AN ABSTRACT OF THE THESIS OF

Keung-Sik,Choi for the degree of Master of Science in Electrical and Computer Engineering presented on May 7, 1998. Title: The Architecture of a Multimedia Multiprocessor.

Abstract approved: Redacted for Privacy

Ben Lee

The multimedia capabilities of computers have recently become the focus of computer developers due to the increasing demand for advanced computer graphics and new media capabilities, such as video conferencing, 3-D visualization, and animation. To support these multimedia capabilities, specialized graphics hardware, such as MPEG encoding/decoding card, 3-D graphics card, video card, and sound card, are widely used today, but the price of a separate board is expensive. Therefore, the processor must be redesigned from the ground up to handle new media applications. Although these multimedia functions are typically consist of simple operations, their sheer volume of computation creates a flood of data. To support such large volumes of multimedia data computation, Sun Microsystems implemented a specialized instruction set, called VIS™ (Visual Instruction Set), which is Single Instruction Multiple Data (SIMD) style of instruction. The basic concept behind VIS is to break the pipeline of the Floating Point Unit (FPU) into two or four parallel pipelines to perform four or eight separate 16-bit or 8-bit integer additions in one cycle, instead of one floating-point addition.

The Electronics and Telecommunications Research Institute (ETRI) in South Korea has researched a 64-bit multimedia enhanced on-chip multiprocessor named Raptor, which has quad processors and shares a common Graphics Control Unit (GCU). Raptor implements multimedia support directly on the processor using specialized instructions, GCU Instructions, which are variant of VIS instructions, and hardware supports. Each processor of Raptor executes multimedia applications independently and

the independent streams or threads of multimedia instructions compute for and share a single GCU.

The major theme of this thesis is to design the GCU architecture and to simulate it. The GCU can simultaneously execute the independent instruction streams from four General Processors (GP) and resolves the dependencies among the instructions dynamically.

The Architecture of a Multimedia Multiprocessor

by

Keung-Sik, Choi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented May 7, 1998
Commencement June 1998

Master of Science thesis of Keung-Sik, Choi presented on May 7, 1998

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Head or Chair of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Keung-Sik Choi, Author

ACKNOWLEDGMENT

This thesis is dedicated to my parents, parents in law, and my wife Eunsuk Kim. Without their unconditional support, this would never have been done.

Special thanks to Dr. Ben Lee, my major professor. His guidance and advice have been extremely valuable to my studies in Oregon State University. Also, special thanks to the committee members, Dr. James H. Herzog, Dr. Shin_Lien Lu, and Dr. Kong Wei.

Thanks to my senior Hantak Kwak. He has been helped me when I first started this study. I will never forget his kindness in helping me. Thanks to Yongseok Seo, who helped me to settle down in Corvallis, and showed me how to live. Thanks to Chuck and Carole, who are like father and mother, and thanks to Danny and Clarice, who are good friends. They have been encouraged me to study and helped me to live in America. And lastly, I would like to thank Republic of Korea Army, which gave me the chance to study abroad and has supported me financially.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES (continued)

Figure Page

LIST OF TABLES

Table Page

THE ARCHITECTURE OF A MULTIMEDIA MULTIPROCESSOR

1 INTRODUCTION

The *multimedia processor* or *media processor* has recently become the focus of both software and hardware developers due to the increasing demand for advanced graphics and new media capabilities, such as video conferencing, 3-D visualization, and animation. Multimedia processor may be defined as a processor that integrates multiple, concurrent media processing tasks required for such applications as virtual reality, interactive presentation of entertainment or educational titles, video teleconferencing, video authoring, or selection and distribution of movies from satellite channels (including real-time data compression and encryption) [1]. Now consumers are demanding a new class of machines, which incorporate extensive network and collaborative workgroup capabilities, rapid access to pictorial databases, and support for real-time video and audio at broadcast-level quality.

There are several approaches for accommodating this explosion in new media processing and workplace interconnection. The conventional approach is to expand the network bandwidth and increase the processor clock rates, but this cannot suffice all the demands. The common method widely used today is specialized graphics hardware such as MPEG encoding/decoding card, 3-D graphics card, video card, and sound card and so on, but the price of separate board is expensive. Therefore, the processor must be redesigned from the ground up to handle new media applications and network computing.

The processor is responsible for handling high-speed graphics, 2-D and 3-D imaging, video processing, and image compression/decompression. Although these functions typically consist of simple operations, their sheer volume creates a flood of data. To support such large volumes of multimedia data computation, Sun Microsystems implemented a specialized instruction set, called VIS^{TM} (Visual Instruction Set), which is Single Instruction Multiple Data (SIMD) style instructions. The basic concept behind VIS is to break the pipeline of Floating Point Unit (FPU) into two or four parallel

pipelines to perform four or eight separate 16-bit or 8-bit integer additions in one cycle, instead of one floating-point addition. This novel idea was quickly accepted and further extended by Intel in their P55C design, and re-emerged as MMX™ [2].

Today's integrated circuit technology gives more options to processor designers and the technology has matured to a point where the development of on-chip multiprocessors is feasible. For instance, the Hydra research group of Stanford has been actively studying on-chip multiprocessors and showed that this design has an advantage over wide-issue superscalar architecture, especially when the application has large number of active processes or independent threads [3]. The Electronics and Telecommunications Research Institute (ETRI) in South Korea has performed research on 64-bit multimedia enhanced on-chip multiprocessor named Raptor, which has quad processors and shares a common Graphics Control Unit (GCU). Raptor implements multimedia support directly on the processor using specialized instructions, GCU Instructions, and hardware supports. Each processor of Raptor executes multimedia applications independently and the independent streams or threads of multimedia instructions compete for and share a single GCU. The GCU executes its own GCU Instructions, which are variant of VIS instructions.

1.1 Objectives and Scope of the Thesis

This thesis describes the design and the simulation study of the GCU architecture. The objective of the thesis is three folds: First, to study the current trends in multimedia processing. Second, to propose and design a new architecture for multimedia processing. Third, to perform performance studies using simulation.

1.2 Outline of the Thesis

The organization of the thesis is as follows: The background of multimedia processing is discussed in Chapter 2. It describes two fundamental aspects of multimedia

processings - pixel and color representation, and data compression. Chapter 3 explains how the addition of SIMD style instructions, such as VIS or MMX, provide improved performance. Chapter 4 discusses the overall architecture of a multimedia enhanced on-chip multiprocessor, called Raptor, and the architecture of GCU. Chapter 5 provides the simulation results of GCU. Finally, a conclusion is presented in Chapter 6.

2 BACKGROUND IN MULTIMEDIA PROCESSING

From the user's perspective, the term multimedia refers to computer information that can be represented through audio and/or video, as well as image, graphics and animation [4]. To understand the architectural requirements of multimedia processing, we first need to understand the basics of computer graphics. First, how pixels and colors are represented in computer systems is introduced. Second, how images are displayed through the monitor is described. And finally, different data compression techniques will be discussed. The amount of the multimedia data, especially motion pictures, are enormous, therefore data has to be first compressed before they are stored and/or transferred.

2.1 Basics of Computer Graphics

Computer graphics refer to any data intended for display on an output device, such as a screen, a printer, a plotter, a film recorder, or a videotape [4]. Moreover, an image is a spatial representation of an object, a two- or three-dimensional scene, or another image. An image may be thought of as a continuous function with resulting values of the light intensity at each point over a planar region [5]. For digital computer operations, this function needs to be sampled at discrete intervals. The sampling quantizes the intensity values into discrete levels. The points at which an image is sampled are known as *picture elements*, commonly abbreviated as *pixels*. The intensity at each pixel is represented by an integer, and is determined from the continuous image by averaging over a small neighborhood around the pixel location. If there are just two intensity values, for example, black and white, they are represented by the numbers 0 and 1. When an 8-bit integer is used to store each pixel value, the values range from 0 (black) to 255 (white).

There are different types of images, e.g., bi-level, gray-scale, color, and photographic-quality images. An example of pixel representations for bi-level, gray-scale, and color images are shown in Figure 2.1. Bi-level or binary images use only two

intensity levels, one for the "information" and another for the "background". Therefore, each pixel is a single bit, whose value is either 0 or 1. Gray-scale images use multiple intensity levels to record the shadings between black and white. Gray-scale imaging techniques are appropriate for monochrome photographs and medical images where an accurate representation of shading is important. Color and photographic-quality images use multiple intensity levels and filtering to capture the brightness level for each of the three primary colors in visible light, i.e., red, green, and blue. Each pixel requires up to 24 bits or more.

