivTXL0;
                2'd1       : TSeTXL1 <= TSdivTXL1;
                2'd2       : TSeTXL1 <= TSdivTXL2;
                default    : TSeTXL1 <= 123'bX;
            endcase
            //PXL 2
            casex(TSlredge2)
                2'd0       : TSeTXL2 <= TSdivTXL0;
                2'd1       : TSeTXL2 <= TSdivTXL1;
                2'd2       : TSeTXL2 <= TSdivTXL2;
                default    : TSeTXL2 <= 123'bX;
            endcase
    end
//Results are shown in the chart.
endmodule
```

Output	SEL	Input
TScount#	TSlredge #	TSsubt2bY#
TSePXL#	TSlredge #	TSdivPXL#
TSeTXL#	TSlredge #	TSdivTXL#

//TS_SEL_VERTEX
Wts_sel_vertex   ssts_sel_vertex();

```
//TS_SEL_VERTEX
//Huge power consumption due to large bit switching
Module    Wts_sel_vertex(  TSt2bb0, TSt2bb1, TSt2bb2, TSpxl0, TSpxl1, TSpxl2,
                TStxl0, TStxl1, TStxl2,
                TSvPXL0, TSvPXL1, TSvPXL2, TSvTXL0,
                TSvTXL1, TSvTXL2);

     /* MUX */
     //PXL
     always @(TSt2bb0 or TSt2bb1 or TSt2bb2 or TSpxl0 or TSpxl1 or TSpxl2)
     begin
          //PXL0
          casex(TSt2bb0)
               2'd0          : TSvPXL0 <= TSpxl0;
               2'd1          : TSvPXL0   <= TSpxl1;
               2'd2          : TSvPXL0   <= TSpxl2;
               default       : TSvPXL0   <= 49'bX;
          endcase
     //PXL1
          casex(TSt2bb1)
               2'd0          : TSvPXL1 <= TSpxl0;
               2'd1          : TSvPXL1   <= TSpxl1;
               2'd2          : TSvPXL1   <= TSpxl2;
               default       : TSvPXL1   <= 49'bX;
          endcase
          //PXL2
          casex(TSt2bb2)
               2'd0           : TSvPXL2 <= TSpxl0;
               2'd1           : TSvPXL2   <= TSpxl1;
               2'd2           : TSvPXL2 <= TSpxl2;
               default        : TSvPXL2   <= 49'bX;
          endcase
     end

     //TXL
     always @(TSt2bb0 or TSt2bb1 or TSt2bb2 or TStxl0 or TStxl1 or TStxl2)
     begin
          //TXL0
          casex(TSt2bb0)
               2'd0           : TSvTXL0 <= TStxl0;
               2'd1           : TSvTXL0   <= TStxl1;
               2'd2           : TSvTXL0   <= TStxl2;
               default        : TSvTXL0   <= 96'bX;
          endcase
          //TXL1
          casex(TSt2bb1)
               2'd0           : TSvTXL1 <= TStxl0;
               2'd1           : TSvTXL1      <= TStxl1;
               2'd2           : TSvTXL1      <= TStxl2;
               default        : TSvTXL1      <= 96'bX;
          endcase
          //TXL2
          casex(TSt2bb2)
               2'd0           : TSvTXL2 <= TStxl0;
               2'd1           : TSvTXL2   <= TStxl1;
               2'd2           : TSvTXL2 <= TStxl2;
               default        : TSvTXL2   <= 96'bX;
          endcase
     end
//Results are shown in the chart.
endmodule
```

Output	SEL	Input
TSvPXL#	TSt2bb#	TSpxl#
TSvTXL#	TSt2bb#	TStxl#

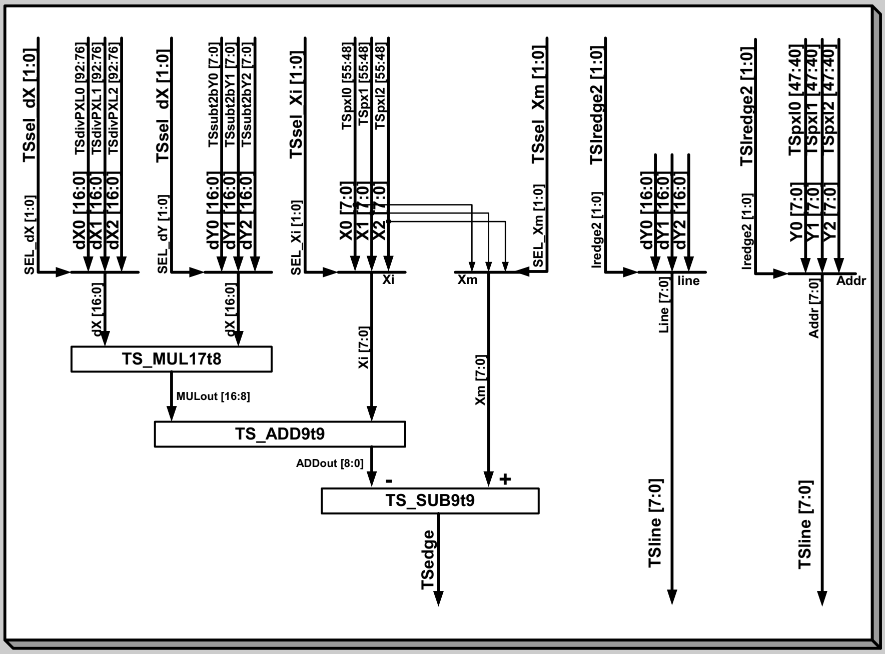
//Since the middle point of the polygon, the edge should be changed.

//The MID_CAL unit computes the middle point of the vertex and controls the change of the edges
Wts_mid_cal   ssts_mid_cal();

```
//TS_MID_CAL
module     Wts_mid_cal(TSsel_dX, TSsel_dY, TSsel_Xi, TSsel_Xm,
                TSdivPXL0, TSdivPXL1, TSdivPXL2, TSsubt2bY0,
                TSsubt2bY1, TSsubt2bY2,
                TSpxl0_X, TSpxl1_X, TSpxl2_X,
                TSpxl0_Y, TSpxl1_Y, TSpxl2_Y,
                TSlredge2, TSt2bb0, TSedge, TSline, TSaddrY);
```

```
/* MUX */
//dX
always @(TSsel_dX or TSdivPXL0 or TSdivPXL1 or TSdivPXL2)
begin
     casex(TSsel_dX)
          2'd0           : dX <= TSdivPXL0;
          2'd1           : dX <= TSdivPXL1;
          2'd2           : dX <= TSdivPXL2;
          default        : dX <= 18'bX;
     endcase
end

//dY
always @(TSsel_dY or TSsubt2bY0 or TSsubt2bY1 or TSsubt2bY2)
begin
     casex(TSsel_dY)
          2'd0           : dY <= TSsubt2bY0;
          2'd1           : dY <= TSsubt2bY1;
          2'd2           : dY <= TSsubt2bY2;
          default        : dY <= 8'bX;
     endcase
end

//Xi
always @(TSsel_Xi or TSpxl0_X or TSpxl1_X or TSpxl2_X)
begin
     casex(TSsel_Xi)
          2'd0           : Xi <= TSpxl0_X;
          2'd1           : Xi <= TSpxl1_X;
          2'd2           : Xi <= TSpxl2_X;
          default        : Xi <= 9'bX;
     endcase
end

//Xm
always @(TSsel_Xm or TSpxl0_X or TSpxl1_X or TSpxl2_X)
begin
     casex(TSsel_Xm)
          2'd0           : Xm <= TSpxl0_X;
          2'd1           : Xm <= TSpxl1_X;
          2'd2           : Xm <= TSpxl2_X;
          default        : Xm <= 9'bX;
     endcase
end

//TSline
always @(TSlredge2 or TSsubt2bY0 or TSsubt2bY1 or TSsubt2bY2)
begin
     casex(TSlredge2)
          2'd0           : TSline <= TSsubt2bY0;
          2'd1           : TSline <= TSsubt2bY1;
          2'd2           : TSline <= TSsubt2bY2;
          default        : TSline <= 8'bX;
     endcase
end
```

```
      //TSaddrY
      always @(TSt2bb0 or TSpxl0_Y or TSpxl1_Y or TSpxl2_Y)
      begin
           casex(TSt2bb0)
                2'd0          : TSaddrY <= TSpxl0_Y;
                2'd1          : TSaddrY <= TSpxl1_Y;
                2'd2          : TSaddrY <= TSpxl2_Y;
                default       : TSaddrY <= 8'bX;
           endcase
      end

      /* Module interconnection */
      WMUL18x8        TS_MUL18t8(  .x      (dX),
                                   .y      (dY),
                                   .z      (MULout));
      DW01_add #(10)  TS_ADD9t9 (   .A      (MULout[17:8]),
                                   .B      ({1'b0, Xi}),
                                   .C      (1'b0),
                                   .SUM    (ADDout),
                                   .CO     (ADDcout));
      DW01_sub #(10)  TS_SUB9t9 (   .A      ({1'b0, Xm}),
                                   .B      (ADDout),
                                   .CI     (1'b0),
                                   .DIFF   (EDGEout),
                                   .CO     (EDGEcout));

      /* Edge assignment: MSB of EDGEout */
      assign  TSedge = EDGEout[9];
//See diagram D and the chart.
endmodule
```



Datapath signals	Description
MULout	Multiplier output: 18-bit {s, d 8, f8}
	MULout = {sign, data8}
ADDout	Adder input: 10-bit { 0, data8}
	ADDout = 10-bit Adder output {s, d8}
edge	MSB of TS_SUB10t10 (sign: minus)

```
endmodule
```

7.5.4.5 EP: Edge Setup Processor

In the EP stage (Figure 7.16), the pixels on the slope of three edges of the polygon are computed. The slope of the edge is used to compute edge pixel values, like this:

$$(N+1)\text{th pixel} = N\text{th pixel} + \text{slope of the edge.}$$

Pipeline Input

- REclk: rendering engine clock
- Control signals: same as TS
- Auxiliary data: same as TS
- Pixel/texel data: same as TS

Pipeline Output

- Control signals: same as TS
- Auxiliary data: same as TS
- EPpxlY [7:0]: Y address
- EPpxlL [47:0]: left edge pixel data
- EPtxlL [47:0]: left edge texel data
- EPpxlR [47:0]: right edge pixel data
- EPtxlR [47:0]: right edge texel data
- EPcntX [8:0]: edge processor X-count (determine PP cycle)
- EPpxlX [8:0]: edge processor X0-address for horizontal intp.
- EPdivPXL [75:0]: EP-div pixel data
- EPdivTXL [74:0]: EP-div texel data

Functions

- Edge processor interpolation
- Multicycle operation management

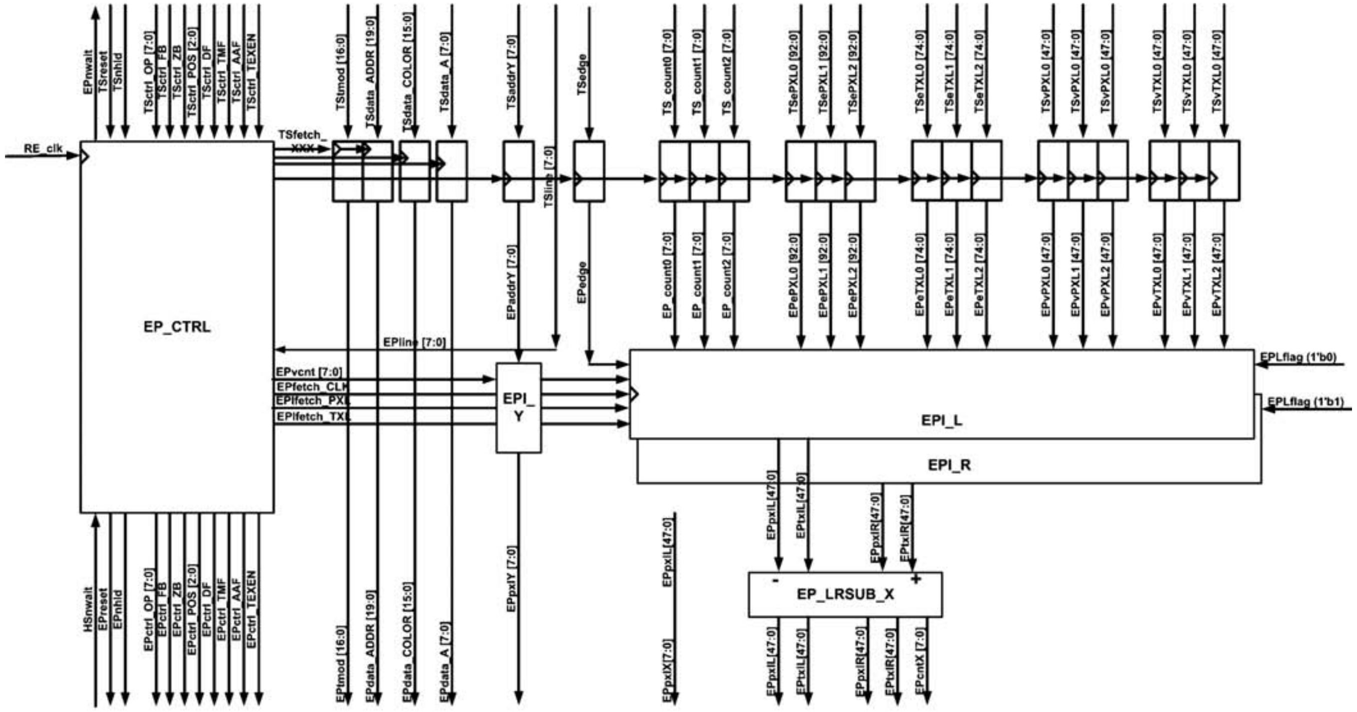


Figure 7.16 Edge processor

Signal Descriptions

Control signals	Description
Main	Rule @ Verilog following ID2 & ID & TS Supporting multicycle wait – generation
EPline	Setting-up EP vertical counter Used for EP cycle control
EPvcnt	Count starts from zero to EPline @ (REclk & nHld)

Datapath signals	Description

TSePXL#	{X[92:76],Z[75:51],R[50:34],G[33:17],B[16:0]}
	X, R, G, B: 17-bit signed data
	{sign_1b, data_8b, frac_8b}
	Z: 24-bit signed data
	{sign_1b, data_15b, frac_8b}

TSeTXL#	{U[74:50],V[49:25],W[24:0]}
	25-bit signed data
	{sign_1b, data_16b, frac_8b}

TSvPXL#	{X[47:39],Z[38:24],R[23:16],G[15:8],B[7:0]} unsigned data
TSvTXL#	{U[47:32],V[31:16],W[15:0]} unsigned data
EPpxlY	Y address 8-bit unsigned data
EPpxlLR	{X[47:39],Z[38:24],R[23:16],G[15:8],B[7:0]} X, R, G, B: 8bit unsigned data Z: 16-bit unsigned data
EPtxlLR	{U[47:32],V[31:16],W[15:0]} U, V, W: 16-bit unsigned data
EPcntX	Edge processor X-count - Determines the number of PP operation cycles 8-bit unsigned data

Datapath: EPI

RTL Code

```
/*
* RAMP-GR
* RAMP-GR  EP Module  :  Edge Process
* by Jeong-Ho Woo ( denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

Module Wep(  REclk, TSreset, TSnhld, HSnwait, TScOP, TScFB, TScZB, TScDF,
             TScTMF, TScAF, TScTEXEN, TScFMB, TScTMB, TScSCR, TStmod,
             TSdADDR, TSdA, TSaddrY, TSline, TSedge, TScount0, TScount2,
           TSePXL0, TSePXL1, TSePXL2, TSeTXL0, TSeTXL1, TSeTXL2,
           TSvPXL0, TSvPXL1, TSvPXL2, TSvTXL0, TSvTXL1, TSvTXL2,
           EPnwait, EPreset, EPnhld, EPcOP, EPcFB, EPcZB, EPcDF,
           EPcTMF, EPcAF, EPcTEXEN, EPcFMB, EPcTMB, EPcSCR,
           EPtmod, EPdADDR, EPdA, EPpxlY, EPpxlX, EPpxlL, EPpxlR,
           EPtxlL, EPtxlR, EPcntX);

     //Feedthrough signals
     assign     EPline = TSline;

     /* Module interconnection */
```

```
// Main Controller
Wep_ctrl    ssep_ctrl();
```

```
//Main controller
module    Wep_ctrl(  REclk, TSreset, TSnhld, HSnwait, TScOP, TScFB,
                TScZB, TScDF, TScTMF, TScAF, TScTEXEN, TScFMB,
                TScTMB, TScSCR, EPline, EPnwait, EPreset, EPnhld,
                EPcOP, EPcFB, EPcZB, EPcDF, EPcTMF, EPcAF,
                EPcTEXEN, EPcFMB, EPcTMB, EPcSCR, EPvcnt, EPfetch_CLK,
                EPfetch_TMOD, EPfetch_ADDR, EPfetch_ALPHA, EPfetch_PXL,
                EPfetch_TXL, EPIfetch_CLK, EPIfetch_PXL, EPIfetch_TXL);

    //Pipeline control signal
    //EP stage controls the multi-cycle operation by y count.
    //Therefore, until the multi-cycle operation, the previous
    //pipeline stage should be stalled.
    //That means the EPnwait is high until the multi-cycle operation is done.

    //nwait
    assign  #1 EPnwait_gen = ~(EPnhld_int & EPcDF_int & (|EPcounter));
    assign  #1 EPnwait = (EPnwait_gen & HSnwait) | ~TSreset;

    //nhld
    assign  #1 EPnhld_gen = ~EPnwait_gen;
    assign  #1 EPnhld = EPnhld_int | EPnhld_gen;

    //Fetch
    assign  #1 EPfetch_CLK = EPnwait & TSnhld;
    assign  EPfetch_TMOD = TScTMF;
    assign  EPfetch_ADDR = TScAF;
    assign  EPfetch_ALPHA = TScDF;
    assign  EPfetch_PXL = TScDF;
    assign  EPfetch_TXL = TScDF & TScTEXEN;

    //EPI fetch signals
    assign  EPIfetch_CLK = HSnwait & EPnhld;
    assign  EPIfetch_PXL = EPcDF;
    assign  EPIfetch_TXL = EPcDF & EPcTEXEN;
    always @(posedge REclk)
    begin
        if(EPnwait)
```

```
          begin
               //State controls
               EPnhld_int <= TSnhld;
               EPreset <= TSreset;
               //Fetch_Signal
               EPcDF_int <= TScDF;
               EPcTMF_int <= TScTMF;
               EPcAF_int <= TScAF;
               EPcTEXEN_int <= TScTEXEN;
               EPcFMB_int <= TScFMB;
               EPcTMB_int <= TScTMB;
               //Trivial Signals
               EPcOP <= TScOP;
               EPcFB <= TScFB;
               EPcZB <= TScZB;
               EPcSCR <= TScSCR;
          end

          /* Multicycle counter */

          // Set to EPline
          if(EPnwait & TScDF & (TScOP[5]|TScOP[4]))
          begin
               EPvcnt <= 8'b0;
               EPcounter <= EPline;
          end
          //Count when {Next OK, RDAT Mode, Run Mode} until EPcounter =0
          else if((HSnwait | ~TSreset) & (|EPcounter) & EPnhld)
          begin
               EPvcnt <= EPvcnt +1;              // Up counter
               EPcounter <= EPcounter -1;        // Down counter
          end
     end

     always @(negedge REclk)
     begin
          EPcDF <= EPcDF_int & EPnhld;
          EPcTMF <= EPcTMF_int & EPnhld;
          EPcAF <= EPcAF_int & EPnhld;
          EPcTEXEN <= EPcTEXEN_int & EPnhld;
          EPcFMB <= EPcFMB_int & EPnhld;
          EPcTMB <= EPcTMB_int & EPnhld;
     end
endmodule
```

//Pipeline data latches
Wep_latch   ssep_latch();

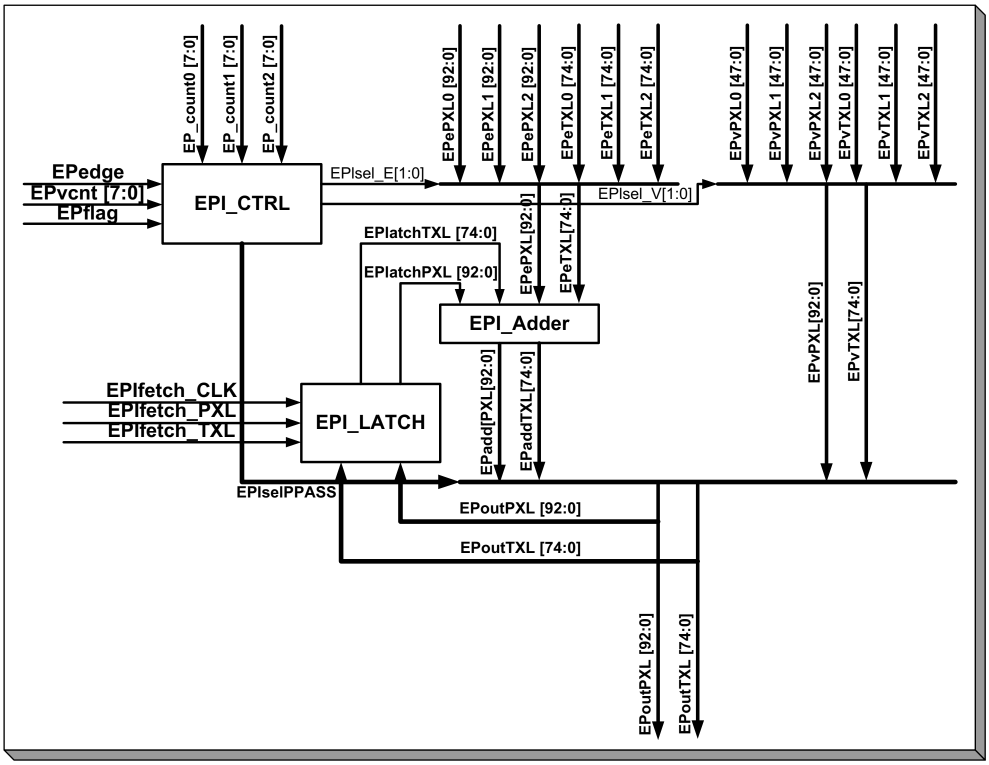
```
//Data latches
module    Wep_latch(  REclk, clk, EPfetch_TMOD, EPfetch_ADDR, EPfetch_ALPHA,
          EPfetch_PXL, EPfetch_TXL, TStmod, TSdADDR, TSdA, TSaddrY, TSedge,
          TScount0, TScount2, TSePXL0, TSePXL1, TSePXL2, TSeTXL0,
          TSeTXL1, TSeTXL2, TSvPXL0, TSvPXL1, TSvPXL2, TSvTXL0, TSvTXL1,
          TSvTXL2, EPtmod, EPdADDR, EPdA, EPaddrY, EPedge, EPcount0,
          EPcount2, EPePXL0, EPePXL1, EPePXL2, EPeTXL0, EPeTXL1, EPeTXL2,
          EPvPXL0, EPvPXL1, EPvPXL2, EPvTXL0, EPvTXL1, EPvTXL2);

     /* Pipeline latches */
     //EP_LATCH_TMOD
     always @(posedge REclk)
     begin
          if(clk & EPfetch_TMOD)
          begin
               EPtmod <= TStmod;
          end
     end
     //EP_LATCH_ADDR
     always @(posedge REclk)
     begin
          if(clk & EPfetch_ADDR) EPdADDR <= TSdADDR;
     end

     //EP_LATCH_ALPHA
     always @(posedge REclk)
     begin
          if(clk & EPfetch_ALPHA) EPdA <= TSdA;
     end

     //EP_LATCH_PXL
     always @(posedge REclk)
     begin
          if(clk & EPfetch_PXL)
          begin
               //Triangle setup values
               EPaddrY <= TSaddrY;
               EPedge <= TSedge;
               EPcount0 <= TScount0;
               EPcount2 <= TScount2;
               //Normal pixels: edge
               EPePXL0 <= TSePXL0;
               EPePXL1 <= TSePXL1;
               EPePXL2 <= TSePXL2;
               //Normal pixels: vertex
               EPvPXL0 <= TSvPXL0;
               EPvPXL1 <= TSvPXL1;
               EPvPXL2 <= TSvPXL2;
          end
     end

     //EP_LATCH_TXL
     always @(posedge REclk)
     begin
          if(clk & EPfetch_TXL)
          begin
               //Edge
               EPeTXL0 <= TSeTXL0;
               EPeTXL1 <= TSeTXL1;
               EPeTXL2 <= TSeTXL2;
               //Vertex
```

```
                EPvTXL0 <= TSvTXL0;
                EPvTXL1 <= TSvTXL1;
                EPvTXL2 <= TSvTXL2;
            end
        end
endmodule
```

//EPI_Y
Wep_epi_y   ssep_epi_y();

```
//EPI_Y
module    Wep_epi_y( inY, vcnt, outY);
     input [7:0]       inY;
     input [7:0]       vcnt;
     output [7:0]      outY;

     DW01_add #(8)     add_Y(  .A    (inY),
                               .B    (vcnt),
                               .CI   (1'b0),
                               .SUM  (outY),
                          .CO  (cout));
endmodule
```

```
//The EPI generates the pixel values of left and right edges.
//The EPI block consists of controller and controller, latch, and SIMD_adder block;
see diagram E and the chart.
```



Control signal	Description	
	EPedge==EPIflag	EPedge!=EPIflag
EPIsel_PASS	Set to 1 (PASS) when	Set to 1 (PASS) when
	EPIv_0 (First)	EPIv_0 (First)
	EPIv_cnt2 (Last)	EPIv_cnt0 (Middle)
		EPIv_cnt2 (Last)
EPIsel_V	Select	Select
	0 when EPIv_0	0 when EPIv_0 & ~v_cnt0
	2 when EPIv_cnt2	1 when EPIv_cnt0
		2 when EPIv_cnt2
EPIsel_E	Select	Select
	2 whenever	0 when EPImode==0
		1 when EPImode==1
EPImode	MSB of Sign Extended {EPIcount0-EPIvcnt}	
EPIv_0	1 when EPvcnt==0	
EPIv_cnt0	1 when EPvcnt==EPcount0	
EPIv_cnt2	1 when EPvcnt==EPcount2	

```
//EPI_L
Wep_epi    ssep_epi_L();
//EPI_R
Wep_epi    ssep_epi_R();
```

```
//EPI controller
module    Wep_epi_ctrl(EPvcnt, EPedge, EPflag, EPcount0, EPcount2,
          EPIsel_E, EPIsel_V, EPIsel_PASS);

     DW01_sub #(9)      sub_MODE(    .A     ({1'b0, EPcount0}),
                                     .B     ({1'b0, EPvcnt}),
                                     .CI    (1'b0),
                                     .DIFF  (EPIsubY),
                                     .CO    (EPIcarry));

     //Basic state signals
     always @(EPvcnt or EPcount0 or EPcount2 or EPflag or EPedge)
     begin
          EPIv_0      = (EPvcnt == 8'b0)       ?     1'b1 : 1'b0;
          EPIv_cnt0   = (EPvcnt == EPcount0)   ?     1'b1 : 1'b0;
          EPIv_cnt2   = (EPvcnt == EPcount2)   ?     1'b1 : 1'b0;
          EPIfe       = (EPflag != EPedge)     ?     1'b1 : 1'b0;
     end

     //Control signals
     //EPIsel_PASS, EPIsel_V
     always @(EPIfe or EPIv_0 or EPIv_cnt0 or EPIv_cnt2)
     begin
          //EPIsel_PASS
          casex({EPIfe, EPIv_0, EPIv_cnt0, EPIv_cnt2})
          4'bx1xx      : EPIsel_PASS <= 1'b1;
          4'bxxx1      : EPIsel_PASS <= 1'b1;
          4'b1x1x      : EPIsel_PASS <= 1'b1;
          default      : EPIsel_PASS <= 1'b0;
      endcase

      //EPIsel_V
      casex({EPIfe, EPIv_0, EPIv_cnt0, EPIv_cnt2})
          4'b01xx      : EPIsel_V <= 2'd0;
          4'b0xx1      : EPIsel_V <= 2'd2;
          4'b110x      : EPIsel_V <= 2'd0;
          4'b1x1x      : EPIsel_V <= 2'd1;
          4'b1x01      : EPIsel_V <= 2'd2;
          default      : EPIsel_V <= 2'bX;
      endcase
  end

  //EPIsel_E
  always @(EPIfe or EPIsubY)
  begin
      casex({EPIfe, EPIsubY[8]})
          2'b0x      : EPIsel_E <= 2'd2;
          2'b10      : EPIsel_E <= 2'd0;
          2'b11      : EPIsel_E <= 2'd1;
      endcase
  end
endmodule

//7-SIMD interpolation adders
module   Wep_epi_add_7simd( EPlatchPXL, EPlatchTXL, EPePXL, EPeTXL,
                            EPaddPXL, EPaddTXL);

  /* Adder interconnections */
  DW01_add #(18)   add_X(  .A      (EPlatchPXL[93:76]),
                           .B      (EPePXL[93:76]),
                           .CI     (1'b0),
                           .SUM    (EPaddPXL[93:76]),
                           .CO     (coutX));
  DW01_add #(25)   add_Z(  .A      (EPlatchPXL[75:51]),
                           .B      (EPePXL[75:51]),
                           .CI     (1'b0),
                           .SUM    (EPaddPXL[75:51]),
                           .CO     (coutZ));
  DW01_add #(17)   add_R(  .A      (EPlatchPXL[50:34]),
                           .B      (EPePXL[50:34]),
                           .CI     (1'b0),
                           .SUM    (EPaddPXL[50:34]),
                           .CO     (coutR));
  DW01_add #(17)   add_G(  .A      (EPlatchPXL[33:17]),
                           .B      (EPePXL[33:17]),
                           .CI     (1'b0),
                           .SUM    (EPaddPXL[33:17]),
                           .CO     (coutG));
  DW01_add #(17)   add_B(  .A      (EPlatchPXL[16:0]),
                           .B      (EPePXL[16:0]),
                           .CI     (1'b0),
                           .SUM    (EPaddPXL[16:0]),
                           .CO     (coutB));
  DW01_add #(41)    add_U(  .A      (EPlatchTXL[122:82]),
                           .B      (EPeTXL[122:82]),
                           .CI     (1'b0),
                           .SUM    (EPaddTXL[122:82]),
                           .CO     (coutU));
  DW01_add #(41)    add_V(  .A      (EPlatchTXL[81:41]),
                           .B      (EPeTXL[81:41]),
                           .CI     (1'b0),
                           .SUM    (EPaddTXL[81:41]),
                           .CO     (coutV));
  DW01_add #(41)    add_W(  .A      (EPlatchTXL[40:0]),
                           .B      (EPeTXL[40:0]),
                           .CI     (1'b0),
                           .SUM    (EPaddTXL[40:0]),
                           .CO     (coutW));

endmodule

//EPI top
module     Wep_epi(  REclk, EPvcnt, EPedge, EPflag, EPIfetch_CLK, EPIfetch_PXL,
           EPIfetch_TXL, EPcount0, EPcount2, EPePXL0, EPePXL1, EPePXL2,
           EPeTXL0, EPeTXL1, EPeTXL2,
           EPvPXL0, EPvPXL1, EPvPXL2, EPvTXL0, EPvTXL1, EPvTXL2, EPpxlLR,
           EPtxlLR, EPlatchPXL);

  //Module interconnection
```

```
//Main controller for EPI
Wep_epi_ctrl     ssep_epi_ctrl();

// 7-SIMD adder
Wep_epi_add_7simd     ssep_add_7simd();
```

```
      //EPI_MUX_EDGE_PXL
      always @(EPIsel_E or EPePXL0 or EPePXL1 or EPePXL2)
      begin
           casex(EPIsel_E)
                2'd0           : EPePXL <= EPePXL0;
                2'd1           : EPePXL <= EPePXL1;
                2'd2           : EPePXL <= EPePXL2;
                default        : EPePXL <= 94'b0;
           endcase
      end

      //EPI_MUX_EDGE_TXL
      always @(EPIsel_E or EPeTXL0 or EPeTXL1 or EPeTXL2)
      begin
           casex(EPIsel_E)
```

```
          2'd0           : EPeTXL <= EPeTXL0;
          2'd1           : EPeTXL <= EPeTXL1;
          2'd2           : EPeTXL <= EPeTXL2;
          default        : EPeTXL <= 123'b0;
     endcase
end

//EPI_MUX_VERTEX_PXL
always @(EPIsel_V or EPvPXL0 or EPvPXL1 or EPvPXL2)
begin
     casex(EPIsel_V)
          2'd0           : EPvPXL <= EPvPXL0;
          2'd1           : EPvPXL <= EPvPXL1;
          2'd2           : EPvPXL <= EPvPXL2;
          default        : EPvPXL <= 49'b0;
     endcase
end

//EPI_MUX_VERTXL_TXL
always @(EPIsel_V or EPvTXL0 or EPvTXL1 or EPvTXL2)
begin
     casex(EPIsel_V)
          2'd0           : EPvTXL <= EPvTXL0;
          2'd1           : EPvTXL <= EPvTXL1;
          2'd2           : EPvTXL <= EPvTXL2;
          default        : EPvTXL <= 96'b0;
     endcase
end

//EPI_MUX_OUTPUT_PXL with sign and fraction extention
always @(EPIsel_PASS or EPaddPXL or EPvPXL)
begin
     casex(EPIsel_PASS)
          1'b0      : EPoutPXL <= EPaddPXL;
          1'b1      :
          begin
               EPoutPXL[93:76]     <= {1'b0, EPvPXL[48:40], 8'b0}; //X
               EPoutPXL[75:51]     <= {1'b0, EPvPXL[39:24], 8'b0}; //Z
               EPoutPXL[50:34]     <= {1'b0, EPvPXL[23:16], 8'b0}; //R
               EPoutPXL[33:17]     <= {1'b0, EPvPXL[15:8], 8'b0}; //G
               EPoutPXL[16:0]      <= {1'b0, EPvPXL[7:0], 8'b0};      //B
          end
     endcase
end

always @(EPIsel_PASS or EPaddTXL or EPvTXL)
begin
     casex(EPIsel_PASS)
          1'b0      : EPoutTXL <= EPaddTXL;
          1'b1      :
          begin
               EPoutTXL[122:82]    <= {1'b0, EPvTXL[95:64], 8'b0}; //U
               EPoutTXL[81:41]     <= {1'b0, EPvTXL[63:32], 8'b0}; //V
```

```
                    EPoutTXL[40:0]      <= {1'b0, EPvTXL[31:0], 8'b0}; //W
               end
          endcase
     end

     /* Datapath latch */

     //EPI_LATCH_PXL
     always @(posedge REclk)
     begin
          if(EPIfetch_CLK & EPIfetch_PXL) EPlatchPXL     <= EPoutPXL;
     end
     //EPI_LATCH_TXL
     always @(posedge REclk)
     begin
          if(EPIfetch_CLK & EPIfetch_TXL) EPlatchTXL     <= EPoutTXL;
     end

     /* Output reassignment */

     //Pixel
     assign #1  EPpxlLR[48:40]     = EPoutPXL[92:84]; //X
     assign #1  EPpxlLR[39:24]     = EPoutPXL[74:59]; //Z
     assign #1  EPpxlLR[23:16]     = EPoutPXL[49:42]; //R
     assign #1  EPpxlLR[15:8]      = EPoutPXL[32:25]; //G
     assign #1  EPpxlLR[7:0]     = EPoutPXL[15:8]; //B

     //Texel
     assign #1  EPtxlLR[95:64]     = EPoutTXL[121:90]; //U
     assign #1  EPtxlLR[63:32]     = EPoutTXL[80:49]; //V
     assign #1  EPtxlLR[31:0]      = EPoutTXL[39:8]; //W
endmodule
```

```
     //EPI_LRSUB_X
     DW01_sub #(10)    sub0_X(  .A     ({1'b0, EPpxlR[48:40]}),
                                .B     ({1'b0, EPpxlL[48:40]}),
                                .CI    (1'b0),
                                .DIFF  (EPcntX_int),
                                .CO    (coutX));