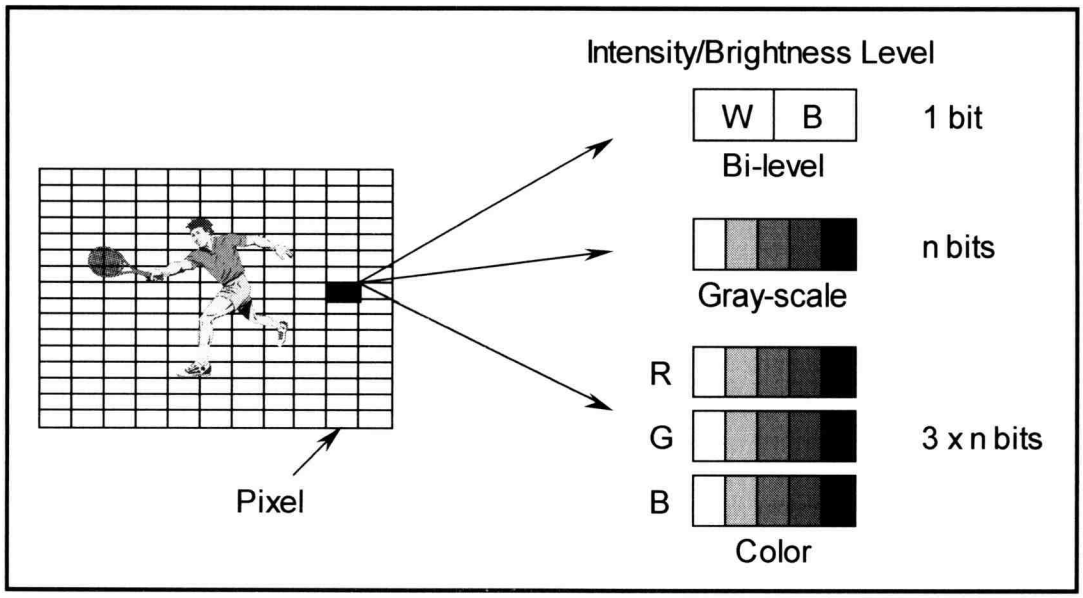


Figure 2.1: Pixel Representation [6]

2.1.1 Color Representation

The most common color systems used in computer graphics are the *primary 3-color systems* [5]. With such systems, a color is defined by specifying an ordered set of three values. Composite colors are created by mixing varying amounts of the three primary colors, which results in the creation of a new color. This subsection describes

some of the common color systems. Table 2.1 shows the corresponding values for the primary and achromatic colors using the RGB, CMY, and HSV color systems.

	RGB	**CMY**	**HSV**
Red	255,0,0	0,255,255	0,240,120
Yellow	255,255,0	0,0,255	40,240,120
Green	0,255,0	255,0,255	80,240,120
Cyan	0,255,255	255,0,0	120,240,120
Blue	0,0,255	255,255,0	160,240,120
Magenta	255,0,255	0,255,0	200,240,120
Black	0,0,0	255,255,255	160,0,0
Shades of Gray	63,63,63	191,191,191	160,0,59
Shades of Gray	127,127,127	127,127,127	160,0,120
Shades of Gray	191,191,191	63,63,63	160,0,180
White	255,255,255	0,0,0	160,0,240

Table 2.1: Equivalent RGB, CMY, and HSV values [5]

- RGB (Red-Green-Blue)

RGB is the most widely used color system today. It is called an *additive system* because colors are created by adding colors to black to create new colors. Graphics files using the RGB color system represent each pixel as a *color triplet*, three numerical values in the form of (R,G,B), each representing the amount of red, green, and blue in the pixel, respectively. For example, if the color system uses 24-bit color, 8-bits each for the three primary colors. For instance, (0,0,0) represents black, and (255,255,255) represents white. When the three RGB values are set to the same value such as (65,65,65) or (120,120,120), the resulting color is a shade of gray.

- CMY (Cyan-Magenta-Yellow)

CMY is a *subtractive color system* used by printers and photographers for the rendering of colors with ink or emulsion, normally on a white surface. It is used by most hard-copy devices that deposit color pigments on white paper, such as laser and ink-jet printers. When illuminated, each of the three colors absorbs its complementary light color. For example, cyan absorbs red, magenta absorbs green, and yellow absorbs blue. This color system is referred to as subtractive because the colors are subtracted from the white light by pigments to create new colors. For example, when cyan and magenta are absorbed, the resulting color is yellow. The yellow pigment is said to "subtract" the cyan and magenta components from the reflected light. When all of the CMY components are subtracted, or absorbed, the resulting color is black. The representation of CMY is just the opposite of RGB. For example, (255,255,255) is black, and (0,0,0) is white.

- HSV (Hue, Saturation, and Value)

In the HSV color system, hue specifies a "color" in the common use of the term, such as red, orange, blue, and so on. Saturation refers to the amount of white in a hue. A fully saturated hue contains no white, and appears pure, and a partly saturated hue appears lighter in color due to the mixture of white. For example, red hue with 50 percent saturation appears pink. Value (also called brightness) is the degree of self-luminescence of a color, which means how much light it emits. A hue with high intensity is very bright, while a hue with low intensity is dark. HSV closely resembles the color system used by painters and other artists, who create colors by adding white, black, and gray to pure pigments to create tints, shades, and tones.

- YUV (Y-signal, U-signal, and V-signal)

The YUV model utilizes the characteristics of human eyes, which are more sensitive to brightness than any color information. Therefore, it is more suitable color system than others. It is basically a linear transformation of RGB image data and is most

widely used to encode color for use in television transformation. Y specifies the brightness information (luminance). The Y of any color can be calculated from the following weighted sum:

$$Y = 0.3R + 0.6G + 0.1B;$$

The scaling is chosen such that the luminance is also expressed by a relative scale from 0 to 1 and the weights reflect the contributions of the individual primaries to the total luminance. The term chrominance is defined as the difference between a color and a reference white at the same luminance. Therefore, the chrominance information can be expressed by a set of color differences, U and V, where U and V are defined by:

$$U = B - Y;$$
$$V = R - Y;$$

These color differences are zero whenever R = G = B, as this condition produces gray, which has no chrominance. The U component controls colors ranging from blue ($U > 0$) to yellow ($U < 0$), whereas the V component controls colors ranging from red ($V > 0$) to blue-green ($V < 0$).

- YC_BC_R

This color system is closely related to YUV. It uses the same Y coordinate as the YUV system, whereas U and V are scaled and zero-shifted to produce the variables C_B and C_R, respectively. The equations are:

$$C_B = (U/2) + 0.5;$$
$$C_R = (V/1.6) + 0.5;$$

With this scaling and zero shifting the chrominance values are always in the range 0 to 1.

2.1.2 Raster Display

Current output technology uses *raster display*, which stores display primitives in a Refresh Buffer in terms of their component pixels. Figure 2.2 shows the architecture of a raster display.

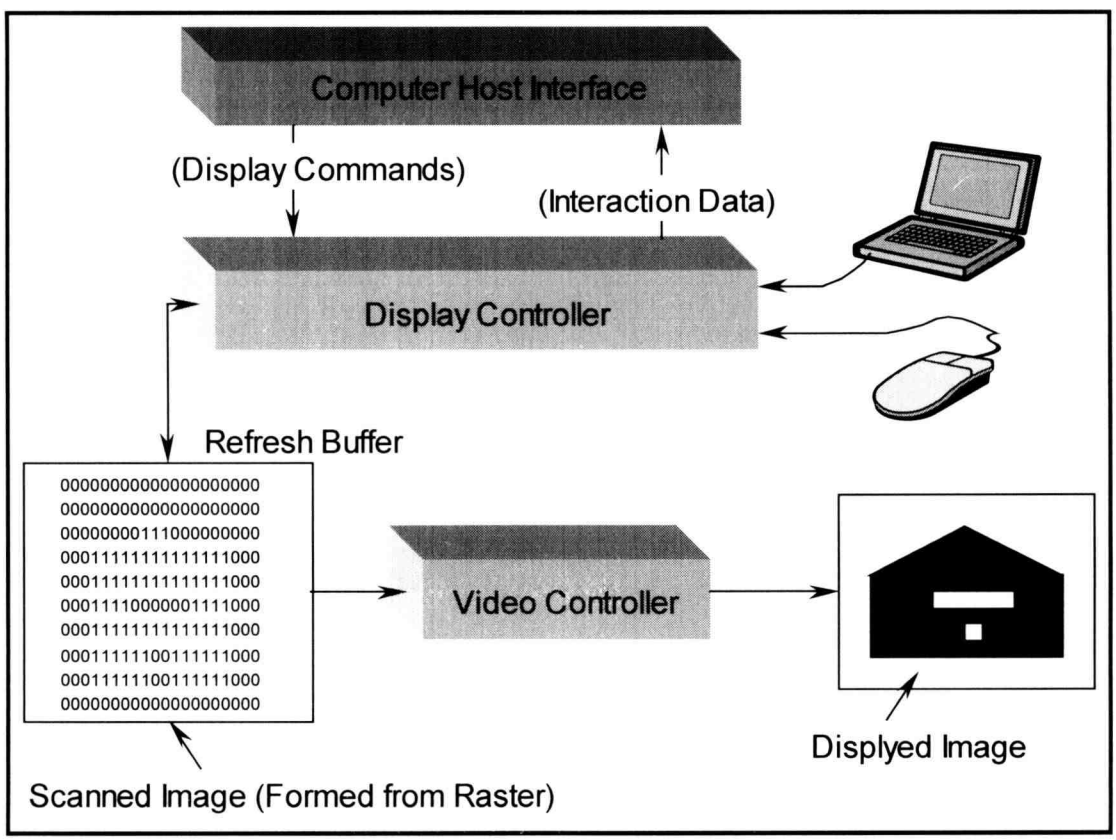


Figure 2.2: Architecture of a Raster Display [4]

The Display Controller receives and interprets sequences of display commands from the CPU and figures out which pixels are being drawn and what color or value they should be, and the pixels are drawn by writing the new values into the Refresh Buffer (bitmap). The Video Controller reads the Refresh Buffer and interprets the pixel values of the Refresh Buffer into their colors and creates the video signals that drive the monitor. The Refresh Buffer is reread each time the monitor image is refreshed. Because this

typically happens 60 to 80 times per second, the Refresh Buffer is effectively displayed flawlessly.

2.1.3 Color Display

When the Video Controller interprets the bitmap pixel values into their resulting colors, it uses *Look-Up Tables* (LUTs) or called a *palette*, which is a 1-dimensional array of color values. These colors are not stored in the bitmap directly. The main reason is to save bitmap memory. If the system uses RGB color system with 256 colors, it would require about 2.5 MB for the bitmap if the bitmap has a resolution of 1,024×800 pixels, since 256 levels requires 8 bits ($2^8 = 256$), or 1 byte, and a full color requires 3 bytes. But in practice, 8 bits per pixel is used with LUT. Eight bits allow up to 256 different colors on the screen at the same time. The entire 1,024×800 bitmap would then fit into just 1 MB (1,024×800 = 819,200) of memory. Then, the color numbers have to be interpreted into real RGB colors.

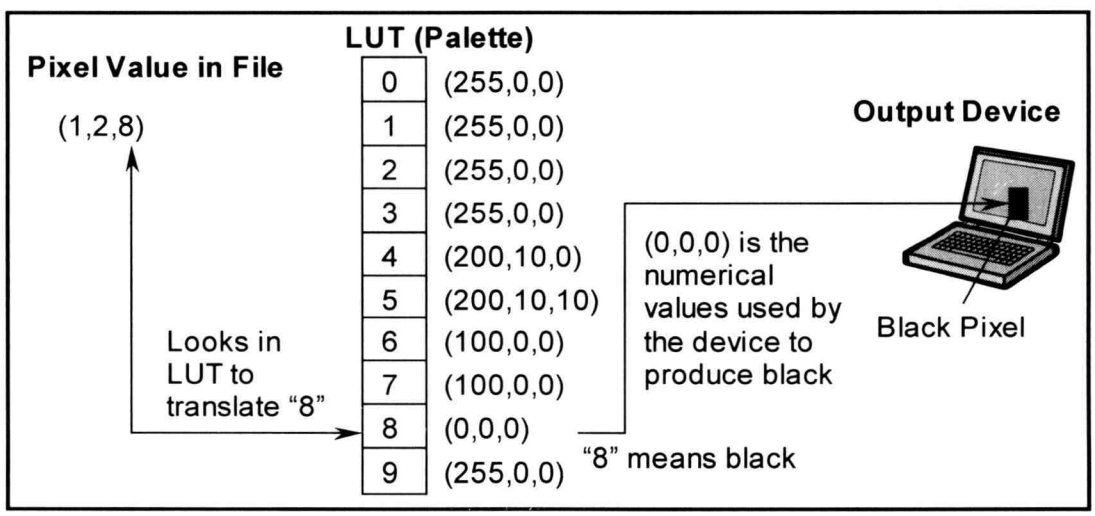


Figure 2.3: Using a LUT to specify a color [5]

This interpretation is done in the LUT shown in Figure 2.3. The LUT converts the color numbers, usually called the *color index values* or *pseudo colors,* from the bitmap into their assigned RGB colors. There are 256 entries in LUT and each entry holds a 24-bit (8 bits per color component) RGB value. For instance, if the pixel value in a file is (1,2,8), the rendering application reads and examines the pixel value from the file and uses it as an index into the LUT, and it retrieves the value of the color from the LUT. Finally, it uses the LUT to specify a colored pixel on an output device.

2.2 Data Compression

This subsection examines the function of data compression in transforming multimedia information. Even the most effective compression technique available today cannot overcome all the storage and bandwidth demands of text, graphics, speech, audio, image, and video data. Figure 2.4 depicts the storage requirements of the case of a single FAX page, image, slide, 5-minute audio, or video presentations.

Whatever presentation method is chosen, the storage for uncompressed audio, images, and video is huge. This makes it difficult to deliver multimedia data in real time. But with data compression, the situation is more manageable, except for video. Therefore, an efficient compression technique is a critical element in a multimedia system. Many advances have been made in compressing multimedia data by exploiting the limitations of human auditory and visual systems. Humans, who usually are the end receivers of the data, do not need, or cannot use, all the information captured during digitization. Powerful and complex models for speech, audio, image, and video data have been created using what is described as perceptual coding techniques that exploit the limitations of human ears and eyes [6]. Among these compression techniques, JPEG (Joint Photographic Experts Group) is an international standard for still images, and MPEG (Moving Picture Experts group) is an international standard of motion pictures. The following subsections briefly introduce these two compression algorithms.

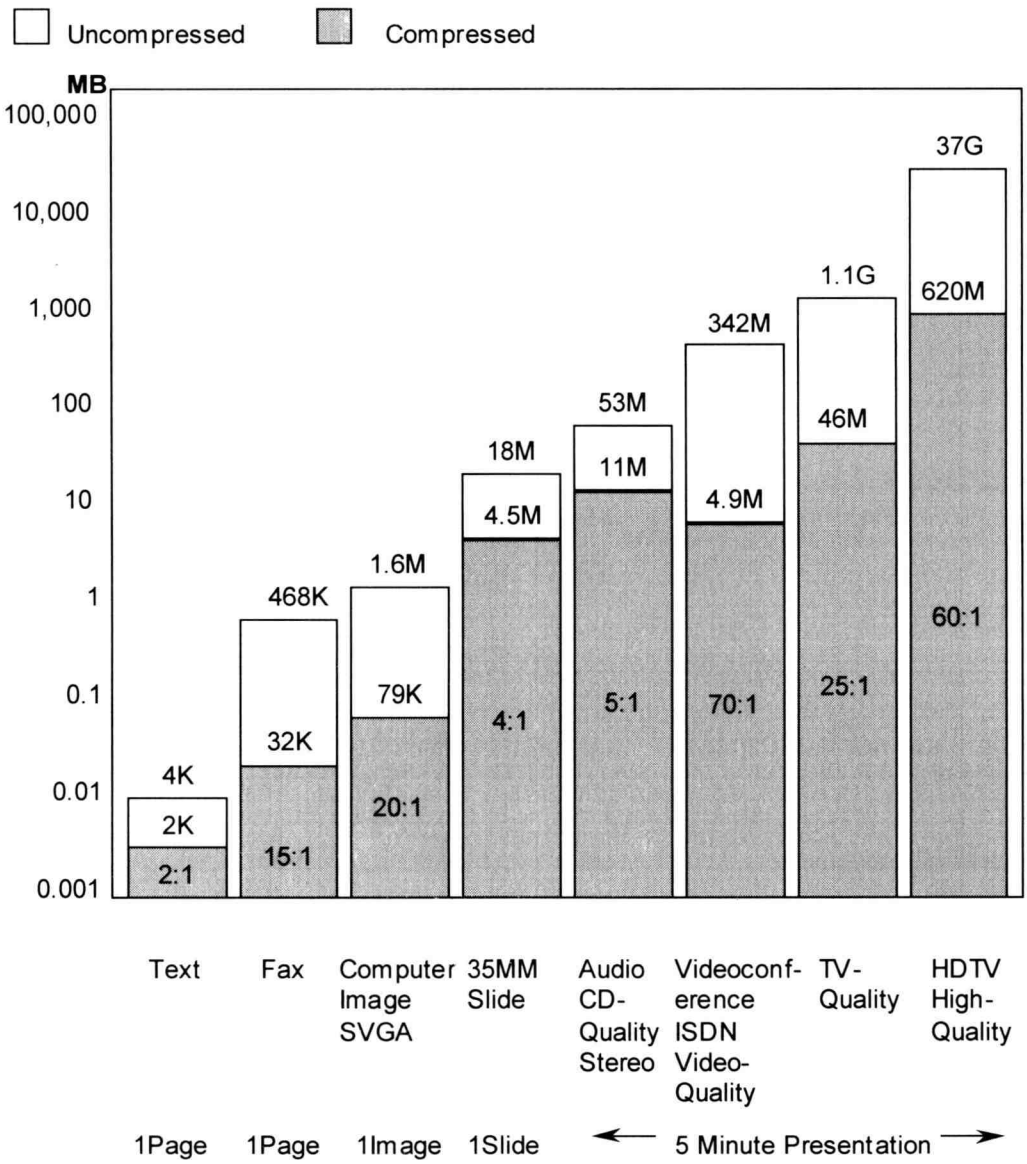


Figure 2.4: Storage Requirement [6]