     //Minus saturation due to lacking of precision @ TS_MID_CAL
     assign  EPcntX = (EPcntX_int[9])      ?     9'b0 : EPcntX_int[8:0];
     //X0 Address
     assign #1  EPpxlX = EPpxlL [48:40];
endmodule
```

7.5.4.6 HS: Horizontal Setup

In the HS stage (Figure 7.17), the horizontal slopes (used for pixel interpolation) are computed. The horizontal slopes are computed using the left and right edge pixels, which are located on the same *y* axis. Also in this stage, depth requests occur. Since the

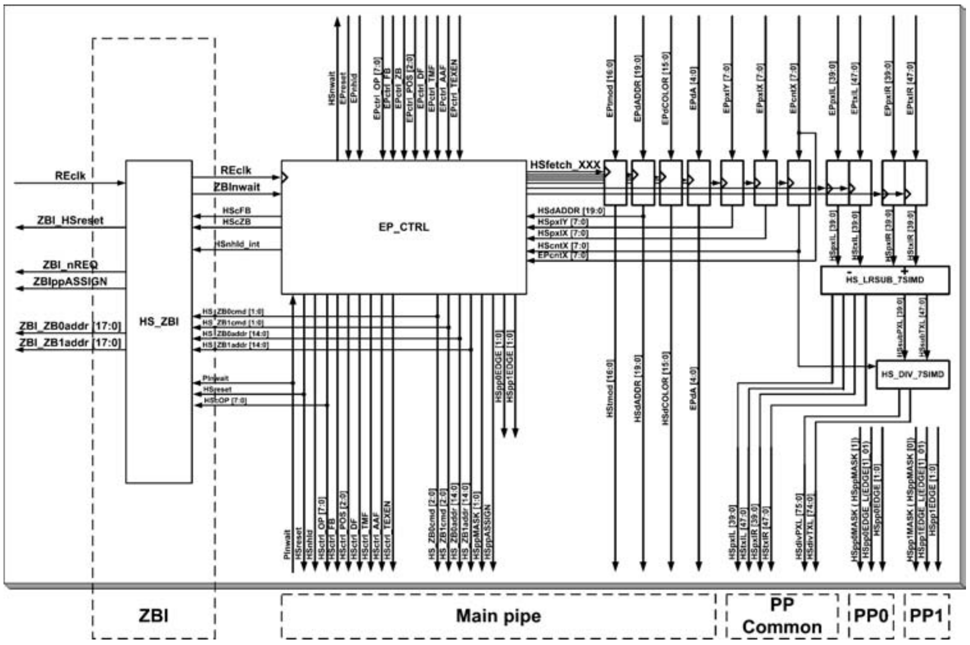


Figure 7.17 Horizontal setup

embedded memory requires 1 cycle latency to read data, the depth buffer request and buffer address are transferred to the depth buffer in the HS stage.

Since the rasterizer has of two pixel processors, it is important to assign pixels to their correct processor. Basically, this is done according to the x coordinate – odd/ even – and the start of the line assigned to pixel processor (PP)0. For line starts with even x coordinate it is easy to assign PPs. In other cases the PP assignment is more difficult.

Since the horizontal setup is performed on a line basis, it contains several pixels. The next pipeline pixel interpolation generates one pixel in a cycle, so the HS stage has to wait until the pixel interpolation is ended. Therefore, the HS stage controls multi-cycle operation.

Pipeline Input

- REclk: rendering engine clock
- Control signals: same as EP
- Auxiliary signals: same as EP
- Pixel/texel: same as EP

Pipeline Output

- Pipe control signals: same as EP
- HS_ZB0cmd [2:0]: ZB0 command
- HS_ZB1cmd [2:0]: ZB1 command
- HS_ZB0addr [15:0]: ZB0 address
- HS_ZB1addr [15:0]: ZB1 address
- HSppMASK [1:0]: pixel processor operation mask
- HSppASSIGN: pixel processor assignment flag

Memory Output

- ZBI_nREQ: ZB operation request
- ZBI_ZB0addr [15:0]: ZB0 address
- ZBI_ZB1addr [15:0]: ZB1 address
- ZBIppASSIGN: PP assignment for arbiter

Pixel Processor Output

- HSpp0MASK: PP0 operation mask
- HSpp1MASK: PP1 operation mask
- HSpp0EDGE [1:0]: PP0 edge selection flag
- HSpp1EDGE [1:0]: PP1 edge selection flag

Functions

Type	HScOP[7:0]	Cycle	Description
Rendering	CTRL_OP_RDAT	EPcntX/2	Horizontal setup Pixel processor management Pixel processor assignment (PP interpolation by 2X)
Texture	CTRL_OP_TMOD	1	Set texture mode Bypass
Auxiliary	CTRL_OP_ASTR	1	Store data to front buffer

PP Control Scheme

See Figure 7.18.

Pipeline Timing

See Figure 7.19.

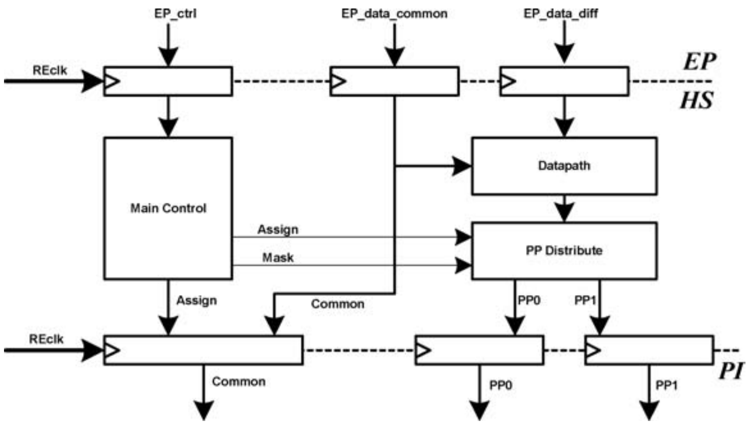


Figure 7.18 PP control scheme: block diagram

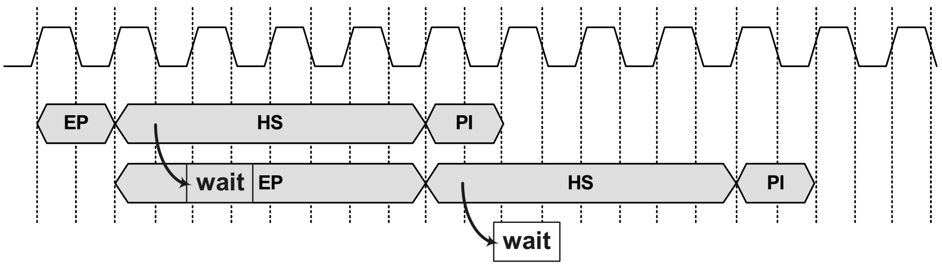


Figure 7.19 PP control scheme: pipeline timing

Signal Descriptions

Control signals	Description
MPnwait	nWait from memory programmer
ZBI_selBUS	Front/back buffer selection (alpha/beta memory)
ZBI_ZB0cmd[2:0]@n	Z-buffer eDRAM command
ZBI_ZB1cmd[2:0]@n	Bypass from HS_ZBcmd
ZBI_ZB0addr[14:0]@n	Z-buffer eDRAM address
ZBI_ZB1addr[14:0]@n	Bypass from HS_ZBaddr considering ZB flag
ZBInwait	nWait to previous pipe
HScOP[7:0]	Original OP code without refresh
	Same as EPcOP[7:0]
	Required to generate ZBInwait from Mpnwait
HScZB	Z-buffer flag
	RBUF mode: buffer select
	ASTR, RCLR mode: masking out
HS_ZB0cmd[2:0]	Z-buffer eDRAM command

HS_ZB1cmd[2:0]	3’b100: dRMW (different ROW)
	3’b010: sRMW (same ROW)
	3’b000: NOP
	ROW-address register must be reset to dRMW for non-consecutive operation at every Refresh/CMD (NOP, RDAT, RCLR, ASTR)
	ex: {MSB_OP_flag + ROW}
	Commonly generated and masked by HSppMASK
	Overridden by HSnhld
HS_ZB0addr[14:0]	15-bit Z-buffer address (128 × 256)
HS_ZB1addr[14:0]	{ROW,COL}
	Normal operation: generated by counter
	ASTR: with HSdADDR[15:0]
	Commonly generated and masked by HSppMASK
HSppMASK[1:0]	PP enable mask
	Overrides nhld at the next stage and command generation
	2’b1X: PP0 enable
	2’bX1: PP1 enable
HSppASSIGN	PP assign
	Assign-A: PP0 → PP1
	Assign-B: PP1 → PP0
HSpp0EDGE[1:0]	PP edge selection
HSpp1EDGE[1:0]	2’b10: left edge
	2’b01: right edge
HScOP[7:0]	OP code

Datapath signal	Description
HSpxlLR	{Z[39:24],R[23:16],G[15:8],B[7:0]}
	R, G, B: 8-bit unsigned data
	Z: 16-bit unsigned data
HStxlLR	{U[47:32],V[31:16],W[15:0]}
	U, V, W: 16-bit unsigned data
HSsubPXL	{Z[43:27],R[26:18],G[17:9],B[8:0]}
	Z: 17-bit signed data
	{sign, data}
	R, G, B: 9-bit signed data
	{sign, data}
HSsubTXL	{U[50:34],V[33:17],W[16:0]}
	U, V, W: 17-bit signed data
	{sign, data}
HSdivPXL	{Z[75:51],R[50:34],G[33:17],B[16:0]}
	Z: 25-bit signed data
	{sign_1b, data_16b, frac_8b}
	R, G, B: 17-bit signed data
	{sign_1b, data_8b, frac_8b}
HSdivTXL	{U[74:50],V[49:25],W[24:0]}
	25-bit signed data
	{sign_1b, data_16b, frac_8b}

Horizontal Setup

See Figure 7.20.

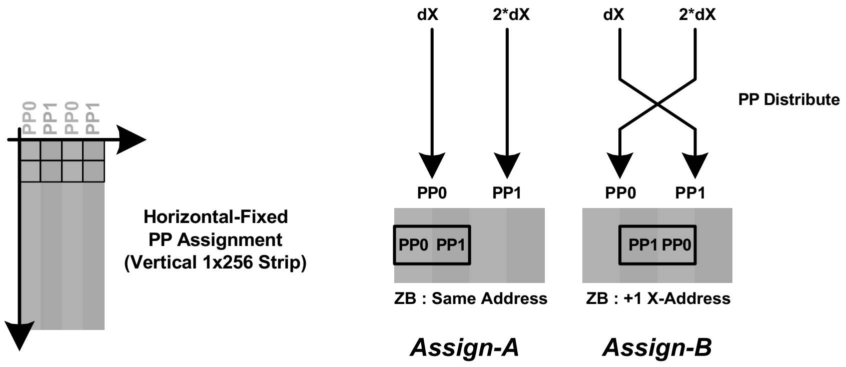


Figure 7.20 Horizontal setup: PP assignment

- Prevents unnecessary PP from transition
- Horizontal-fixed PP assignment eliminates "2-to-2 memory crossbar"

Pipeline Timing

See Figure 7.21.

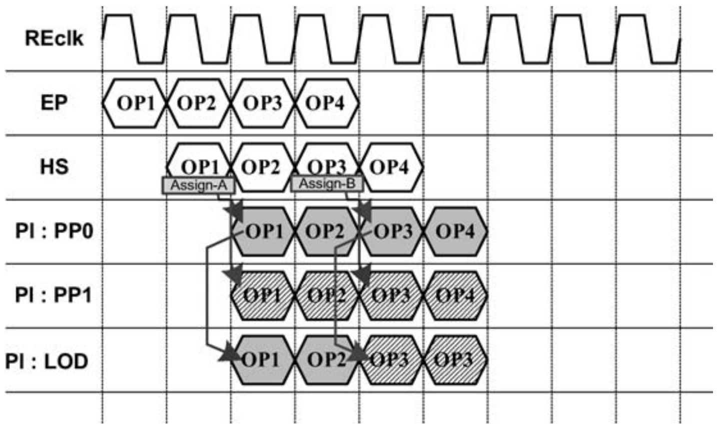


Figure 7.21 Horizontal setup: pipeline timing

RTL Code

```
/*
* RAMP-GR
* RAMP-GR  HS Module: Horizontal Setup
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Whs(  REclk, PInwait, EPreset, EPnhld, EPcOP, EPcFB, EPcZB,
             EPcDF, EPcTMF, EPcAF, EPcTEXEN, EPcFMB, EPcTMB, EPcSCR,
             EPtmod, EPdADDR, EPdA, EPpxlY, EPpxlX, EPcntX,
             EPpxlL, EPtxlL, EPpxlR, EPtxlR,
             HSnwait, HSreset, HSnhld, HScOP, HScFB,
             HScDF, HScTMF, HScAF, HScTEXEN, HScFMB, HScTMB, HScSCR,
             HS_ZB0cmd, HS_ZB1cmd, HS_ZB0addr, HS_ZB1addr, HSppMASK, HSppASSIGN,
             HStmod, HSdADDR, HSdA, HSpxlL, HStxlL, HSpxlR, HStxlR, HSdivPXL,
             HSdivTXL, HSpp0MASK, HSpp0EDGE_L, HSpp0EDGE, HSpp1MASK, HSpp1EDGE_L,
             HSpp1EDGE, ZBI_HSreset, ZBI_ZB0cmd, ZBI_ZB1cmd, ZBI_ZB0addr,
             ZBI_ZB1addr);

    //Wiring
    assign #1  HSpp0MASK = HSppMASK[1];
    assign #1  HSpp1MASK = HSppMASK[0];
    assign #1  HSpp0EDGE_L = HSpp0EDGE[1] | HSpp1EDGE[1];
    assign #1  HSpp1EDGE_L = HSpp0EDGE[1] | HSpp1EDGE[1];

    /* Module interconnection */


//Main controller
Whs_ctrl   sshs_ctrl(   );
```

```
//Main controller
module   Whs_ctrl(  REclk, PInwait, EPreset, EPnhld, EPcOP, EPcFB, EPcZB,
         EPcDF, EPcTMF, EPcAF, EPcTEXEN, EPcFMB, EPcTMB, EPcSCR,
         HSdADDR, HSpxlY, HSpxlX, EPcntX, HScntX,
         HSnwait, HSreset, HSnhld, HScOP, HScFB,
         HScDF, HScTMF, HScAF, HScTEXEN, HScFMB, HScTMB, HScSCR,
         HSfetch_CLK, HSfetch_TMOD, HSfetch_ADDR, HSfetch_ALPHA,
         HSfetch_PXL, HSfetch_TXL,
         HS_ZB0cmd, HS_ZB1cmd, HS_ZB0addr, HS_ZB1addr,
         HSppMASK, HSppASSIGN, HSpp0EDGE, HSpp1EDGE, HScZB, HSnhld_int,
         HScounter_X, HScounter_Y);

    /* Combinational state-control logics */
    //nWait
    assign  HSnwait_gen1 = ~(HSnhld_int & (HScDF_int HScOP[4]))
                  & (|{HScounter_Y, HScounter_X}));
    assign  HSnwait_gen2 = ~(HSnhld_int & HScOP[0] & (|{HScounter_Y,
                 HScounter_X}));
    assign  HSnwait_gen = (HSnwait_gen1 & HSnwait_gen2);
    assign  HSnwait  = (HSnwait_gen & PInwait) | ~EPreset;
    //nHld
    assign  HSnhld_gen1 = ~HSnwait_gen;
    assign  HSnhld   = HSnhld_int|HSnhld_gen1;

    //Fetch
    assign #1  HSfetch_CLK = HSnwait & EPnhld;
    assign  HSfetch_TMOD = EPcTMF;
    assign  HSfetch_ADDR = EPcAF;
    assign  HSfetch_ALPHA = EPcDF;
    assign  HSfetch_PXL = EPcDF;
    assign  HSfetch_TXL = EPcDF & EPcTEXEN;
    always @(posedge REclk)
    begin
         //Control signals
         if(HSnwait)
         begin
              //State control
              HSnhld_int <= EPnhld;
              HSreset <= EPreset;
              //Fetch singals
              HScDF_int <= EPcDF;
              HScTMF_int <= EPcTMF;
              HScAF_int <= EPcAF;
              HScTEXEN_int <= EPcTEXEN;
              HScFMB_int <= EPcFMB;
              HScTMB_int <= EPcTMB;
              //Trivial Signals
              HScOP <= EPcOP;
              HScFB <= EPcFB;
              HScZB <= EPcZB;
              HScSCR <= EPcSCR;
         end

         /* Multicycle counter */
         //Set: RDAT to EPcntX/2
         if(HSnwait & EPcDF & (EPcOP[5]|EPcOP[4]))
         begin
              HSxcnt <= 8'b0;
              //HScounter <= {8'b0,EPcntX[8:1]};
              HScounter_X <= EPcntX[8:1];
              HScounter_Y <= 8'b0;
         end
         //Set: RCLR to EPcnt
         else if(HSnwait & EPcOP[0])
         begin
              HScounter_X <= 8'd159;
              HScounter_Y <= 8'd239;
         end
         //Count
         else if((PInwait | ~EPreset) & (|{HScounter_Y, HScounter_X}) & HSnhld)
         begin
              if(HScOP[5])
              begin
                   //Up counter
```

```
                    HSxcnt <= HSxcnt + 1;
                    HScounter_X <= HScounter_X -1;
               end
               else if(HScOP[0])
               begin
                    //Down counter
                    HScounter_X <= HScounter_X - 1;
                    if(~|HScounter_X & |HScounter_Y)
                    begin
                         HScounter_X <= 8'd159;
                         HScounter_Y <= HScounter_Y - 1;
                    end
               end
               //Down counter
               //HScounter <= HScounter - 1;
          end
     end

     always @(negedge REclk)
     begin
          HScDF <= HScDF_int & HSnhld;
          HScTMF <= HScTMF_int & HSnhld;
          HScAF <= HScAF_int & HSnhld;
          HScTEXEN <= HScTEXEN_int & HSnhld;
          HScFMB <= HScFMB_int & HSnhld;
          HScTMB <= HScTMB_int & HSnhld;
     end

     /* PP control signals: fully combinational logic */

     //Assignment, edge, mask: left and rignt engine (not PP0 nor PP1)
     assign #1 HSppASSIGN = HSpxlX[0];
     assign #1 HSppEDGE_LR = {~(|HSxcnt), ~(|{HScounter_Y, HScounter_X})};
     assign #1 HSppMASK_LR = {1'b1, ((|{HScounter_Y, HScounter_X})|HScntX[0])};
     //PP0 and PP1 correction
     assign #1 HSppMASK = (HSppASSIGN) ?
                    {HSppMASK_LR[0], HSppMASK_LR[1]} : HSppMASK_LR;
     always @(HSppASSIGN or HSppEDGE_LR or HScntX)
     begin
          casex({HSppASSIGN, HScntX[0]})
               2'b00 : begin
                         HSpp0EDGE <= HSppEDGE_LR;
                         HSpp1EDGE <= 2'b00; end
               2'b01 : begin
                         HSpp0EDGE <= {HSppEDGE_LR[1], 1'b0};
                         HSpp1EDGE <= {1'b0, HSppEDGE_LR[0]}; end
               2'b10 : begin
                         HSpp0EDGE <= 2'b00;
                         HSpp1EDGE <= HSppEDGE_LR; end
               2'b11 : begin
                         HSpp0EDGE <= {1'b0, HSppEDGE_LR[0]};
                         HSpp1EDGE <= {HSppEDGE_LR[1], 1'b0}; end
          endcase
     end
```

```
  //Z-buffer address generation
    //RDAT
    DW01_add #(8)    add_ZB_XL (  .A     (HSpxlX[8:1]),
                                  .B     (HSxcnt),
                                  .CI    (1'b0),
                                  .SUM   (HSZBaddrX_L),
                                  .CO    (cout0));
    DW01_add #(8)    add_ZB_XR (  .A     (HSpxlX[8:1]),
                                  .B     (HSxcnt),
                                  .CI    (HSpxlX[0]),
                                  .SUM   (HSZBaddrX_R),
                                  .CO    (cout1));
    //320xy
    WMUL_add    mul_ZB_Y (  .x    (10'd320),
                            .y    (HSpxlY),
                            .z    (HS_yout));
    DW01_add #(16)   add_addr_L(  .A    (HS_yout[16:1]),
                                  .B    ({8'b0, HSZBaddrX_L}),
                                  .CI   (1'b0),
                                  .SUM  (HS_RDATaddrL),
                                  .CO   (a_coutL));
    DW01_add #(16)   add_addr_R(  .A    (HS_yout[16:1]),
                                  .B    ({8'b0, HSZBaddrX_R}),
                                  .CI   (1'b0),
                                  .SUM  (HS_RDATaddrR),
                                  .CO   (a_coutR));
    //Clear address
    WMUL_add    mul_CLR_Y ( .x    (10'd320),
                            .y    (HScounter_Y),
                            .z    (HS_Yclr));
    DW01_add #(16)   add_CLR (     .A    (HS_Yclr[16:1]),
                                  .B    ({8'b0, HScounter_X}),
                                  .CI   (1'b0),
                                  .SUM  (HS_clraddr),
                                  .CO   (a_coutclr));
    //assign #1  HSpp0_RDATaddr = (HSppASSIGN) ?
                    {HSpxlY, HSZBaddrX_R} : {HSpxlY, HSZBaddrX_L};
    //assign #1  HSpp1_RDATaddr = (HSppASSIGN) ?
                    {HSpxlY, HSZBaddrX_L} : {HSpxlY, HSZBaddrX_R};
    assign #1  HSpp0_RDATaddr = (HSppASSIGN) ? HS_RDATaddrR : HS_RDATaddrL;
    assign #1  HSpp1_RDATaddr = (HSppASSIGN) ? HS_RDATaddrL : HS_RDATaddrR;
    //Address multiplexing
    always @(HScOP or HSpp0_RDATaddr or HSpp1_RDATaddr or HS_clraddr)
    begin
         casex(HScOP)
              'WCTRL_OP_RSHA : begin
                   HS_ZB0addr <= HSpp0_RDATaddr;
                   HS_ZB1addr <= HSpp1_RDATaddr; end
              'WCTRL_OP_RCLR : begin
                   HS_ZB0addr <= HS_clraddr;
                   HS_ZB1addr <= HS_clraddr; end
              default : begin
                   HS_ZB0addr <= 16'b0;
```

```
                    HS_ZB1addr <= 16’b0; end
          endcase
      end

      /* Z-buffer command generation */
      always @(HScOP or HSnhld or PInwait)
      begin
          if((HScOP[5]|HScOP[4]|HScOP[0]) & HSnhld & PInwait)
          //RSHA, RTEX, RCLR
               HS_ZBcmd_int <= ‘WHS_MEMCMD_RMW;
          else HS_ZBcmd_int <= ‘WHS_MEMCMD_NOP;
      end
      //Command generation
      always @(HS_ZBcmd_int or HSppMASK or HScOP)
      begin
      //Other operations
           casex(HS_ZBcmd_int)
               ‘WHS_MEMCMD_NOP :
               begin
                    HS_ZB0cmd <= ‘WFZB_CMD_NOP;
                    HS_ZB1cmd <= ‘WFZB_CMD_NOP;
               end
               ‘WHS_MEMCMD_RMW :
               begin
                    if(HScOP[0])
                    //RCLR
                    begin
                         HS_ZB0cmd <= ‘WFZB_CMD_RMW;
                         HS_ZB1cmd <= ‘WFZB_CMD_RMW;
                    end
                    else if(HScOP[5]|HScOP[4])
                    //RSHA / RTEX
                    begin
                         HS_ZB0cmd <= (HSppMASK[1]) ?
                          ‘WFZB_CMD_RMW : ‘WFZB_CMD_NOP;
                         HS_ZB1cmd <= (HSppMASK[0]) ?
                         ‘WFZB_CMD_RMW : ‘WFZB_CMD_NOP;
                    end
                    else
                    begin
                         HS_ZB0cmd <= ‘WFZB_CMD_NOP;
                         HS_ZB1cmd <= ‘WFZB_CMD_NOP;
                    end
               end
               default :
               begin
                    HS_ZB0cmd <= ‘WFZB_CMD_NOP;
                    HS_ZB1cmd <= ‘WFZB_CMD_NOP;
               end
          endcase
      end
endmodule
```

//Data latches
Whs_latch   sshs_latch();

```
//Data latches
module    Whs_latch(REclk, clk, HSfetch_TMOD, HSfetch_ADDR,
          HSfetch_ALPHA, HSfetch_PXL, HSfetch_TXL,
          EPtmod, EPdADDR, EPdA, EPpxlY, EPpxlX, EPcntX,
          EPpxlL, EPtxlL, EPpxlR, EPtxlR,
          HStmod, HSdADDR, HSdA, HSpxlY, HSpxlX, HScntX,
          HSpxlL, HStxlL, HSpxlR, HStxlR);

     //HS_LATCH_TMOD
     always @(posedge REclk)
     begin
          if(clk & HSfetch_TMOD)     HStmod <= EPtmod;
     end
     //HS_LATCH_ADDR
     always @(posedge REclk)
     begin
          if(clk & HSfetch_ADDR) HSdADDR <= EPdADDR;
     end
     //HS_LATCH_ALPHA
     always @(posedge REclk)
     begin
          if(clk & HSfetch_ALPHA) HSdA <= EPdA;
     end
     //HS_LATCH_PXL
     always @(posedge REclk)
     begin
          if(clk & HSfetch_PXL)
          begin
               HSpxlY <= EPpxlY;
               HSpxlX <= EPpxlX;
               HScntX <= EPcntX;
               HSpxlL <= EPpxlL;
               HSpxlR <= EPpxlR;
          end
     end
     //HS_LATCH_TXL
     always @(posedge REclk)
     begin
          if(clk & HSfetch_TXL)
          begin
               HStxlL <= EPtxlL;
               HStxlR <= EPtxlR;
          end
     end
endmodule
```

//Depth buffer interface
Whs_zbi   sshs_zbi();

```
// Z-buffer interface
module    Whs_zbi(REclk, HScZB, HSnhld_int,
          HS_ZB0cmd, HS_ZB1cmd, HS_ZB0addr, HS_ZB1addr, PInwait, HSreset, HScOP,
          ZBI_HSreset, ZBI_ZB0cmd, ZBI_ZB1cmd, ZBI_ZB0addr, ZBI_ZB1addr);

     //Reset
     assign #1 ZBI_HSreset = HSreset;

     /* Z-buffer command/address interface */
     //Latched @ negative REclk for large skew-free window
     always @(negedge REclk)
     begin
          //Command - RCLR : masking out depending on HScZB
          if(HScOP[0]) begin
               ZBI_ZB0cmd <= (HScZB) ? HS_ZB0cmd : `WFZB_CMD_NOP;
               ZBI_ZB1cmd <= (HScZB) ? HS_ZB1cmd : `WFZB_CMD_NOP; end
          //Command otherwise
          else begin
               ZBI_ZB0cmd <= HS_ZB0cmd;
               ZBI_ZB1cmd <= HS_ZB1cmd; end
          //Address
          ZBI_ZB0addr <= HS_ZB0addr;
          ZBI_ZB1addr <= HS_ZB1addr;
     end
endmodule
```

//7-way SIMD subtractor
Whs_lrsub_7simd   sshs_lrsub_7simd();

```
module     Whs_lrsub_7simd(HSpxlL, HStxlL, HSpxlR, HStxlR, HSsubPXL, HSsubTXL);

     /* Substractor */
     DW01_sub #(17)    sub_Z(   .A           ({1'b0, HSpxlR[39:24]}),
                                .B           ({1'b0, HSpxlL[39:24]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubPXL[43:27]),
                                .CO          (coutZ));
     DW01_sub #(9)     sub_R(   .A           ({1'b0, HSpxlR[23:16]}),
                                .B           ({1'b0, HSpxlL[23:16]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubPXL[26:18]),
                                .CO          (coutR));
     DW01_sub #(9)     sub_G(   .A           ({1'b0, HSpxlR[15:8]}),
                                .B           ({1'b0, HSpxlL[15:8]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubPXL[17:9]),
                                .CO          (coutG));
     DW01_sub #(9)     sub_B(   .A           ({1'b0, HSpxlR[7:0]}),
                                .B           ({1'b0, HSpxlL[7:0]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubPXL[8:0]),
                                .CO          (coutB));
     DW01_sub #(33)    sub_U(   .A           ({1'b0, HStxlR[95:64]}),
                                .B           ({1'b0, HStxlL[95:64]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubTXL[98:66]),
                                .CO          (coutU));
      DW01_sub #(33)    sub_V(  .A           ({1'b0, HStxlR[63:32]}),
                                .B           ({1'b0, HStxlL[63:32]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubTXL[65:33]),
                                .CO          (coutV));
      DW01_sub #(33)    sub_W(  .A           ({1'b0, HStxlR[31:0]}),
                                .B           ({1'b0, HStxlL[31:0]}),
                                .CI          (1'b0),
                                .DIFF        (HSsubTXL[32:0]),
                                .CO          (coutW));
endmodule
```

```
//7-way SIMD divider
Whs_div_7simd   sshs_div_7simd();
```

```
//7-way SIMD divider
module     Whs_div_7simd(HSsubPXL, HSsubTXL, HScntX, HSdivPXL, HSdivTXL);

      /* Divide look-up table */
      WDIVLUT9    lut(    .tbin       (HScntX),
                          .tbout      (DIVsel),
                          .tbshift    (DIVshift));

      /* Bunch of multipliers */
      WMUL17x8    mul_Z(   .x      (HSsubPXL[43:27]),
                           .y      (DIVsel),
                           .z      (HSdivPXLs[75:51]));
      WMUL9x8     mul_R(   .x      (HSsubPXL[26:18]),
                           .y      (DIVsel),
                           .z      (HSdivPXLs[50:34]));
      WMUL9x8     mul_G(   .x      (HSsubPXL[17:9]),
                           .y      (DIVsel),
                           .z      (HSdivPXLs[33:17]));
      WMUL9x8     mul_B(   .x      (HSsubPXL[8:0]),
                           .y      (DIVsel),
                           .z      (HSdivPXLs[16:0]));
      WMUL33x8    mul_U(   .x      (HSsubTXL[98:66]),
                           .y      (DIVsel),
                           .z      (HSdivTXLs[122:82]));
      WMUL33x8    mul_V(   .x      (HSsubTXL[65:33]),
                           .y      (DIVsel),
                           .z      (HSdivTXLs[81:41]));
      WMUL33x8    mul_W(   .x      (HSsubTXL[32:0]),
                           .y      (DIVsel),
                           .z      (HSdivTXLs[40:0]));


      /* Shift module */
      Whs_shift_25    shift_Z(   .din(HSdivPXLs[ 75:51]),
                                 .dout(HSdivPXL[ 75:51]), .shift(DIVshift));
      Whs_shift_17    shift_R(   .din(HSdivPXLs[ 50:34]),
                                 .dout(HSdivPXL[ 50:34]), .shift(DIVshift));
      Whs_shift_17    shift_G(   .din(HSdivPXLs[ 33:17]),
                                 .dout(HSdivPXL[ 33:17]), .shift(DIVshift));
      Whs_shift_17    shift_B(   .din(HSdivPXLs[ 16: 0]),
                                 .dout(HSdivPXL[ 16: 0]), .shift(DIVshift));
      Whs_shift_41    shift_U(   .din(HSdivTXLs[122:82]),
                                 .dout(HSdivTXL[122:82]), .shift(DIVshift));
      Whs_shift_41    shift_V(   .din(HSdivTXLs[ 81:41]),
                                 .dout(HSdivTXL[ 81:41]), .shift(DIVshift));
      Whs_shift_41    shift_W(   .din(HSdivTXLs[ 40: 0]),
                                 .dout(HSdivTXL[ 40: 0]), .shift(DIVshift));
endmodule
```

```
endmodule
```

7.5.4.7 PI: Pixel Interpolation

The PI stage (Figure 7.22) computes pixel values using the edge pixel and the horizontal slope. Also, the depth test is performed in the PI stage. The previous depth data stored in the depth buffer and the depth data of the current pixel are compared and the prior pixel, from the view point, is stored. If the current pixel is a later pixel than the previous pixel, the rest of the pipelines are turned off.

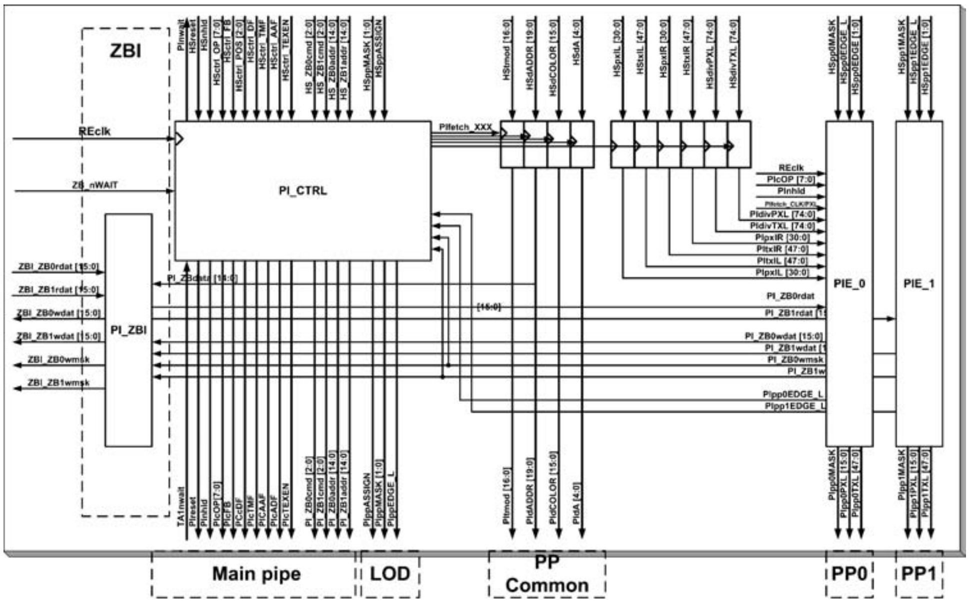


Figure 7.22 Pixel interpolation

Pipeline Input

- REclk: redering engine clock
- Pipe control signals: same as HS
- Memory bypass signals: ZB0, ZB1, TM control signals
- PP common signals: edge pixel data
- PP dependent signals
- ZB_nWAIT: arbitration wait signal

Pipeline Output

- Main pipe signals
- LOD signals
- Common signals
- PP dependent signals

Functions

- Pixel/texel horizontal interpolation
- Depth test
- Z-buffer databus interface
- Turing off following "datapath" if z-test fails

Signal Descriptions

Datapath signal	Description
HSpxlLR	{Z[30:16],R[15:11],G[10:5],B[4:0]}
PIpp#PXL	R, G, B: 5:6:5 unsigned data
	Z: 15-bit unsigned data
HStxlLR	{U[47:32],V[31:16],W[15:0]}
PIpp#TXL	U, V, W: 16-bit unsigned data
HSdivPXL	{Z[74:51],R[50:34],G[33:17],B[16:0]}
	Z: 24-bit signed data
	- {sign_1b, data_16b, frac_8b}
	R, G, B: 17-bit signed data
	- {sign_1b, data_8b, frac_8b}
HSdivTXL	{U[74:50],V[49:25],W[24:0]}
	25-bit signed data
	- {sign_1b, data_16b, frac_8b}

Control signal	Description
PI_ZBmode[2:0]	Z-buffer interface write-bus-mux control
	ZBI_ZB#wdat[15:0]
	3'b100: Write 16'b0

	3’b010: Driving {1’b0, PI_ZBdata[14:0]}
	3’b001: Bypass PI_ZB#wdat(Default)
	ZBI_ZB#wmsk
	3’b100: Force to 1
	3’b010: Force to 1
	3’b001: Bypass PI_ZB#wmsk (default)
PIppMASK[1:0]	“Real mask”
	HSppMASK & PI_ZB#wmsk (bitwise operation)
PIpp0MASK	“Bypass mask”
PIpp1MASK	Bypass HSpp#MASK
	To conserve u,v,w for LOD calculation

RTL Code

```
/*
* RAMP-GR
* RAMP-GR      PI Module: Pixel Interpolation
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wpi(  REclk, TA1nwait, HSreset, HSnhld, HScOP, HScFB,
             HScDF, HScTMF, HScAF, HScTEXEN, HScFMB, HScTMB, HScSCR,
             HS_ZB0cmd, HS_ZB1cmd, HS_ZB0addr, HS_ZB1addr, HSppMASK,
             HSppASSIGN, HStmod, HSdADDR, HSdA, HSpxlL, HStxlL, HSpxlR,
             HStxlR, HSdivPXL, HSdivTXL, HSpp0MASK, HSpp0EDGE_L,
             HSpp0EDGE, HSpp1MASK, HSpp1EDGE_L, HSpp1EDGE,
             ZBI_ZB0rdat, ZBI_ZB1rdat,
             PInwait, PIreset, PInhld, PIcOP, PIcFB, PIcDF, PIcTMF, PIcAF,
             PIcTEXEN, PIcFMB, PIcTMB, PIcSCR,
             PI_ZB0cmd, PI_ZB1cmd, PI_ZB0addr, PI_ZB1addr, PIppASSIGN,
             PIppMASK, PIppEDGE_L, PItmod, PIdADDR, PIdA,
             PIpp0MASK, PIpp0PXL, PIpp0TXL, PIpp1MASK, PIpp1PXL, PIpp1TXL,
             ZBI_ZB0wdat, ZBI_ZB1wdat, ZBI_ZB0wmsk, ZBI_ZB1wmsk);

     /* Module interconnection */

//Main controller
Wpi_ctrl    sspi_ctrl();
```

```
//Main controller
module   Wpi_ctrl(  REclk, TA1nwait, HSreset, HSnhld, HScOP, HScFB, HScDF,
HScTMF,
                HScAF, HScTEXEN, HScFMB, HScTMB, HScSCR,
                HS_ZB0cmd, HS_ZB1cmd, HS_ZB0addr, HS_ZB1addr, HSppMASK,
                HSppASSIGN, PIpp0EDGE_L, PIpp1EDGE_L, PI_ZB0wmsk,
                PI_ZB1wmsk,  PInwait,  PIreset,  PInhld,  PIcOP,  PIcFB,  PIcDF,
PIcTMF,
                PIcAF, PIcTEXEN, PIcFMB, PIcTMB, PIcSCR,
                PI_ZB0cmd, PI_ZB1cmd, PI_ZB0addr, PI_ZB1addr,
                PIppASSIGN, PIppMASK, PIppMASK_int, PIppEDGE_L, PI_ZBmode,
                PIfetch_CLK, PIfetch_TMOD, PIfetch_ADDR, PIfetch_ALPHA,
                PIfetch_PXL, PIfetch_TXL);

    /* Pipeline control signal generation */

    //nWait
    assign   PInwait = TA1nwait | ~HSreset;
    //nHld
    assign #1   PInhld = PInhld_int;
    //Fetch enable
    assign #1   PIfetch_CLK = PInwait & HSnhld;
    assign   PIfetch_TMOD = HScTMF;
    assign   PIfetch_ADDR = HScAF;
    assign   PIfetch_ALPHA = HScDF;
    assign   PIfetch_PXL = HScDF;
    assign   PIfetch_TXL = HScDF & HScTEXEN;
    //LOD control
    assign #1   PIppMASK[1] = PIppMASK_int[1] & PI_ZB0wmsk;
    assign #1   PIppMASK[0] = PIppMASK_int[0] & PI_ZB1wmsk;
    assign #1   PIppEDGE_L = PIpp0EDGE_L | PIpp1EDGE_L;
    always @(posedge REclk)
    begin
         if(PInwait)
         begin
              //State controls
              PInhld_int <= HSnhld;
              PIreset <= HSreset;
              //Trivial signals
              PIcOP <= HScOP;
              PIcFB <= HScFB;
              PIcSCR <= HScSCR;
              PI_ZB0cmd <= HS_ZB0cmd;
              PI_ZB1cmd <= HS_ZB1cmd;
              PI_ZB0addr <= HS_ZB0addr;
              PI_ZB1addr <= HS_ZB1addr;
              //Fetch Signals
              PIcDF_int <= HScDF;
              PIcTMF_int <= HScTMF;
              PIcAF_int <= HScAF;
              PIcTEXEN_int <= HScTEXEN;
              PIcFMB_int <= HScFMB;
              PIcTMB_int <= HScTMB;
              //LOD Signals
              PIppASSIGN <= HSppASSIGN;
              PIppMASK_int <= HSppMASK;
         end
    end

    always @(negedge REclk)
    begin
         PIcDF <= PIcDF_int & PInhld;
         PIcTMF <= PIcTMF_int & PInhld;
         PIcAF <= PIcAF_int & PInhld;
         PIcTEXEN <= PIcTEXEN_int & PInhld;
```

```
          PIcFMB <= PIcFMB_int & PInhld;
          PIcTMB <= PIcTMB_int & PInhld;
     end