2.2.1 JPEG

Uncompressed photographic-quality images are too expensive to transmit or store, because they typically use 24-bits per pixel (8-bits each for the RGB components). JPEG group accomplishes 10:1 compression ratio with visually *lossless* image fidelity. It takes

advantage of the *redundancy* and the *irrelevancy* of the image. There are two kinds of redundancy in image compression. One is a *statistical redundancy*, which occurs in a sequence of symbols. This can be removed by lossless compression algorithms, such as Huffman and arithmetic entropy coding, which JPEG uses in its final stage of coding. The basic idea of statistical coding, which includes Huffman and arithmetic coding, is that by observing how often particular symbols occur, shorter codewords are assigned to frequently occurring, more-probable symbols, and longer codewords are assigned to infrequently occurring, less-probable symbols. For example, the Huffman code is constructed by building a binary tree where the leaves of the tree are the probabilities of the symbols to be coded. The tree is built starting at the leaves and working toward the root of the tree, which is shown in Figure 2.5.

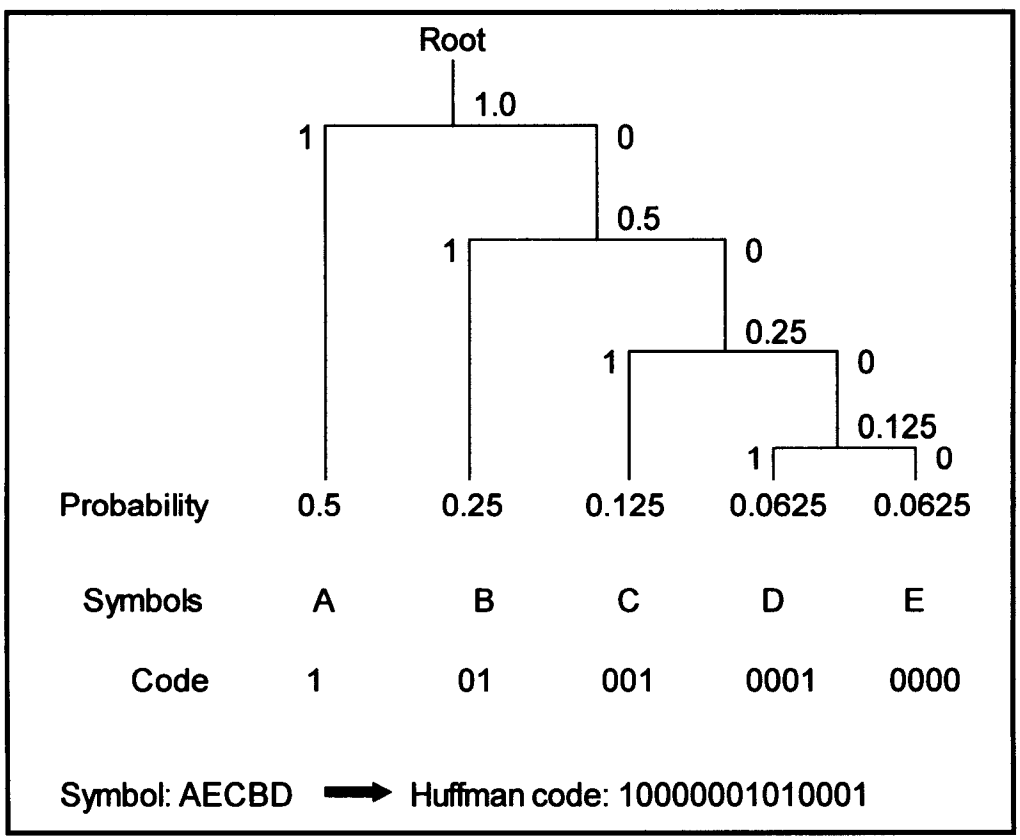


Figure 2.5: Huffman Coding [5]

To begin the process, the symbols are ranked in order of their frequency of occurrence. Next, the two symbols with the smallest probabilities are replaced by an intermediate node representing a subgroup whose probability is the sum of both symbols. Then, the next least frequent pair of symbols or subgroups is located and replaced by an intermediate node. The process continues until all the symbols have been combined into a single structure called a Huffman code tree. The final step is to assign codewords to the symbols. This is accomplished by tracing the tree beginning at the root node and continuing to the leaf node for each symbol.

The other redundancy is a *spatial redundancy*. Within images, neighboring elements for any one of the RGB or YC_BC_R matrices are highly correlated because all the pixels in a region of the image tend to represent the same luminance, or chrominance.

JPEG also uses irrelevancy of the image through the *quantization* technique. The human eye is less sensitive to chrominance than to luminance, meaning that fewer bits are needed for chrominance information, and it can be coded more coarsely than for luminance information. And the human eye is most sensitive to mid-spatial frequencies and not so sensitive to low- and high-spatial frequencies. This allows designers to make compromises in the fidelity of edge contours where rapid transitions in brightness occur. Also, the human eye is less sensitive to quantization distortions at high luminance levels, and this allows more coarse quantization to save bits.

These data compression concepts are embodied in the JPEG standard for still, color image compression. Figure 2.6 shows the basic mode of JPEG encoding and decoding. To begin the process, the components of digitized image are decomposed into sequence of 8×8 pixels and placed in a *Frame Store*. The algorithm operates on each component of the image. It produces a single compressed image for gray-scale images, one for each of the RGB color components or YC_BC_R luminance-chrominance components, or whatever the chosen image representation it is. JPEG does not specify how the image should be represented, so any color transformation, for example RGB to YC_BC_R transformation, where luminance-chrominance differences can be best exploited, must precede before it is placed in a frame store.

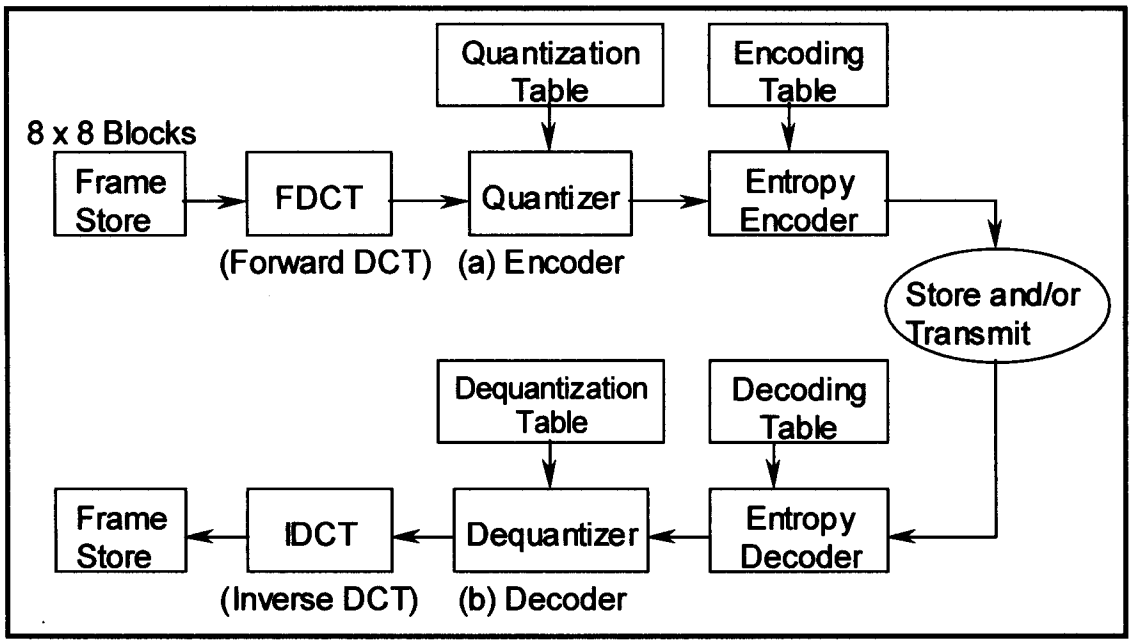


Figure 2.6: JPEG Encoding and Decoding [6]

The JPEG encoding algorithm breaks each image component into blocks of 8×8 pixels and processes each block as follows:

- Spatial redundancy within the image is removed with Discrete Cosine Transform (DCT) coding. The objective of transform coding is to decorrelate the image pixels; that is, statistically dependent image elements are converted into independent coefficients in a transformed space, where the energy of the image is concentrated onto as few coefficients as possible. This allows removing the irrelevant components within the image during the next step.
- Irrelevancy reduction occurs when lossy compression is applied and the DCT coefficients are quantized. Quantization, which is the process of scaling the DCT coefficients and truncating them to integer values, allows us to reduce the accuracy. This is very important in image compression, as it tends to make many coefficients zeros, especially those images with high spatial frequencies.

- Statistical redundancy reduction occurs when lossless compression is performed using Run-Length and Huffman entropy coding.

By reversing the encoding sequences, the JPEG decoder produces an approximate representation of the original input image. Let us examine the JPEG encoding and decoding process more closely as shown in Figure 2.7, which shows how JPEG processes blocks of 8×8 pixels. This figure shows the transformation of a small, a set of shaded gray squares in a field of white. After decoding the actual value is changed, but the human eye cannot distinguish the differences. The steps JPEG uses to process images are also used in the video compression algorithm such as MPEG.

The encoding process starts with the DCT step that transforms the two-dimensional block of pixels from the spatial domain to a two-dimensional array of frequency coefficients in the frequency domain, where only a few data points are required to represent the image. Before the DCT step, the spatial domain pixel values are threshold shifted by 128, becoming positive and negative values. Figure 2.7.a shows a *spatial representation or spatial domain* [6], which represents the 8×8 matrix of 64 values, each with x and y coordinates.

Figure 2.7.b shows the *DCT coefficients* after DCT conversion step. Row 0, column 0 has the DCT coefficient of -784, which is much larger than the other 63 coefficients, and is called *DC coefficient*. The DC coefficient represents an average of the overall value of the 8×8 input matrix. The other 63 coefficients are called AC coefficients and the AC coefficients become smaller and smaller in value as the distance from the DC coefficient increases. This means the reduction in data representation, since only a small set of values is really useful in defining the matrix of Figure 2.7.b. Obviously this is lossy step, but fortunately the human eye may not detect the loss of data.

Figure 2.7.c shows the quantization step, which reduces the number of bits required to store the values of the matrix. JPEG uses *quantization tables*, derived from extensive empirical experimentation. This process attempts to determine what information can be safely discarded without a significant loss in visual fidelity.

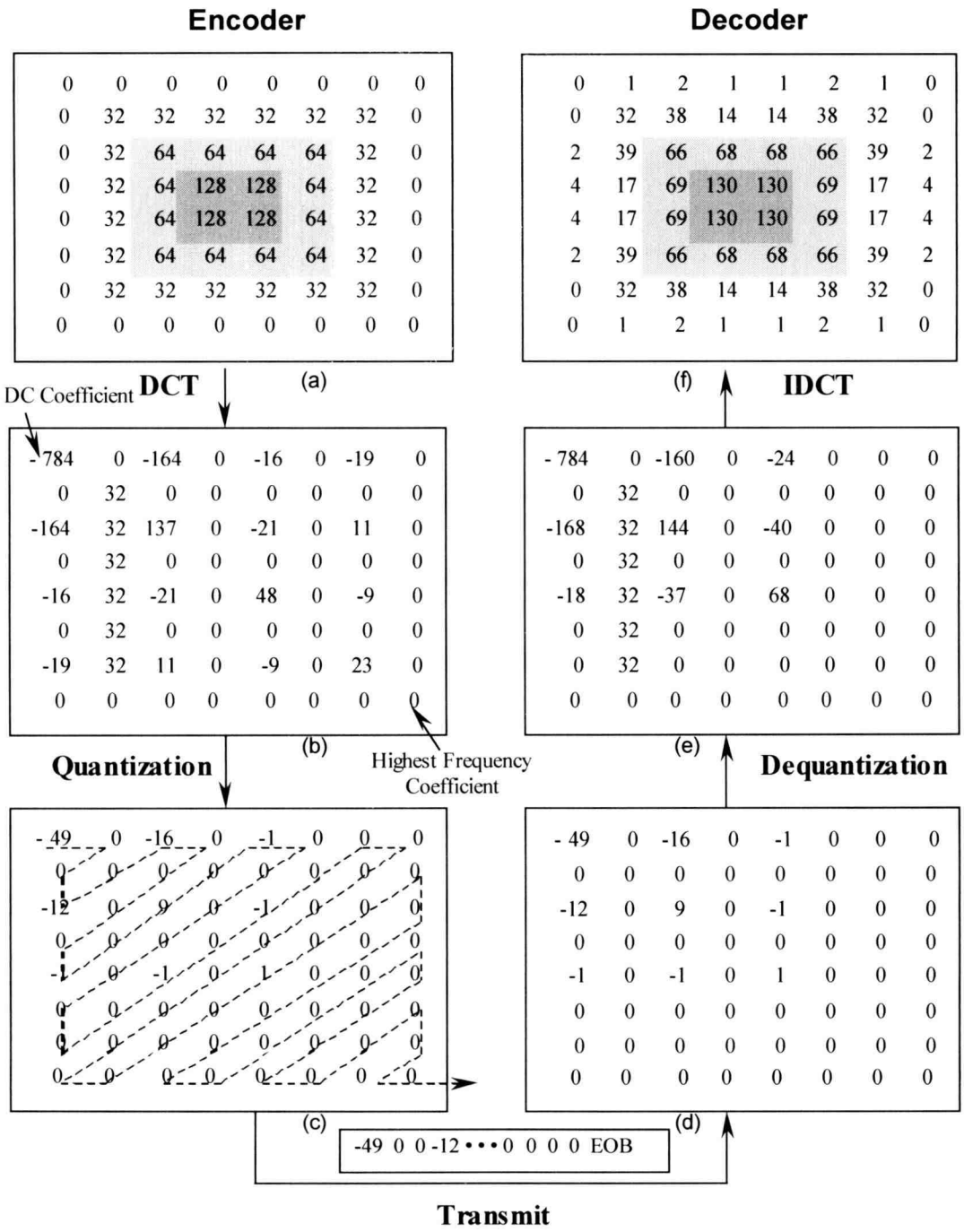


Figure 2.7: JPEG 8×8 Pixel Block Coding [6]

Table 2.2 shows a sample quantization table. For example, DC coefficient is quantized to integer number - 49, which is calculated by - 784 divided by 16, the first row of first column of the quantization table.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 2.2: Quantization Table [7]

The quantized coefficients are further compressed by the entropy coding such as Huffman coding and arithmetic coding. To get better compression ratio, JPEG reorders the sequence of the quantized DCT coefficients in a zigzag sequence, shown in Figure 2.7.c. By doing this, the zero coefficients are grouped well. Finally, the decoding sequence is reverse the order of the encoding sequence.

2.2.2 MPEG

In 1998, the Moving Picture Experts Group (MPEG) began working to develop international standards for digital audio and video transmission and storage. They produced MPEG-1 and MPEG-2, and MPEG-4 will be introduced in 1998 [6]. Table 2.3 shows these standards.

	MPEG-1	MPEG-2	MPEG-4
Final Draft	1992	1995	1998
Data Rate	≤ 1.86 Mbps	≤ 4 Mbps	≤ 64 Kbps
Resolution - Maximum	720 × 576	1920 × 1152	na
- Typical	352 × 240	Varies by Application	na
Frame Rate - Maximum	30 fps	60 fps	na
Coding Methods	DCT & BMCP (bidirectional motion-compensated prediction)	DCT & BMCP	DCT & BMCP ? Object Oriented ?
Applications	< SDTV-quality video storage on CD-ROM & transmission	SDTV → HDTV-quality storage & transmission	Video on PSTN and mobile networks Video on low-capacity storage devices

Table 2.3: MPEG Parameters [6]

MPEG-1 was designed for playback of stored multimedia and, by performing significant between-frame compression on limited-resolution pictures, aimed to get visually acceptable picture quality at the 1.2 Mbps video data rate provided by CD-ROM. In contrast, MPEG-2 is intended for both playback and conferencing, providing considerable flexibility in the amount of between-frame processing and the use of much higher-resolution pictures at higher data rates. MPEG-2 can produce the video quality needed for multimedia entertainment piped to the home and for more demanding business and scientific applications as well. MPEG-2 also supports the picture resolution and quality needed for HDTV. MPEG-4, still in the process of becoming a standard, has a different set of objectives. It is to provide video on low-bandwidth transmission links or for low-capacity storage devices where no existing standardized video compression algorithm has proved satisfactory [6].