     /* Z-buffer interface signal generation */

     //This signal is overridden by ZBcmd
     always @(PIcOP or PInhld)
     begin
          if(PInhld & PIcOP[0])   PI_ZBmode <= 'WPI_ZBI_ZERO;    //RCLR;
          else                    PI_ZBmode <= 'WPI_ZBI_BYPASS;  //RSHA / RTEX
     end
endmodule
```

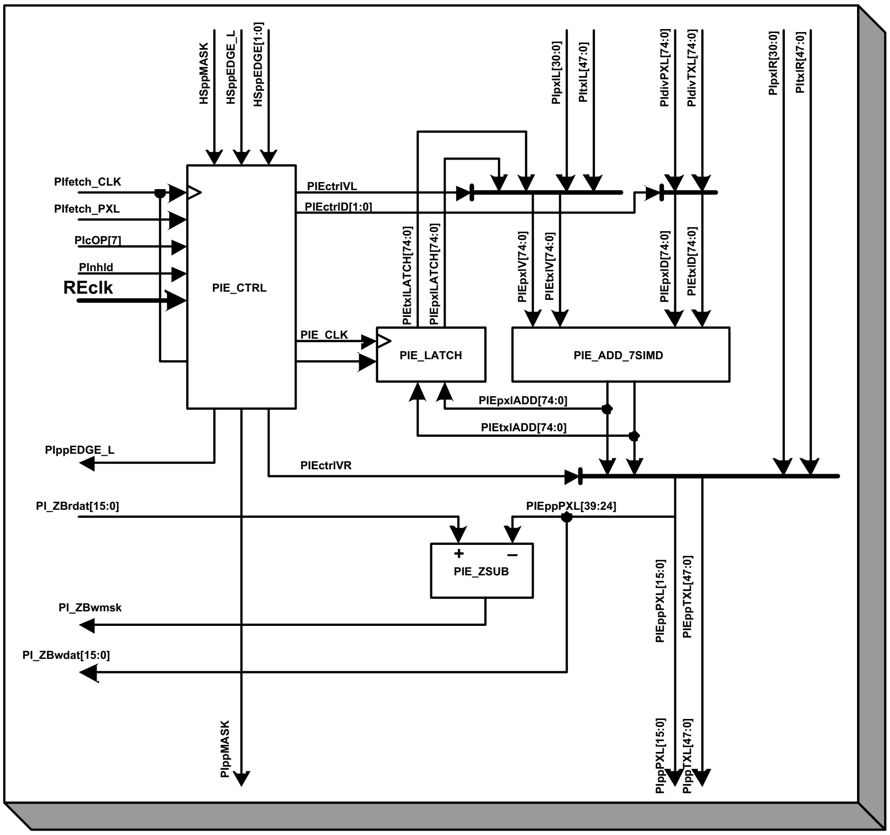
//Data latches
Wpi_latch   sspi_latch();

```
//Data latches
module   Wpi_latch(  REclk, clk, PIfetch_TMOD, PIfetch_ADDR, PIfetch_ALPHA,
               PIfetch_PXL, PIfetch_TXL, HStmod, HSdADDR, HSdA,
               HSpxlL, HStxlL, HSpxlR, HStxlR, HSdivPXL, HSdivTXL,
               PItmod, PIdADDR, PIdA, PIpxlL, PItxlL, PIpxlR, PItxlR,
               PIdivPXL, PIdivTXL);

     //PI_LATCH_TMOD
     always @(posedge REclk)
     begin
          if(clk&PIfetch_TMOD)
          PItmod <= HStmod;
     end
     //PI_LATCH_ADDR
     always @(posedge REclk)
     begin
          if(clk&PIfetch_ADDR)
          PIdADDR <= HSdADDR;
     end
     //PI_LATCH_ALPHA
     always @(posedge REclk)
     begin
          if(clk&PIfetch_ALPHA)
          PIdA <= HSdA;
     end
     //PI_LATCH_PIXEL
     always @(posedge REclk)
     begin
          if(clk &PIfetch_PXL)
          begin
               PIpxlL <= HSpxlL;
               PIpxlR <= HSpxlR;
               PIdivPXL <= HSdivPXL;
          end
     end
     //PI_LATHCH_TEXEL
     always @(posedge REclk)
     begin
          if(clk & PIfetch_TXL)
          begin
               PItxlL <= HStxlL;
               PItxlR <= HStxlR;
               PIdivTXL <= HSdivTXL;
          end
     end
endmodule
```

```
// PI_ZBI
Wpi_zbi       sspi_zbi();
```

```
//Z-buffer interface
module    Wpi_zbi(    PI_ZBmode, PIppMASK_int, ZBI_ZB0rdat, ZBI_ZB1rdat,
          ZBI_ZB0wdat, ZBI_ZB1wdat, ZBI_ZB0wmsk, ZBI_ZB1wmsk,
          PI_ZB0rdat, PI_ZB1rdat, PI_ZB0wdat, PI_ZB1wdat,
          PI_ZB0wmsk, PI_ZB1wmsk);

     //ZB -> PI
     assign #1  PI_ZB0rdat = ZBI_ZB0rdat;
     assign #1  PI_ZB1rdat = ZBI_ZB1rdat;

     //PI -> ZB
     always @(PI_ZBmode or PIppMASK_int or PI_ZB0wdat or PI_ZB1wdat or PI_ZB0wmsk
           or PI_ZB1wmsk)
     begin
          casex(PI_ZBmode)
               `WPI_ZBI_ZERO : begin
                    ZBI_ZB0wdat <= 16'b0;
                    ZBI_ZB1wdat <= 16'b0;
                    ZBI_ZB0wmsk <= 1'b1;
                    ZBI_ZB1wmsk <= 1'b1; end
               `WPI_ZBI_BYPASS : begin
                    ZBI_ZB0wdat <= PI_ZB0wdat;
                    ZBI_ZB1wdat <= PI_ZB1wdat;
                    ZBI_ZB0wmsk <= PI_ZB0wmsk & PIppMASK_int[1];
                    ZBI_ZB1wmsk <= PI_ZB1wmsk & PIppMASK_int[0]; end
               default : begin
                    ZBI_ZB0wdat <= 16'bX;
                    ZBI_ZB1wdat <= 16'bX;
                    ZBI_ZB0wmsk <= 1'bX;
                    ZBI_ZB1wmsk <= 1'bX; end
          endcase
     end
endmodule
```

```
//PIE_0
Wpi_engine     sspi_pie0();
//PIE_1
Wpi_engine     sspi_pie1();
//PI engine is in charge of the pixel interpolation operation.
//See the chart.
```



Datapath signals	Description
PIEctrlVL	Left vertex selection
	= HSppEDGE_L
	0: select LATCH
	1: bypass PI_L
PIEctrlVR	Right vertex selection
	= HSppEDGE[0]
	0: select ADD
	1: select PI_R
PIEctrlD[1:0]	Interpolation delta selection
	={EDGE_L,EDGE[1]}
	0X: PI_D<<1 (x2 generation)
	10: PI_D (x1 bypass)
	11: 0 (zero blocking)
PIE_CLK	Fetch signal

	= (PIcOP & PInhld & PIppMASK) & REclk
	Internal enable signal (@ negedge REclk)
PIE_ADD_7SIMD	Input: sign extension + data + fraction padding
	Output: data only
PIE_ZSUB	Input: sign extension
	Output: MSB 1-bit only

```
//PIE main controller
module   Wpie_ctrl(  REclk, PIfetch_CLK, PIfetch_PXL, PIcOP, PInhld,
         PIppEDGE_L, HSppMASK, HSppEDGE_L, HSppEDGE,
         PIppMASK, PIEctrlVL, PIEctrlVR, PIEctrlD, PIE_CLK);

     assign  PIE_CLK = PIE_FETCH;

     always @(posedge REclk)
     begin
          if(PIfetch_CLK&PIfetch_PXL)
          begin
               PIppEDGE_L <= HSppEDGE_L;
               PIppMASK  <= HSppMASK;
               PIEctrlVL <= HSppEDGE_L;
               PIEctrlVR <= HSppEDGE[0];
               PIEctrlD  <= {HSppEDGE_L,HSppEDGE[1]};
          end
     end

     always @(negedge REclk)
     begin
          PIE_FETCH <= PIcOP & PInhld & PIppMASK;
     end
endmodule

//7-way SIMD adder with sign extended input
//and trimmed output
module  Wpie_add_7simd(PIEpxlV, PIEtxlV, PIEpxlD, PIEtxlD, PIEpxlADD,
                       PIEtxlADD);

     /* Adders */
     DW01_add #(25)   add_Z(  .A          (PIEpxlV[75:51]),
                              .B          (PIEpxlD[75:51]),
                              .CI         (1'b0),
                              .SUM        (PIEpxlADD[75:51]),
                              .CO         (coutZ));
     DW01_add #(17)   add_R(  .A          (PIEpxlV[50:34]),
                              .B          (PIEpxlD[50:34]),
                              .CI         (1'b0),
                              .SUM        (PIEpxlADD[50:34]),
                              .CO         (coutR));
     DW01_add #(17)   add_G(  .A          (PIEpxlV[33:17]),
                              .B          (PIEpxlD[33:17]),
                              .CI         (1'b0),
                              .SUM        (PIEpxlADD[33:17]),
                              .CO         (coutG));
     DW01_add #(17)    add_B(  .A          (PIEpxlV[16:0]),
                              .B          (PIEpxlD[16:0]),
                              .CI         (1'b0),
                              .SUM        (PIEpxlADD[16:0]),
                              .CO         (coutB));
     DW01_add #(41)    add_U(  .A          (PIEtxlV[122:82]),
                              .B          (PIEtxlD[122:82]),
                              .CI         (1'b0),
                              .SUM        (PIEtxlADD[122:82]),
                              .CO         (coutU));
     DW01_add #(41)    add_V(  .A          (PIEtxlV[81:41]),
                              .B          (PIEtxlD[81:41]),
                              .CI         (1'b0),
                              .SUM        (PIEtxlADD[81:41]),
                              .CO         (coutV));
     DW01_add #(41)    add_W(  .A          (PIEtxlV[40:0]),
                              .B          (PIEtxlD[40:0]),
                              .CI         (1'b0),
                              .SUM        (PIEtxlADD[40:0]),
                              .CO         (coutW));
endmodule

module   Wpi_engine (REclk, PIcOP, PInhld, PIfetch_CLK, PIfetch_PXL,
          HSppMASK, HSppEDGE_L, HSppEDGE,
          PIpxlL, PItxlL, PIpxlR, PItxlR, PIdivPXL, PIdivTXL,
          PI_ZBrdat, PIppEDGE_L, PI_ZBwdat, PI_ZBwmsk,
          PIppMASK, PIppPXL, PIppTXL);

     //PIE main controller
     Wpie_ctrl          sspie_ctrl();
     //PIE_ADD_7SIMD
     Wpie_add_7simd     sspie_add_7simd();
     //PIE_ZSBUB
     DW01_sub #(17)    sspie_zsub(
                       .A          ({1'b0, PI_ZBrdat}),
                       .B          ({1'b0, PIEppPXL[39:24]}),
                       .CI         (1'b0),
                       .DIFF       ({PI_ZBwmsk, subZ}),
                       .CO         (coutZ));

     /* Flattened logics */

     //PIE_SEL_VL
     always @(PIEctrlVL or PIpxlL or PItxlL or PIEpxlLATCH or PIEtxlLATCH)
     begin
          casex (PIEctrlVL)
          1'b0    : begin
                  PIEpxlV <= PIEpxlLATCH;
                  PIEtxlV <= PIEtxlLATCH; end
          1'b1    :begin
                  PIEpxlV[ 75:51] <= {1'b0, PIpxlL[39:24], 8'b0}; //Z
                  PIEpxlV[ 50:34] <= {1'b0, PIpxlL[23:16], 8'b0}; //R
                  PIEpxlV[ 33:17] <= {1'b0, PIpxlL[15: 8], 8'b0}; //G
                  PIEpxlV[ 16: 0] <= {1'b0, PIpxlL[ 7: 0], 8'b0}; //B
```

```
               PIEtxlV[122:82] <= {1'b0, PItxlL[95:64], 8'b0}; //U
               PIEtxlV[ 81:41] <= {1'b0, PItxlL[63:32], 8'b0}; //V
               PIEtxlV[ 40: 0] <= {1'b0, PItxlL[31: 0], 8'b0}; end //W
         endcase
     end
     //PIE_SEL_VR
     always @(PIEctrlVR or PIEpxlADD or PIEtxlADD or PIpxlR or PItxlR)
     begin
         casex(PIEctrlVR)
         1'b0    : begin
               PIEppPXL [39:24] <= PIEpxlADD[ 74:59]; //Z
               PIEppPXL [23:16] <= PIEpxlADD[ 49:42]; //R
               PIEppPXL [15: 8] <= PIEpxlADD[ 32:25]; //G
               PIEppPXL [ 7: 0] <= PIEpxlADD[ 15: 8]; //B
               PIEppTXL [95:64] <= PIEtxlADD[121:90]; //U
               PIEppTXL [63:32] <= PIEtxlADD[ 81:49]; //V
               PIEppTXL [31: 0] <= PIEtxlADD[ 39: 8]; end //W
         1'b1    : begin
               PIEppPXL <= PIpxlR;
               PIEppTXL <= PItxlR; end
         endcase
     end
     //PIE_SEL_D
     always @(PIEctrlD or PIdivPXL or PIdivTXL)
     begin
         casex(PIEctrlD)
         2'b00   : begin // x2 generation
               PIEpxlD[ 75:51] <= {PIdivPXL[ 73:51], 1'b0}; //Z
               PIEpxlD[ 50:34] <= {PIdivPXL[ 49:34], 1'b0}; //R
               PIEpxlD[ 33:17] <= {PIdivPXL[ 32:17], 1'b0}; //G
               PIEpxlD[ 16: 0] <= {PIdivPXL[ 15: 0], 1'b0}; //B
               PIEtxlD[122:82] <= {PIdivTXL[121:82], 1'b0}; //U
               PIEtxlD[ 81:41] <= {PIdivTXL[ 80:41], 1'b0}; //V
               PIEtxlD[ 40: 0] <= {PIdivTXL[ 39: 0], 1'b0}; end //W
         2'b10   : begin //x1 bypass
               PIEpxlD <= PIdivPXL;
               PIEtxlD <= PIdivTXL; end
         2'b11   : begin //zero blocking
               PIEpxlD <= 76'b0;
               PIEtxlD <= 123'b0; end
               default : begin
               PIEpxlD <= 76'bX;
               PIEtxlD <= 123'bX; end
         endcase
     end
     //PIE_LATCH
     always @(posedge REclk)
     begin
         //if(PIfetch_CLK&PIE_CLK)
         if(PIfetch_CLK)
         begin
             PIEpxlLATCH <= PIEpxlADD;
             PIEtxlLATCH <= PIEtxlADD;
         end
```

```
      end

      /* Output Re-wiring */

      assign #1  PIppPXL = {PIEppPXL[23:16], PIEppPXL[15:8], PIEppPXL[7:0]};
      assign #1  PIppTXL = PIEppTXL;
      assign #1  PI_ZBwdat = PIEppPXL[39:24];
 endmodule
```

endmodule

7.5.4.8 TA1: Texture Address #1

See Figure 7.23.

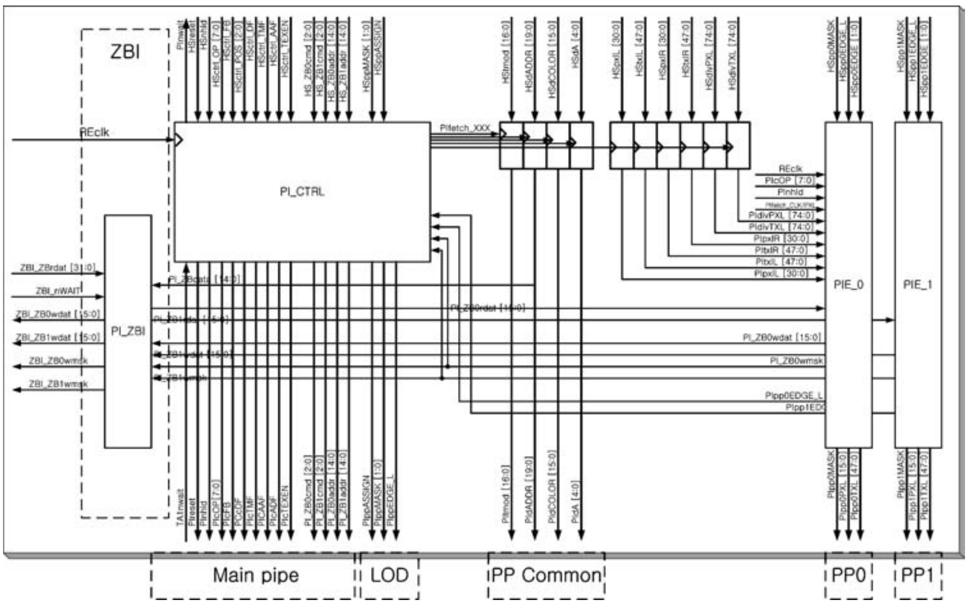


Figure 7.23 Texture address #1

Pipeline Input

- REclk: rendering clock
- Pipeline control signal: same as PI
- Memory bypass signal: same as PI
- PP common signal
- PP dependent signal

Pipeline Output

- Pipeline control signals
- Common signals
- PP dependent signals

Functions

- UV calculation with u/w, v/w
- “Really” turning off following datapath if PPmask = 0

Signal Descriptions

Control signal	Description
TA1nhld	Normal
TA1ppFETCH	TA1enable & REclk TA1enable = TA1nhld_int & TA1cOP[7] & TA1cTEXEN (@ negedge)
TA1tmod[12:0]	BLND:FILT:SIZE = {PItmod[16:13],Pitmod[8:0]}
TA1tmod_BIAS[3:0]	Texture LOD bias control = PItmod[12:9] Bias control 0XXX: unbiased 1AAA: Set to AAA
TA1tmod_SIZE[8:0]	Texture size = PItmod[8:0] Size control 1_0000_0000: 256×256 0_1000_0000: 128×128 . . .
TA1lod[2:0]	Texture LOD 000: LOD_0 001: LOD_1 010: LOD_2 . . .

Datapath signal	Description
TA1pp#U[11:0]	8-bit texture address with 4-bit fraction
TA1pp#V[11:0]	Multiplied (shifted) by texture size

RTL Code

```
/*
* RAMP-GR
* RAMP-GR      TA1 Module : Texture Address #1
* by Jeong-Ho Woo ( denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wta1(    REclk, TA2nwait, PIreset, PInhld, PIcOP, PIcFB, PIcDF, PIcTMF,
          PIcAF, PIcTEXEN, PIcFMB, PIcTMB, PIcSCR,
          PI_ZB0cmd, PI_ZB1cmd, PI_ZB0addr, PI_ZB1addr,
          PIppASSIGN, PIppMASK, PIppEDGE_L, PItmod, PIdADDR, PIdA,
          PIpp0MASK, PIpp0PXL, PIpp0TXL, PIpp1MASK, PIpp1PXL, PIpp1TXL,
          TA1nwait, TA1reset, TA1nhld, TA1cOP, TA1cFB, TA1cDF, TA1cTMF,
          TA1cAF, TA1cTEXEN, TA1cFMB, TA1cTMB, TA1cSCR,
          TA1_ZB0cmd, TA1_ZB1cmd, TA1_ZB0addr, TA1_ZB1addr,
          TA1ppMASK, TA1ppASSIGN, TA1tmod, TA1dADDR, TA1dA,
          TA1pp0VALID, TA1pp0RGB, TA1pp0U, TA1pp0V,
          TA1pp1VALID, TA1pp1RGB, TA1pp1U, TA1pp1V);

    /* Wire renaming */
    assign #1  TA1pp0VALID = TA1ppMASK[1];
    assign #1  TA1pp1VALID = TA1ppMASK[0];

    /* Module interconnection */


//Main controller
Wta1_ctrl     ssta1_ctrl();
```

```
//Main controller
module   Wta1_ctrl(  REclk, TA2nwait, PIreset, PInhld, PIcOP, PIcFB, PIcDF,
         PIcTMF, PIcAF, PIcTEXEN, PIcFMB, PIcTMB, PIcSCR,
         PI_ZB0cmd, PI_ZB1cmd, PI_ZB0addr, PI_ZB1addr, PIppASSIGN, PIppMASK,
         PIppEDGE_L, TA1nwait, TA1reset, TA1nhld, TA1cOP, TA1cFB, TA1cDF,
         TA1cTMF, TA1cAF, TA1cTEXEN, TA1cFMB, TA1cTMB, TA1cSCR,
         TA1_ZB0cmd, TA1_ZB1cmd, TA1_ZB0addr, TA1_ZB1addr,
         TA1ppMASK, TA1ppASSIGN,
         TA1fetch_CLK, TA1fetch_TMOD, TA1fetch_ADDR, TA1fetch_ALPHA,
         TA1fetch_PXL, TA1fetch_TXL);

    //nWait
    assign  TA1nwait = TA2nwait | ~PIreset;
    //nhld
    assign #1  TA1nhld = TA1nhld_int;
    //Fetch enable
    assign #1  TA1fetch_CLK = TA1nwait & PInhld;
    assign  TA1fetch_TMOD = PIcTMF;
    assign  TA1fetch_ADDR = PIcAF;
    assign  TA1fetch_ALPHA = PIcDF;
    assign  TA1fetch_PXL = PIcDF;
    assign  TA1fetch_TXL = PIcDF & PIcTEXEN;

    always @(posedge REclk)
    begin
        if(TA1nwait)
        begin
            //State controls
```

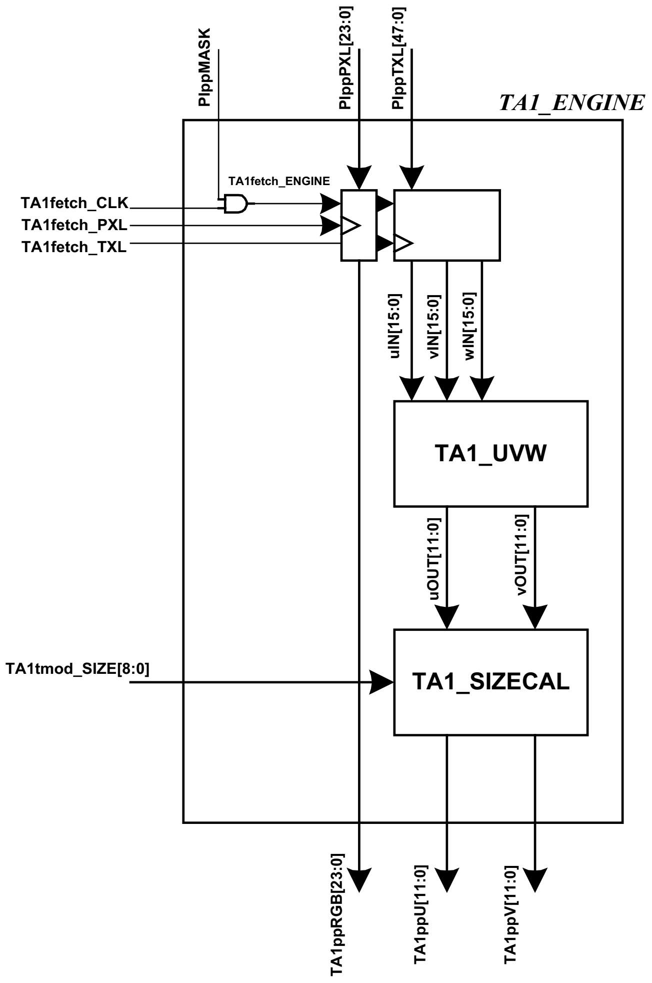
```
                TA1nhld_int <= PInhld;
                TA1reset <= PIreset;
                //Fetch signals
                TA1cDF_int <= PIcDF;
                TA1cTMF_int <= PIcTMF;
                TA1cAF_int <= PIcAF;
                TA1cTEXEN_int <= PIcTEXEN;
                TA1cFMB_int <= PIcFMB;
                TA1cTMB_int <= PIcTMB;
                //Trivial signals
                TA1cOP <= PIcOP;
                TA1cFB <= PIcFB;
                TA1cSCR <= PIcSCR;
                TA1_ZB0cmd <= PI_ZB0cmd;
                TA1_ZB1cmd <= PI_ZB1cmd;
                TA1_ZB0addr <= PI_ZB0addr;
                TA1_ZB1addr <= PI_ZB1addr;
                //PP control
                TA1ppMASK <= PIppMASK;
                TA1ppASSIGN <= PIppASSIGN;
           end
      end

      always @(negedge REclk)
      begin
           TA1cDF <= TA1cDF_int;
           TA1cTMF <= TA1cTMF_int;
           TA1cAF <= TA1cAF_int;
           TA1cTEXEN <= TA1cTEXEN_int;
           TA1cFMB <= TA1cFMB_int;
           TA1cTMB <= TA1cTMB_int;
      end
 endmodule
```

//Data latches
Wta1_latch   ssta1_latch();

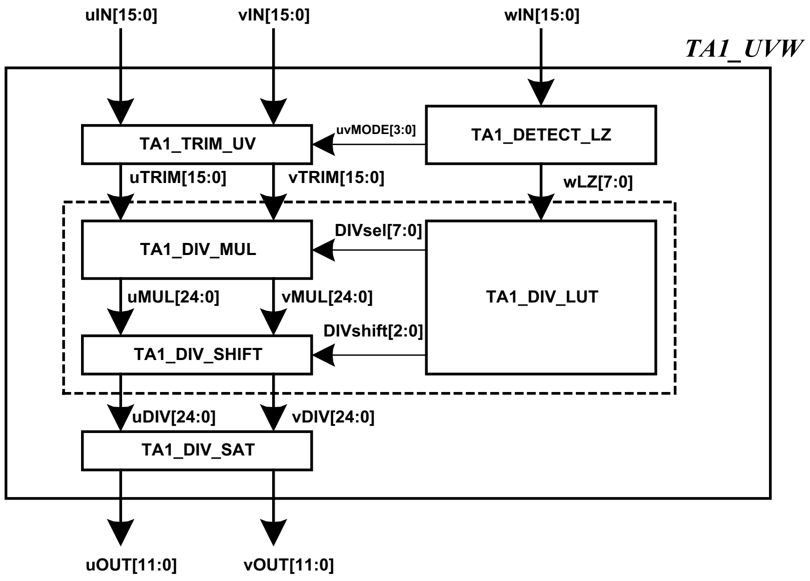
```
//Data latches
module   Wta1_latch(  REclk, clk, TA1fetch_TMOD, TA1fetch_ADDR, TA1fetch_ALPHA,
               PItmod, PIdADDR, PIdA,
               TA1tmod, TA1tmod_BIAS, TA1tmod_SIZE, TA1dADDR, TA1dA);.

     //TMOD
     always @(posedge REclk)
     begin
         if(clk & TA1fetch_TMOD)
         begin
           TA1tmod<={PItmod[16:13],PItmod[8:0]}; //BLND[1] : FILT [3] : SIZE[9]
           TA1tmod_BIAS <= PItmod [12:9];
           TA1tmod_SIZE <= PItmod [8:0];
         end
     end
```

```
      //ADDR
      always @(posedge REclk)
      begin
           if(clk & TA1fetch_ADDR) TA1dADDR <= PIdADDR;
      end
      //ALPHA
      always @(posedge REclk)
      begin
           if(clk & TA1fetch_ALPHA) TA1dA <= PIdA;
      end
endmodule
```

```
     /* Texture address generation unit */
     //TA_engine generates perspective correct texture address.
     //Datapath #1: see diagram G.
```



```
//Datapath #2: see diagram H.
```



```
Wta1_engine   ssta1_ta1e_0();
Wta1_engine   ssta1_ta1e_1();
```

```
//TA1_UVW
//TA1_UVW unit divides texture coordinate U and V by W.
//To reduce computation latency and powe consumption, it employs
//logarithmic datapath in divider.
//Details of the HDIV block are in the datapath file.

module      Wta1_uvw(uIN, vIN, wIN, uOUT, vOUT);

     //U/W
     Hdiv       div_U(    .inx          (uIN),
                          .iny          (wIN),
                          .outx         (uDIV_out));
     //V/W
     Hdiv       div_V(    .inx          (vIN),
                          .iny          (wIN),
                          .outx         (vDIV_out));

     //Zero devide by zero
     assign #1  uDIV = (uIN == 32'b0) ? 32'b0 : uDIV_out;
     assign #1  vDIV = (vIN == 32'b0) ? 32'b0 : vDIV_out;
     //TA1_DIV_SAT_U
     always @(uDIV)
     begin
          if(uDIV[16]) uOUT <= 12'b1111_1111_1111;
          else uOUT <= uDIV[15:4];
     end
```

```
      //TA1)DIV_SAT_V
      always @(vDIV)
      begin
           if(vDIV[16]) vOUT <= 12'b1111_1111_1111;
           else vOUT <= vDIV[15:4];
      end
endmodule
```

/* Main engine */

```
module   Wta1_engine(REclk, TA1fetch_CLK, TA1fetch_PXL, TA1fetch_TXL, PIpp-
MASK,
         PIppPXL, PIppTXL, TA1tmod_SIZE, TA1ppRGB, TA1ppU, TA1ppV);

      //Fetch signals
      assign #1  TA1fetch_ENGINE = TA1fetch_CLK & PIppMASK;

      /* Module interconnection */

      //TA1_ENGINE_LATCH_PXL
      always @(posedge REclk)
      begin
           if(TA1fetch_ENGINE & TA1fetch_PXL) TA1ppRGB <= PIppPXL;
      end
      //TA1_ENGINE_LATCH_TXL
      always @(posedge REclk)
      begin
           if(TA1fetch_ENGINE & TA1fetch_TXL)
           begin
                uIN <= PIppTXL[95:64];
                vIN <= PIppTXL[63:32];
                wIN <= PIppTXL[31:0];
           end
      end
      // UVW unit
      Wta1_uvw        ssta1_uvw();
      //TA1_Sizecal_u
      always @(TA1tmod_SIZE or uOUT)
      begin
           casex(TA1tmod_SIZE)
                9'b1_XXXX_XXXX   :   TA1ppU <= uOUT;
                9'bX_1XXX_XXXX   :   TA1ppU <= {1'b0, uOUT[11:1]};
                9'bX_X1XX_XXXX   :   TA1ppU <= {2'b0, uOUT[11:2]};
                9'bX_XX1X_XXXX   :   TA1ppU <= {3'b0, uOUT[11:3]};
                9'bX_XXX1_XXXX   :   TA1ppU <= {4'b0, uOUT[11:4]};
                9'bX_XXXX_1XXX   :   TA1ppU <= {5'b0, uOUT[11:5]};
                9'bX_XXXX_X1XX   :   TA1ppU <= {6'b0, uOUT[11:6]};
                9'bX_XXXX_XX1X   :   TA1ppU <= {7'b0, uOUT[11:7]};
                9'bX_XXXX_XXX1   :   TA1ppU <= {8'b0, uOUT[11:8]};
                default          :   TA1ppU <= 12'bX;
           endcase
      end
      //TA1_Sizecal_u
      always @(TA1tmod_SIZE or vOUT)
```

```
     begin
          casex(TA1tmod_SIZE)
               9'b1_XXXX_XXXX    :    TA1ppV <= vOUT;
               9'bX_1XXX_XXXX    :    TA1ppV <= {1'b0, vOUT[11:1]};
               9'bX_X1XX_XXXX    :    TA1ppV <= {2'b0, vOUT[11:2]};
               9'bX_XX1X_XXXX    :    TA1ppV <= {3'b0, vOUT[11:3]};
               9'bX_XXX1_XXXX    :    TA1ppV <= {4'b0, vOUT[11:4]};
               9'bX_XXXX_1XXX    :    TA1ppV <= {5'b0, vOUT[11:5]};
               9'bX_XXXX_X1XX    :    TA1ppV <= {6'b0, vOUT[11:6]};
               9'bX_XXXX_XX1X    :    TA1ppV <= {7'b0, vOUT[11:7]};
               9'bX_XXXX_XXX1    :    TA1ppV <= {8'b0, vOUT[11:8]};
               default           :    TA1ppV <= 12'bX;
          endcase
     end
endmodule
```

endmodule

7.5.4.9 TA2: Texture Address #2

In the TA2 stage (Figure 7.24), the bilinear texture addresses are generated and the cache address alignment (CAL) logic, the same as the address alignment logic (AAL) mentioned in Chapter 5, is activated. First the bilinear texture addresses are generated and then the addresses are compared with neighboring texture addresses and previous texture addresses. Finally, only the new texture addresses are transferred to the next pipeline stage.

Pipeline Inputs

- REclk: rendering engine clock
- Pipe control signals: same as TA1
- Memory bypass signals: same as TA1
- LOD signals
- PP common signals
- PP dependent signals

Pipeline Outputs

- Pipe controls
- Common signals
- CAL control
- PP dependent signals
- Texture memory address calculation

Functions

- Cache alignment logic core
- Generate $\times 8$ physical texture address (bilinear filtering)

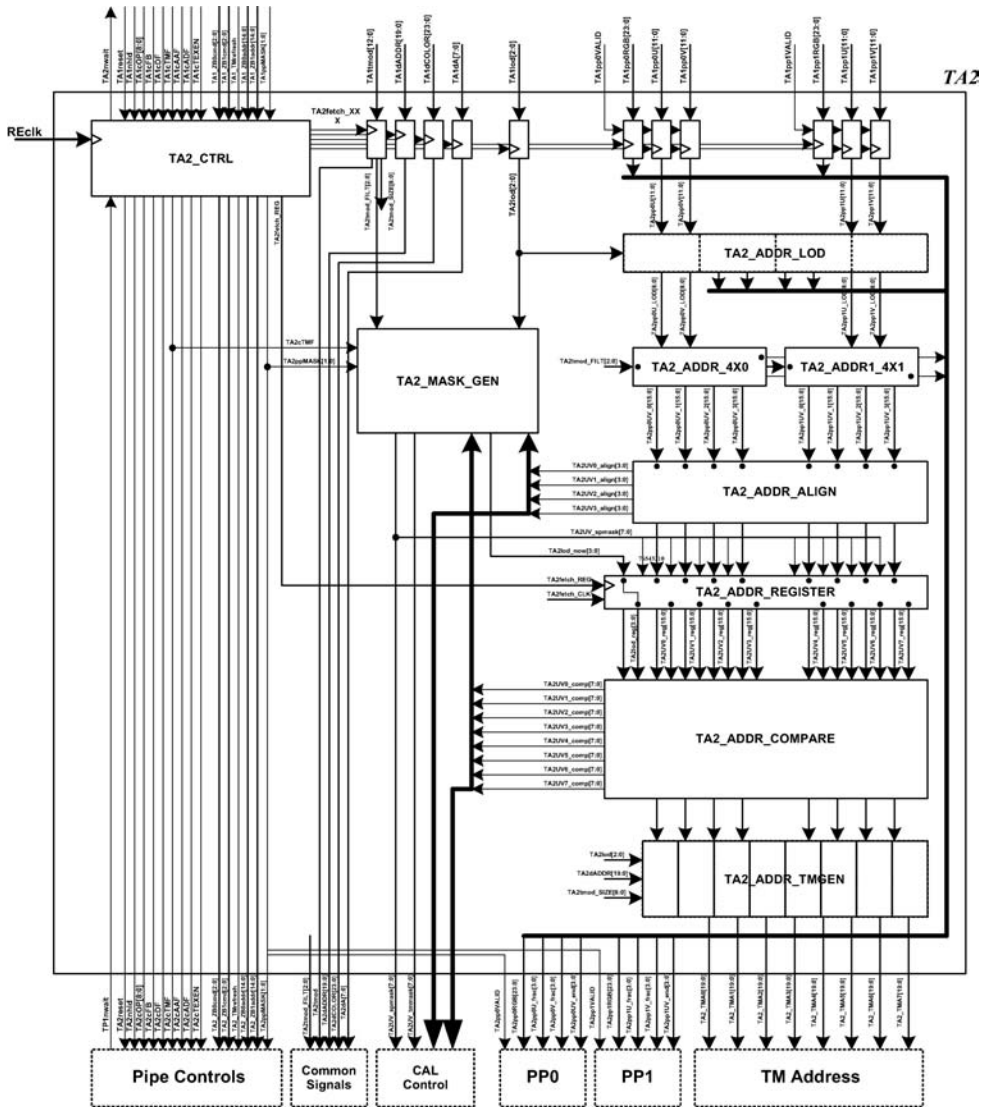
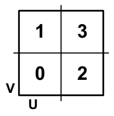


Figure 7.24 Texture address generation #2

Signal Descriptions

Control signal	Description
TA2pp#U_LOD[8:0]	Post LOD address: real address on LOD map
TA2pp#V_LOD[8:0]	{8-bit integer + 1-bit fraction}

TA2pp#U_frac[3:0]	Fraction of texture address	
TA2pp#V_frac[3:0]	For bilinear interpolation	
TA2pp#UV_N[15:0]	For bilinear texture address	
	{U[15:8],V[7:0]}	
	Generation method	
	0: {U-0.5, V-0.5}	1: {U-0.5, V + 0.5}
	2: {U + 0.5, V-0.5}	3: {U + 0.5, V + 0.5}



TA2pp#UV_end[3:0]	End point flag
	{U[3:2],V[1:0]}
	Flag
	2'd0: no end point
	2'd1: equals to zero
	2'd2: equals to one
TA2UV#_align[3:0]	Address align flag
	PP1 == PP0[0∼3] ?
	4'b0000: unique
	4'b1000: same as UV_0
	4'b0100: same as UV_1
	4'b0010: same as UV_2
	4'b0001: same as UV_3
TA2UV#_reg[15:0]	Cached address
TA2lod_now[3:0]	LOD address with RESET status
	Force to reset by TMF signal
	{TA2cTMF,LOD[2:0]}
TA2lod_reg[3:0]	Cached LOD address
TA2fetch_REG	Texture register fetch
	Verilog rule @ TA1
TA2UV#_comp[7:0]	Address register flag
	PP == PP_reg[0∼7] ?
	8'b0000_0000: unique
	8'b1000_0000: same as REG[0]
	8'b0100_0000: same as REG[1]
TA2tmod	Texture blending mode
	= TA1tmod[12]
	1'b0: decal
	1'b1: modulate
TA2tmod_FILT[2:0]	Texture filtering mode
	= TA1tmod[11:9]
	3'b001: point sampling
	3'b010: bilinear filtering
	3'b100: trilinear filtering
TA2tmod_SIZE[8:0]	Texture size
	= TA1tmod[8:0]
	9'b1_0000_0000: 256 × 256

	9’b0_1000_0000: 128 × 128 9’b0_0100_0000: 64 × 64
TA2UV_spmask[7:0]	Spatial locality mask Generation rule @ TA2_MASK_GEN
TA2UV_tmmask[7:0]	Temporal locality mask Generation rule @ TA2_MASK_GEN
TA2_TMA#[19:0]	Texture memory address Generation rule @ TA2_ADDR_TMGEN

RTL Code

```
/*
* RAMP-GR
* RAMP-GR    TA2 Module: Texture Address #2
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wta2(   REclk, TP1nwait, TA1reset, TA1nhld, TA1cOP, TA1cFB, TA1cDF, TA1cTMF,
               TA1cAF, TA1cTEXEN, TA1cFMB, TA1cTMB, TA1cSCR, TA1_ZB0cmd,
               TA1_ZB1cmd, TA1_ZB0addr, TA1_ZB1addr, TA1ppASSIGN,
               TA1ppMASK, TA1tmod, TA1dADDR, TA1dA,
               TA1pp0VALID, TA1pp0RGB, TA1pp0U, TA1pp0V, TA1pp1VALID,
               TA1pp1RGB, TA1pp1U, TA1pp1V, TA2nwait, TA2reset, TA2nhld, TA2cOP,
               TA2cFB, TA2cDF, TA2cTMF, TA2cAF, TA2cTEXEN, TA2cFMB, TA2cTMB,
               TA2cSCR, TA2_ZB0cmd, TA2_ZB1cmd, TA2_ZB0addr, TA2_ZB1addr,
               TA2ppASSIGN, TA2ppMASK, TA2tmod_FILT, TA2tmod, TA2dADDR,
               TA2dA, TA2UV_spmask, TA2UV_tmmask, TA2UV0_align, TA2UV1_align,
               TA2UV2_align, TA2UV3_align, TA2UV0_comp, TA2UV1_comp,
               TA2UV2_comp, TA2UV3_comp, TA2UV4_comp, TA2UV5_comp,
               TA2UV6_comp, TA2UV7_comp, TA2pp0VALID, TA2pp0RGB,
               TA2pp0U_frac, TA2pp0V_frac, TA2pp0UV_end,
               TA2pp1VALID, TA2pp1RGB, TA2pp1U_frac, TA2pp1V_frac,
               TA2pp1UV_end, TA2_TMA0, TA2_TMA1, TA2_TMA2, TA2_TMA3,
               TA2_TMA4, TA2_TMA5, TA2_TMA6, TA2_TMA7);