The MPEG video compression standard has the identical algorithms to the JPEG algorithm during the initial stages. In addition to the JPEG algorithm, MPEG applies several more levels of block-based *motion-compensation* techniques to reduce *temporal*

redundancy that all video sequences contain because persons or objects in the scene do not move very much from frame to frame. Motion compensation compression and coding discard frames whose absence would not significantly deteriorate the perception of the human eye and brain.

Moving pictures consist of sequences of *video pictures* or *frames* that are played back a fixed number of frames per second. Furthermore, to achieve the requirement of random access, a set of pictures can be defined to form a Group of Pictures (GOP), consisting of one or more of the three types of pictures, which are intrapictures (I), unidirectionally predicted pictures (P), and bidirectionally predicted pictures (B) [8].

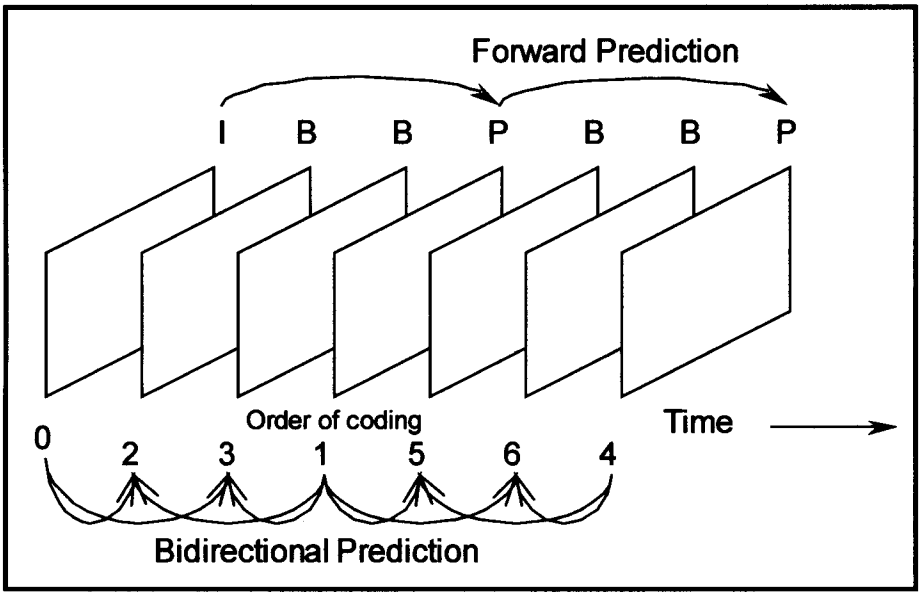


Figure 2.8: MPEG Group of Pictures (GOP) [5]

Figure 2.8 shows a MPEG GOP. A GOP consists of consecutive pictures that begin with an I-picture. The I-picture is coded without any reference to any other picture in the group. I-pictures can be placed anywhere in the sequence and can be utilized for random access to the sequence. P-pictures are interframe coded with a reference to the closest past I- or P-pictures, and motion compensated prediction is used for interframe coding. P-pictures provide more compression and serve as references for bi-directional

prediction for coding B-pictures (those between I- and P-pictures). B-pictures are coded using both past and future pictures as reference. They provide the most compression. B-pictures are never used as references. In other words, an I-picture or P-picture is utilized to code the current B-picture by using motion compensation.

Motion compensation is the basis for the MPEG compression algorithm. The idea is to reduce the number of bits to be transmitted by finding the temporal redundancy that exists between frames in the video data stream. The intensity and color of an object will change only slightly from frame to frame. Also, if the object moves, its motion is slight. Therefore, when coding a new frame, it uses frames that have already been encoded to predict what the object will be and where it will be. To make it easier to compare frames, a frame is not encoded as a whole. Instead, it is split into blocks, and the blocks are encoded. For each block in the frame to be encoded (that is, the current frame being addressed), the best matching block in the reference frame (for example, the I-frame) is searched among a number of candidate blocks. For each block, a motion vector is generated. A motion vector may be viewed as an analytical indication of the new location in the frame being encoded from an existing block in the reference frame. In a sense this is an attempt to match up the new location of a moved object. Figure 2.9 shows the motion compensated prediction.

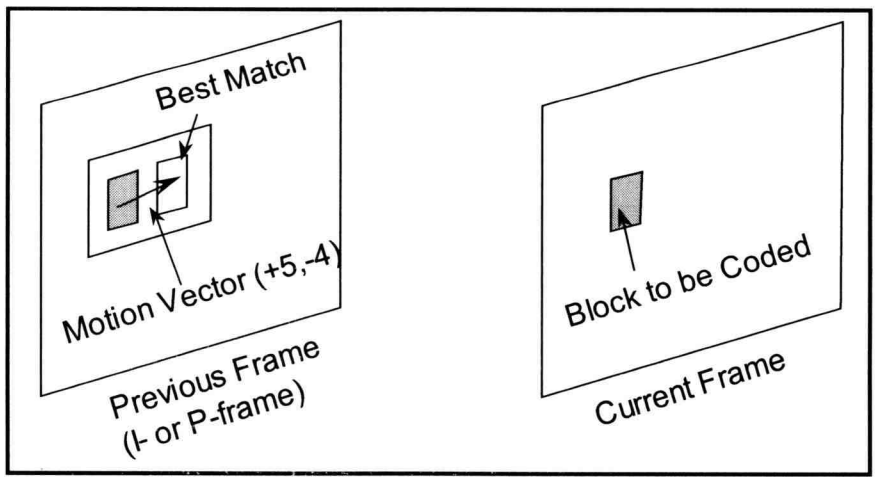


Figure 2.9: Motion Compensated Prediction [6].

The process of matching up can be based on prediction or interpolation. Prediction requires only the current frame and the reference frame. Based on the motion vector values generated, the prediction approach attempts to find the relative new position of the object and confirms it by comparing some blocks exhaustively. The motion vector is a pair of x-offset and y-offset, that specifies the differences from current block. In the interpolation approach, the motion vectors are generated in relation to two reference frames, one from the past and the next predicted frame. The best-matching blocks in both reference frames are searched, and the average is taken as the position of the block in the current frame.

The MPEG standard allows the encoder to choose the frequency of occurrence of I-, P-, and B-pictures based on application-specific needs for random accessibility, coding delay, visual image fidelity, and compressed video stream bit rates. Coding the bit stream with only I-pictures improves random accessibility and editibility but achieves low compression. Coding with only I- and P-pictures achieves moderate compression while still allowing random access and fast-forward/fast-reverse searches through the encoded bit stream. Using I-, P-, and B-pictures provides the most compression, as B-pictures can be coded in very few bits, leaving more bits for I- or P-pictures. It is advantageous to use more B-pictures but, as can be deduced from Figure 2.8, there is a drawback. When the number of B-pictures in a GOP is increased, the random access points into the video stream grow further apart and the encoding (and decoding too) delay time grows longer, characteristics that many applications cannot tolerate [6].

3 BENEFITS OF SIMD APPROACH

This chapter explains how SIMD (Single Instruction Multiple Data) model such as the VIS™ from Sun Microsystems or the MMX™ from Intel provide the performance improvement on multimedia applications. This chapter also presents some examples that highlight the performance benefits of SIMD model on multimedia processing.

3.1 Introduction

The multimedia applications, which include MPEG 1/2 video encoder/decoder, music synthesis, speech compression, speech recognition, image processing, 3-D graphics in games, video conferencing, modem, and audio applications, are built out of a few key computing-intensive routines. Where the applications spend most of its execution time to that routines and have the following common characteristics [9]:

- Small, native data types (for example, 8-bit pixels)
- Regular and recurring memory access patterns
- Localized, recurring operations performed on the data computing-intensive

Most of the multimedia data operands' sizes are either 8-bits or 16-bits, and multimedia processing typically involves performing the same computation on a large number of adjacent data elements. This common behavior is well suited with SIMD model of computation because the data elements (8 and 16-bits) can be packed together and operated in parallel. Figure 3.1 illustrates the performance advantages of a packed four 8-bit by 16-bit data multiplication. It performs four 8×16 multiplication in a single cycle instead of four separate multiplication. This type of computation is needed when manipulating the images. For example, alpha blending blends two images together. The formula of this function is as follows:

$$dst = (alpha/256) * s1 + (1-alpah/256) * s2;$$

For each pair of corresponding pixels in two images 's1' and 's2', a corresponding pixel is computed from a third control image 'alpha', which is loaded as a 16-bit variable. Without SIMD model instruction, only one multiplication can be performed, but SIMD model instruction shown in Figure 3.1 can perform four multiplication in a cycle.