     /* Wire renaming */
     assign #1  TA2pp0VALID = TA2ppMASK[1];
     assign #1  TA2pp1VALID      = TA2ppMASK[0];

     /* Module interconnection */


//Main Controller
Wta2_ctrl    ssta2_ctrl();
```

```
//Main controller
module    Wta2_ctrl(  REclk, TP1nwait, TA1reset, TA1nhld, TA1cOP,
          TA1cFB, TA1cDF, TA1cTMF, TA1cAF, TA1cTEXEN, TA1cFMB,
          TA1cTMB, TA1cSCR, TA1_ZB0cmd, TA1_ZB1cmd, TA1_ZB0addr,
          TA1_ZB1addr, TA1ppASSIGN,
          TA1ppMASK, TA2nwait, TA2reset, TA2nhld, TA2cOP, TA2cFB, TA2cDF,
          TA2cTMF, TA2cAF, TA2cTEXEN, TA2cFMB, TA2cTMB, TA2cSCR,
           TA2_ZB0cmd, TA2_ZB1cmd, TA2_ZB0addr, TA2_ZB1addr, TA2ppASSIGN,
          TA2ppMASK, TA2fetch_CLK, TA2fetch_TMOD, TA2fetch_ADDR,
          TA2fetch_ALPHA, TA2fetch_PXL, TA2fetch_TXL, TA2fetch_REG);

     /* Combinational logics */

     //nwait
     assign  TA2nwait = TP1nwait | ~TA1reset;
     //nhld
     assign #1  TA2nhld = TA2nhld_int;
     //Fetch enables
     assign #1  TA2fetch_CLK = TA2nwait & TA1nhld;
     assign  TA2fetch_TMOD = TA1cTMF;
     assign  TA2fetch_ADDR = TA1cAF;
     assign  TA2fetch_ALPHA = TA1cDF;
     assign  TA2fetch_PXL = TA1cDF;
     assign  TA2fetch_TXL = TA1cDF & TA1cTEXEN;
     //PP controls
     assign  TA2fetch_REG = TA2enable;
     always @(posedge REclk)
     begin
          if(TA2nwait)
          begin
               //State control
               TA2nhld_int <= TA1nhld;
               TA2reset <= TA1reset;
               //Fetch signals
               TA2cDF_int <= TA1cDF;
               TA2cTMF_int <= TA1cTMF;
               TA2cAF_int <= TA1cAF;
               TA2cTEXEN_int <= TA1cTEXEN;
               TA2cFMB_int <= TA1cFMB;
               TA2cTMB_int <= TA1cTMB;
               //Trivial signals
               TA2cOP <= TA1cOP;
               TA2cFB <= TA1cFB;
               TA2cSCR <= TA1cSCR;
               TA2_ZB0cmd <= TA1_ZB0cmd;
               TA2_ZB1cmd <= TA1_ZB1cmd;
               TA2_ZB0addr <= TA1_ZB0addr;
               TA2_ZB1addr <= TA1_ZB1addr;
               TA2ppASSIGN <= TA1ppASSIGN;
               //PP Controls
               TA2ppMASK <= TA1ppMASK;
          end
     end

     always @(negedge REclk)
     begin
```

```
            //Fetch signals
            TA2cDF <= TA2cDF_int;
            TA2cTMF <= TA2cTMF_int;
            TA2cAF <= TA2cAF_int;
            TA2cTEXEN <= TA2cTEXEN_int;
            TA2cFMB <= TA2cFMB_int;
            TA2cTMB <= TA2cTMB_int;
            //PP controls @ RDAT or TMOD
            TA2enable <= TA2nhld_int & (TA2cOP[5] | TA2cOP[2]) & TA2cTEXEN_int;
       end
 endmodule
```

```
//Pipeline latch
Wta2_latch       ssta2_latch();
```

```
//Data latches
module   Wta2_latch( REclk, clk, TA2fetch_TMOD, TA2fetch_ADDR, TA2fetch_ALPHA,
         TA2fetch_PXL, TA2fetch_TXL, TA1tmod, TA1dADDR, TA1dA,
         TA1pp0VALID, TA1pp0RGB, TA1pp0U, TA1pp0V,
         TA1pp1VALID, TA1pp1RGB, TA1pp1U, TA1pp1V,
         TA2tmod, TA2tmod_FILT, TA2tmod_SIZE, TA2dADDR, TA2dA, TA2lod,
         TA2pp0RGB, TA2pp0U, TA2pp0V, TA2pp1RGB, TA2pp1U, TA2pp1V);

     //TMOD
     always @(posedge REclk)
     begin
          if(clk & TA2fetch_TMOD)
          begin
               TA2tmod          <= TA1tmod [12];
               TA2tmod_FILT     <= TA1tmod [11:9];
               TA2tmod_SIZE     <= TA1tmod [8:0];
          end
     end
     //ADDR
     always @(posedge REclk)
     begin
          if(clk & TA2fetch_ADDR) TA2dADDR <= TA1dADDR;
     end
     //ALPHA
     always @(posedge REclk)
     begin
          if(clk & TA2fetch_ALPHA)    TA2dA <= TA1dA;
     end
     //LOD
     always @(posedge REclk)
     begin
          if(clk & TA2fetch_TXL) TA2lod <= 3'b0;
     end
     //PP0_PXL
     always @(posedge REclk)
     begin
          if(clk & TA1pp0VALID & TA2fetch_PXL) TA2pp0RGB <= TA1pp0RGB;
     end
     //PP0_TXL
     always @(posedge REclk)
     begin
          if(clk & TA1pp0VALID & TA2fetch_TXL)
          begin
               TA2pp0U <= TA1pp0U;
               TA2pp0V <= TA1pp0V;
          end
     end
     //PP1_PXL
     always @(posedge REclk)
     begin
          if(clk & TA1pp1VALID & TA2fetch_PXL) TA2pp1RGB <= TA1pp1RGB;
     end
     //PP1_TXL
     always @(posedge REclk)
     begin
          if(clk & TA1pp1VALID & TA2fetch_TXL)
          begin
               TA2pp1U <= TA1pp1U;
               TA2pp1V <= TA1pp1V;
          end
     end

endmodule
```

```
/* Address after LOD calculation */

//In this block, the LOD ("level of detail") addresses of texture.
//But, in this example, the texture unit does not support LOD operation,
//it just bypasses the texture address.
```

`Wta2_addr_lod     ssta2_addr_lod();`

```
//Address after LOD calculation unit
module     Wta2_addr_lod_unit(TA2uvIN, TA2uvOUT, TA2fracOUT);

     //Bus renaming
     assign #1  TA2uvOUT = TA2uvIN[11:3];
     assign #1  TA2fracOUT = {~TA2uvIN[3],TA2uvIN[3:1]};
endmodule

//Address after LOD calculation
module     Wta2_addr_lod(    TA2pp0U,TA2pp0V,TA2pp1U,TA2pp1V,
                TA2pp0U_LOD,TA2pp0V_LOD,TA2pp1U_LOD,TA2pp1V_LOD,
                TA2pp0U_frac,TA2pp0V_frac,TA2pp1U_frac,TA2pp1V_frac);

     /* Unit interconnection */
     Wta2_addr_lod_unit     lod0(  .TA2uvIN       (TA2pp0U),
                                   .TA2uvOUT      (TA2pp0U_LOD),
                                   .TA2fracOUT    (TA2pp0U_frac));
     Wta2_addr_lod_unit     lod1(  .TA2uvIN       (TA2pp0V),
                                   .TA2uvOUT      (TA2pp0V_LOD),
                                   .TA2fracOUT    (TA2pp0V_frac));
     Wta2_addr_lod_unit     lod2(  .TA2uvIN       (TA2pp1U),
                                   .TA2uvOUT      (TA2pp1U_LOD),
                                   .TA2fracOUT    (TA2pp1U_frac));
     Wta2_addr_lod_unit     lod3(  .TA2uvIN       (TA2pp1V),
                                   .TA2uvOUT      (TA2pp1V_LOD),
                                   .TA2fracOUT    (TA2pp1V_frac));
 endmodule
```

```
//Bilinear address generation
Wta2_addr_4x     ssta2_addr_4x();
```

```
//Bilinear address generation unit block
module     Wta2_addr_4x_unit(     TA2tmod_FILT, TA2u_LOD, TA2v_LOD,
                                  TA2uv0, TA2uv1, TA2uv2, TA2uv3, TA2uvEND);

     /* Adders & Subs */

     //u-0.5
     DW01_sub    #(9)    u_minus(    .A       (TA2u_LOD),
                                     .B       ({8'b0, 1'b1}),
                                     .CI      (1'b0),
                                     .DIFF    (TA2u_MINUS),
                                     .CO      (carry0));
     //u+0.5
     DW01_add    #(9)    u_plus (    .A       (TA2u_LOD),
                                     .B       ({8'b0, 1'b1}),
                                     .CI      (1'b0),
                                     .SUM     (TA2u_PLUS),
                                     .CO      (carry1));
     //v-0.5
     DW01_sub    #(9)    v_minus(    .A       (TA2v_LOD),
                                     .B       ({8'b0, 1'b1}),
                                     .CI      (1'b0),
                                     .DIFF    (TA2v_MINUS),
                                     .CO      (carry2));
     //v+0.5
     DW01_add    #(9)    v_plus (    .A       (TA2v_LOD),
                                     .B       ({8'b0, 1'b1}),
                                     .CI      (1'b0),
                                     .SUM     (TA2v_PLUS),
                                     .CO      (carry3));
     //For bilinear address
     assign #1  TA2uv0 = (TA2tmod_FILT[0]) ? {TA2u_LOD[8:1], TA2v_LOD[8:1]} :
                              {TA2u_MINUS[8:1] , TA2v_MINUS[8:1]};
     assign #1  TA2uv1 = {TA2u_MINUS[8:1], TA2v_PLUS[8:1]};
     assign #1  TA2uv2     = {TA2u_PLUS[8:1], TA2v_MINUS[8:1]};
     assign #1  TA2uv3     = {TA2u_PLUS[8:1], TA2v_PLUS[8:1]};
     //End-point detection
     assign #1  TA2uvEND_int[3] = &(TA2u_MINUS[8:1]);
     assign #1  TA2uvEND_int[2] = ~(|(TA2u_PLUS[8:1]));
     assign #1  TA2uvEND_int[1] = &(TA2v_MINUS[8:1]);
     assign #1  TA2uvEND_int[0]  =~(|(TA2v_PLUS[8:1]));
     //Zero priority
     always @(TA2uvEND_int)
     begin
          //U
          if(TA2uvEND_int[3:2] == 2'b11) TA2uvEND[3:2] <= 2'b01;
          else
               TA2uvEND[3:2] <= TA2uvEND_int[3:2];
          //V
          if(TA2uvEND_int[1:0] == 2'b11) TA2uvEND[1:0] <= 2'b01;
          else     TA2uvEND[1:0] <= TA2uvEND_int[1:0];
     end
endmodule

//Bilinear address generation
module    Wta2_addr_4x(TA2tmod_FILT,TA2pp0U_LOD,TA2pp0V_LOD,TA2pp1U_LOD,
               TA2pp1V_LOD, TA2pp0UV_0,TA2pp0UV_1,TA2pp0UV_2,TA2pp0UV_3,
               TA2pp1UV_0,TA2pp1UV_1,TA2pp1UV_2,TA2pp1UV_3,
               TA2pp0UV_end,TA2pp1UV_end);

     /* Unit interconnection */
     Wta2_addr_4x_unit   addr_4x0(   .TA2tmod_FILT    (TA2tmod_FILT),
                                     .TA2u_LOD   (TA2pp0U_LOD),
                                     .TA2v_LOD        (TA2pp0V_LOD),
                                     .TA2uv0          (TA2pp0UV_0),
                                     .TA2uv1          (TA2pp0UV_1),
                                     .TA2uv2          (TA2pp0UV_2),
                                     .TA2uv3          (TA2pp0UV_3),
                                     .TA2uvEND   (TA2pp0UV_end));
     Wta2_addr_4x_unit   addr_4x1(   .TA2tmod_FILT    (TA2tmod_FILT),
                                     .TA2u_LOD   (TA2pp1U_LOD),
                                     .TA2v_LOD        (TA2pp1V_LOD),
                                     .TA2uv0          (TA2pp1UV_0),
                                     .TA2uv1          (TA2pp1UV_1),
                                     .TA2uv2          (TA2pp1UV_2),
                                     .TA2uv3          (TA2pp1UV_3),
                                     .TA2uvEND   (TA2pp1UV_end));
endmodule
```

//Address alignment logic
Wta2_addr_align   ssta2_addr_align();

```
//Address alignment logic
module     Wta2_addr_align(    TA2pp0UV_0,TA2pp0UV_1,TA2pp0UV_2,TA2pp0UV_3,
                TA2pp1UV_0,TA2pp1UV_1,TA2pp1UV_2,TA2pp1UV_3,
                TA2UV0_align,TA2UV1_align,TA2UV2_align,TA2UV3_align);

     /* Parallel compare logic for PP1 */
     //#0
     assign #1  TA2UV0_align[3] = (TA2pp1UV_0 == TA2pp0UV_0) ? 1'b1 : 1'b0;
     assign #1  TA2UV0_align[2] = (TA2pp1UV_0 == TA2pp0UV_1) ? 1'b1 : 1'b0;
     assign #1  TA2UV0_align[1] = (TA2pp1UV_0 == TA2pp0UV_2) ? 1'b1 : 1'b0;
     assign #1  TA2UV0_align[0] = (TA2pp1UV_0 == TA2pp0UV_3) ? 1'b1 : 1'b0;
     //#1
     assign #1  TA2UV1_align[3] = (TA2pp1UV_1 == TA2pp0UV_0) ? 1'b1 : 1'b0;
     assign #1  TA2UV1_align[2] = (TA2pp1UV_1 == TA2pp0UV_1) ? 1'b1 : 1'b0;
     assign #1  TA2UV1_align[1] = (TA2pp1UV_1 == TA2pp0UV_2) ? 1'b1 : 1'b0;
     assign #1  TA2UV1_align[0] = (TA2pp1UV_1 == TA2pp0UV_3) ? 1'b1 : 1'b0;
     //#2
     assign #1  TA2UV2_align[3] = (TA2pp1UV_2 == TA2pp0UV_0) ? 1'b1 : 1'b0;
     assign #1  TA2UV2_align[2] = (TA2pp1UV_2 == TA2pp0UV_1) ? 1'b1 : 1'b0;
     assign #1  TA2UV2_align[1] = (TA2pp1UV_2 == TA2pp0UV_2) ? 1'b1 : 1'b0;
     assign #1  TA2UV2_align[0] = (TA2pp1UV_2 == TA2pp0UV_3) ? 1'b1 : 1'b0;
     //#3
     assign #1  TA2UV3_align[3] = (TA2pp1UV_3 == TA2pp0UV_0) ? 1'b1 : 1'b0;
     assign #1  TA2UV3_align[2] = (TA2pp1UV_3 == TA2pp0UV_1) ? 1'b1 : 1'b0;
     assign #1  TA2UV3_align[1] = (TA2pp1UV_3 == TA2pp0UV_2) ? 1'b1 : 1'b0;
     assign #1  TA2UV3_align[0] = (TA2pp1UV_3 == TA2pp0UV_3) ? 1'b1 : 1'b0;

endmodule
```

```
//Address register
Wta2_addr_register     ssta2_addr_register();
```

```
//Address register
module    Wta2_addr_register(  REclk,TA2fetch_REG,TA2fetch_CLK,TA2lod_now,
               TA2lod_reg,TA2UV_spmask, TA2pp0UV_0,TA2pp0UV_1,TA2pp0UV_2,
               TA2pp0UV_3, TA2pp1UV_0,TA2pp1UV_1,
               TA2pp1UV_2,TA2pp1UV_3,
               TA2UV0_reg,TA2UV1_reg,TA2UV2_reg,TA2UV3_reg,
               TA2UV4_reg,TA2UV5_reg,TA2UV6_reg,TA2UV7_reg);

     always @(posedge REclk)
     begin
          if(TA2fetch_REG)
          begin
               TA2lod_reg = TA2lod_now;
               //Initial status
               if(TA2lod_now[3])
               begin
                    TA2UV0_reg <= 16'b1111_1111_1111_1111;
                    TA2UV1_reg <= 16'b1111_1111_1111_1111;
                    TA2UV2_reg <= 16'b1111_1111_1111_1111;
                    TA2UV3_reg <= 16'b1111_1111_1111_1111;
                    TA2UV4_reg <= 16'b1111_1111_1111_1111;
                    TA2UV5_reg <= 16'b1111_1111_1111_1111;
                    TA2UV6_reg <= 16'b1111_1111_1111_1111;
                    TA2UV7_reg <= 16'b1111_1111_1111_1111;
               end
               else
               begin
                    if(TA2UV_spmask[7] & TA2fetch_CLK)
                         TA2UV0_reg <= TA2pp0UV_0;
                    if(TA2UV_spmask[6] & TA2fetch_CLK)
```

```
                        TA2UV1_reg <= TA2pp0UV_1;
                    if(TA2UV_spmask[5] & TA2fetch_CLK)
                        TA2UV2_reg <= TA2pp0UV_2;
                    if(TA2UV_spmask[4] & TA2fetch_CLK)
                        TA2UV3_reg <= TA2pp0UV_3;
                    if(TA2UV_spmask[3] & TA2fetch_CLK)
                        TA2UV4_reg <= TA2pp1UV_0;
                    if(TA2UV_spmask[2] & TA2fetch_CLK)
                        TA2UV5_reg <= TA2pp1UV_1;
                    if(TA2UV_spmask[1] & TA2fetch_CLK)
                        TA2UV6_reg <= TA2pp1UV_2;
                    if(TA2UV_spmask[0] & TA2fetch_CLK)
                        TA2UV7_reg <= TA2pp1UV_3;
                end
            end
        end
endmodule
```

//Address compare logic
Wta2_addr_compare ssta2_addr_compare();

```
//Address compare logic endmodule
module    Wta2_addr_compare(  TA2lod_now,TA2lod_reg, TA2pp0UV_0,TA2pp0UV_1,
                TA2pp0UV_2,TA2pp0UV_3, TA2pp1UV_0,TA2pp1UV_1,
                TA2pp1UV_2,TA2pp1UV_3, TA2UV0_reg,TA2UV1_reg,
                TA2UV2_reg,TA2UV3_reg,
                TA2UV4_reg,TA2UV5_reg,TA2UV6_reg,TA2UV7_reg,
                TA2UV0_comp,TA2UV1_comp,TA2UV2_comp,TA2UV3_comp,
                TA2UV4_comp,TA2UV5_comp,TA2UV6_comp,TA2UV7_comp);

    /* Bunch of comparators */

    //Overridden by LOD results
    //#0
    assign #1  comp0[7] = (TA2pp0UV_0 == TA2UV0_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[6] = (TA2pp0UV_0 == TA2UV1_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[5] = (TA2pp0UV_0 == TA2UV2_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[4] = (TA2pp0UV_0 == TA2UV3_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[3] = (TA2pp0UV_0 == TA2UV4_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[2] = (TA2pp0UV_0 == TA2UV5_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[1] = (TA2pp0UV_0 == TA2UV6_reg) ? 1'b1 : 1'b0;
    assign #1  comp0[0] = (TA2pp0UV_0 == TA2UV7_reg) ? 1'b1 : 1'b0;
    //#1
    assign #1  comp1[7] = (TA2pp0UV_1 == TA2UV0_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[6] = (TA2pp0UV_1 == TA2UV1_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[5] = (TA2pp0UV_1 == TA2UV2_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[4] = (TA2pp0UV_1 == TA2UV3_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[3] = (TA2pp0UV_1 == TA2UV4_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[2] = (TA2pp0UV_1 == TA2UV5_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[1] = (TA2pp0UV_1 == TA2UV6_reg) ? 1'b1 : 1'b0;
    assign #1  comp1[0] = (TA2pp0UV_1 == TA2UV7_reg) ? 1'b1 : 1'b0;
```

```
        //#2
        assign #1  comp2[7] = (TA2pp0UV_2 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[6] = (TA2pp0UV_2 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[5] = (TA2pp0UV_2 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[4] = (TA2pp0UV_2 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[3] = (TA2pp0UV_2 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[2] = (TA2pp0UV_2 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[1] = (TA2pp0UV_2 == TA2UV6_reg) ? 1'b1 : 1'b0;
        assign #1  comp2[0] = (TA2pp0UV_2 == TA2UV7_reg) ? 1'b1 : 1'b0;
        //#3
        assign #1  comp3[7] = (TA2pp0UV_3 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[6] = (TA2pp0UV_3 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[5] = (TA2pp0UV_3 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[4] = (TA2pp0UV_3 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[3] = (TA2pp0UV_3 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[2] = (TA2pp0UV_3 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[1] = (TA2pp0UV_3 == TA2UV6_reg) ? 1'b1 : 1'b0;
        assign #1  comp3[0] = (TA2pp0UV_3 == TA2UV7_reg) ? 1'b1 : 1'b0;
        //#4
        assign #1  comp4[7] = (TA2pp1UV_0 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[6] = (TA2pp1UV_0 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[5] = (TA2pp1UV_0 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[4] = (TA2pp1UV_0 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[3] = (TA2pp1UV_0 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[2] = (TA2pp1UV_0 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[1] = (TA2pp1UV_0 == TA2UV6_reg) ? 1'b1 : 1'b0;
        assign #1  comp4[0] = (TA2pp1UV_0 == TA2UV7_reg) ? 1'b1 : 1'b0;
        //#5
        assign #1  comp5[7] = (TA2pp1UV_1 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[6] = (TA2pp1UV_1 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[5] = (TA2pp1UV_1 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[4] = (TA2pp1UV_1 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[3] = (TA2pp1UV_1 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[2] = (TA2pp1UV_1 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[1] = (TA2pp1UV_1 == TA2UV6_reg) ? 1'b1 : 1'b0;
        assign #1  comp5[0] = (TA2pp1UV_1 == TA2UV7_reg) ? 1'b1 : 1'b0;
        //#6
        assign #1  comp6[7] = (TA2pp1UV_2 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[6] = (TA2pp1UV_2 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[5] = (TA2pp1UV_2 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[4] = (TA2pp1UV_2 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[3] = (TA2pp1UV_2 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[2] = (TA2pp1UV_2 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[1] = (TA2pp1UV_2 == TA2UV6_reg) ? 1'b1 : 1'b0;
        assign #1  comp6[0] = (TA2pp1UV_2 == TA2UV7_reg) ? 1'b1 : 1'b0;
        //#7
        assign #1  comp7[7] = (TA2pp1UV_3 == TA2UV0_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[6] = (TA2pp1UV_3 == TA2UV1_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[5] = (TA2pp1UV_3 == TA2UV2_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[4] = (TA2pp1UV_3 == TA2UV3_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[3] = (TA2pp1UV_3 == TA2UV4_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[2] = (TA2pp1UV_3 == TA2UV5_reg) ? 1'b1 : 1'b0;
        assign #1  comp7[1] = (TA2pp1UV_3 == TA2UV6_reg) ? 1'b1 : 1'b0;
```

```
     assign #1  comp7[0] = (TA2pp1UV_3 == TA2UV7_reg) ? 1'b1 : 1'b0;
     //LOD override
     assign #1  TA2UV0_comp = (TA2lod_now==TA2lod_reg) ? comp0 : 8'b0;
     assign #1  TA2UV1_comp = (TA2lod_now==TA2lod_reg) ? comp1 : 8'b0;
     assign #1  TA2UV2_comp = (TA2lod_now==TA2lod_reg) ? comp2 : 8'b0;
     assign #1  TA2UV3_comp = (TA2lod_now==TA2lod_reg) ? comp3 : 8'b0;
     assign #1  TA2UV4_comp = (TA2lod_now==TA2lod_reg) ? comp4 : 8'b0;
     assign #1  TA2UV5_comp = (TA2lod_now==TA2lod_reg) ? comp5 : 8'b0;
     assign #1  TA2UV6_comp = (TA2lod_now==TA2lod_reg) ? comp6 : 8'b0;
     assign #1  TA2UV7_comp = (TA2lod_now==TA2lod_reg) ? comp7 : 8'b0;
endmodule
```

//Texture memory address generation
Wta2_addr_tmgen   ssta2_addr_tmgen();

```
//Texture memory address generation unit block
module     Wta2_addr_tmgen_unit(size, uvin, tmout);

     /* Inputs */

     input [8:0]      size;
     input [15:0]     uvin;
     output [15:0]    tmout;

     //Intermediate wires
     wire [7:0]       u;
     wire [7:0]       v;
     wire             tmcarry;
     wire [15:0]      tmout;
     reg [15:0]       uvshift;
     //reg [17:0]     tmout;

     assign #1  u = uvin[15:8];
     assign #1  v = uvin[7:0];

     //UVshift
     always @(size or u or v)
     begin
          casex(size)
               9'b1_XXXX_XXXX    : uvshift <= {v[7:0], u[7:0]};
               9'bX_1XXX_XXXX    : uvshift <= {2'b0, v[6:0], u[6:0]};
               9'bX_X1XX_XXXX    : uvshift <= {4'b0, v[5:0], u[5:0]};
               9'bX_XX1X_XXXX    : uvshift <= {6'b0, v[4:0], u[4:0]};
               9'bX_XXX1_XXXX    : uvshift <= {8'b0, v[3:0], u[3:0]};
               9'bX_XXXX_1XXX    : uvshift <= {10'b0, v[2:0], u[2:0]};
               9'bX_XXXX_X1XX    : uvshift <= {12'b0, v[1:0], u[1:0]};
               9'bX_XXXX_XX1X    : uvshift <= {14'b0, v[0], u[0]};
               9'bX_XXXX_XXX1    : uvshift <= 16'b0;
               default           : uvshift <= 16'bX;
          endcase
     end

     assign  tmout = uvshift;
endmodule
```

```
//Texture memory Address Generation
module     Wta2_addr_tmgen(TA2tmod_SIZE, TA2pp0UV_0, TA2pp0UV_1, TA2pp0UV_2,
                TA2pp0UV_3, TA2pp1UV_0, TA2pp1UV_1, TA2pp1UV_2, TA2pp1UV_3,
                TA2_TMA0, TA2_TMA1, TA2_TMA2, TA2_TMA3,
                TA2_TMA4, TA2_TMA5, TA2_TMA6, TA2_TMA7);
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit0(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp0UV_0),
                    .tmout              (TA2_TMA0));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit1(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp0UV_1),
                    .tmout              (TA2_TMA1));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit2(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp0UV_2),
                    .tmout              (TA2_TMA2));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit3(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp0UV_3),
                    .tmout              (TA2_TMA3));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit4(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp1UV_0),
                    .tmout              (TA2_TMA4));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit5(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp1UV_1),
                    .tmout              (TA2_TMA5));
     Wta2_addr_tmgen_unit  ssta2_addr_tmgen_unit6(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp1UV_2),
                    .tmout              (TA2_TMA6));
     Wta2_addr_tmgen_unit ssta2_addr_tmgen_unit7(
                    .size         (TA2tmod_SIZE),
                    .uvin         (TA2pp1UV_3),
                    .tmout              (TA2_TMA7));
endmodule
```

```
//Alignment mask generation module
Wta2_mask_gen     ssta2_mask_gen();
```

```
//Alignment mask generation module
module     Wta2_mask_gen(   TA2cTMF, TA2ppMASK, TA2tmod_FILT, TA2lod,
                TA2UV0_align, TA2UV1_align, TA2UV2_align, TA2UV3_align,
                TA2UV0_comp, TA2UV1_comp, TA2UV2_comp, TA2UV3_comp,
                TA2UV4_comp, TA2UV5_comp, TA2UV6_comp, TA2UV7_comp,
                TA2lod_now, TA2UV_spmask, TA2UV_tmmask);

     //Filtering mask seed generator
     always @(TA2tmod_FILT)
     begin
          casex(TA2tmod_FILT)
```

```
                3'b001:     SEEDfilt <= 8'b1000_1000;     //Point Sampling
                3'b010:     SEEDfilt <= 8'b1111_1111;     //Bilinear Filtering
                default:    SEEDfilt <= 8'bX;
            endcase
        end
        //PP mask
        assign #1 MASKpp = SEEDfilt & { {4{TA2ppMASK[1]}}, {4{TA2ppMASK[0]}} };
        //Alignment mask
        assign #1  MASKalign[3] = ~(|TA2UV0_align);
        assign #1  MASKalign[2] = ~(|TA2UV1_align);
        assign #1  MASKalign[1] = ~(|TA2UV2_align);
        assign #1  MASKalign[0] = ~(|TA2UV3_align);
        //spmask: spatial locality mask - special care for Pt sample
        always @(TA2tmod_FILT or MASKalign or MASKpp or TA2ppMASK)
        begin
            if(TA2tmod_FILT[1] & (&TA2ppMASK))
                TA2UV_spmask <= MASKpp & {4'b1111, MASKalign};
            else     TA2UV_spmask <= MASKpp;
        end
        //Compare mask
        assign #1  MASKcomp[7] = ~(|TA2UV0_comp);
        assign #1  MASKcomp[6] = ~(|TA2UV1_comp);
        assign #1  MASKcomp[5] = ~(|TA2UV2_comp);
        assign #1  MASKcomp[4] = ~(|TA2UV3_comp);
        assign #1  MASKcomp[3] = ~(|TA2UV4_comp);
        assign #1  MASKcomp[2] = ~(|TA2UV5_comp);
        assign #1  MASKcomp[1] = ~(|TA2UV6_comp);
        assign #1  MASKcomp[0] = ~(|TA2UV7_comp);
        //tmmask: temporal locality mask
        assign #1  TA2UV_tmmask = TA2UV_spmask & MASKcomp;
        assign #1  TA2lod_now = {TA2cTMF, TA2lod};
endmodule
```

endmodule

7.5.4.10 TP1: Texture Prefetch #1

In the TP1 stage (Figure 7.25), texture cache requests are generated. Since the rasterizer has four independent texture caches, this stage decides which cache is used. The other important task of this stage is to hold whole pipeline texture cache status. When the texture cache misses, it requires a few tens of cycles and this stage holds all pipeline stages by generating nHld and nWait signals.

Pipe Inputs

- REclk: rendering clock
- Pipe controls: same as TA2
- Common signals: same as TA2
- CAL controls: same as TA2

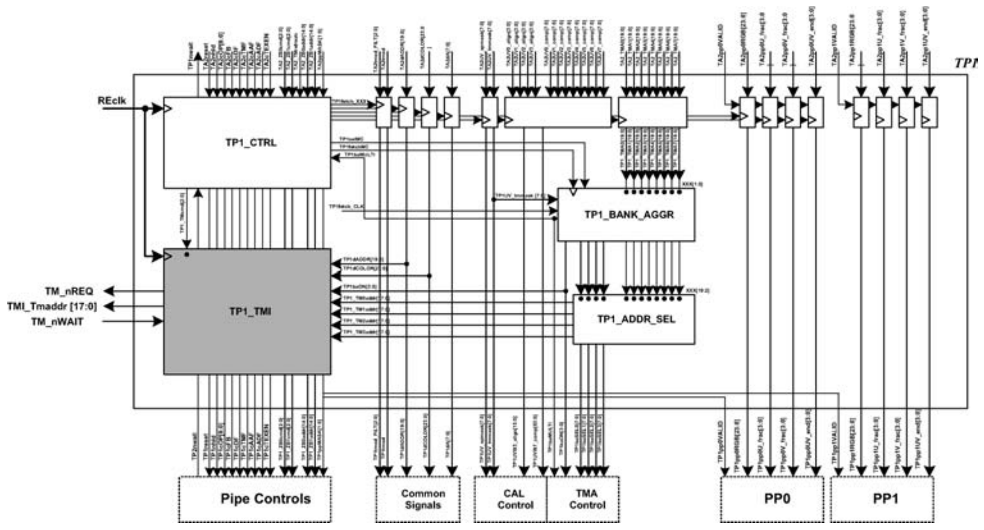


Figure 7.25 Texture prefetch #1

- PP dependent signals: same as TA2
- Texture memory address: same as TA2

Pipe Outputs

- Pipe controls
- Common signals
- CAL control: cache alignment logic
- TMA control: texture memory aggregation control
- PP dependent signals

Memory Input/Outputs

- Texture memory request (out)
- Texture memory address (out)
- Texture memory nWAIT (in)

Functions

- Texture address generation
- Multi-cycle control
- Texture address control

Type	TP1cOP[8:0]	Cycle	Functional description @ this stage
Rendering	CTRL_OP_RDAT	Multi-cycle (Depends on TM bank flag)	Texture bank aggregation Multi-cycle control
			Independent texture memory control (×4)
Texture	CTRL_OP_TSTR	1	Store texture map Write one of TM @ TA1dADDR
	CTRL_OP_TMOD	1	Bypass
	CTRL_OP_TF2T	1	Bypass [This instruction is deleted to reduce design complexity. It may be implemented later on the next revision.]
Auxiliary	CTRL_OP_ASTR	1	Bypass

Signal Descriptions

Control signal	Description
TP1selMC	Multi-cycle selection signal Generated by using TEXEN,OP_CODE,nHLD,baMULTI 1’b0: initial/no texture 1’b1: multi-cycle
TP1fetchMC	Multi-cycle fetch signal Rule @ Verilog
TP1baMULTI	Multi-cycle flag from bank aggregation 1’b0: single cycle 1’b1: multi-cycle
TP1baON	Texture bank active flag
TP1baSEL#[7:0]	Texture bank selection signal
TP1_TM#addr[17:0]	Texture memory address
TP1_TMcmd[2:0]	Texture memory command from CTRL to TMI 3’b000: NOP 3’b001: REFRESH 3’b010: READ 3’b100: WRITE
TMI_TM#cmd[2:0]	Texture memory command
TMI_TM#addr[17:0]	Texture memory address

Datapath #1: TP1_BANK_AGGR

See Figure 7.26.

Multi-cycle Timing

See Figure 7.27.

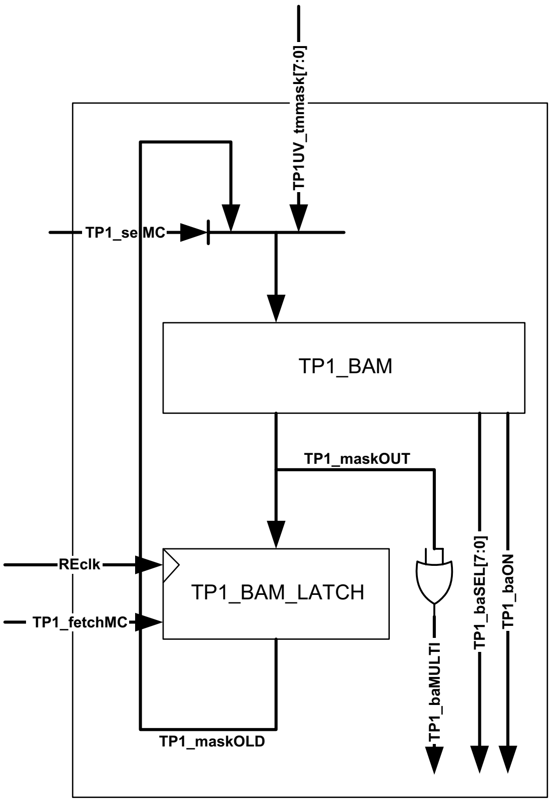


Figure 7.26 Datapath #1: TP1_BANK_AGGR

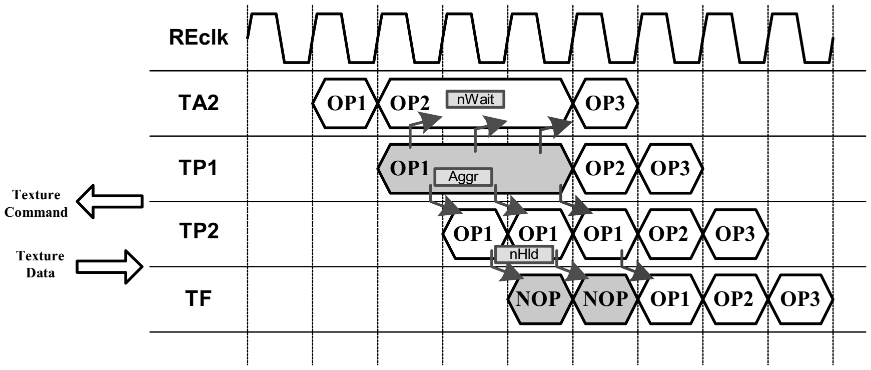


Figure 7.27 Multi-cycle timing

RTL Code

```
/*
* RAMP-GR
* RAMP-GR TP1 Module: Texture Prefetch #1
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wtp1(  REclk, TP1nwait, TA2reset, TA2nhld, TA2cOP, TA2cFB, TA2cDF, TA2cTMF,
            TA2cAF, TA2cTEXEN, TA2cFMB, TA2cTMB, TA2cSCR,
            TA2_ZB0cmd, TA2_ZB1cmd, TA2_ZB0addr, TA2_ZB1addr,
            TA2ppMASK, TA2tmod_FILT, TA2tmod, TA2dADDR, TA2dA,
            TA2UV_spmask, TA2UV_tmmask, TA2UV0_align, TA2UV1_align,
            TA2UV2_align, TA2UV3_align,
            TA2UV0_comp, TA2UV1_comp, TA2UV2_comp, TA2UV3_comp,
            TA2UV4_comp, TA2UV5_comp, TA2UV6_comp, TA2UV7_comp,
            TA2_TMA0, TA2_TMA1, TA2_TMA2, TA2_TMA3,
            TA2_TMA4, TA2_TMA5, TA2_TMA6, TA2_TMA7,
            TA2pp0VALID, TA2pp0RGB, TA2pp0U_frac, TA2pp0V_frac, TA2pp0UV_end,
            TA2pp1VALID, TA2pp1RGB, TA2pp1U_frac, TA2pp1V_frac, TA2pp1UV_end,
            TP2nwait, TP1reset, TP1nhld, TP1cOP, TP1cFB, TP1cDF, TP1cTMF,
            TP1cAF, TP1cTEXEN, TP1cFMB, TP1cSCR,
            TP1_ZB0cmd, TP1_ZB1cmd, TP1_ZB0addr, TP1_ZB1addr,
            TP1ppMASK, TP1tmod_FILT, TP1tmod, TP1dADDR, TP1dA,
            TP1UV_spmask, TP1UV_tmmask, TP1UV0t3_align, TP1UV0t7_comp,
            TP1baMULTI, TP1baON, TP1baSEL0, TP1baSEL1, TP1baSEL2, TP1baSEL3,
            TP1pp0VALID, TP1pp0RGB, TP1pp0U_frac, TP1pp0V_frac, TP1pp0UV_end,
            TP1pp1VALID, TP1pp1RGB, TP1pp1U_frac, TP1pp1V_frac, TP1pp1UV_end,
            TMI_TMBADDR, TMI_TM0nREQ, TMI_TM0cmd, TMI_TM0addr,
            TMI_TM1nREQ, TMI_TM1cmd, TMI_TM1addr,
            TMI_TM2nREQ, TMI_TM2cmd, TMI_TM2addr,
            TMI_TM3nREQ, TMI_TM3cmd, TMI_TM3addr);