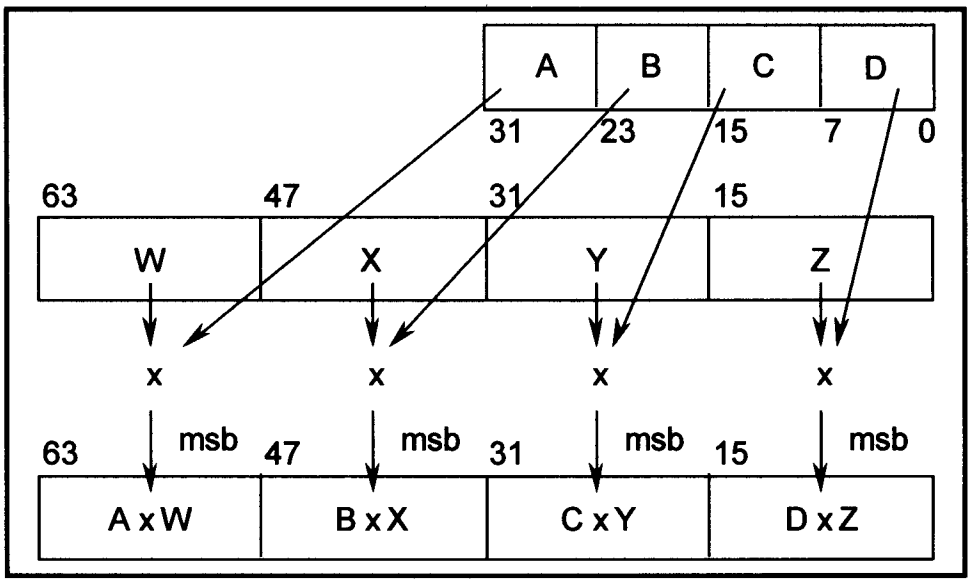


Figure 3.1: Four Multiplication Performed in a Single Cycle [10]

3.2 Data Formats for Graphics

The data types used in SIMD model are implementation dependent, but VIS and MMX chose 64-bits based on their design considerations. They use *partitioned data format* or *packed data format* to store pixel information. Usually, pixel information is stored as four 8-bit integer values. Typically, these four values represent the RGB color components and the alpha information. Fixed data formats provide an intermediate format with enough precision and dynamic range for filtering and simple image computations on pixel values. For example, 3-D rendering requires a smooth interpolation between the maximum and minimum intensity. By expanding an 8-bit color component to a 16-bit format, this operation is able to retain sufficient precision for a

smooth interpolation. In the case of VIS, the vis_fexpand() instruction converts 8-bit element to 16-bit element by inserting four zeroes to the right and to the left of each byte. Figure 3.2 shows these two data formats.

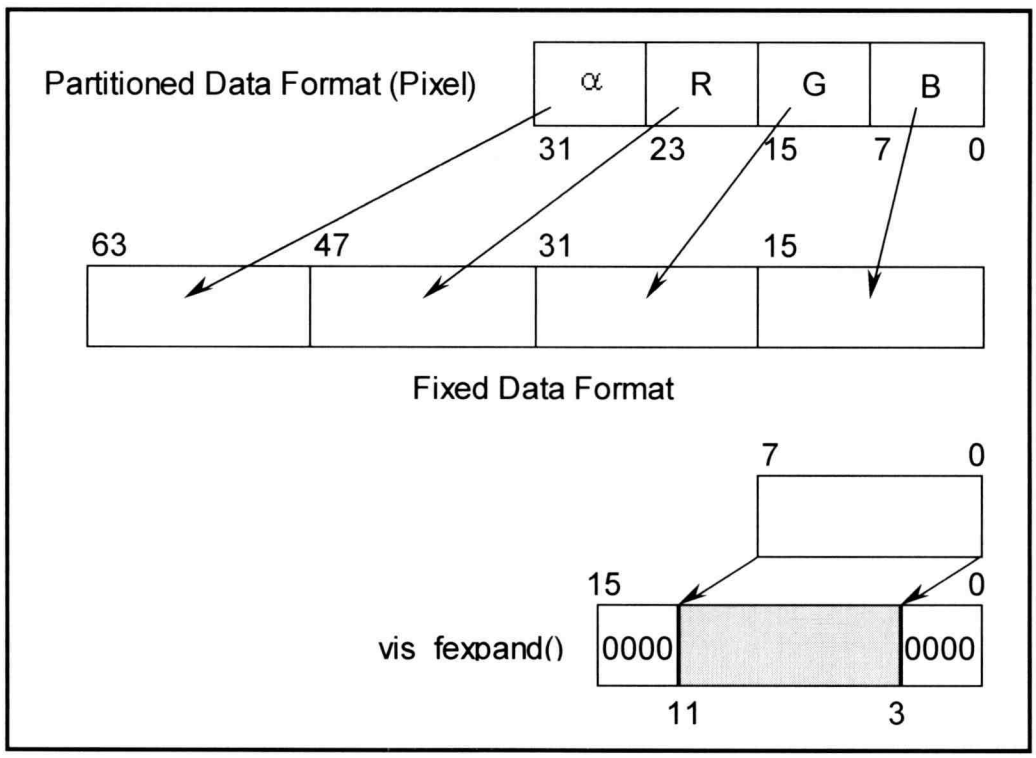


Figure 3.2: Data Formats [10]

3.3 Performance Advantages in Multimedia Applications

Let us inspect some examples of multimedia applications to understand how SIMD-style instruction set boosts the performance.

3.3.1 Alpha Blending

Alpha Blending is an application where two images are blended together and is used by software game developers. For example, alpha blending allows racing cars to

drive realistically through fog or smoke, provides a more realistic view of fishes in water, or a rabbit in a translucent tube. The alpha portion of the pixel value (RGBα) is a measure of how transparent the pixel is. The equation to calculate the blended image is as follows:

$$dst = (alpha/256) * s1 + (1\text{-}alpah/256) * s2 \quad [9]$$
$$= (s1\text{-}s2) * (alpah/256) + s1$$

For example, image s1 and s2 are to be blended, and the alpha value is 255, the image s1 is fully opaque and we won't be able to see the image s2. If the alpha value is 0, the image s1 is fully transparent and has no effect on the result pixel values. Figure 3.3 illustrates the images of a flower and a swan blending when alpha is 230. When the alpha value is 230, the resulting picture is 90 percent flower and 10 percent swan. On close examination, some of the swan image appears in the resulting picture.

Figure 3.3: Alpha Blending [11]

Figure 3.4 shows the main loop of alpha blending from VIS User's Guide [9]. The 'db1_s1' and 'db1_s2' represent 64-bit source data of images s1 and s2, and each source data has two pixels (α, R, G, B, α, R, G, B). The VIS code executes two images at a time, where the regular application executes only one component at a time. Therefore, VIS can perform six times more execution in a cycle. The algorithm used in VIS code is as follows:

First, the high half of the source data is read and the 8-bit component is expanded to 16-bit component, and the image s2 is subtracted by the image s1. Second, the result of the subtraction is multiplied by the α value, and added to image s1. The lower half of the source data is processed in the same manner as the higher half. As a final step, the 16-bit component of the computation result of the higher half and lower half are packed to an 8-bit, and stored as a 64-bit variable.

```
db1_s1_e = vis_fexpand (vis_read_hi(db1_s1)); /* High half of image s1 */
db1_s2_e = vis_fexpand (vis_read_hi(db1_s2)); /* High half of image s2 */
db1_tmp2 = vis_fpsub16 (db1_s2_e, db1_s1_e); /* s2 - s1 */
 /* alpha * (s2 - s1) */
db1_tmp1 = vis_fmul8x16 (vis_read_hi(quad_a), db1_tmp2);
db1_sum1 = vis_fpadd16 (db1_s1_e, db1_tmp1); /* s1 + (s2-s1) * alpha */

db1_s1_e = vis_fexpand (vis_read_lo(db1_s1)); /* Low half of image s1 */
db1_s2_e = vis_fexpand (vis_read_lo(db1_s2)); /* Low half of image s2 */
db1_tmp2 = vis_fpsub16 (db1_s2_e, db1_s1_e);
db1_tmp1 = vis_fmul8x16 (vis_read_lo(quad_a), db1_tmp2);
db1_sum2 = vis_fpadd16 (db1_s1_e, db1_tmp1);

/* rd [63-32] = packed sum1, rd [31-0] = packed sum2 */
db1_d = vis_freg_pair (vis_fpack16 (db1_sum1), vis_fpack16 (db1_sum2));
```

Figure 3.4: Blending two images using VIS [10].

The vis_fpack16 instruction is controlled by the Graphics Status Register (GSR), which allows programmers to clip the input values to the dynamic range of the output format as the application demands. The vis_fpack16 instruction takes four 16-bit values, scales, truncates and clips them into four 8-bit components. This is accomplished by left shifting the 16-bit component as determined from the scale factor field of GSR and truncating to an 8-bit unsigned integer by rounding and then discarding the least significant digits. If the resulting value is negative, zero is returned, and if the value is

greater than 255, then 255 is returned. Otherwise, the scaled value is returned. Figure 3.5 illustrates vis_fpack16 operation.