    /* Wire renaming */
    assign #1  TP1pp0VALID = TP1ppMASK[1];
    assign #1  TP1pp1VALID = TP1ppMASK[0];

    /* Module interconnection */


//Main controller
Wtp1_ctrl     sstp1_ctrl();
```

```
//Main controller
module    Wtp1_ctrl(    REclk, TP1nwait, TA2reset, TA2nhld,
          TA2cOP, TA2cFB, TA2cDF, TA2cTMF,
          TA2cAF, TA2cTEXEN, TA2cFMB, TA2cTMB, TA2cSCR,
          TA2_ZB0cmd, TA2_ZB1cmd, TA2_ZB0addr, TA2_ZB1addr, TA2ppMASK,
          TP2nwait, TP1reset, TP1nhld, TP1cOP, TP1cFB, TP1cDF, TP1cTMF,
          TP1cAF, TP1cTEXEN, TP1cFMB, TP1cSCR,
          TP1_ZB0cmd, TP1_ZB1cmd, TP1_ZB0addr, TP1_ZB1addr, TP1ppMASK,
          TP1_TMcmd, TP1fetch_CLK, TP1fetch_TMOD, TP1fetch_ADDR,
          TP1fetch_TADDR, TP1fetch_ALPHA,
          TP1fetch_PXL, TP1fetch_TXL, TP1fetchMC, TP1selMC, TP1baMULTI);

     //nwait
     assign  TP1nwait_gen = ~(TP1nhld_int & TP1cDF_int & TP1cOP[5]
                      & TP1cTEXEN_int & TP1baMULTI);
     assign  TP1nwait = (TP1nwait_gen & TP2nwait) | ~TA2reset;
     //nhld
     assign #1  TP1nhld = TP1nhld_int;
     //Fetch enable
     assign #1  TP1fetch_CLK = TP1nwait & TA2nhld;
     assign  TP1fetch_TMOD = TA2cTMF;
     assign  TP1fetch_ADDR = TA2cAF;
     assign  TP1fetch_TADDR = TA2cAF & TA2cTMB;
     assign  TP1fetch_ALPHA= TA2cDF;
     assign  TP1fetch_PXL = TA2cDF;
     assign  TP1fetch_TXL = TA2cDF & TA2cTEXEN;
     assign  TP1fetchMC = TP1enable & TP1cTEXEN;
     //MC select signal
     assign #1  TP1selMC = TP1nhld_int & TP1cDF_int & TP1cOP[5]
                      & TP1cTEXEN_int & TP1selMC_int;
     //TM command
     assign #1  TP1_TMcmd = TP1_TMcmd_int;

     always @(posedge REclk)
     begin
          if(TP1nwait)
          begin
               //State controls
               TP1nhld_int <= TA2nhld;
               TP1reset <= TA2reset;
               //Fetch signals
               TP1cDF_int <= TA2cDF;
               TP1cTMF_int <= TA2cTMF;
               TP1cAF_int <= TA2cAF;
               TP1cTEXEN_int <= TA2cTEXEN;
               TP1cFMB_int <= TA2cFMB;
               // Trivial Signals
               TP1cOP <= TA2cOP;
               TP1cFB <= TA2cFB;
               TP1cSCR <= TA2cSCR;
               TP1_ZB0cmd <= TA2_ZB0cmd;
               TP1_ZB1cmd <= TA2_ZB1cmd;
               TP1_ZB0addr <= TA2_ZB0addr;
               TP1_ZB1addr <= TA2_ZB1addr;
               // PP Control Signals
               TP1ppMASK <= TA2ppMASK;
          end
          TP1selMC_int <= ~TP1nwait_gen_lat;
     end

     always @(negedge REclk)
     begin
          //Fetch signals
          TP1cDF <= TP1cDF_int;
          TP1cTMF <= TP1cTMF_int;
          TP1cAF <= TP1cAF_int;
          TP1cTEXEN <= TP1cTEXEN_int;
          TP1cFMB <= TP1cFMB_int;
          //BA fetch control
          TP1enable <= TP1nhld_int & TP1cOP[5];
          //MC control
          TP1nwait_gen_lat <= TP1nwait_gen;
     end
     //TM command generation
     always @(TP1cOP or TP1nhld_int or TP2nwait or TP1cTEXEN_int)
     begin
          casex({TP1nhld_int, TP2nwait, TP1cTEXEN_int, TP1cOP})
          {3'b111,'WCTRL_OP_RSHA}  :    TP1_TMcmd_int<= 'WTM_CMD_READ;
          default                       TP1_TMcmd_int <= 'WTM_CMD_NOP;
          endcase
     end
endmodule
```

//Data latches
Wtp1_latch   sstp1_latch();

```
//Data latches
module     Wtp1_latch(REclk, clk, TP1fetch_TMOD, TP1fetch_ADDR,
                TP1fetch_TADDR,
                TP1fetch_ALPHA, TP1fetch_PXL, TP1fetch_TXL,
                TA2tmod_FILT, TA2tmod, TA2dADDR, TA2dA,
                TA2UV_spmask, TA2UV_tmmask, TA2UV0_align, TA2UV1_align,
                TA2UV2_align, TA2UV3_align,
                TA2UV0_comp, TA2UV1_comp, TA2UV2_comp, TA2UV3_comp,
                TA2UV4_comp, TA2UV5_comp, TA2UV6_comp, TA2UV7_comp,
                TA2_TMA0, TA2_TMA1, TA2_TMA2, TA2_TMA3,
                TA2_TMA4, TA2_TMA5, TA2_TMA6, TA2_TMA7,
                TA2pp0VALID, TA2pp0RGB, TA2pp0U_frac,
                TA2pp0V_frac, TA2pp0UV_end,
                TA2pp1VALID, TA2pp1RGB, TA2pp1U_frac, TA2pp1V_frac,
                TA2pp1UV_end, TP1tmod_FILT, TP1tmod, TP1dADDR, TP1dA,
                TP1UV_spmask, TP1UV_tmmask, TP1UV0t3_align,
                TP1UV0t7_comp,
                TP1_TMBADDR, TP1_TMA0, TP1_TMA1, TP1_TMA2,
                TP1_TMA3,
                TP1_TMA4, TP1_TMA5, TP1_TMA6, TP1_TMA7,
                TP1pp0RGB, TP1pp0U_frac, TP1pp0V_frac, TP1pp0UV_end,
                TP1pp1RGB, TP1pp1U_frac, TP1pp1V_frac, TP1pp1UV_end);

     //TMOD
     always @(posedge REclk)
     begin
          if(clk & TP1fetch_TMOD)
          begin
                TP1tmod_FILT <= TA2tmod_FILT;
                TP1tmod <= TA2tmod;
          end
```

```
      end
      //ADDR
      always @(posedge REclk)
      begin
           if(clk & TP1fetch_ADDR)      TP1dADDR <= TA2dADDR;
      end
      //TADDR
      always @(posedge REclk)
      begin
           if(clk & TP1fetch_TADDR) TP1_TMBADDR <= TA2dADDR;
      end
      //ALPHA
      always @(posedge REclk)
      begin
           if(clk & TP1fetch_ALPHA) TP1dA <= TA2dA;
      end
      //TXL
      always @(posedge REclk)
      begin
           if(clk & TP1fetch_TXL)
           begin
           TP1UV_spmask <= TA2UV_spmask;
           TP1UV_tmmask <= TA2UV_tmmask;
           TP1UV0t3_align <= { TA2UV0_align, TA2UV1_align, TA2UV2_align,
                  TA2UV3_align};
           TP1UV0t7_comp <= {TA2UV0_comp, TA2UV1_comp, TA2UV2_comp, TA2UV3_comp,
                  TA2UV4_comp, TA2UV5_comp, TA2UV6_comp, TA2UV7_comp};
           TP1_TMA0 <= TA2_TMA0;
           TP1_TMA1 <= TA2_TMA1;
           TP1_TMA2 <= TA2_TMA2;
           TP1_TMA3 <= TA2_TMA3;
           TP1_TMA4 <= TA2_TMA4;
           TP1_TMA5 <= TA2_TMA5;
           TP1_TMA6 <= TA2_TMA6;
           TP1_TMA7 <= TA2_TMA7;
           end
      end
      //PP0 PXL
      always @(posedge REclk)
      begin
           if(clk & TA2pp0VALID & TP1fetch_PXL) TP1pp0RGB <= TA2pp0RGB;
      end
      //PP0 TXL
      always @(posedge REclk)
      begin
           if(clk & TA2pp0VALID & TP1fetch_TXL)
           begin
                TP1pp0U_frac <= TA2pp0U_frac;
                TP1pp0V_frac <= TA2pp0V_frac;
                TP1pp0UV_end <= TA2pp0UV_end;
           end
      end
      //PP1 PXL
      always @(posedge REclk)
```

```
     begin
          if(clk & TA2pp1VALID & TP1fetch_PXL) TP1pp1RGB <= TA2pp1RGB;
     end
     //PP1 TXL
     always @(posedge REclk)
     begin
          if(clk & TA2pp1VALID & TP1fetch_TXL)
          begin
               TP1pp1U_frac <= TA2pp1U_frac;
               TP1pp1V_frac <= TA2pp1V_frac;
               TP1pp1UV_end <= TA2pp1UV_end;
          end
     end
endmodule
```

//Bank aggregation
Wtp1_bank_aggr   sstp1_bank_aggr();

```
//Bank aggregation module
module     Wtp1_bam_unit(   moduleID,maskIN,maskOUT,baON,baSEL,
                 tma0,tma1,tma2,tma3,tma4,tma5,tma6,tma7);

     /* Combinational logics */

     //ID==?TMA
     assign  bankSAME[7] = (moduleID == tma0) ? 1'b1 : 1'b0;     // TMA0
     assign  bankSAME[6] = (moduleID == tma1) ? 1'b1 : 1'b0;     // TMA1
     assign  bankSAME[5] = (moduleID == tma2) ? 1'b1 : 1'b0;     // TMA2
     assign  bankSAME[4] = (moduleID == tma3) ? 1'b1 : 1'b0;     // TMA3
     assign  bankSAME[3] = (moduleID == tma4) ? 1'b1 : 1'b0;     // TMA4
     assign  bankSAME[2] = (moduleID == tma5) ? 1'b1 : 1'b0;     // TMA5
     assign  bankSAME[1] = (moduleID == tma6) ? 1'b1 : 1'b0;     // TMA6
     assign  bankSAME[0] = (moduleID == tma7) ? 1'b1 : 1'b0;     // TMA7
     //Bitwise-AND with tmmask
     assign  bankANDmask = maskIN & bankSAME;
     // Bank address ON
     assign  baON = (|baSEL);
     //Mask out
     assign  maskOUT = maskIN & (~baSEL);
     //Bank address select: priority selection
     always @(bankANDmask)
     begin
          casex(bankANDmask)
               8'b1XXX_XXXX : baSEL <= 8'b1000_0000;
               8'b01XX_XXXX : baSEL <= 8'b0100_0000;
               8'b001X_XXXX : baSEL <= 8'b0010_0000;
               8'b0001_XXXX : baSEL <= 8'b0001_0000;
               8'b0000_1XXX : baSEL <= 8'b0000_1000;
               8'b0000_01XX : baSEL <= 8'b0000_0100;
               8'b0000_001X : baSEL <= 8'b0000_0010;
               8'b0000_0001 : baSEL <= 8'b0000_0001;
               8'b0000_0000 : baSEL <= 8'b0000_0000;
```

```
                default      : baSEL <= 8'b0000_0000;
            endcase
        end
endmodule

/* Bank aggregation */
module     Wtp1_bank_aggr(    REclk,TP1selMC,TP1fetchMC,TP1fetch_CLK,
                  TP1UV_tmmask, TP1_TMA0,TP1_TMA1,TP1_TMA2,TP1_TMA3,
                  TP1_TMA4,TP1_TMA5,TP1_TMA6,TP1_TMA7,
                  TP1baMULTI,TP1baON,
                  TP1baSEL0,TP1baSEL1,TP1baSEL2,TP1baSEL3);

      /* Combinational Logics */
      assign #1  TP1maskIN = (TP1selMC) ? TP1maskOLD : TP1UV_tmmask;
      assign #1  TP1baMULTI = (|TP1maskOUT);

      /* Latches */
      always @(posedge REclk)
      begin
            //if(TP1fetch_CLK)  TP1maskOLD = TP1maskOUT;
            if(TP1fetchMC) TP1maskOLD <= TP1maskOUT;
      end

      /* Module interconnection */
      Wtp1_bam_unit     sstp1_bam_0();
      Wtp1_bam_unit     sstp1_bam_1();
      Wtp1_bam_unit     sstp1_bam_2();
      Wtp1_bam_unit     sstp1_bam_3();
endmodule
```

//Address select MUX
Wtp1_addr_sel   sstp1_addr_sel();

```
//Address select MUX
module     Wtp1_addr_sel(    TP1baSEL0,TP1baSEL1,TP1baSEL2,TP1baSEL3,
                  TP1_TMA0,TP1_TMA1,TP1_TMA2,TP1_TMA3,
                  TP1_TMA4,TP1_TMA5,TP1_TMA6,TP1_TMA7,
                  TP1_TM0addr,TP1_TM1addr,TP1_TM2addr,TP1_TM3addr);

      //TM
      always @(TP1baSEL0 or TP1baSEL1 or TP1baSEL2 or TP1baSEL3 or
            TP1_TMA0 or TP1_TMA1 or TP1_TMA2 or TP1_TMA3 or
            TP1_TMA4 or TP1_TMA5 or TP1_TMA6 or TP1_TMA7)
      begin
            //TM#0
            casex(TP1baSEL0)
                  8'b1XXX_XXXX : TP1_TM0addr <= TP1_TMA0;
                  8'bX1XX_XXXX : TP1_TM0addr <= TP1_TMA1;
                  8'bXX1X_XXXX : TP1_TM0addr <= TP1_TMA2;
                  8'bXXX1_XXXX : TP1_TM0addr <= TP1_TMA3;
                  8'bXXXX_1XXX : TP1_TM0addr <= TP1_TMA4;
                  8'bXXXX_X1XX : TP1_TM0addr <= TP1_TMA5;
```

```
                8'bXXXX_XX1X : TP1_TM0addr <= TP1_TMA6;
                8'bXXXX_XXX1 : TP1_TM0addr <= TP1_TMA7;
                default      : TP1_TM0addr <= 16'bX;
           endcase
           //TM#1
           casex(TP1baSEL1)
                8'b1XXX_XXXX : TP1_TM1addr <= TP1_TMA0;
                8'bX1XX_XXXX : TP1_TM1addr <= TP1_TMA1;
                8'bXX1X_XXXX : TP1_TM1addr <= TP1_TMA2;
                8'bXXX1_XXXX : TP1_TM1addr <= TP1_TMA3;
                8'bXXXX_1XXX : TP1_TM1addr <= TP1_TMA4;
                8'bXXXX_X1XX : TP1_TM1addr <= TP1_TMA5;
                8'bXXXX_XX1X : TP1_TM1addr <= TP1_TMA6;
                8'bXXXX_XXX1 : TP1_TM1addr <= TP1_TMA7;
                default      : TP1_TM1addr <= 16'bX;
           endcase
           //TM#2
           casex(TP1baSEL2)
                8'b1XXX_XXXX : TP1_TM2addr <= TP1_TMA0;
                8'bX1XX_XXXX : TP1_TM2addr <= TP1_TMA1;
                8'bXX1X_XXXX : TP1_TM2addr <= TP1_TMA2;
                8'bXXX1_XXXX : TP1_TM2addr <= TP1_TMA3;
                8'bXXXX_1XXX : TP1_TM2addr <= TP1_TMA4;
                8'bXXXX_X1XX : TP1_TM2addr <= TP1_TMA5;
                8'bXXXX_XX1X : TP1_TM2addr <= TP1_TMA6;
                8'bXXXX_XXX1 : TP1_TM2addr <= TP1_TMA7;
                default      : TP1_TM2addr <= 16'bX;
           endcase
           //TM#3
           casex(TP1baSEL3)
                8'b1XXX_XXXX : TP1_TM3addr <= TP1_TMA0;
                8'bX1XX_XXXX : TP1_TM3addr <= TP1_TMA1;
                8'bXX1X_XXXX : TP1_TM3addr <= TP1_TMA2;
                8'bXXX1_XXXX : TP1_TM3addr <= TP1_TMA3;
                8'bXXXX_1XXX : TP1_TM3addr <= TP1_TMA4;
                8'bXXXX_X1XX : TP1_TM3addr <= TP1_TMA5;
                8'bXXXX_XX1X : TP1_TM3addr <= TP1_TMA6;
                8'bXXXX_XXX1 : TP1_TM3addr <= TP1_TMA7;
                default      : TP1_TM3addr <= 16'bX;
           endcase
      end
endmodule
```

//Texture memory interface
Wtp1_tmi   sstp1_tmi();

```
//Texture memory interface
module    Wtp1_tmi(    REclk, TP1_TMcmd, TP1baON,
          TP1_TMBADDR, TP1_TM0addr, TP1_TM1addr,
          TP1_TM2addr,TP1_TM3addr,
          TMI_TMBADDR, TMI_TM0nREQ, TMI_TM0cmd, TMI_TM0addr,
          TMI_TM1nREQ, TMI_TM1cmd, TMI_TM1addr,
          TMI_TM2nREQ, TMI_TM2cmd, TMI_TM2addr,
          TMI_TM3nREQ, TMI_TM3cmd, TMI_TM3addr);

     /* Combinational logics */
     //TMmask
     always @(TP1_TMcmd or writeON or TP1baON)
     begin
          casex(TP1_TMcmd)
               'WTM_CMD_READ      : TMmask <= TP1baON;
               default            : TMmask <= 4'b1111;
          endcase
     end
     //TM#cmd
     assign  TM0cmd = TP1_TMcmd & {3{TMmask[3]}};          // TM0
     assign  TM1cmd = TP1_TMcmd                            // TM1
     assign  TM2cmd = TP1_TMcmd & {3{TMmask[1]}};          // TM2
     assign  TM3cmd = TP1_TMcmd & {3{TMmask[0]}};          // TM3
     //TM#addr
     always @(TP1_TMcmd or tmADDR or
          TP1_TM0addr or TP1_TM1addr or TP1_TM2addr or TP1_TM3addr)
     begin
          casex(TP1_TMcmd)
               'WTM_CMD_READ :
               begin
                    TM0addr <= TP1_TM0addr;
                    TM1addr <= TP1_TM1addr;
                    TM2addr <= TP1_TM2addr;
                    TM3addr <= TP1_TM3addr;
               end
               default :
               begin
                    TM0addr <= 16'b0;
                    TM1addr <= 16'b0;
                    TM2addr <= 16'b0;
                    TM3addr <= 16'b0;
               end
          endcase
     end

     //Command & address
     always @(negedge REclk)
     begin
          TMI_TMBADDR <= TP1_TMBADDR;
          //REQ & command & address
          TMI_TM0nREQ <= ~(|TM0cmd);
          TMI_TM0cmd <= TM0cmd;
          TMI_TM0addr <= TM0addr;
          TMI_TM1nREQ <= ~(|TM1cmd);
          TMI_TM1cmd <= TM1cmd;
          TMI_TM1addr <= TM1addr;
          TMI_TM2nREQ <= ~(|TM2cmd);
          TMI_TM2cmd <= TM2cmd;
          TMI_TM2addr <= TM2addr;
          TMI_TM3nREQ <= ~(|TM3cmd);
```

```
            TMI_TM3cmd <= TM3cmd;
            TMI_TM3addr <= TM3addr;
        end
endmodule
```

```
endmodule
```

7.5.4.11 TP2: Texture Prefetch #2

The TP2 stage (Figure 7.28) latches the texture data from the texture cache and it transfers to the next pipeline stage.

Pipe Inputs

- REclk: rendering clock
- Pipe controls
- Common signals
- CAL controls
- TMA Controls
- PP dependent signals

Pipe Outputs

- Pipe controls
- Common signals
- CAL controls

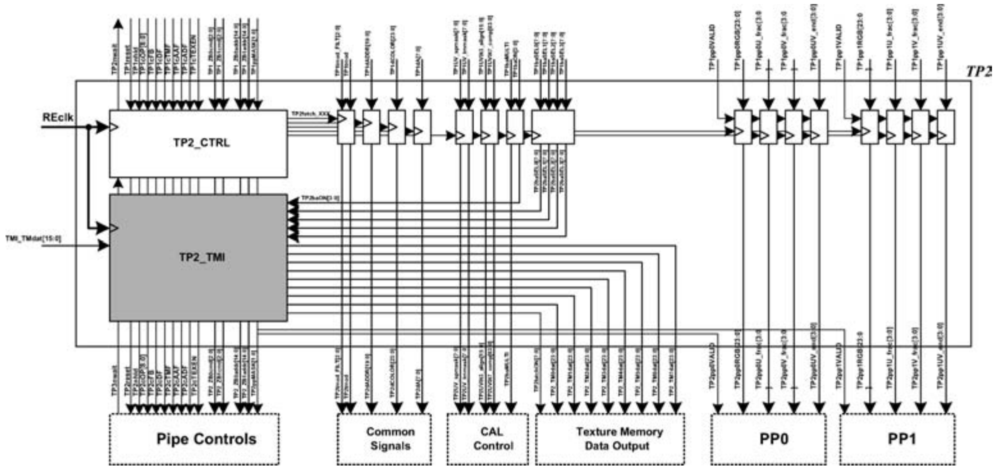


Figure 7.28 Texture prefetch #2

- Texture memory data outputs
- PP dependent signals

Memory Inputs

- Texture data (in)

Functions

- Store texture memory data from texture memory
- Control MUX of data bus to prevent race at negative edge REclk

Datapath #1: TP2_TMI
See Figure 7.29.

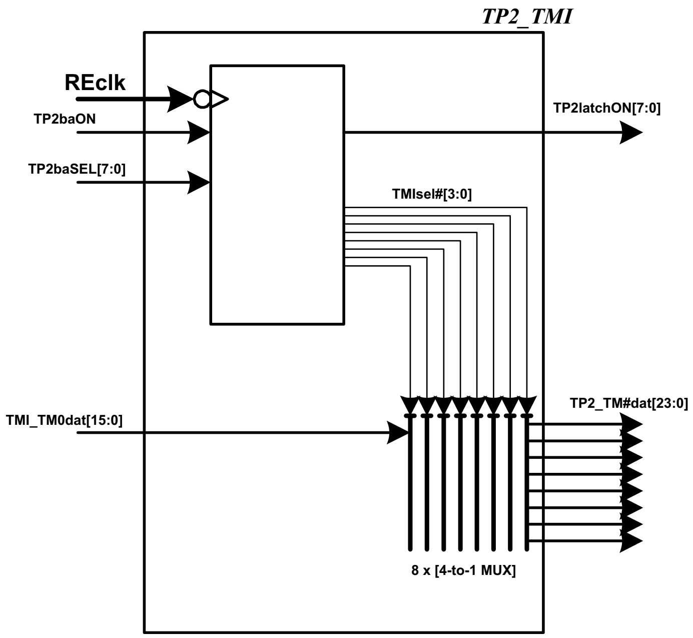


Figure 7.29 Datapath #1: TP2_TMI

Control signal	Description
TP2latchON[7:0]	Latch enable signal “Enable signal for next-stage latch” ({8{TP2baON[3]}} & TP2baSEL0) | . . .

TMsel[7:0]	MUX select control Latch @ negative latch (if TP2baON to reduce bus transition)

RTL Code

```
/*
* RAMP-GR
* RAMP-GR    TP2 Module  :    Texture Prefetch #2
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wtp2(     REclk, TP3nwait, TP1reset, TP1nhld, TP1cOP, TP1cFB, TP1cDF,
        TP1cTMF, TP1cAF, TP1cTEXEN, TP1cFMB, TP1cSCR,
        TP1_ZB0cmd, TP1_ZB1cmd, TP1_ZB0addr, TP1_ZB1addr,
        TP1ppMASK, TP1tmod_FILT, TP1tmod, TP1dADDR, TP1dA,
        TP1UV_spmask, TP1UV_tmmask, TP1UV0t3_align, TP1UV0t7_comp, TP1baMULTI,
        TP1baON, TP1baSEL0, TP1baSEL1, TP1baSEL2, TP1baSEL3,
        TP1pp0VALID, TP1pp0RGB, TP1pp0U_frac, TP1pp0V_frac, TP1pp0UV_end,
        TP1pp1VALID, TP1pp1RGB, TP1pp1U_frac, TP1pp1V_frac, TP1pp1UV_end,
        TMI_TM0dat, TMI_TM1dat, TMI_TM2dat, TMI_TM3dat,
        TMI_TM0nWAIT, TMI_TM1nWAIT, TMI_TM2nWAIT, TMI_TM3nWAIT,
        TP2nwait, TP2reset, TP2nhld, TP2cOP, TP2cFB, TP2cDF,
        TP2cTMF, TP2cAF, TP2cTEXEN, TP2cFMB, TP2cSCR,
        TP2_ZB0cmd, TP2_ZB1cmd, TP2_ZB0addr, TP2_ZB1addr, TP2ppMASK,
        TP2tmod_FILT, TP2tmod, TP2dADDR, TP2dA,
        TP2UV_spmask, TP2UV_tmmask, TP2UV0t3_align, TP2UV0t7_comp, TP2baMULTI,
        TP2latchON, TP2_TM0dat, TP2_TM1dat, TP2_TM2dat, TP2_TM3dat, TP2_TM4dat,
        TP2_TM5dat, TP2_TM6dat, TP2_TM7dat,
        TP2pp0VALID, TP2pp0RGB, TP2pp0U_frac, TP2pp0V_frac, TP2pp0UV_end,
        TP2pp1VALID, TP2pp1RGB, TP2pp1U_frac, TP2pp1V_frac, TP2pp1UV_end);

    /* Wire renaming */

    assign #1  TP2pp0VALID = TP2ppMASK[1];
    assign #1  TP2pp1VALID = TP2ppMASK[0];
    reg     TMI_TMnWAIT;
    always @(posedge REclk)
    begin
        TMI_TMnWAIT <= TMI_TM0nWAIT & TMI_TM1nWAIT & TMI_TM2nWAIT
                       & TMI_TM3nWAIT;
    end

    /* Module interconnection */


//Main controller
Wtp2_ctrl      sstp2_ctrl();
```

```
//Main controller
module    Wtp2_ctrl(    REclk, TP3nwait, TP1reset, TP1nhld,
          TP1cOP, TP1cFB, TP1cDF, TP1cTMF,
          TP1cAF, TP1cTEXEN, TP1cFMB, TP1cSCR,
          TP1_ZB0cmd, TP1_ZB1cmd, TP1_ZB0addr, TP1_ZB1addr, TP1ppMASK,
          TMI_TMnWAIT,
          TP2nwait, TP2reset, TP2nhld, TP2cOP, TP2cFB, TP2cDF, TP2cTMF,
          TP2cAF, TP2cTEXEN, TP2cFMB, TP2cSCR,
          TP2_ZB0cmd, TP2_ZB1cmd, TP2_ZB0addr, TP2_ZB1addr, TP2ppMASK,
          TP2fetch_CLK, TP2fetch_TMOD, TP2fetch_ADDR, TP2fetch_ALPHA,
          TP2fetch_PXL, TP2fetch_TXL);

     /* Combinational logics */

     //nwait
     assign  TP2nwait = ((TP3nwait & TMI_TMnWAIT) | ~TP1reset) ;
     //nhld
     assign  TP2nhld = TP2nhld_int & TMI_TMnWAIT;
     //Fetch enables
     assign #1  TP2fetch_CLK = TP2nwait & TP1nhld;
     assign  TP2fetch_TMOD = TP1cTMF;
     assign  TP2fetch_ADDR = TP1cAF;
     assign  TP2fetch_ALPHA = TP1cDF;
     assign  TP2fetch_PXL = TP1cDF;
     assign  TP2fetch_TXL = TP1cDF & TP1cTEXEN;

     always @(posedge REclk)
     begin
          if(TP2nwait)
          begin
               //State controls
               TP2nhld_int <= TP1nhld;
               TP2reset <= TP1reset;
               //Fetch signals
               TP2cDF_int <= TP1cDF;
               TP2cTMF_int <= TP1cTMF;
               TP2cAF_int <= TP1cAF;
               TP2cTEXEN_int <= TP1cTEXEN;
               TP2cFMB_int <= TP1cFMB;
               //Trivial signals
               TP2cOP <= TP1cOP;
               TP2cFB <= TP1cFB;
               TP2cSCR <= TP1cSCR;
               TP2_ZB0cmd <= TP1_ZB0cmd;
               TP2_ZB1cmd <= TP1_ZB1cmd;
               TP2_ZB0addr <= TP1_ZB0addr;
               TP2_ZB1addr <= TP1_ZB1addr;
               TP2ppMASK <= TP1ppMASK;
          end
     end
     always @(negedge REclk)
     begin
          TP2cDF <= TP2cDF_int;
          TP2cTMF <= TP2cTMF_int;
          TP2cAF <= TP2cAF_int;
          TP2cTEXEN <= TP2cTEXEN_int;
          TP2cFMB <= TP2cFMB_int;
     end
endmodule
```

//Data latches
Wtp2_latch   sstp2_latch();

```
//Data latches
module    Wtp2_latch( REclk, clk, TP2fetch_TMOD, TP2fetch_ADDR, TP2fetch_ALPHA,
          TP2fetch_PXL, TP2fetch_TXL,
          TP1tmod_FILT, TP1tmod, TP1dADDR, TP1dA, TP1UV_spmask,
          TP1UV_tmmask,TP1UV0t3_align, TP1UV0t7_comp, TP1baMULTI,
          TP1baON, TP1baSEL0, TP1baSEL1, TP1baSEL2, TP1baSEL3,
          TP1pp0VALID, TP1pp0RGB, TP1pp0U_frac, TP1pp0V_frac,
          TP1pp0UV_end, TP1pp1VALID, TP1pp1RGB, TP1pp1U_frac,
          TP1pp1V_frac, TP1pp1UV_end,
          TP2tmod_FILT, TP2tmod, TP2dADDR, TP2dA,
          TP2UV_spmask, TP2UV_tmmask, TP2UV0t3_align, TP2UV0t7_comp,
          TP2baMULTI, TP2baON, TP2baSEL0, TP2baSEL1,
          TP2baSEL2, TP2baSEL3,
          TP2pp0RGB, TP2pp0U_frac, TP2pp0V_frac, TP2pp0UV_end,
          TP2pp1RGB, TP2pp1U_frac, TP2pp1V_frac, TP2pp1UV_end);

     /* Latching */
     //TP2_LATCH_TMOD
     always @(posedge REclk)
     begin
          if(clk&TP2fetch_TMOD)
          begin
               TP2tmod_FILT <= TP1tmod_FILT;
               TP2tmod <= TP1tmod;
          end
     end
     //TP2_LATCH_ADDR
     always @(posedge REclk)
     begin
          if(clk&TP2fetch_ADDR) TP2dADDR <= TP1dADDR;
     end
     //TP2_LATCH_ALPHA
     always @(posedge REclk)
     begin
          if(clk&TP2fetch_ALPHA) TP2dA <= TP1dA;
     end
     //TP2_LATCH_CAL : TXL
     always @(posedge REclk)
     begin
          if(clk&TP2fetch_TXL)
          begin
               TP2UV_spmask <= TP1UV_spmask;
               TP2UV_tmmask <= TP1UV_tmmask;
               TP2UV0t3_align <= TP1UV0t3_align;
               TP2UV0t7_comp <= TP1UV0t7_comp;
               TP2baMULTI <= TP1baMULTI;
```

```
                TP2baON <= TP1baON;
                TP2baSEL0 <= TP1baSEL0;
                TP2baSEL1 <= TP1baSEL1;
                TP2baSEL2 <= TP1baSEL2;
                TP2baSEL3 <= TP1baSEL3;
           end
      end
      //TP2_LATCH_PP0_PXL
      always @(posedge REclk)
      begin
           if(clk & TP1pp0VALID&TP2fetch_PXL) TP2pp0RGB <= TP1pp0RGB;
      end
      //TP2_LATCH_PP0_TXL
      always @(posedge REclk)
      begin
           if(clk & TP1pp0VALID&TP2fetch_TXL)
           begin
                TP2pp0U_frac <= TP1pp0U_frac;
                TP2pp0V_frac <= TP1pp0V_frac;
                TP2pp0UV_end <= TP1pp0UV_end;
           end
      end
      //TP2_LATCH_PP1_PXL
      always @(posedge REclk)
      begin
           if(clk & TP1pp1VALID&TP2fetch_PXL) TP2pp1RGB <= TP1pp1RGB;
      end
      //TP2_LATCH_PP1_TXL
      always @(posedge REclk)
      begin
           if(clk & TP1pp1VALID&TP2fetch_TXL)
           begin
                TP2pp1U_frac <= TP1pp1U_frac;
                TP2pp1V_frac <= TP1pp1V_frac;
                TP2pp1UV_end <= TP1pp1UV_end;
           end
      end
endmodule
```

//Texture memory interface
Wtp2_tmi   sstp2_tmi();

```
//Texture memory interface: MUX Unix
module     Wtp2_tmi_muxunit(   TMsel,TMin0,TMin1,TMin2,TMin3,TMout);

      /* Inputs */
      input  [3:0]      TMsel;
      input  [23:0]     TMin0;
      input  [23:0]     TMin1;
      input  [23:0]     TMin2;
      input  [23:0]     TMin3;
      /* Outputs */
      output     [23:0]     TMout;
      reg        [23:0]     TMout;
```

```
     always @(TMsel or TMin0 or TMin1 or TMin2 or TMin3)
     begin
          casex(TMsel)
               4'b1XXX : TMout <= TMin0;
               4'bX1XX : TMout <= TMin1;
               4'bXX1X : TMout <= TMin2;
               4'bXXX1 : TMout <= TMin3;
               default : TMout <= 24'bX;
          endcase
     end
endmodule

//Texture memory interface
module    Wtp2_tmi(  REclk,TP2baON,TP2baSEL0,TP2baSEL1,TP2baSEL2,TP2ba-
SEL3,
          TP2latchON,TP2_TM0dat,TP2_TM1dat,TP2_TM2dat,TP2_TM3dat,
          TP2_TM4dat,TP2_TM5dat,TP2_TM6dat,TP2_TM7dat,
          TMI_TM0dat,TMI_TM1dat,TMI_TM2dat,TMI_TM3dat);

     /* Combinational logic */

     assign #1  TP2latchON =    ({8{TP2baON[3]}} & TP2baSEL0) |     //TM0
                                ({8{TP2baON[2]}} & TP2baSEL1) |     //TM1
                                ({8{TP2baON[1]}} & TP2baSEL2) |     //TM2
                                ({8{TP2baON[0]}} & TP2baSEL3);      //TM3

     /* Control Latch */
     always @(negedge REclk)
     begin
          if(|TP2baON)
          begin
          TMsel0 <= {TP2baSEL0[7],TP2baSEL1[7],TP2baSEL2[7],TP2baSEL3[7]};
          TMsel1 <= {TP2baSEL0[6],TP2baSEL1[6],TP2baSEL2[6],TP2baSEL3[6]};
          TMsel2 <= {TP2baSEL0[5],TP2baSEL1[5],TP2baSEL2[5],TP2baSEL3[5]};
          TMsel3 <= {TP2baSEL0[4],TP2baSEL1[4],TP2baSEL2[4],TP2baSEL3[4]};
          TMsel4 <= {TP2baSEL0[3],TP2baSEL1[3],TP2baSEL2[3],TP2baSEL3[3]};
          TMsel5 <= {TP2baSEL0[2],TP2baSEL1[2],TP2baSEL2[2],TP2baSEL3[2]};
          TMsel6 <= {TP2baSEL0[1],TP2baSEL1[1],TP2baSEL2[1],TP2baSEL3[1]};
          TMsel7 <= {TP2baSEL0[0],TP2baSEL1[0],TP2baSEL2[0],TP2baSEL3[0]};
          end
     end

     /* Buffer insertion for synthesis: one more inverter @ WTP2_TOP */
     assign TMI_TM0dat_buf = {TMI_TM0dat[15:11], 3'b0, TMI_TM0dat[10:5], 2'b0,
                    TMI_TM0dat[4:0], 3'b0};
     assign TMI_TM1dat_buf = {TMI_TM1dat[15:11], 3'b0, TMI_TM1dat[10:5], 2'b0,
                    TMI_TM1dat[4:0], 3'b0};
     assign TMI_TM2dat_buf = {TMI_TM2dat[15:11], 3'b0, TMI_TM2dat[10:5], 2'b0,
                    TMI_TM2dat[4:0], 3'b0};
     assign TMI_TM3dat_buf = {TMI_TM3dat[15:11], 3'b0, TMI_TM3dat[10:5], 2'b0,
                    TMI_TM3dat[4:0], 3'b0};

     /* Module interconnection */
     Wtp2_tmi_muxunit   sstp2_tmi_mux0();
     Wtp2_tmi_muxunit   sstp2_tmi_mux1();
     Wtp2_tmi_muxunit   sstp2_tmi_mux2();
     Wtp2_tmi_muxunit   sstp2_tmi_mux3();
     Wtp2_tmi_muxunit   sstp2_tmi_mux4();
     Wtp2_tmi_muxunit   sstp2_tmi_mux5();
     Wtp2_tmi_muxunit   sstp2_tmi_mux6();
     Wtp2_tmi_muxunit   sstp2_tmi_mux7();
endmodule
```

```
endmodule
```

7.5.4.12 TP3: Texture Prefetch #3

In the TP3 stage (Figure 7.30), the reverse operation of the address alignment logic is performed. Using the previous used texels, which are stored in the register, and the new texels from the texture caches, four bilinear texels are aligned in this stage; and the aligned texels are transferred to the next pipeline stage, texture filtering.

Pipe Inputs

- Pipe controls
- Common signals
- CAL controls
- Texture memory data output
- PP dependent signals

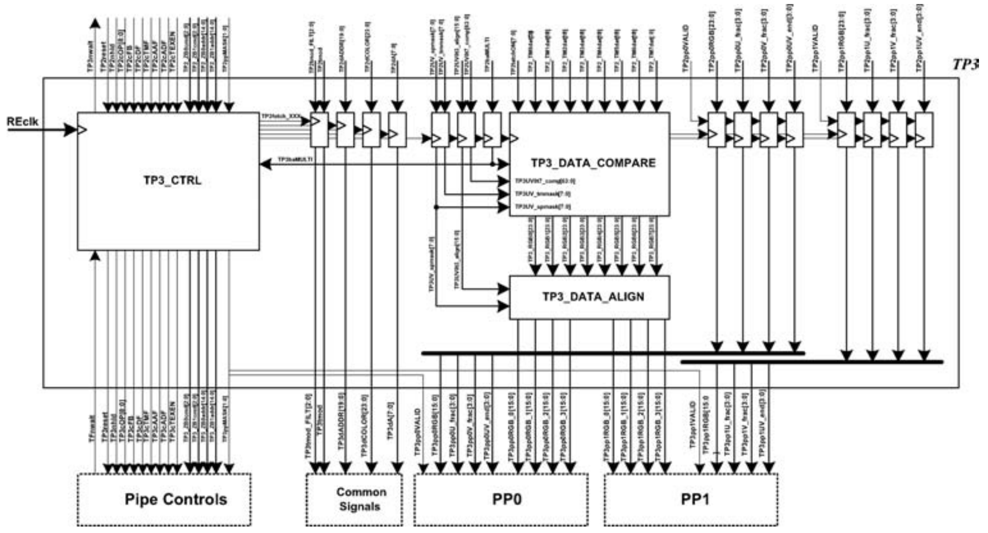


Figure 7.30 Texture prefetch #3

Pipe Outputs

- Pipe controls
- Common signals
- PP dependent signals
  - ALID
  - Pixel RGB
  - Texel U,V, End
  - Texel RGB ×4

Functions

- Cache alignment logic: reverse operation
- Realign texel

Signal Descriptions

Control signal	Description
TP3baMULTI	Multi-cycle control signal TP3nhld = 0 (if MULTI==1)

Datapath #1: TP3_DATA_COMPARE
See Figure 7.31.

Datapath #2: TP3_DA_COMP_REVUNIT
See Figure 7.32.

RTL Code

```
/*
* RAMP-GR
* RAMP-GR      TP3 Module: Texture Prefetch #3
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wtp3(    REclk, TFnwait, TP2reset, TP2nhld, TP2cOP, TP2cFB, TP2cDF,
                TP2cTMF, TP2cAF, TP2cTEXEN, TP2cFMB, TP2cSCR,
                TP2_ZB0cmd, TP2_ZB1cmd, TP2_ZB0addr, TP2_ZB1addr, TP2ppMASK,
                TP2tmod_FILT, TP2tmod, TP2dADDR, TP2dA,
                TP2UV_spmask, TP2UV_tmmask, TP2UV0t3_align, TP2UV0t7_comp,
                TP2baMULTI, TP2latchON,
                TP2_TM0dat, TP2_TM1dat, TP2_TM2dat, TP2_TM3dat,
                TP2_TM4dat, TP2_TM5dat, TP2_TM6dat, TP2_TM7dat,
                TP2pp0VALID, TP2pp0RGB, TP2pp0U_frac, TP2pp0V_frac, TP2pp0UV_end,
```

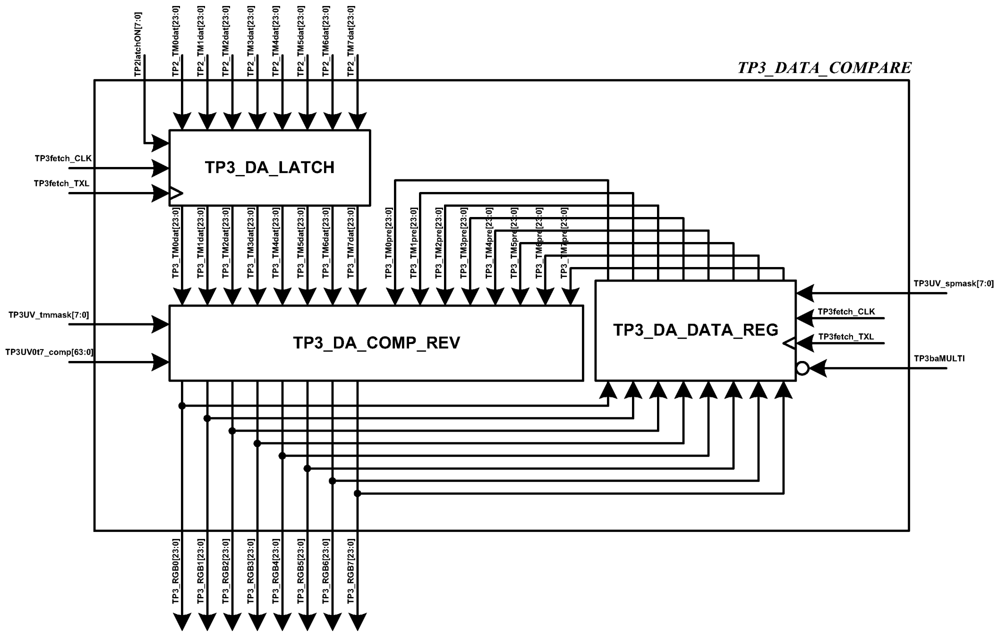


Figure 7.31 Datapath #1: TP3_DATA_COMPARE

```
          TP2pp1VALID, TP2pp1RGB, TP2pp1U_frac, TP2pp1V_frac, TP2pp1UV_end,
          TP3nwait, TP3reset, TP3nhld, TP3cOP, TP3cFB, TP3cDF,
          TP3cTMF, TP3cAF, TP3cTEXEN, TP3cFMB, TP3cSCR,
          TP3_ZB0cmd, TP3_ZB1cmd, TP3_ZB0addr, TP3_ZB1addr, TP3ppMASK,
          TP3tmod_FILT, TP3tmod, TP3dADDR, TP3dA,
          TP3pp0VALID, TP3pp0RGB, TP3pp0U_frac, TP3pp0V_frac, TP3pp0UV_end,
          TP3pp0RGB_0, TP3pp0RGB_1, TP3pp0RGB_2, TP3pp0RGB_3,
          TP3pp1VALID, TP3pp1RGB, TP3pp1U_frac, TP3pp1V_frac, TP3pp1UV_end,
          TP3pp1RGB_0, TP3pp1RGB_1, TP3pp1RGB_2, TP3pp1RGB_3);

     /* Renaming */
     assign #1  TP3pp0VALID = TP3ppMASK[1];
     assign #1  TP3pp1VALID = TP3ppMASK[0];

     /* Module interconnection */
```

//Main controller
Wtp3_ctrl   Wtp3_ctrl();

```
//Main controller
module    Wtp3_ctrl (  REclk, TFnwait, TP2reset, TP2nhld, TP2cOP, TP2cFB, TP2cDF,
          TP2cTMF, TP2cAF, TP2cTEXEN, TP2cFMB, TP2cSCR,
          TP2_ZB0cmd, TP2_ZB1cmd, TP2_ZB0addr, TP2_ZB1addr, TP2ppMASK,
          TP3baMULTI,
          TP3nwait, TP3reset, TP3nhld, TP3cOP, TP3cFB, TP3cDF,
```

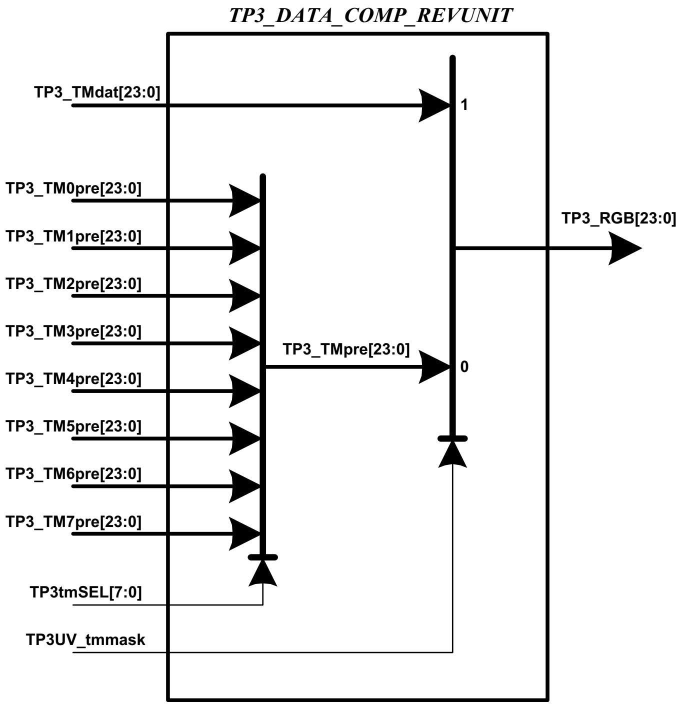


Figure 7.32 Datapath #2: TP3_DA_COMP_REVUNIT

```
          TP3cTMF, TP3cAF, TP3cTEXEN, TP3cFMB, TP3cSCR,
          TP3_ZB0cmd, TP3_ZB1cmd, TP3_ZB0addr, TP3_ZB1addr, TP3ppMASK,
          TP3fetch_CLK, TP3fetch_TMOD, TP3fetch_ADDR,
          TP3fetch_ALPHA, TP3fetch_PXL, TP3fetch_TXL);

     //nwait
     assign  TP3nwait = TFnwait | ~TP2reset;
     //nhld
     assign TP3nhld_gen = ~(TP3cDF_int & TP3cOP[5] & TP3cTEXEN_int & TP3baMULTI);
     assign  TP3nhld = TP3nhld_int & TP3nhld_gen;
     //Fetch signals
     assign #1  TP3fetch_CLK = TP3nwait & TP2nhld;
     assign  TP3fetch_TMOD = TP2cTMF;
     assign  TP3fetch_ADDR = TP2cAF;
     assign  TP3fetch_ALPHA = TP2cDF;
     assign  TP3fetch_PXL = TP2cDF;
     assign  TP3fetch_TXL = TP2cDF & TP2cTEXEN;

always @(posedge REclk)
     begin
          if(TP3nwait)
          begin
               //State control
               TP3nhld_int <= TP2nhld;
```

```
                TP3reset <= TP2reset;
                //Fetch signals
                TP3cDF_int <= TP2cDF;
                TP3cTMF_int <= TP2cTMF;
                TP3cAF_int <= TP2cAF;
                TP3cTEXEN_int <= TP2cTEXEN;
                TP3cFMB_int <= TP2cFMB;
                //Trivial signals
                TP3cOP <= TP2cOP;
                TP3cFB <= TP2cFB;
                TP3cSCR <= TP2cSCR;
                TP3_ZB0cmd <= TP2_ZB0cmd;
                TP3_ZB1cmd <= TP2_ZB1cmd;
                TP3_ZB0addr <= TP2_ZB0addr;
                TP3_ZB1addr <= TP2_ZB1addr;
                TP3ppMASK <= TP2ppMASK;
           end
      end
      always @(negedge REclk)
      begin
           TP3cDF <= TP3cDF_int;
           TP3cTMF <= TP3cTMF_int;
           TP3cAF <= TP3cAF_int;
           TP3cTEXEN <= TP3cTEXEN_int;
           TP3cFMB <= TP3cFMB_int;
      end
endmodule
```

//Data latching
Wtp3_latch   Wtp3_latch();

```
//Data latching
module    Wtp3_latch(  REclk, clk, TP3fetch_TMOD, TP3fetch_ADDR,
                TP3fetch_ALPHA, TP3fetch_PXL, TP3fetch_TXL,
                TP2tmod_FILT, TP2tmod, TP2dADDR, TP2dA,
                TP2UV_spmask, TP2UV_tmmask, TP2UV0t3_align, TP2UV0t7_comp,
                TP2baMULTI, TP2pp0VALID, TP2pp0RGB, TP2pp0U_frac,
                TP2pp0V_frac,
                TP2pp0UV_end, TP2pp1VALID, TP2pp1RGB, TP2pp1U_frac,
                TP2pp1V_frac, TP2pp1UV_end,
                TP3tmod_FILT, TP3tmod, TP3dADDR, TP3dA,
                TP3UV_spmask, TP3UV_tmmask, TP3UV0t3_align, TP3UV0t7_comp,
                TP3baMULTI,
                TP3pp0RGB, TP3pp0U_frac, TP3pp0V_frac, TP3pp0UV_end,
                TP3pp1RGB, TP3pp1U_frac, TP3pp1V_frac, TP3pp1UV_end);

      //TMOD
      always @(posedge REclk)
      begin
           if(clk & TP3fetch_TMOD)
           begin
                TP3tmod_FILT <= TP2tmod_FILT;
                TP3tmod <= TP2tmod;
           end
```

```
     end
     //ADDR
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_ADDR) TP3dADDR <= TP2dADDR;
     end
     //ALPHA
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_ALPHA) TP3dA <= TP2dA;
     end
     //TXL
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_TXL)
          begin
               TP3UV_spmask <= TP2UV_spmask;
               TP3UV_tmmask <= TP2UV_tmmask;
               TP3UV0t3_align <= TP2UV0t3_align;
               TP3UV0t7_comp <= TP2UV0t7_comp;
               TP3baMULTI <= TP2baMULTI;
          end
     end
     //PP0_PXL'
     always @(posedge REclk)
     begin
          if (clk & TP3fetch_PXL)  TP3pp0RGB <= TP2pp0RGB;
     end
     //PP0_TXL
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_TXL)
          begin
               TP3pp0U_frac <= TP2pp0U_frac;
               TP3pp0V_frac <= TP2pp0V_frac;
               TP3pp0UV_end <= TP2pp0UV_end;
          end
     end
     //PP1_PXL
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_PXL)  TP3pp1RGB <= TP2pp1RGB;
     end
     //PP1_TXL
     always @(posedge REclk)
     begin
          if(clk & TP3fetch_TXL)
          begin
               TP3pp1U_frac <= TP2pp1U_frac;
               TP3pp1V_frac <= TP2pp1V_frac;
               TP3pp1UV_end <= TP2pp1UV_end;
          end
     end
endmodule
```

//Data compare
Wtp3_data_compare   sstp3_data_compare();

```
//Data compare reverse unit block
module     Wtp3_data_comp_revunit( TP3tmSEL, TP3UV_tmmask, TP3_TMdat,
                 TP3_TM0pre, TP3_TM1pre, TP3_TM2pre, TP3_TM3pre,
                 TP3_TM4pre, TP3_TM5pre, TP3_TM6pre, TP3_TM7pre, TP3_RGB);

     /* MUX */
     //PRE-SEL: priority decoder
     always @(TP3tmSEL or TP3_TM0pre or TP3_TM1pre or TP3_TM2pre or TP3_TM3pre or
                 TP3_TM4pre or TP3_TM5pre or TP3_TM6pre or TP3_TM7pre)
     begin
          casex(TP3tmSEL)
               8'b1XXX_XXXX : TP3_TMpre <= TP3_TM0pre;
               8'b01XX_XXXX : TP3_TMpre <= TP3_TM1pre;
               8'b001X_XXXX : TP3_TMpre <= TP3_TM2pre;
               8'b0001_XXXX : TP3_TMpre <= TP3_TM3pre;
               8'b0000_1XXX : TP3_TMpre <= TP3_TM4pre;
               8'b0000_01XX : TP3_TMpre <= TP3_TM5pre;
               8'b0000_001X : TP3_TMpre <= TP3_TM6pre;
               8'b0000_0001 : TP3_TMpre <= TP3_TM7pre;
               default      : TP3_TMpre <= 24'bX;
          endcase
     end
     //OUT-SEL
     always @(TP3UV_tmmask or TP3_TMdat or TP3_TMpre)
     begin
          if(TP3UV_tmmask)      TP3_RGB <= TP3_TMdat;    // Pass-thru
          else          TP3_RGB <= TP3_TMpre;      // Internal
     end
endmodule


//Data compare reverse unit block
module     Wtp3_da_comp_rev( TP3UV_tmmask, TP3UV0t7_comp,
                 TP3_TM0dat, TP3_TM1dat, TP3_TM2dat, TP3_TM3dat,
                 TP3_TM4dat, TP3_TM5dat, TP3_TM6dat, TP3_TM7dat,
                 TP3_TM0pre, TP3_TM1pre, TP3_TM2pre, TP3_TM3pre,
                 TP3_TM4pre, TP3_TM5pre, TP3_TM6pre, TP3_TM7pre,
                 TP3_RGB0, TP3_RGB1, TP3_RGB2, TP3_RGB3,
                 TP3_RGB4, TP3_RGB5, TP3_RGB6, TP3_RGB7);

     //Reassign wires
     assign #1  TP3tmSEL0 = TP3UV0t7_comp[63:56];
     assign #1  TP3tmSEL1 = TP3UV0t7_comp[55:48];
     assign #1  TP3tmSEL2 = TP3UV0t7_comp[47:40];
     assign #1  TP3tmSEL3 = TP3UV0t7_comp[39:32];
     assign #1  TP3tmSEL4 = TP3UV0t7_comp[31:24];
     assign #1  TP3tmSEL5 = TP3UV0t7_comp[23:16];
     assign #1  TP3tmSEL6 = TP3UV0t7_comp[16:8];
     assign #1  TP3tmSEL7 = TP3UV0t7_comp[7:0];

     /* Module interconnection */
     Wtp3_data_comp_revunit     sstp3_data_comp_rev0();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev1();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev2();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev3();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev4();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev5();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev6();
     Wtp3_data_comp_revunit     sstp3_data_comp_rev7();
endmodule

//Data compare
module   Wtp3_data_compare( REclk, TP3fetch_CLK, TP3fetch_TXL, TP3UV_tmmask,
               TP3UV_spmask, TP3UV0t7_comp, TP2latchON,
               TP2_TM0dat, TP2_TM1dat, TP2_TM2dat, TP2_TM3dat,
               TP2_TM4dat, TP2_TM5dat, TP2_TM6dat, TP2_TM7dat,
               TP3_RGB0, TP3_RGB1, TP3_RGB2, TP3_RGB3,
               TP3_RGB4, TP3_RGB5, TP3_RGB6, TP3_RGB7,TP3baMULTI);

     //TP3_DA_LATCH
     always @(posedge REclk)
     begin
          if(TP3fetch_CLK&TP3fetch_TXL)
          begin
               if(TP2latchON[7])     TP3_TM0dat <= TP2_TM0dat;
               if(TP2latchON[6])     TP3_TM1dat <= TP2_TM1dat;
               if(TP2latchON[5])     TP3_TM2dat <= TP2_TM2dat;
               if(TP2latchON[4])     TP3_TM3dat <= TP2_TM3dat;
               if(TP2latchON[3])     TP3_TM4dat <= TP2_TM4dat;
               if(TP2latchON[2])     TP3_TM5dat <= TP2_TM5dat;
               if(TP2latchON[1])     TP3_TM6dat <= TP2_TM6dat;
               if(TP2latchON[0])     TP3_TM7dat <= TP2_TM7dat;
          end
     end
     //TP3_DA_DATA_REG
     always @(posedge REclk)
     begin
          if(TP3fetch_CLK & ~TP3baMULTI & TP3fetch_TXL)
          begin
               if(TP3UV_spmask[7])     TP3_TM0pre <= TP3_RGB0;
               if(TP3UV_spmask[6])     TP3_TM1pre <= TP3_RGB1;
               if(TP3UV_spmask[5])     TP3_TM2pre <= TP3_RGB2;
               if(TP3UV_spmask[4])     TP3_TM3pre <= TP3_RGB3;
               if(TP3UV_spmask[3])     TP3_TM4pre <= TP3_RGB4;
               if(TP3UV_spmask[2])     TP3_TM5pre <= TP3_RGB5;
               if(TP3UV_spmask[1])     TP3_TM6pre <= TP3_RGB6;
               if(TP3UV_spmask[0])     TP3_TM7pre <= TP3_RGB7;
          end
     end

     Wtp3_da_comp_rev   sstp3_da_comp_rev();

endmodule
```

```
//Data align
Wtp3_data_align     sstp3_data_align();
```

```
//Data align unit
module      Wtp3_data_alignunit(TP3tmSEL, TP3UV_spmask, TP3_RGB,
               TP3_RGB0, TP3_RGB1, TP3_RGB2, TP3_RGB3, TP3pp1RGB);

     /* MUX */
     //PRE-SEL: pririty decoder
     always @(TP3tmSEL or TP3_RGB0 or TP3_RGB1 or TP3_RGB2 or TP3_RGB3)
     begin
          casex(TP3tmSEL)
               4'b1XXX : TP3pp1PRE <= TP3_RGB0;
               4'b01XX : TP3pp1PRE <= TP3_RGB1;
               4'b001X : TP3pp1PRE <= TP3_RGB2;
               4'b0001 : TP3pp1PRE <= TP3_RGB3;
               default : TP3pp1PRE <= 24'bX;
          endcase
     end
     //OUT-SEL
     always @(TP3UV_spmask or TP3pp1PRE or TP3_RGB)
     begin
          if(TP3UV_spmask)      TP3pp1RGB <= TP3_RGB; //Pass-through
          else                 TP3pp1RGB <= TP3pp1PRE; //Internal
     end
endmodule

//Data align
module     Wtp3_data_align(    TP3UV_spmask, TP3UV0t3_align,
          TP3_RGB0, TP3_RGB1, TP3_RGB2, TP3_RGB3,
          TP3_RGB4, TP3_RGB5, TP3_RGB6, TP3_RGB7,
          TP3pp0RGB_0, TP3pp0RGB_1, TP3pp0RGB_2, TP3pp0RGB_3,
          TP3pp1RGB_0, TP3pp1RGB_1, TP3pp1RGB_2, TP3pp1RGB_3);

     /* Reassign wires */
     //Select signals
     assign #1  TP3tmSEL0 = TP3UV0t3_align[15:12];
     assign #1  TP3tmSEL1 = TP3UV0t3_align[11:8];
     assign #1  TP3tmSEL2 = TP3UV0t3_align[7:4];
     assign #1  TP3tmSEL3 = TP3UV0t3_align[3:0];
     //PP0 feed-through data
     assign #1  TP3pp0RGB_0 = TP3_RGB0;
     assign #1  TP3pp0RGB_1 = TP3_RGB1;
     assign #1  TP3pp0RGB_2 = TP3_RGB2;
     assign #1  TP3pp0RGB_3 = TP3_RGB3;

     /* Module interconnection */
     Wtp3_data_alignunit     sstp3_data_align0();
     Wtp3_data_alignunit     sstp3_data_align1();
     Wtp3_data_alignunit     sstp3_data_align2();
     Wtp3_data_alignunit     sstp3_data_align3();
endmodule
```

```
endmodule
```

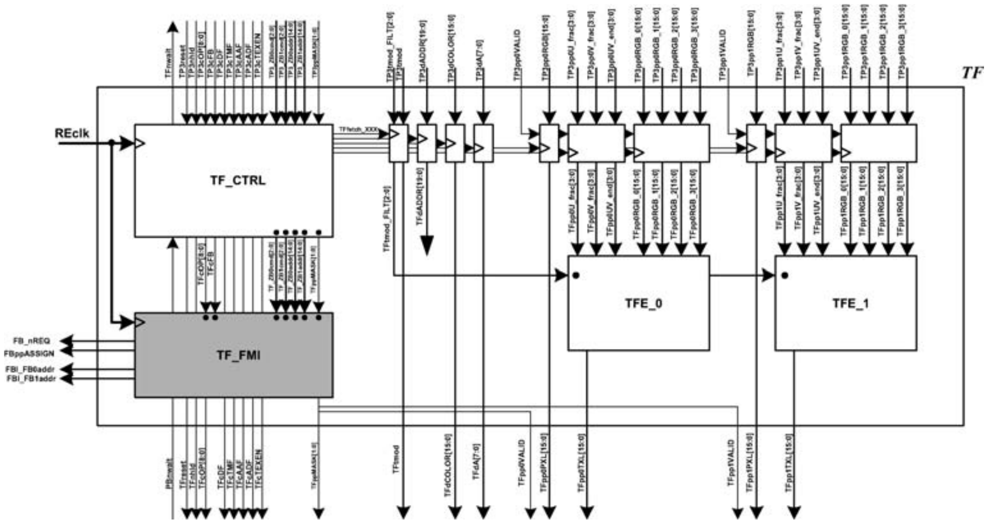


Figure 7.33 Texture filting unit

7.5.4.13 TF: Texture Filtering

The task of this stage (Figure 7.33) is to generate filtered texels and to generate the frame buffer requests. Using four texels from the previous stage, texture filtering is performed according to the texture mode. This stage generates the frame buffer read requests for the next pixel blending stage.

Pipe Inputs

- Pipe controls
- Common signals
- PP dependent signals

Pipe Outputs

- Pipe contorls
- Common signals
- PP valid signals
- Pixel RGB
- Bilinear filtered texel RGB.

Memory Outputs

- Frame buffer request
- Frame buffer address.

Functions

- Texture filtering (point sampling/bilinear filter)
- Frame buffer memory control

Frame Buffer Commands

Command	Functional description
RDAT	FBI_FB#cmd[2:0] = TF_ZB#cmd[2:0] FB0/FB1 Control <= TFppMASK[1:0]
TMOD	NOP
ASTR	FBI_FB#cmd[2:0] = TF_ZB#cmd[2:0] FB0/FB1 Control <= TFcFB (Both)

Datapath #1: TF_ENGINE

See Figure 7.34.

Signal	Description
TFpp#UV_end[1:0]	End-point select flag Flag 2'd0: no end point – select BLEND 2'd1: equals zero – select HIGH 2'd2: equals one – select LOW
TFtmod_FILT[2:0]	Texture filtering mode 3'b001: point sampling – select RGB_0 3'b010: bilinear filtering – select BLEND

Datapath #2: TF Blend Module

See Figure 7.35.

$$\begin{aligned} C &= A \times f + B \times (1-f) \\ &= A \times f + B - B \times f \\ &= f \times (A-B) + B \\ &= (2^4 \times f \times (A-B) + 2^4 \times B)/2^4 \\ &= (\mathit{frac}[3:0] \times (A-B) + \{B \ll 4, 4'b0\}) \gg 4 \end{aligned}$$

RTL Code

```
/*
* RAMP-GR
* RAMP-GR TF Module: Textue Filtering
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/
```

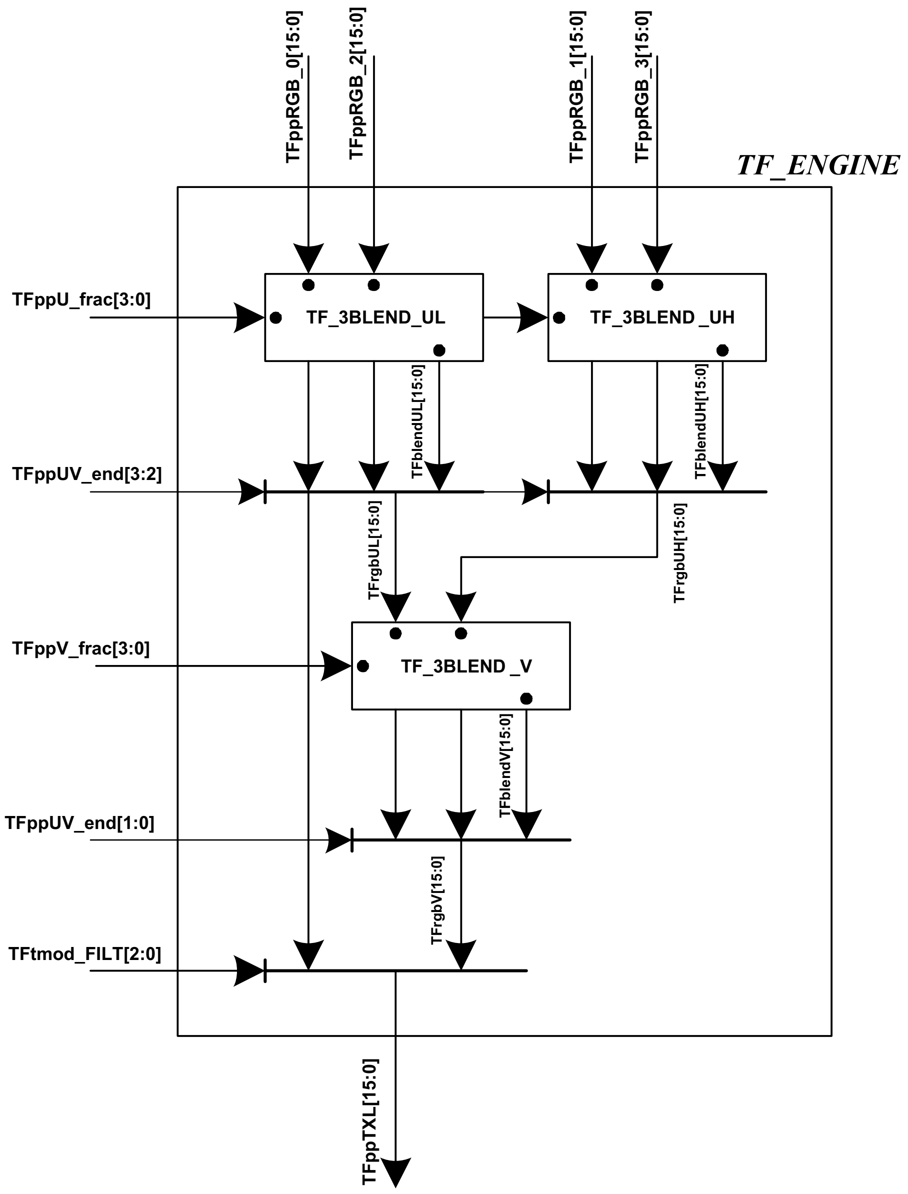


Figure 7.34 Datapath #1: TF_ENGINE

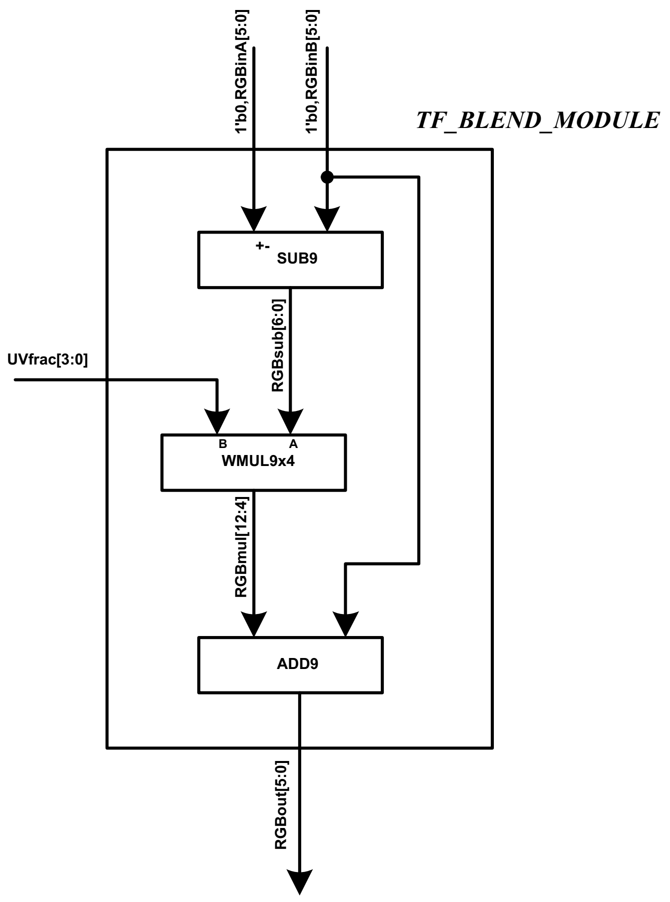


Figure 7.35 Datapath #2: TF blend module

```
module Wtf(  REclk, memclk1, PBnwait, TP3reset, TP3nhld, TP3cOP, TP3cFB, TP3cDF,
           TP3cTMF, TP3cAF, TP3cTEXEN, TP3cFMB, TP3cSCR,
           TP3_ZB0cmd, TP3_ZB1cmd, TP3_ZB0addr, TP3_ZB1addr, TP3ppMASK,
           TP3tmod_FILT, TP3tmod, TP3dADDR, TP3dA,
           TP3pp0VALID, TP3pp0RGB, TP3pp0U_frac, TP3pp0V_frac, TP3pp0UV_end,
           TP3pp0RGB_0, TP3pp0RGB_1, TP3pp0RGB_2, TP3pp0RGB_3,
           TP3pp1VALID, TP3pp1RGB, TP3pp1U_frac, TP3pp1V_frac, TP3pp1UV_end,
           TP3pp1RGB_0, TP3pp1RGB_1, TP3pp1RGB_2, TP3pp1RGB_3,
           TFnwait, TFreset, TFnhld, TFcOP, TFcFB, TFcDF,
           TFcTMF, TFcAF, TFcTEXEN, TFcSCR, TFppMASK, TFtmod, TFdA,
           TFpp0VALID, TFpp0PXL, TFpp0TXL, TFpp1VALID, TFpp1PXL, TFpp1TXL,
           FB_nREQ, FBI_FB0addr, FBI_FB1addr, FBI_FB0cmd, FBI_FB1cmd,
           FBI_FBnWAIT);
```

```
     /* Renaming */
     assign #1  TFpp0VALID = TFppMASK[1];
     assign #1  TFpp1VALID = TFppMASK[0];

     /*Module Interconnection*/
```

```
//Main controller
Wtf_ctrl     sstf_ctrl   ();
```

```
 //Pipeline controller
 module    Wtf_ctrl(  REclk, PBnwait, TP3reset, TP3nhld, TP3cOP, TP3cFB, TP3cDF,
           TP3cTMF, TP3cAF, TP3cTEXEN, TP3cFMB, TP3cSCR,
           TP3_ZB0cmd, TP3_ZB1cmd, TP3_ZB0addr, TP3_ZB1addr, TP3ppMASK,
           FBI_FBnWAIT, TFnwait, TFreset, TFnhld, TFcOP, TFcFB, TFcDF,
           TFcTMF, TFcAF, TFcTEXEN, TFcSCR,
           TF_ZB0cmd, TF_ZB1cmd, TF_ZB0addr, TF_ZB1addr, TFppMASK,
           TFfetch_CLK, TFfetch_TMOD, TFfetch_FADDR, TFfetch_ALPHA,
           TFfetch_PXL, TFfetch_TXL);

      /* Combinational logics */

      //nwait
      assign  TFnwait = (PBnwait&FBI_FBnWAIT) | ~TP3reset;
      //nhld
      assign  TFnhld = TFnhld_int;
      //Fetch signals
      assign #1  TFfetch_CLK = TFnwait & TP3nhld;
      assign  TFfetch_TMOD = TP3cTMF;
      assign  TFfetch_FADDR = TP3cAF & TP3cFMB;
      assign  TFfetch_ALPHA = TP3cDF;
      assign  TFfetch_PXL = TP3cDF;
      assign  TFfetch_TXL = TP3cDF & TP3cTEXEN;

      always @(posedge REclk)
      begin
           if(TFnwait)
           begin
                TFnhld_int <= TP3nhld;
                TFreset <= TP3reset;
                //Fetch signals
                TFcDF_int <= TP3cDF;
                TFcTMF_int <= TP3cTMF;
                TFcAF_int <= TP3cAF;
                TFcTEXEN_int <= TP3cTEXEN;
                //Trivial signals
                TFcOP <= TP3cOP;
                TFcFB <= TP3cFB;
                TFcSCR <= TP3cSCR;
                TF_ZB0cmd <= TP3_ZB0cmd;
                TF_ZB1cmd <= TP3_ZB1cmd;
                TF_ZB0addr <= TP3_ZB0addr;
                TF_ZB1addr <= TP3_ZB1addr;
                TFppMASK <= TP3ppMASK;
           end
      end
```

```
     always @(negedge REclk)
     begin
          TFcDF <= TFcDF_int;
          TFcTMF <= TFcTMF_int;
          TFcAF <= TFcAF_int;
          TFcTEXEN <= TFcTEXEN_int;
     end
endmodule
```

//Data latches
Wtf_latch   sstf_latch();

```
//Data latches
module    Wtf_latch(  REclk, clk, TFfetch_TMOD, TFfetch_FADDR, TFfetch_ALPHA,
          TFfetch_PXL, TFfetch_TXL,
          TP3tmod_FILT, TP3tmod, TP3dADDR, TP3dA,
          TP3pp0VALID, TP3pp0RGB, TP3pp0U_frac, TP3pp0V_frac, TP3pp0UV_end,
          TP3pp0RGB_0, TP3pp0RGB_1, TP3pp0RGB_2, TP3pp0RGB_3,
          TP3pp1VALID, TP3pp1RGB, TP3pp1U_frac, TP3pp1V_frac, TP3pp1UV_end,
          TP3pp1RGB_0, TP3pp1RGB_1, TP3pp1RGB_2, TP3pp1RGB_3,
          TFtmod_FILT, TFtmod, TFdA, FBI_FBBADDR,
          TFpp0PXL, TFpp0U_frac, TFpp0V_frac, TFpp0UV_end,
          TFpp0RGB_0, TFpp0RGB_1, TFpp0RGB_2, TFpp0RGB_3,
          TFpp1PXL, TFpp1U_frac, TFpp1V_frac, TFpp1UV_end,
          TFpp1RGB_0, TFpp1RGB_1, TFpp1RGB_2, TFpp1RGB_3);

     /*     Latching     */

     //TMOD
     always @(posedge REclk)
     begin
          if(clk & TFfetch_TMOD)
          begin
               TFtmod_FILT <= TP3tmod_FILT;
               TFtmod <= TP3tmod;
          end
     end
     //ADDR
     always @(posedge REclk)
     begin
          if(clk & TFfetch_FADDR) FBI_FBBADDR <= TP3dADDR;
     end
     //ALPHA
     always @(posedge REclk)
     begin
          if(clk & TFfetch_ALPHA) TFdA <= TP3dA;
     end
     //PPO PXL
     always @(posedge REclk)
     begin
          if(clk & TP3pp0VALID & TFfetch_PXL) TFpp0PXL <= TP3pp0RGB;
     end
     //PP0 TXL
     always @(posedge REclk)
     begin
          if(clk & TP3pp0VALID & TFfetch_TXL)
          begin
               TFpp0U_frac <= TP3pp0U_frac;
               TFpp0V_frac <= TP3pp0V_frac;
               TFpp0UV_end <= TP3pp0UV_end;
               TFpp0RGB_0 <= TP3pp0RGB_0;
               TFpp0RGB_1 <= TP3pp0RGB_1;
               TFpp0RGB_2 <= TP3pp0RGB_2;
               TFpp0RGB_3 <= TP3pp0RGB_3;
          end
     end
     //PP1 PXL
     always @(posedge REclk)
     begin
          if(clk & TP3pp1VALID & TFfetch_PXL) TFpp1PXL <= TP3pp1RGB;
     end
     //PP1 TXL
     always @(posedge REclk)
     begin
          if(clk & TP3pp1VALID & TFfetch_TXL)
          begin
               TFpp1U_frac <= TP3pp1U_frac;
               TFpp1V_frac <= TP3pp1V_frac;
               TFpp1UV_end <= TP3pp1UV_end;
               TFpp1RGB_0 <= TP3pp1RGB_0;
               TFpp1RGB_1 <= TP3pp1RGB_1;
               TFpp1RGB_2 <= TP3pp1RGB_2;
               TFpp1RGB_3 <= TP3pp1RGB_3;
          end
     end
endmodule
```

//Frame buffer interface
Wtf_fbi   sstf_fbi();

```
//Frame buffer interface
module    Wtf_fbi(  REclk, memclk1, TFnhld, TFcOP, TFcFB, TF_ZB0cmd, TF_ZB1cmd,
          TF_ZB0addr, TF_ZB1addr, TFppMASK, FBI_FBBADDR,
          FBI_FBnREQ, FBI_FB0cmd, FBI_FB1cmd, FBI_FB0addr, FBI_FB1addr);

     /* Request generation */
     assign  FBI_FBnREQ_int = ~((|FBI_FB0cmd_int) | (|FBI_FB1cmd_int));

     /* Address generation */
     DW01_add    #(18)    addr0(    .A      (FBI_FBBADDR),
                                    .B      ({1'b0, TF_ZB0addr, 1'b0}),
                                    .CI      (1'b0),
                                    .SUM      (FBI_FB0addr_int),
                                    .CO      (FB0_cout));
     DW01_add    #(18)    addr1(    .A      (FBI_FBBADDR),
                                    .B      ({1'b0, TF_ZB1addr, 1'b1}),
                                    .CI      (1'b0),
                                   .SUM      (FBI_FB1addr_int),
                                   .CO      (FB1_cout));

      /* Command generation */
      always @(TFcOP or TFnhld or TF_ZB0cmd or TF_ZB1cmd or TFppMASK or TFcFB)
      begin
           casex(TFcOP)
              'WCTRL_OP_RSHA : begin //RDAT
              FBI_FB0cmd_int<= (TFppMASK[1]&TFnhld) ? TF_ZB0cmd : 'WFZB_CMD_NOP;
              FBI_FB1cmd_int<= (TFppMASK[0]&TFnhld) ? TF_ZB1cmd : 'WFZB_CMD_NOP;
              end
              'WCTRL_OP_RCLR : begin //RCLR
              FBI_FB0cmd_int <= (TFcFB&TFnhld) ? TF_ZB0cmd : 'WFZB_CMD_NOP;
              FBI_FB1cmd_int <= (TFcFB&TFnhld) ? TF_ZB1cmd : 'WFZB_CMD_NOP;
              end
              default : begin //Other commands
              FBI_FB0cmd_int <= 'WFZB_CMD_NOP;
              FBI_FB1cmd_int <= 'WFZB_CMD_NOP; end
           endcase
      end

      // Command and address fetch @ negative Clk
      always @(negedge memclk1)
      begin
           FBI_FBnREQ <= FBI_FBnREQ_int;
           FBI_FB0cmd <= FBI_FB0cmd_int;
           FBI_FB1cmd <= FBI_FB1cmd_int;
           FBI_FB0addr <= FBI_FB0addr_int;
           FBI_FB1addr <= FBI_FB1addr_int;
      end
endmodule
```

//Pixel processor engine #0
Wtf_engine   sstfe_0();
//Pixel processor engine #1
Wtf_engine   sstfe_1();

```
/* Blend unit module */
module     Wtf_blend_module(    UVfrac, RGBinA, RGBinB, RGBout);

     //B-A: sign extension
     DW01_sub    #(9)     sub9(  .A           ({1'b0,RGBinB}),
                                 .B           ({1'b0,RGBinA}),
                                 .CI          (1'b0),
                                 .DIFF        (RGBsub),
                                 .CO          (scout));
     //(F*(B-A)) << 4
     WMUL9x4              mul9x4(  .x          (RGBsub),
                                 .y          (UVfrac),
                                 .z          (RGBmul));
     //(F*(A-B) + A) = B*F + A*(1-F)
     DW01_add    #(9)     add9(  .A           (RGBmul[12:4]),
                                 .B           ({1'b0,RGBinA}),
                                 .CI          (1'b0),
                                .SUM         (RGBsum),
                                .CO          (cout));
     //Output renaming
     assign #1  RGBout = RGBsum[7:0];
endmodule

/* 3-way SIMD blend unit */
module     Wtf_3blend_unit(     UVfrac, RGBinA, RGBinB, RGBout);

     /* Module interconnection */
     Wtf_blend_module     sstf_blend_R(
                          .UVfrac          (UVfrac),
                          .RGBinA          (RGBinA[23:16]),
                          .RGBinB          (RGBinB[23:16]),
                          .RGBout          (RGBout[23:16]));
     Wtf_blend_module     sstf_blend_G(
                          .UVfrac          (UVfrac),
                          .RGBinA          (RGBinA[15:8]),
                          .RGBinB          (RGBinB[15:8]),
                          .RGBout          (RGBout[15:8]));
     Wtf_blend_module     sstf_blend_B(
                          .UVfrac          (UVfrac),
                          .RGBinA          (RGBinA[7:0]),
                          .RGBinB          (RGBinB[7:0]),
                          .RGBout          (RGBout[7:0]));
endmodule

/* Pixel processor engine */
module     Wtf_engine(TFppRGB_0, TFppRGB_1, TFppRGB_2, TFppRGB_3,
          TFppU_frac, TFppV_frac, TFppUV_end, TFtmod_FILT, TFppTXL);

     /* Module interconnection */
     Wtf_3blend_unit      sstf_3blend_ul();
     Wtf_3blend_unit      sstf_3blend_uh();
     Wtf_3blend_unit      sstf_3blend_v();

     /* Mux */
     //TF_MUX_UL
     always @(TFppUV_end or TFppRGB_0 or TFppRGB_2 or TFblendUL)
     begin
          casex(TFppUV_end[3:2])
               2'd0 : TFrgbUL <= TFblendUL;      //No end point
               2'd1 : TFrgbUL <= TFppRGB_2;      //Equals to zero
               2'd2 : TFrgbUL <= TFppRGB_0;      //Equals to one
               default : TFrgbUL <= 24'b0;
          endcase
     end
     //TF_MUX_UH
     always @(TFppUV_end or TFppRGB_1 or TFppRGB_3 or TFblendUH)
     begin
          casex(TFppUV_end[3:2])
               2'd0 : TFrgbUH <= TFblendUH;      //No end point
               2'd1 : TFrgbUH <= TFppRGB_3;      //Equals to zero
               2'd2 : TFrgbUH <= TFppRGB_1;      //Equals to one
               default : TFrgbUH <= 24'b0;
          endcase
     end
     //TF_MUX_V
     always @(TFppUV_end or TFrgbUL or TFrgbUH or TFblendV)
     begin
          casex(TFppUV_end[1:0])
               2'd0 : TFrgbV <= TFblendV;      //No end point
               2'd1 : TFrgbV <= TFrgbUH;      //Equals to zero
               2'd2 : TFrgbV <= TFrgbUL;      //Equals to one
               default : TFrgbV <= 24'b0;
          endcase
     end
     //TF_MUX_FILTER_MODE
     always @(TFtmod_FILT or TFppRGB_0 or TFrgbV)
     begin
          casex(TFtmod_FILT)
               3'bXX1 : TFppTXL <= TFppRGB_0;      //Point sampling
               3'bX1X : TFppTXL <= TFrgbV;           //Bilinear filtering
               default : TFppTXL <= 24'b0;
          endcase
     end
endmodule
```

endmodule

7.5.4.14 PB: Pixel Blending

This is the last stage of the rasterizer. In this stage, the pixels and texels are blended and the pixel alpha blending is performed using the previous frame data (Figure 7.36).

Pipe Inputs

- Pipe controls
- Common signals
- PP valid signal
- Pixel RGB
- Texel RGB

Pipe Outputs

- None: the last pipeline stage

Memory Inputs/Outputs

- FRAME read data (in)
- FRAME write data (out)
- Mask signal (out)

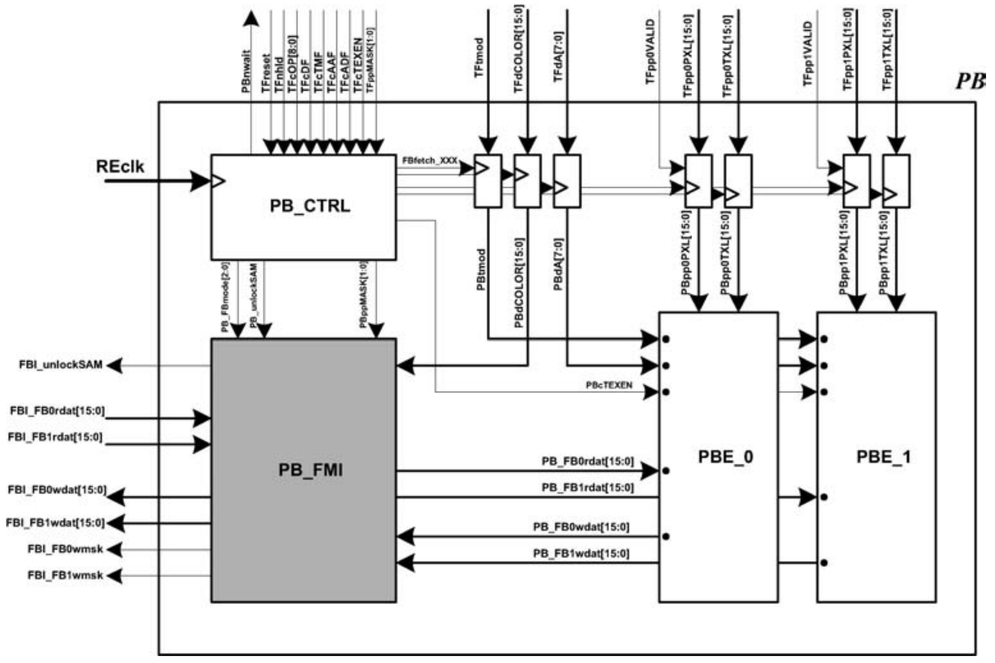


Figure 7.36 Pixel blending

Functions

- Pixel color blending with textured color
- Alpha blending

Datapath #1: PB_ENGINE
See Figure 7.37.

Datapath #2: PB_ALPHA_MODULE
See Figure 7.38.

RTL Code

```
/*
* RAMP-GR
* RAMP-GR PB Module: Pixel Blending
* by Jeong-Ho Woo (denber@eeinfo.kaist.ac.kr)
* Semiconductor System Laboratory, KAIST
* All rights reserved
*/

module Wpb(  REclk, PBnwait, TFreset, TFnhld, TFcOP, TFcDF,
             TFcTMF,TFcAF, TFcTEXEN, TFcSCR, TFppMASK, TFtmod, TFdA, PBcOP,
```

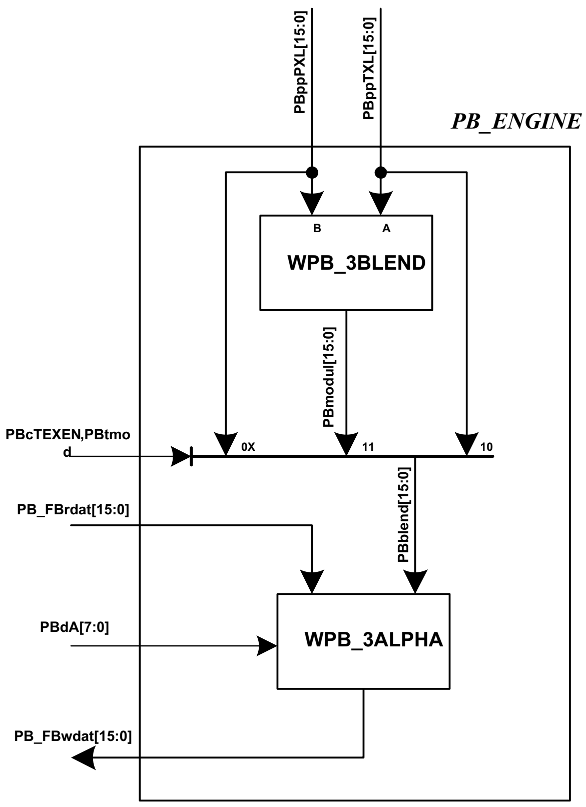


Figure 7.37 Datapath #1: PB_ENGINE

```
            TFpp0VALID, TFpp0PXL, TFpp0TXL, TFpp1VALID, TFpp1PXL, TFpp1TXL,
            RDON_clr, FBI_FB0rdat, FBI_FB1rdat, FBI_FB0wdat, FBI_FB1wdat,
            FBI_FB0wmsk, FBI_FB1wmsk, RDON);

     /* Module interconnection */
```

```
//Main controller
Wpb_ctrl      sspb_ctrl();
```

```
//Main controller
module     Wpb_ctrl(    REclk, PBnwait, TFreset, TFnhld, TFcOP, TFcDF,
           TFcTMF, TFcAF, TFcTEXEN, TFcSCR, TFppMASK, RDON_clr,
           PBcOP, PB_FBmode, PBppMASK, PBcSCR,
           PBfetch_CLK, PBfetch_TMOD, PBfetch_ALPHA,
           PBfetch_PXL, PBfetch_TXL, PBcTEXEN, RDON);

     /* Combinational logics */

     //nwait
     assign  PBnwait = 1'b1;      // End of Pipeline
     //nhld
```

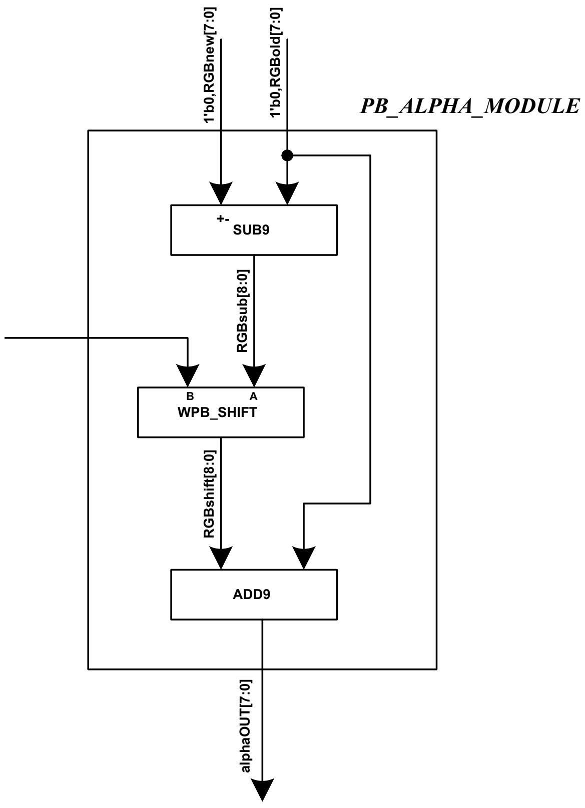


Figure 7.38 Datapath #2: PB_ALPHA_MODULE

```
      assign  PBnhld = PBnhld_int;
      //Fetch signals
      assign #1  PBfetch_CLK = PBnwait & TFnhld;
      assign  PBfetch_TMOD = TFcTMF;
      assign  PBfetch_ALPHA = TFcDF;
      assign  PBfetch_PXL = TFcDF;
      assign  PBfetch_TXL = TFcDF & TFcTEXEN;
      //Texture blending control
      assign #1  PBcTEXEN = PBcTEXEN_int;

      always @(posedge REclk)
      begin
           if(PBnwait)
           begin
                //State controls
                PBnhld_int <= TFnhld;
                PBreset <= TFreset;
                //Fetch signals
                PBcDF_int <= TFcDF;
                PBcTMF_int <= TFcTMF;
                PBcAF_int <= TFcAF;
                PBcTEXEN_int <= TFcTEXEN;
                //Trivial signals
                PBcOP <= TFcOP;
                PBcSCR <= TFcSCR;
                //PP control signals
                PBppMASK <= TFppMASK;
            end
        end

        /* Frame buffer interface signal generation */

        // This signal is overridden by FBcmd
        always @(PBcOP or PBnhld)
        begin
            if(PBnhld & PBcOP[0])           PB_FBmode <= 'WPI_ZBI_ZERO;
            else if(PBnhld & PBcOP[5])      PB_FBmode <= 'WPI_ZBI_BYPASS;
            else                            PB_FBmode <= 'WPI_ZBI_BYPASS;
        end
        always @(PBcOP or RDON_clr)
        begin
            if((PBcOP == 'WCTRL_OP_RDON)  (RDON_clr == 1'b0)) RDON <= 1'b1;
            else  RDON <= 1'b0;
        end
    endmodule
```

```
//Data latches
Wpb_latch     sspb_latch();
```

```
//Data latches
module    Wpb_latch(    REclk, clk, PBfetch_TMOD, PBfetch_ALPHA, PBfetch_PXL,
          PBfetch_TXL, TFtmod, TFdA, TFpp0VALID, TFpp0PXL, TFpp0TXL,
          TFpp1VALID, TFpp1PXL, TFpp1TXL, PBtmod, PBdA,
          PBpp0PXL, PBpp0TXL, PBpp1PXL, PBpp1TXL);

     /* Latches */

     //PB_LATCH_TMOD
     always @(posedge REclk)
     begin
          if(clk&PBfetch_TMOD) PBtmod <= TFtmod;
     end
     //PB_LATCH_ALPHA
     always @(posedge REclk)
     begin
          if(clk&PBfetch_ALPHA) PBdA <= TFdA;
     end
     //PB_LATCH_PP0_PXL
     always @(posedge REclk)
     begin
          if(clk & TFpp0VALID&PBfetch_PXL) PBpp0PXL <= TFpp0PXL;
     end
     //PB_LATCH_PP0_TXL
```

```
      always @(posedge REclk)
      begin
           if(clk & TFpp0VALID&PBfetch_TXL) PBpp0TXL <= TFpp0TXL;
      end
      //PB_LATCH_PP1_PXL
      always @(posedge REclk)
      begin
           if(clk & TFpp1VALID&PBfetch_PXL) PBpp1PXL <= TFpp1PXL;
      end
      //PB_LATCH_PP1_TXL
      always @(posedge REclk)
      begin
           if(clk & TFpp1VALID&PBfetch_TXL) PBpp1TXL <= TFpp1TXL;
      end
endmodule
```

```
//PP #0
Wpb_engine    sspbe_0();
//PP #1
Wpb_engine    sspbe_1();
```

```
/* Alpha blend shift module */
module     Wpb_alpha_shift(shiftIN, shiftVAL, shiftOUT);

      /* Inputs */
      input [8:0]       shiftIN;
      input [7:0]       shiftVAL;

      /* Outputs */
      output [8:0]     shiftOUT;
      wire [16:0]      shift_OUT;
      WMUL9x8          alpha(    .x      (shiftIN),
                                 .y      ({1'b0,shiftVAL[6:0]}),
                                 .z      (shift_OUT));

      assign  shiftOUT = shift_OUT[15:7];
endmodule

/* Alpha blend module */
module     Wpb_alpha_module(RGBnew, RGBold, PBdA, alphaOUT);

      //NEW-OLD: sign extension
      DW01_sub    #(9)    sub9(    .A      ({1'b0,RGBnew}),
                                   .B      ({1'b0,RGBold}),
                                   .CI     (1'b0),
                                   .DIFF   (RGBsub),
                                   .CO     (scout));
      //(NEW-OLD)*ALPHA: shift with sign extension
      Wpb_alpha_shift      sspb_alpha_shift();
      // (NEW-OLD)*ALPHA + OLD = NEW*ALPHA + OLD*(1-ALPHA)
      DW01_add    #(9)    add9(    .A      ({1'b0,RGBold}),
                                   .B      (RGBshift),
                                   .CI     (1'b0),
                                   .SUM    (RGBsum),
                                   .CO     (cout));
      // Output Select
      assign #1  alphaOUT = (PBdA[7]) ? RGBnew : RGBsum[7:0];
endmodule

/* 3-way SIMD alpha blend logic
module     Wpb_3alpha(PB_FBrdat, PBblend, PBdA, PB_FBwdat);

      /* Inputs */
      input [15:0]      PB_FBrdat;
      input [23:0]      PBblend;
      input [7:0]       PBdA;

      /* Outputs */
      output [23:0]     PB_FBwdat;

      /* Module interconnection */
      Wpb_alpha_module         alpha_R();
      Wpb_alpha_module         alpha_G();
      Wpb_alpha_module         alpha_B();
endmodule

/* 3-way SIMD texture blend logic */
module     Wpb_3blend(PBppPXL, PBppTXL, PBmodul);

      /* Module interconnection */
      //TXL * PXL
      WMUL9x8    mul_R(    .x      ({1'b0,PBppTXL[23:16]}),
                           .y      (PBppPXL[23:16]),
                           .z      (mulR));
      WMUL9x8    mul_G(    .x      ({1'b0,PBppTXL[15:8]}),
                           .y      (PBppPXL[15:8]),
                           .z      (mulG));
      WMUL9x8    mul_B(    .x      ({1'b0,PBppTXL[7:0]}),
                           .y      (PBppPXL[7:0]),
                           .z      (mulB));

      /* Wire connection */
      //(TXL * PXL) / 255
      assign #1  PBmodul = {mulR[15:8],mulG[15:8],mulB[15:8]};
endmodule

/* Pixel processor engine
module     Wpb_engine(PBtmod, PBdA, PBcTEXEN, PBppPXL,
                PBppTXL, PB_FBrdat, PB_FBwdat);

      /* Module interconnection */
      //3-way SIMD texture blend logic
      Wpb_3blend  sspb_3blend(
                     .PBppPXL     (PBppPXL),
                     .PBppTXL     (PBppTXL),
                     .PBmodul     (PBmodul));
```

```
      // 3-Way SIMD Alpha Blend Logic
      Wpb_3alpha  sspb_3alpha(
                     .PB_FBrdat      (PB_FBrdat),
                     .PBblend       (PBblend),
                     .PBdA           (PBdA),
                     .PB_FBwdat      (PB_FBwdat));


      /* Mux */
      always @(PBcTEXEN or PBtmod or PBmodul or PBppPXL or PBppTXL)
      begin
           casex({PBcTEXEN,PBtmod})
                2'b0X : PBblend <= PBppPXL; //No texture
                2'b10 : PBblend <= PBppTXL; //Decal
                2'b11 : PBblend <= PBmodul; //Modulate
                default : PBblend <= 24'bX;
           endcase
      end
 endmodule
```

//Frame buffer interface
Wpb_fbi   sspb_fbi();

```
 //Frame buffer interface
 module     Wpb_fbi(    PB_FBmode, PBppMASK, PB_FB0rdat, PB_FB1rdat,
            PB_FB0wdat, PB_FB1wdat, FBI_FB0rdat, FBI_FB1rdat,
            FBI_FB0wdat,FBI_FB1wdat,FBI_FB0wmsk,FBI_FB1wmsk);

      /* Logics */
      //FB -> PB
      assign  PB_FB0rdat = FBI_FB0rdat;
      assign  PB_FB1rdat = FBI_FB1rdat;
      //PB -> FB
      always @(PB_FBmode or PBppMASK or PB_FB0wdat or PB_FB1wdat)
      begin
           casex(PB_FBmode)
           `WPI_ZBI_ZERO :
           begin
                 FBI_FB0wdat <= 24'b0;
                FBI_FB1wdat <= 24'b0;
                FBI_FB0wmsk <= 1'b1;
                FBI_FB1wmsk <= 1'b1;
           end
           `WPI_ZBI_BYPASS :
           begin
                 FBI_FB0wdat <= {PB_FB0wdat[23:19], PB_FB0wdat[15:10],
 PB_FB0wdat[7:3]};
                 FBI_FB1wdat <= {PB_FB1wdat[23:19], PB_FB1wdat[15:10],
 PB_FB1wdat[7:3]};
                 FBI_FB0wmsk <= PBppMASK[1];
                 FBI_FB1wmsk <= PBppMASK[0];
           end
           default :
```

```
           begin
                FBI_FB0wdat <= 16'bX;
                FBI_FB1wdat <= 16'bX;
                FBI_FB0wmsk<= 1'b1;
                FBI_FB1wmsk <= 1'b1;
           end
           endcase
      end
 endmodule
```

```
endmodule
```

8

The Future of Mobile 3D Graphics

8.1 Game and Mapping Applications Involving Networking

Key applications of mobile 3D graphics in the future will be games and mapping, just as these are already very popular in typical PCs and game consoles. Users have experienced the rich graphics scenes of many fantastic 3D games, so it is easy to imagine the demand for comparable 3D graphics quality in mobile devices, despite their small screen size. Early mobile 3D games used only a software game engine with limited graphics capabilities. Users were satisfied with just running a 3D game on a small screen. However, recently many hardware architectures have been proposed and commercialized for mobile 3D graphics applications. Today's graphics power of a premium mobile device is comparable to that of PCs and game consoles of a few years ago. By adopting HW acceleration in mobile 3D graphics, we can now extend the experience of PCs and game consoles to mobile devices.

In mapping applications such as car navigation devices, the transition from simple 2D bitmaps to complete 3D modeling is already occurring. The modeling of geographical information using 3D graphics is very useful. Roads and buildings can be located in x, y, and z coordinates. Users can see the whole of the Earth through the Google Map application, which uses texture mapping to show details of land and oceans.

These popular applications will be enhanced with the help of networking capabilities. Many people are already connected through the Internet and by means of mobile cellphones and wireless networks, so now the position of mobile 3D graphics cannot be considered separately from the connected environment. While the use of 3D modeling will initially reduce the capacity to represent quantities of information, it will encourage people to become thirsty for even more information. Network service providers such as mobile communications carriers will have to cater for more people wanting to use more network capacity. They will provide an online games service and other applications through the Internet. The challenge for mobile 3D graphics is to provide ever more efficient tools to represent these vast quantities of information.

In this book we have introduced the OpenGL-ES 2.0 library and its features. These can provide more graphics functions with lower power consumption. Moreover, current mobile devices incorporate many semiconductor chips that integrate hardware accelerators compatible with OpenGL-ES 2.0. PC graphics have already adopted the benefits of programmability in graphics architectures such as Direct3D and OpenGL 2.0, but the introduction of OpenGL-ES 2.0 opens the way for users to experience the quality of PC graphics on a small screen. This evolution drives the development of more efficient graphics accelerators. Currently, some newly developed mobile 3D graphics make use of multiple cores of very powerful unified shaders in a single graphics system. We can look forward to seeing many mobile devices using OpenGL-ES 2.0 or more advanced graphics architectures.

8.2 Moves Towards More User-centered Applications

Research on the use of highly featured mobile handsets shows that they are increasingly used to support daily information management, such as e-mail Web browsing. Figure 8.1 shows an analysis of Apple iPhone usage which was presented by Rubion Consulting in 2008 [1]. The graph shows that about 72% of users did e-mail reading at least once a day. Web browsing and calendar checking were performed more than

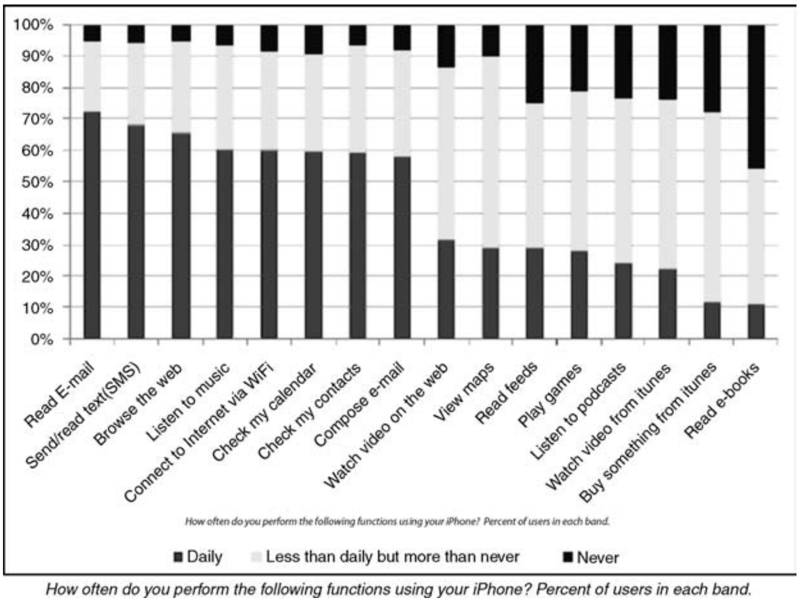


Figure 8.1 Apple iPhone usage

gaming. Although the iPhone does not have the most powerful mobile 3D hardware such as OpenGL-ES 2.0 full shaders, this research indicates that user experience and convenience in their daily lives should be the key drivers in developing new high-quality mobile gadgets.

Therefore, mobile 3D graphics applications – such as in Web browsers – should be designed to be used intuitively. Especially after the release of the iPhone, the trend is to get around the small fixed screen by adopting sensors such as a touch interface. Dynamic changes and zooming of scenes and UI elements such as icons and pictures are easily implemented by mobile 3D graphics. OpenGL-ES 2.0 will add a very esthetic feeling to these elements, as if we are touching and navigating in the more sensitive materials, not just plastic phone cases. Moreover, the quantity of information that can be accessed at the same time will also be increased because this information can be represented in three-dimensional spaces, and can be popped up in the front from the background by user inputs.

The mobile's display is much smaller than a PC screen which we use for Web browsing in most cases. That means we need to draw a Web page on the small screen by zooming out, and see the parts we want by zooming in. This interactive action should be carried out swiftly, and mobile 3D graphics is the right solution. Also we can use ideas from mobile 3D graphics to enhance the rendering speed of a Web page. One example is to store a full Web page in texture images and apply texture mapping in zooming operations. Since Web pages nowadays contain not only text but also multimedia content such as images and videos, the composition of multiple sources will be necessary.

Mobile 3D graphics will be developed for more user-centered applications, driven by the UI and Web browsing applications. The combination of UI and Web browsing will lead us to integrate all the information we want into single, simple and comprehensive interfaces (Figure 8.2). Mobile 3D graphics will be the canvas we are drawing on. The content of existing game and mapping applications will also be ported and integrated into more mobile devices.

8.3 Final Remarks

The programmability introduced into mobile 3D graphics is gradually removing the distinction between PC and mobile experiences in terms of quantity and quality of content. This crossover is apparent in the development of both hardware and software libraries. In fact, in terms of programmability, the architectural difference between mobile graphics hardware and PC hardware is not particularly high because the basic algorithms of coordinate transformation and color processing are not very different. The difference relates to the level and number of hardware integrations. Moreover, the standard of a mobile 3D graphics library is approaching that of a PC graphics library. We think that this trend is the result of the applications enlargement described in the

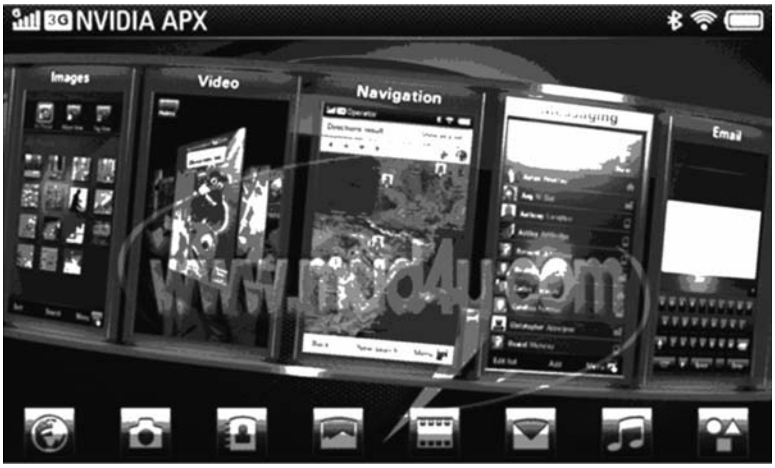


Figure 8.2 Examples of user interfaces from nVidia. © 2009 NVIDIA Corporation. NVIDIA, The NVIDIA logo, SC10 and Tegra are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. All rights reserved

previous section. Now, very recursively, that enlargement will accelerate the programmability and innovations of mobile 3D graphics standards and architecture.

As more functions and scenarios are integrated, the refinement of architecture and physical considerations such as memory management and graphics object compositions will be researched more and more. The modeling and platform approach with a systems engineering concept, for example, will be accompanied by enhancing the programmable architecture.

Reference

1 The Apple iPhone: Successes and Challenges for the Mobile Industry, Rubicon Consulting Inc. (March 2008).

Appendix

Verilog HDL Design

This appendix introduces the hardware description language, *Verilog*. It describes how to use Verilog and create your own hardware models. After a short introduction, the design flow and the level of designs using Verilog are explained. Then the basic Verilog syntax is explained with basic examples. We include RTL code and simulation environments, so that users can simulate the rendering processor on their PC or workstation environment. In addition, the synthesis scripts are also included.

A.1 Introduction to Verilog Design

Verilog is a *hardware description language* (HDL), which is a language to describe a digital system such as microprocessor, application-specific unit, memory, or a simple flip–flop. Anyone can describe any hardware at any level of detail by using a hardware description language.

Verilog was developed initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is very similar to traditional computer language such as C. At that time, Verilog was not standardized and the language became modified in almost all the revisions that came out between 1984 and 1990. A Verilog simulator was first used in early 1985 and was extended substantially through to 1987. In 1990, Cadence Design Systems decided to acquire Gateway, so Cadence became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys was marketing the top-down design methodology, using Verilog. This was the starting point of powerful digital system design. In 1990, to survive in a tough industry that used VHDL, Cadence decided to make Verilog an "open" language. Cadence organized the Open Verilog International (OVI) in 1991 and, in the meantime, the popularity of Verilog was rising exponentially.

The need for a universal standard was recognized at this time. The directors of OVI asked the IEEE to form a working committee to establish Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993, and finally in 1995 the standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed as IEEE standard 1364(1995).

A.2 Design Level

Verilog allows a digital system to be designed at a wide range of abstractions – at behavior level, at register transfer level (RTL), at gate level, at switch level, and so on. Among these, behavior level, RTL level, and gate level design are frequently used for digital system design.

A.2.1 Behavior Level

The behavior level is used to describe a system intuitively. Therefore, this level is frequently used to verify algorithms or system behavior like a high-level language. This level is not focused on synthesizability or structural realization of the design.

A.2.2 Register Transfer Level

After high-level verification at the behavior level, RTL code is used to specify the characteristics of a circuit or system by operations and the transfer of data between the registers. This level is focused on real implementation of the design, so the design should be carefully described according to synthesizable syntax or semantics. At this level the design contains explicit clocks, so RTL design contains exact timing bounds.

A.2.3 Gate Level

After synthesizing RTL code, gate-level code is generated. Within this level, the characteristics of a system are described by standard cell libraries and their logical links and timing properties. All signals are discrete signals. They can only have definite logical values (0, 1, X, Z). The usable operations are predefined logic primitives (gates AND, OR, NOT etc.). Gate-level code is generated by tools like synthesis tools, and this net list is used for gate-level simulation and for back-end design.

Design flow is one of the most difficult decisions. Like other HDLs, Verilog also allows both bottom-up and top-down design.

- **Bottom-up design** This is a type of traditional design flow. Each design is performed at the gate level using standard gates such as AND, OR, NOR, and NAND.

Designers decide which gates will be used and where those gates will be placed. With increasing design complexity, this approach is nearly impossible to maintain. Emerging large-scale systems consist of ASIC or microprocessors with a complexity of millions of transistors. These traditional bottom-up designs have to give way to new, structural, hierarchical design methods.

- **Top-down design** Designers can concretize their ideas from concept to silicon implementation. The major advantages of top-down design are early testing, easy change between different technologies, and a structured system design. Details of top-down design will be discussed below.

A.3 Design Flow

Figure A.1 shows a typical design flow of a digital system using Verilog. In short, the design flow can be divided into two steps: front-end and back-end. It consists of about nine or ten design steps, which will be described briefly.

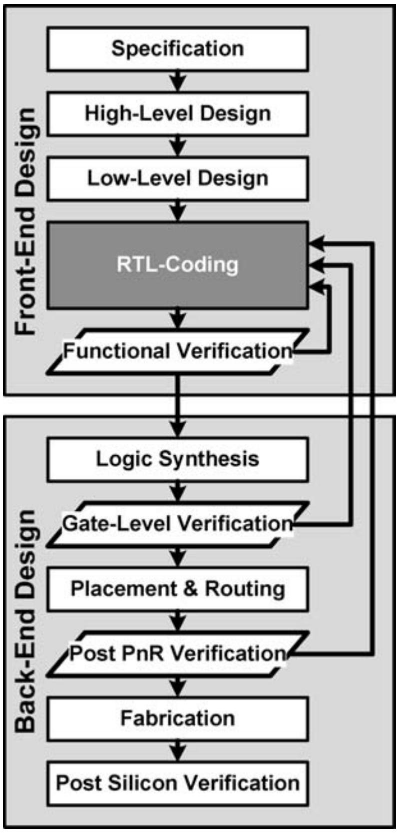


Figure A.1 Typical design flow

A.3.1 Specification

The designer decides what the target performance will be, what the important parameters of the design are, what the interfaces are, and so on (Figure A.2). For example, when designing a microprocessor, the designer should decide the target performance, the reset behavior, the interfaces with other blocks, and input/output format. Normally, the specification does not need a technical tool; designers often use documentation tools such as Microsoft Word, PowerPoint, Visio, and so on.

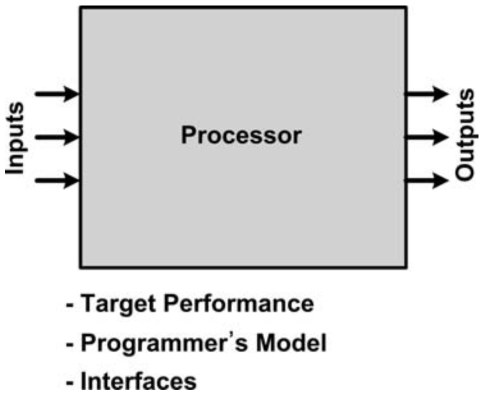


Figure A.2 Specification

A.3.2 High-level Design

After deciding the specification, the designer should describe the target system in high-level language (Figure A.3). Algorithm verification or block definitions are performed in this stage. To design a microprocessor, how to divide functional blocks and how to communicate between them should be decided. For algorithm verification or block definition, a high-level language such as C or C++ or Verilog behavior-level description is widely used in this stage.

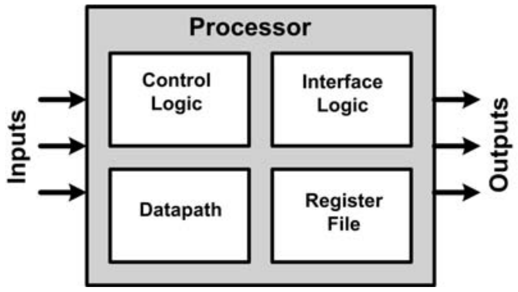


Figure A.3 High-level design

A.3.3 Low-level Design

The designer defines the details about how each block works. In this stage, you should define how to design a finite state machine, registers, and control units of each block (Figure A.4). Of course, the detailed timing diagrams of interfaces are defined with waveforms. It is strongly recommended that you make all items of the design clear in this phase.

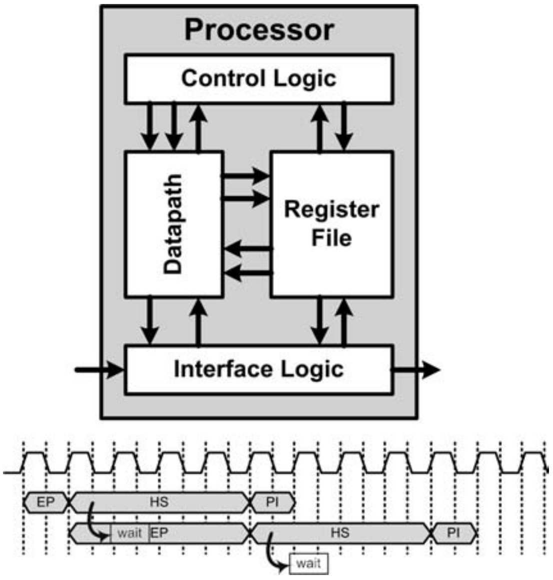


Figure A.4 Low-level design

A.3.4 RTL Coding

In RTL coding, the low-level design is implemented into a hardware description language such as Verilog or VHDL.

```
module processor( clk,      //Clock
                 reset     //Reset
                 inputs;
                 outputs);
      input clk;
      input reset;
      input inputs;
      output     outputs

      controller       proc_ctrl(clk, reset, signals)
      datapath         proc_dp(clk, reset, signals)
      registerfile     proc_rf(clk, reset, signals)
      interfaces       proc_if(clk, reset, signals)
endmodule
```

A.3.5 Simulation

The functionality and timing characteristics of a system should be verified at every step of its design (Figure A.5). A test bench is needed that generates input signals, dummy blocks that are behavioral models of neighboring blocks, and simulators. The initial simulation is done by using RTL codes for functional verification of blocks. After that, gate-level simulation is performed to verify timing bounds as well as functionalities. The final simulation is performed after placement and routing (PnR). After PnR, the design contains wire delays and parasitic resistances and capacitances, so verification is needed that the design meets the target specification.

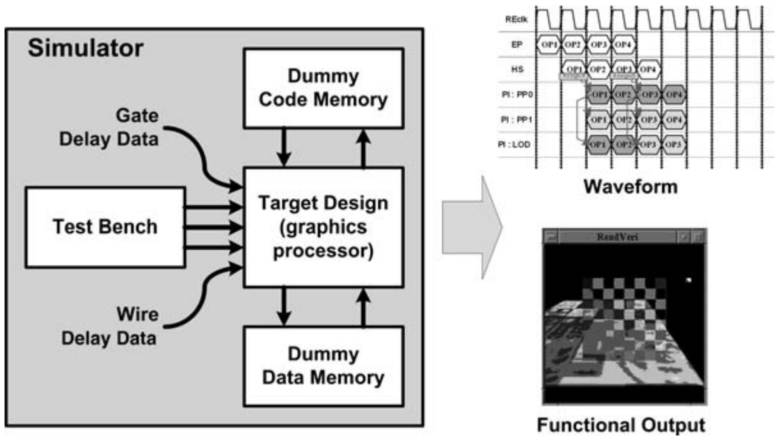


Figure A.5 Simulation

After the simulations, waveform outputs and functional outputs can be obtained, to check functionality and timing bounds.

A.3.6 Synthesis

In the synthesis stage, the RTL code is converted into a gate-level net list (Figure A.6). A synthesis tool such as a design compiler or Synplify takes the RTL code and technology library files as inputs and generates gate-level code that contains standard cells and link data. Basically, the synthesis tool reports on whether or not the timing information meets the timing requirements. However, it does not include wire delays, only gate propagation delays.

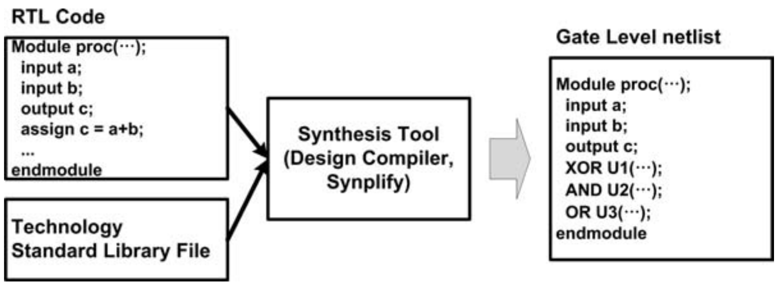


Figure A.6 Synthesis

A.3.7 Placement and Routing

In this stage, the gate-level netlists are converted into a physical layout (Figure A.7). All gates and flip–flops in a gate-level netlist are placed; and clock tree synthesis and global signals such as "reset" are routed. After that, local signals are routed. Then the final GDS file that will be delivered to the foundary, and final net list which includes wire delays as well as gate delays, are generated.

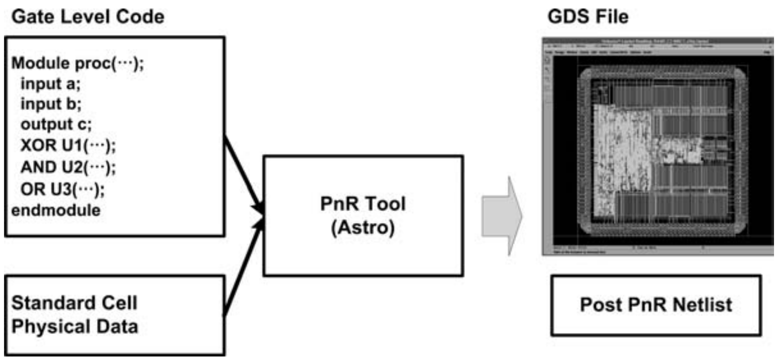


Figure A.7 Placement and routing

A.4 Verilog Syntax

The basic Verilog syntax will be described. The goal of this section is to show how to write Verilog, but it does not cover all the syntax. More information is available at, for example, www.verilog.com and www.verilogtutorial.info.

A.4.1 Modules

The basic unit of Verilog modeling is the *module*. As in the C language, you can use lower-case letters, capital letters, numbers, and underscores (_). Of course, it is case-sensitive. Below we show the basic configuration of a module.

Module Example

```
module  module_name (port-list);     //Declare all inputs and outputs
port declaration
     //Detailed declaration of inputs and outputs, including bitwidth
     //For example:
     //input  clk, res; → 1-bit input signals
     //input  [7:0] bus; → 8-bit input signal
     //output  done; → 1-bit output signal
     //inout  [3:0] dbus; → 4-bit inout signal
     //inout means bidirectional signal
net declaration
     //Declaration of internal net of the block
     //wire  enable;
     //wire [7:0] int_bus;
reg declaration
     //Register declaration
     //reg  enable;
     //reg [3:0] number;
     //You should declare all ports or values which are used in
     //initial, task, function, and always statements
parameter declaration
     //Parameter declaration
     //parameter WORD=32;
     //parameter BW=8;
     //reg [BW:0] temp_out; → example of usage
assign statement
     //Combinational logic declaration
function statement
always statement
Instantiation
     //Instantiate sub-modules
endmodule
     //Declaration of the end of the module
```

Initial Statement Example

"Initial" and "always" statements are *event-driven*. The details of these statements will be discussed later. Here only the configuration is discussed.

```
initial
     //Initial statement does not require an event list.
begin
     //If the initial includes more than one statement, 'begin-end' command
     clk = 0;   //Bind those statements into one command
     reg = 1;   //Use begin-end command with event-driven statements
end
     //Include more than one statement.
```

Always Statement Examples

```
always @ (posedge Clk)
      //Every positive edge of Clk signal drives this statement.
always @ (posedge Clk or negedge reset)
      //Every positive edge of Clk or negative edge or reset drives.
always @ (a or b or c)
      //When one of the conditional signals changes,
      //this statement is executed.
always @ (posedge Clk)
begin
      //An always statement also requires 'begin-end'.
      command_latch <= command_in; //Command
      data_latch <= data_in;
end
```

Examples of Bad Expressions

```
always @ (Clk)
      //This is a correct expression in terms of grammar.
      //However, the clock signal is edge-triggered.
      //Therefore this statement will incur a timing violation.
always @ (a or b or posedge Clk)
      //Do not mix level-triggered and edge-triggered events.
```

A.4.2 Logic Values and Numbers

Verilog supports four logic values:

0	Logic value 0
1	Logic value 1
X	Unknown signal means “don’t care”
Z	High-impedance value

To describe logic values or numbers in Verilog, follow the syntax:

```
<bit width>'<base><value>
```

Here, <bit width> declares the width of the data bit in decimal (the default is 32-bit); and <base> declares the base number of the data (default = decimal)

b, B: binary o,O: octal
d,D: decimal h,H: hexadecimal

The <value> declares the value of data.

When the <base> is decimal, you can declare data without <bit width> and <base>, and you can use underscore (_) for convenience.

Data	Bit width	Base	Binary digit
10	32	10	0000. . .01010
1’b1	1	2	1
8’hf1	8	16	11110001
4’bx	4	2	Xxxx
8’b0000_11xx	8	2	000011xx
‘hf_f	32	16	0000 011111111
4’d5	4	10	0101

A.4.3 Data Types

To declare signals or variables you have two choices, the “register” type or the “net” type. You should declare the type for all variables.

Register type	Declare using “reg”
	Used for latch or flip–flop
	It stores values until the next event occurs
Net type	Declare using “wire”
	Used for combinational logic or intermediate signals for links

When the data is used on the left side there is no limitation. When the data is used on the right side you should follow two rules:

- Use the “reg” type signals only in always, initial, task, and function statements.
- Use the “wire” type signals only in assign statements.

You can omit the wire declaration only when the “net” type signal is in the port list.

```
//Example 1
module  DFF(Clk, D, Q);
      input  Clk;
      input  D;
      output  Q;
      wire  Clk, D; //You can omit this statement
      reg  Q;
      always @(posedge Clk)  Q <=D;
endmodule

//Example 2
module  Func_1(In1, enable, Q1, Q2);
      input  In1;
      input[2:0]  enable;
      output  Q1;
      output  Q2;
      wire  In1; //You can omit this statement
      wire [2:0] enable //You can omit this statement
      wire  int_sig //You can omit this statement
                          but it is useful for debugging
      wire  Q1, Q2 //You can omit this statement
      assign  Q1 = enable[2] enable[0] ;
      nand  nd1(In1, enable[2], int_sig);
      nand  nd2(int_sig, enable[0], Q2);
endmodule
```

When you declare multibit signals, you can declare the bit width of the signal in the format [MSB:LSB]:

```
input [3:0] enable;  //4-bit input signal enable
wire [7:0] bus;  //8-bit net signal bus
output [4:0] out;  //5-bit output signal out
```

Of course you can select bits from multibit signals, or you can assign part of the multibit signal:

```
wire  SB;
wire [2:0]  DB;
assign SB = dbus [7];
assign = dbus [6:4];
assign  dbus[3:0] = enable;
```

You can assign a register array. This is frequently used for a memory block or register file:

```
reg [15:0]  mem[0:127];  //16-bit signal * 128 entries
```

However in this case you *cannot* directly select part of the signal in a register array; you can access the signal only entry level. And Verilog does *not* support multidimensional arrays like C or C++.

```
reg [15:0]   mem[0:127]
//To access the last 4bit of the first entry of the mem array.

//Incorrect example
assign out1 = mem[0][4:0]
//Correct example
wire [15:0]  temp;
assgign temp = mem[0];
assign out1 = temp[4:0];
```

A.4.4 Operators

Verilog supports arithmetic operators, bitwise operators, bit-reduction operators, logic operators, relative operators, shift operators, conditional operators, and concatenation. The operators are listed below.

```
wire a, b, c;
wire [4:0] d, e;
```

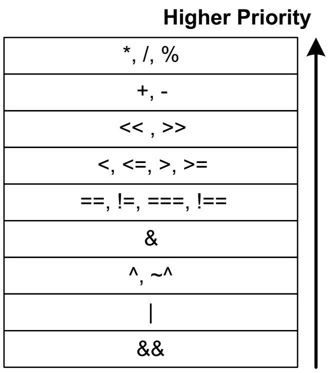
Operator		Meaning
aritmetic	+	Add
	-	Sub
	*	Multiplication
	/	Division
	%	Modulo operation
Bit	~	NOT
	&	AND
	|	OR
	^	XOR
	~^	XNOR
Bit Reduction	&	AND
	~&	NAND
	|	OR
	~|	NOR
	^	XOR
	~^	XNOR

Operator		Meaning
Logic Operation	!	Logical NOT
	&&	Logical AND
	||	Logical OR
	==	Equal
	!=	Not equal
	===	Equal including (x, z)
	!==	Not equal including (x, z)
Relative	<	Less than
	<=	Less than or equal to
	>	Larger than
	>=	Larger than or equal to
Shift	<<	Left Shift
	>>	Right Shift
ETC	? :	Conditional
	{ }	Concatenation

```
//Arithmetic operations
assign  c = a+b;
assign  c = a-b;
assign  c = a*b;
assign  c = a/b;
assign  c = a%b;
      //If the process library supports these arithmetic operations, the
        operators are
      //simply converted into an arithmetic unit, called design ware.
      //The types of those arithmetic units are decided
      //according to timing constraints.
      //If the process library does not support design ware,
      //these expressions are not synthesizable.
//Bit operations
assign  c = ~a;
assign  c = a&b;
//Bit-reduction operations
assign  c = &d;
      //c = d[4] & d[3] & d[2] & d[1] & d[0];
assign  c = |d;
      //c = d[4] | d[3] | d[2] | d[1] | d[0];
//Logical operations
assign  c = (a == 1) && (b==1);
assign  c = (a >= 1) || (b !=3);
      //When you want to decide a certain signal with logical conditions,
      //use logical operators.
      //Since === and !== are not synthesizable, use them carefully.
```

```
//Shift operations
assign  d = a << 3;
      //Shift three bits to the left
assign  e = d >>2;
      //shift two bits to the right
//Conditional operation
assign  d = (a == 0)? 11000 : 00001;
      //If the condition is true, the procedure is selected.
      //If not, the latter is selected.
//Concatenations
assign  {a,b,c} = d[2:0];
      //Assign the last three bits of d to a, b, c, respectively
assign  d = {a, b, c, 2'b11};
```

The *priority of the operators* is very similar that of the C and C++ languages:



Higher Priority
*, /, %
+, -
<< , >>
<, <=, >, >=
==, !=, ===, !==
&
^, ~^
|
&&

When you use concatenation, you also can use repeated expressions:

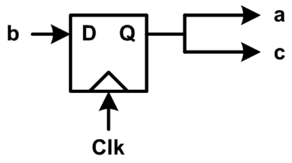
```
wire [7:0]  temp;
assign temp = {4{2'b10}};  //temp = 8'b10101010
wire [15:0]  word;
wire [31:0]  double
assign double = {{16{word[15]}}, word};
```

A.4.5 Assignment

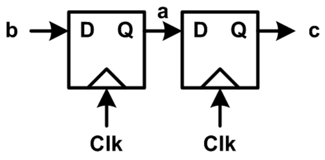
In Verilog there are two assignment statements, blocking and non-blocking. In "net" type signal statements, these produce the same results. But, in a "reg" signal statement they produce different results.

```
//Assume initial values a = 4, b = 5, c = 6
//Blocking assignment
always @(posedge clk)
```

```
begin
      a=b;   //a = b = 5; c=6;
      c=a;   //c = a = b = 5;
end
//Non-blocking assignment
always @(posedge clk)
begin
      a <= b;   //a = 5;
      c <= a;   //c = 4;
end

//Synthesis results:
//Blocking assignment
```



```
//Non-blocking assignment
```



You can easily distinguish the difference between blocking assignments and non-blocking assignments. The data transition of the blocking assignment occurs one by one and that of the non-blocking assignment occurs simultaneously. When you design a digital system, it is best to use one type of assignment only.

A.4.6 Ports and Connections

Verilog has three types of port: input, output, and inout (Figure A.8). Basically, all ports are considered as wires. Therefore, if the ports use “net” type signals, you do not need to declare wire statements. But if the output port needs to hold the value – the left side value in initial or always statements – you should declare a “reg” statement about that signal. Since the input signal cannot hold a value, you cannot declare a “reg” statement for input and inout ports.

A.4.7 Expressions

The most frequently used statements in Verilog are “initial” and “always.” Both of them are used to generate events. Normally, the “always” statement is used to design sequential logic like flip–flops or to design complex combinational logic, and the

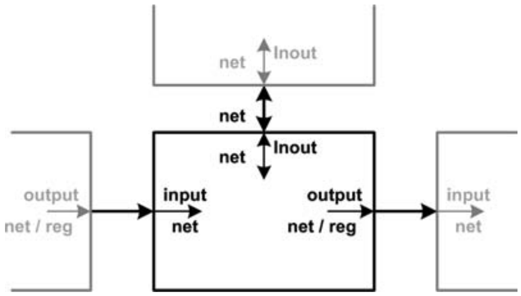


Figure A.8 Ports

“initial” statement is used to declare an initial value. One big difference between “initial” and “always” is the number of executions: the “initial” statement runs only once when the module is activated, but the “always” statement runs every time a specified event occurs. Also, the “initial” statement is omitted when the module is synthesized. Therefore, use the “initial” statement only to describe behavior or to declare an initial value.

```
//Initial statement
initial   //Execute once to initiate ing_sig and temp_wire
begin
      int_sig = 0;
      temp_data = 4'b0;
end
//Always statements
always @ (posedge clk)
      //Execute every positive edge of the clock signal
begin
      int_sig = in_data [4];
      temp_data = in_data[3:0];
end
always @ (a or b or c)

      //Execute whenever value of a or b or c is changed.
      //→ Sensitivity list
begin
      //Write all values used in right-hand side of the equation
      //and all values in conditions to the sensitivity list
      if (c ==1);
      d = (a & b);
end
```

As in the C language, Verilog supports “if–else” statements and “case” statements. These statements should be inside the “initial” or “always” statements. With these conditional statements there is one rule: “*Make a complete conditional statement.*” When you use “if–else” and “case” statements with edge-triggered events you do not need to make the condition complete; but when you use them with level-triggered events, like combinational logic, you should make the condition complete. If you do not do this, undesired latches are generated in the logic, and those latches cause an unrecoverable timing violation like hold time violations and setup time violations.

```
//Edge-triggered event
always @(posedge Clk)
       //An edge-triggered event is synthesized with a flip-flop.
       //Therefore you don't need to complete the condition.
begin
       if (enable == 1) data_latch <= data_in;
end

//Combinational logic:
//if-else statement
always @(a or b or c or d or e)
       //List all right-side signals and condition signals
       //in an if-else statement.
       //There are three choices: if, else if, else.
       //Although you don't need the 'else' case,
       //you should declare it for a complete declaration.
       //If you don't complete the conditions, an undesired latch
       //will be inserted.
begin
       if (a & b == 1)        out = c;
       else if ( b & c == 0)  out = d & e;
       else                   out = 1'b0;
end

//Case statement
always @(a or b or c or d)
begin
       case({a,b})  //There are three case statements: case, casex, casez.
            2'b00 : out = 1;   //case signal can be 0 or 1
            2'b01 : out = 0;   //casex signal can be 0 or 1 or x
            default : out =0;   //casez signal can be 0 or 1 or z
            //With a case statement, declare a default case
            //to make the condition complete.
            //It prevents an unwanted latch.
       endcase
end
```

A.4.8 Instantiation

Digital systems are designed with a hierarchical structure, so that one big system consists of many sub-modules. One reason for this is design complexity. Since the system cannot be verified all at once, the desirability of dividing it into several blocks is increased. After verifying the sub-modules, it is easier to verify the big system so that the efficiency of designing is increased. Other advantages are the shortened design time and efficient use of system resources. Therefore, you often use instantiation in Verilog. There are two methods for instantiation: port assignment based on an order of signals, and port assignment based on number of ports.

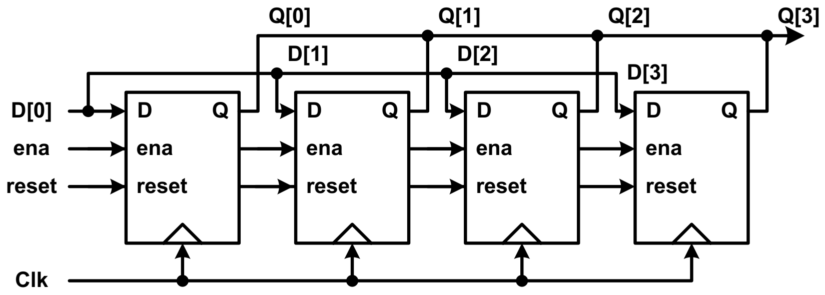


Figure A.9 Four-bit flip–flop

Examples of Instantiation: Flip–Flops

```
//1-bit flip-flop
module   DFF (Clk, reset, ena, D, Q);
      input  Clk, reset, ena;
      input  D;
      output  Q;
      always @(posedge Clk or negedge reset)
      begin
           if(~reset) Q <= 0;
           else if(ena) Q <=D;
     end
endmodule

//4-bit flip-flop (Figure A.9)
module   4b_DFF(Clk, reset, ena, D, Q);
      input  Clk, reset, ena;
      input [3:0]  D;
      output [3:0]  Q;
//Instantiation by order
      //When the sub-block has only a few in/out signals,
      //you can use instantiation by order.
      //As the signal is connected by order, it is easy to make mistakes.
      //This is especially true when debugging the block
      //or when adding or removing signals of sub-blocks.
      //Therefore, this method is not recommended for a block
      //containing lots of ports.
      DFF   DFF0(Clk, reset, ena, D[3], Q[3]);
      DFF   DFF1(Clk, reset, ena, D[2], Q[2]);
      DFF   DFF2(Clk, reset, ena, D[1], Q[1]);
      DFF   DFF3(Clk, reset, ena, D[0], Q[0]);
//Instantiation by name
      //In this method, you should declare the name or port with '.'
      //and put the name of the link signal to the port.
      //.Clk (Clk) means that the Clk signal of the 4-bit flip-flop is
      //connected to the Clk port of the 1-bit flip-flop.
      //This method may seem a little scrappy, but it does not
      //depend on the order of ports. That means this method
      //can add or remove ports of the sub-blocks, which is
      //really helpful for debugging.
      DFF  DFF0(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[3]),
           .Q (Q[3]));
      DFF  DFF1(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[2]),
           .Q (Q[2]));
     DFF  DFF2(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[1]),
           .Q (Q[1]));
     DFF  DFF3(.Clk (Clk),
           .reset (reset),
           .ena (ena),
           .D (D[0]),
           .Q (Q[0]));
endmodule
```

A.4.9 Miscellaneous

Include Directive

Like the C language's #include, Verilog supports an include directive. This is used to include other Verilog code or library file within the current Verilog code. The syntax of the include directive is shown below:

```
'include  "datapath.v"
       //Datapath.v includes adder module, multiplier, divider modules.
       //Then you can use those modules when you declare the include directive.
module
       processor(clk, inputs, output);
             input  clk;
             input  inputs;
             output  outputs;
       //Divider and multiplier are described in datapath.v.
       divider proc_div(.input(input), .output(output));
       multiplier proc_mul(.input(input), .output(output));
endmodule
```

Define Statement

The "define" statement is a kind of complier command to transpose a text. When you use a certain value recursively, the "define" statement can help you.

```
//Do not write semicolons in define statements.
//Once you have declared a define statement,you can use the text in any modules.
```

```
//You can use these defined texts in other files if you include this file.
`define    WISA_RSHA    6'b1000_00
`define    WISA_RTEX    6'b1000_01
`define    WISA_RDON    6'b1000_10
`define    WISA_TMOD    6'b0100_01
`define    WISA_MBAS    6'b0010_00
`define    WISA_RCLR    6'b0010_01
module
      decode(inputs, outputs);
             input inputs;
             output outputs;
...
     //Command decoder
     always @(ID1data)
     begin
          case(ID1data[123:118])
          //In compile time, these texts are converted into defined signals.
          `WISA_RSHA   : ID1ctrl_OP <= `WCTRL_OP_RSHA;
          `WISA_RTEX   : ID1ctrl_OP <= `WCTRL_OP_RTEX;
          `WISA_RDON   : ID1ctrl_OP <= `WCTRL_OP_RDON;
          `WISA_TMOD   : ID1ctrl_OP <= `WCTRL_OP_TMOD;
          `WISA_MBAS   : ID1ctrl_OP <= `WCTRL_OP_MBAS;
          `WISA_RCLR   : ID1ctrl_OP <= `WCTRL_OP_RCLR;
          default      : ID1ctrl_OP <= 6'b0;
          endcase
     end
...
endmodule
```

Timescale Command

In Verilog you can define a timescale. You should define this when you want to perform gate-level simulation or post-PnR simulation. Of course you can define a delay time in your design with timescale. But it will be omitted in synthesis time. Therefore use a time delay statement only for functional verification like gate propagation delay modeling.

```
`timescale   1ns / 1ps
//1ns is the reference time unit. It is a unit of time and delay.
//1ps is the precision of the time.
//Below, #5 means 5ns in simulation time,
//but this is not effective in the synthesis.
module   testbench();
      reg clk;
      initial    clk = 1b0;
      always #5
      begin
           clk = ~clk;
      end
endmodule
```

A.5 Example of Four-bit Adder with Zero Detection

See Figure A.10.

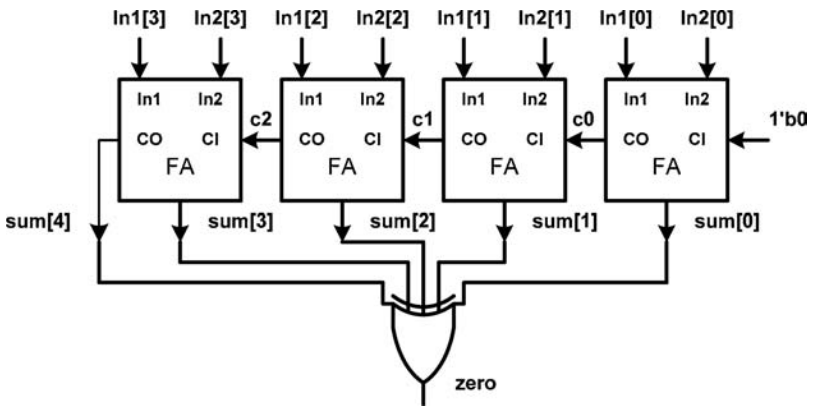


Figure A.10 Four-bit adder with zero detection

Behavior-level Description

This behavior-level description includes an initial statement and thus it is not synthesizable.

```
module   adder4 (in1, in2, sum, zero);
      input [3:0] in1, in2;
      output [4:0] sum;
      output zero;
      reg [4:0] sum;
      reg zero;
      initial
      begin
           sum = 0;
           zero = 1;
      end
      always @ (in1 or in2)
      begin
           sum = in1 + in2;
           if (sum == 0)  zero = 1;
           else  zero = 0;
      end
endmodule
```

Hierarchical Description 1 – Full Adder

```
//1-bit full adder module
module   FA (in1, in2, c_in, sum, c_out);
```

```
      input      in1, in2, c_in;
      output     sum, c_out;
      assign     {c_out, sum} = in1 + in2 + c_in;
endmodule

//4-bit full adder module
module adder4 (in1, in2, sum, zero);
      input [3:0]    in1;
      input [3:0]    in2;
      output [4:0]   sum;
      output         zero;
      wire           c0, c1, c2;
      FA add1 (. in1(in1[0]), .in2(in2[0]), .c_in(1b0), .sum(sum[0]), .c_out(c0));
      FA add2 (. in1(in1[1]), .in2(in2[1]), .c_in(c0), .sum(sum[1]), .c_out(c1));
      FA add3 (. in1(in1[2]), .in2(in2[2]), .c_in(c1), .sum(sum[2]), .c_out(c2));
      FA add4 (. in1(in1[3]), .in2(in2[3]), .c_in(c2), .sum(sum[3]),
.c_out(sum[4]));

      //zero detection
      zero = | sum;
endmodule
```

Hierarchical Description 2 – Designware

The Synopsys design compiler, a synthesis tool, supports designware libraries. A designware library is a type of datapath set – unsigned adder, subtracter, multiplier, divider, floating-point adder, subtracter, multiplier, divider, and so on. When you use designware you do not need to declare basic datapaths. More details of designware are available on the website of Synopsys: www.synopsys.com.

```
//4-bit full adder module
module   adder4 (in1, in2, sum, zero);
      input [3:0]     in1;
      input [3:0]     in2;
      output [4:0]    sum;
      output          zero;
      wire            c0, c1, c2;

      //Designware module #(bitwidth) instant_name(portlist)
      //When you declare the designware adder,
      //the type of adder is decided by timing and area bounds.
      DW01_add #(1) add0(.A(in1[0]), .B(in2[0]), .CI(1b0), .SUM(sum[0]), .CO(c0));
      DW01_add #(1) add1(.A(in1[1]), .B(in2[1]), .CI(c0), .SUM(sum[1]), .CO(c1));
      DW01_add #(1) add2(.A(in1[2]), .B(in2[2]), .CI(c1), .SUM(sum[2]), .CO(c2));
      DW01_add #(1) add3(.A(in1[3]), .B(in2[3]), .CI(c2), .SUM(sum[3]),
        .CO(sum[4]));

      //Zero detection
      zero = | sum;
endmodule
```

A.6 Synthesis Scripts

dont_use Scan Chain

This command is used to exempt some library blocks from the process library file.

```
set_dont_use   {typical/SD*}
set_dont_use   {typical/SE*}
```

read

```
read -format db {"WGR_ctrl.db"}
      //Read database as db format
read -format db {"WGR_datapath.db"}
      //This command is used to read a previously synthesized block
read -format verilog {"WGR_main.v"}
      //Used to read target design Verilog file
check_design
      //Check target design
reset_design
      //Reset target design
remove_clock -all
      //Since each block has its own clock and timing bound,
      //the previously defined clock should be removed.
```

IF: TOP

```
current_design Wif
      //Define current design.
      //One Verilog file can have several modules,
      //so you should declare the current block to be synthesized.
remove_constraint – all
      //Since the new constraints will be declared,
      //the previous constraints should be removed.
create_clock -period 40 REclk -waveform {0,20.0}
      //Create clock: period is 40ns, duty cycle is 50%.
set_dont_touch find(net, "REclk")
      //Since clock net is a global signal, declare ‘‘dont_touch’’ net.
set_input_delay 2.5 -rise -clock REclk {IFmode, SYS_nRESET}
      //Declare input signal delay.
set_load 0.05 all_outputs()
      //Set load to all output signals
      //to adjust driving power of the output load
set_load 0.3 InREQ
      //Set load to output signal.
set_max_transition 0.06 all_outputs()
      //To limit rising or falling time due to RC time constant.
set_max_area 39
      //Declare the area bound.
compile -map_effort high
      //Set compile option.
report_area > Wif_Area.txt
      //Write area report.
```

```
report_timing -from all_inputs() -to all_outputs() > Wif_Timing.txt
      //Write timing report.
      //Timing report of the synthesis only includes gate propagation delay.
write -format db -hierarchy -output "Wif.db"
      //Write result as db format – for hierarchical synthesis
write -format verilog -hierarchy -output "Wif.v"
      //Write result as Verilog file – for gate-level simulation.
write_sdf "/home/denber/project/ramp-GR/verilog/sdf/Wif.sdf"
      //Write SDF file – standard delay file.
```

Script for Top Module: WRE_top

```
current_design WRE
      //Declare current design.
remove_clock – all
      //Remove previously defined clock.
remove_constraint – all
      //Remove previously define constraints.
create_clock "REclk" -period 40.0 -waveform {0, 20.0}
      //Create clock for processor.
create_clock "memclk1" -period 20.0 -waveform {0, 10.0}
      //Create clock for memory module.
set_dont_touch find (net, "REclk")
      //Declare dont_touch net signal – clock.
set_dont_touch find (net, "memclk1")
      //Declare dont_touch net signal – clock.
set_dont_touch find (cell, "GR_IF")
      //Declare previously synthesized module.
      //In top module, it only uses the previously synthesized module.
set_dont_touch find (cell, "GR_ID1")
set_dont_touch find (cell, "GR_ID2")
set_load 0.05 {Dbg_out, CSR}
      //Set output load capacity.
set_max_transition 0.06 {Dbg_out, CSR}
      //Limit rising and falling times.
compile -map_effort high
      //Set compiler option.
report_area > WRE_Area.txt
      //Write area report.
report_timing -from all_inputs() -to all_outputs() > WRE_Timing.txt
      //Write timing report.
write -format db -hierarchy -output "WGR.db"
      //Write result as db format
write -format verilog -hierarchy -output "WGR.v"
      //Write result as Verilog file.
write_sdf "WGR.sdf"
      //Write SDF file.
```

Glossaries

3D Graphics	Three Dimensional Graphics
AAL	Address Alignment Logic
AI	Artificial Intelligence
ALU	Arithmetic and Logic Unit
API	Application Programming Interfaces
ARM	Advanced RISC Machine; a popular embedded processor
ARM9	ARM processor with ARM v5instuction set
ARM11	ARM processor with ARM v6 instruction set
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DVFS	Dynamic Voltage Frequency Scaling
GPU	Graphics Processing Unit
HSR	Hidden Surface Removal
HW	Hardware
IP	Intellectual Property
ISP	Image Signal Processor
KAIST	Korean Advanced Institute of Science and Technology
LOD	Level of Detail
MIPS	Million Instructions per Second
NoC	Network-on-Chip
OpenGL	Open Graphics Library
PC	Personal Computer
PIM	Personal Information Management
RAMP	RAM + Processor; a GPU architecture developed by KAIST
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SERDES	SERializer and DESerializer

SIMD	Single Instruction Multiple Data
SoC	System-on-a-Chip
SRAM	Static Random Access Memory
SW	Software
TBR	Tile Based Rendering
TnL	Transform and Lighting
UML	Unified Modeling Language
VSLI	Very Large Scaled Integration

Index

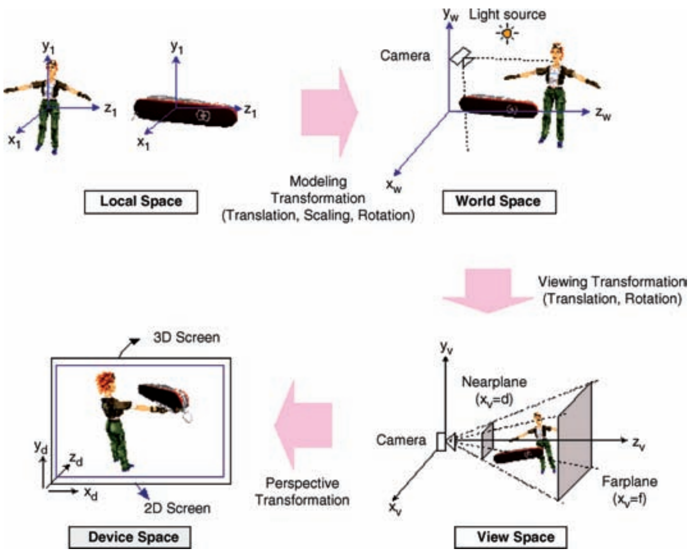


Figure 3.2 A 3D graphics pipeline

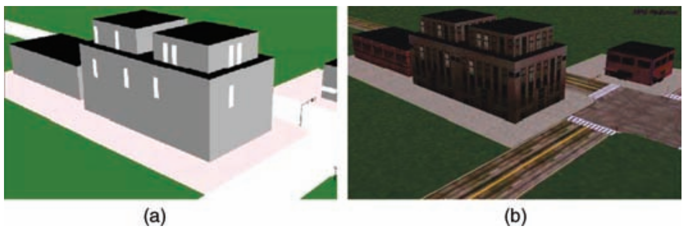


Figure 3.13 Texture mapping effects: (a) flat shaded image, and (b) texture-mapped image

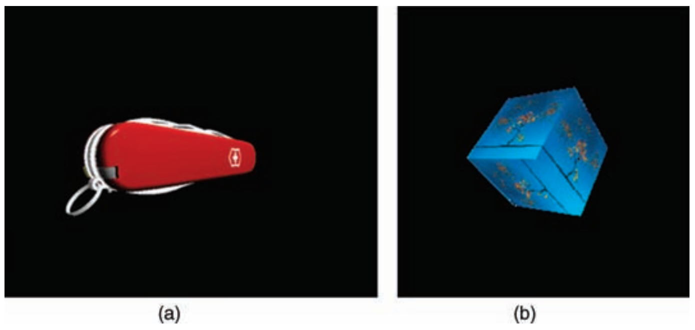


Figure 4.7 Scenes from MobileGL: (a) lighting, and (b) texture mapping

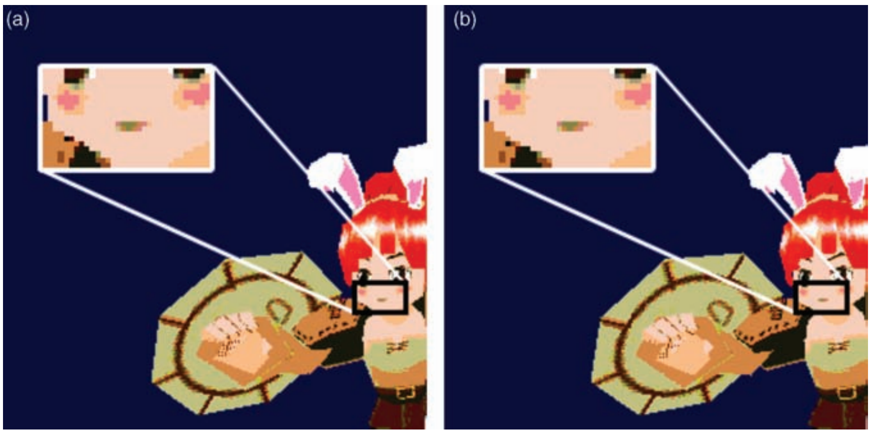


Figure 5.5 Comparison of 3D graphics results: (a) normal fixed-point calculation, and (b) logarithmic number calculation

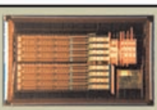
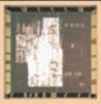
	Architecture		Process Technology	Power Consumption	Chip Area Embedded Memory	Features
Fixed 3D Graphics Pipeline	RAMP-I		0.35μm Embedded Memory Logic	560mW	56mm^2 512kb eDRAM	Gouraud-Shading Alpha Blending Depth Comparison
	RAMP-II		0.18μm Embedded Memory Logic	160mW	84mm^2 6Mb eDRAM	
	RAMP-IV		0.16μm Pure DRAM	210mW	121mm^2 29Mb eDRAM	+ Bilinear MIPMAP Textureing Fogging / Motion Blur Antialiasing
Programmable 3D Graphics Pipeline	RAMP-V		0.18μm CMOS Logic	155mW	36mm^2 96kb eSRAM	Fixed-Point Vertex Shader Dual-Operations
	RAMP-VI		0.18μm CMOS Logic	153mW Peak 52.4mW Avg. (60fps)	25mm^2 29kb eSRAM	Logarithmic Number Vertex Shader DVFS for 3D Graphics Pipeline
	RAMP-VII		0.13μm CMOS Logic	195mW	41mm^2 128kB eSRAM	Mobile Unified Shader Pixel-Vertex Multi-Threading Logarithmic Lighting Engine

Figure 6.1 Summary of RAMP architectures