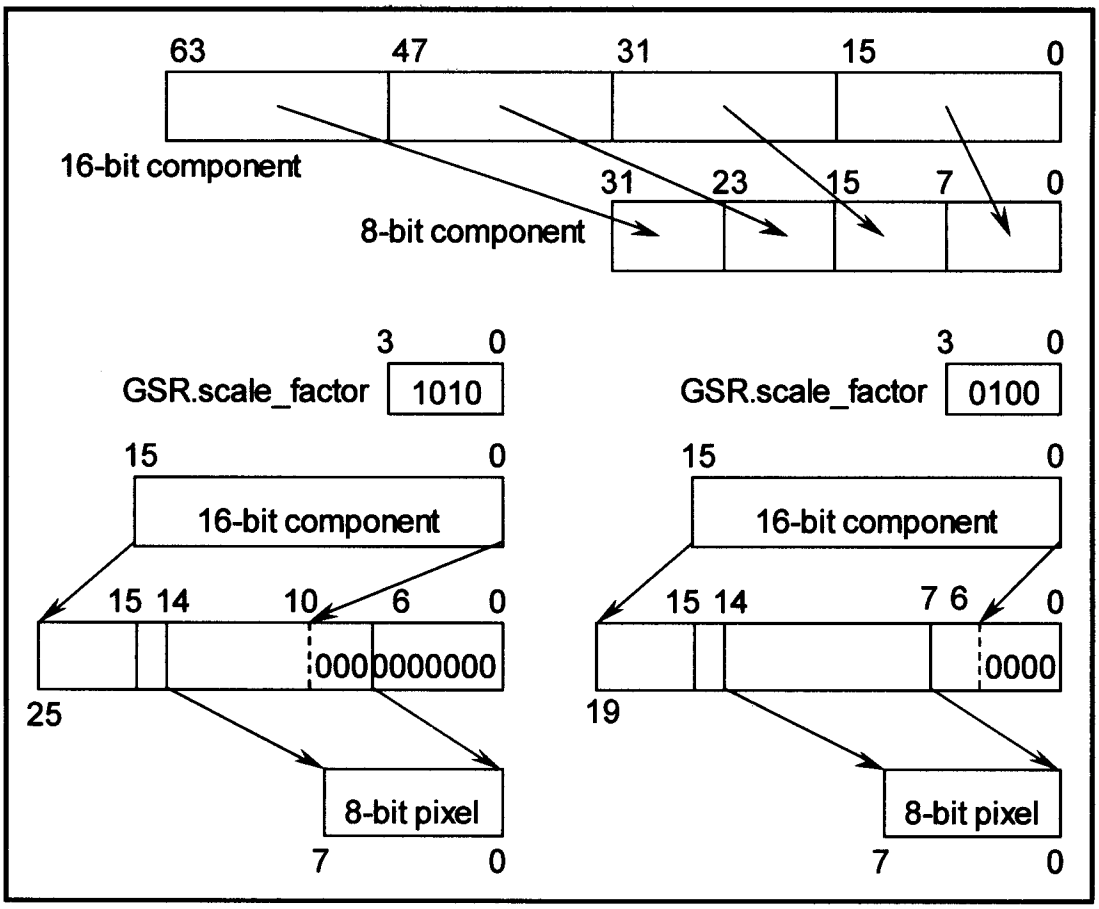


Figure 3.5: vis_fpack16() Operation [10]

Table 3.1 shows the efficiency of the SIMD model of computation. It compares the instruction counts with and without MMX technology for alpha blending. To perform the alpha blending operation, MMX requires 1 billion fewer instructions than without MMX.

Operation	Number of Instructions without MMX Instructions	Number of Instructions with MMX Instructions
Load	470 million	117 million
Unpack	--	117 million
Multiply	470 million	117 million
Add	235 million	58 million
Pack	--	58 million
Store	235 million	58 million
Total	1.4 billion	525 million

Table 3.1: Comparing Instruction Counts with and without MMX [11]

3.3.2 Chroma Keying

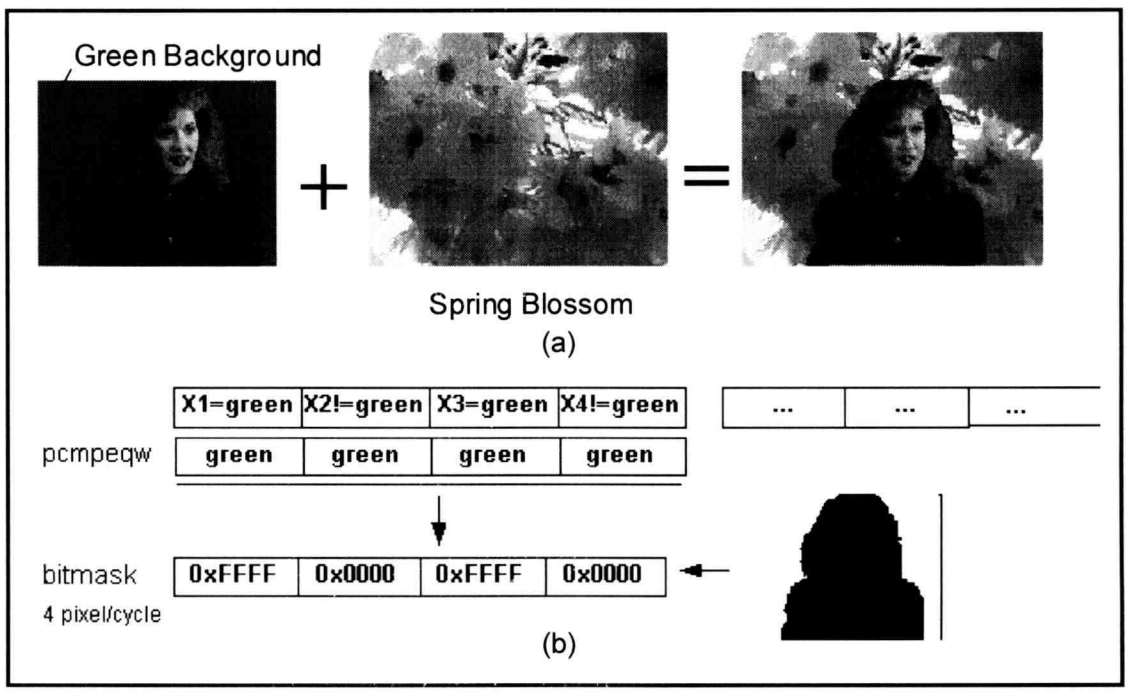


Figure 3.6: Chroma Keying [11].

Chroma keying is an image-composition method used by the television industry shown in Figure 3.6. We can easily see a weatherman who seems to be standing in front of a weather map. The weatherman stands in front of green wall shown in Figure 3.6.a, called a *green screen* [12]. Her image with the green background is chroma-key overlaid onto the video from the remote site. The remote image is displayed wherever the weatherman's image is green, meaning the wall behind her. This gives the impression of the weatherman standing in front of whatever is going on. The color green in the weather map was chosen because it is the most opposite hue (the degree of the brightness) to the skin color.

Figure 3.6.b depicts how the woman from the background is chosen with the MMX instruction. This example executes 16-bit pixels in parallel. The top row of the data of the Figure 3.6.b is generated by taking four pixels from the picture with the woman on a green background, and represents pixels that alternate between green, not green, green, and not green. The compare instruction PCMPEQW (Packed Compare Word for Equality) builds a mask for that data. That mask is a sequence of words that are all ones or all zeros representing the Boolean values of true and false. To extract the shape of a woman, the pixels of the zero masks, which represent a woman, are kept, and the pixels of the one masks are discarded, and the shape of a woman is shown using a shadow picture in Figure 3.6.b.

Figure 3.7 shows the process of overlapping a woman on the spring blossom. The bitmask is now used on the same four pixels from the picture with the woman and the equivalent four pixels from the spring blossom. The PANDN (Bit wise Logical AND NOT) and PAND (Bit wise Logical AND) instructions use the bitmask to identify which pixels to keep from the picture of the woman and the picture of the spring blossom. The first row of Figure 3.7 shows the bitmask and PANDN instruction inverts the bitmask and performs a logical AND with the four pixels of the picture with the woman on a green background. So the result is just X2 and X4 field which are not green pixels, i.e., the shape of the woman. The PAND instruction performs a logical AND with the four pixels of the picture of the spring blossom, and the results are Y1 and Y3 field, which are not woman's shape. And the POR (Bit wise Logical OR) instruction builds the final

overlapped picture. In this example, four pixels were mapped using only four MMX instructions without any branches. Without MMX, each pixel has to be processed separately and requires a conditional branch when choosing which pixel is selected to overlap. Therefore, the MMX instructions provide a significant performance advantage.

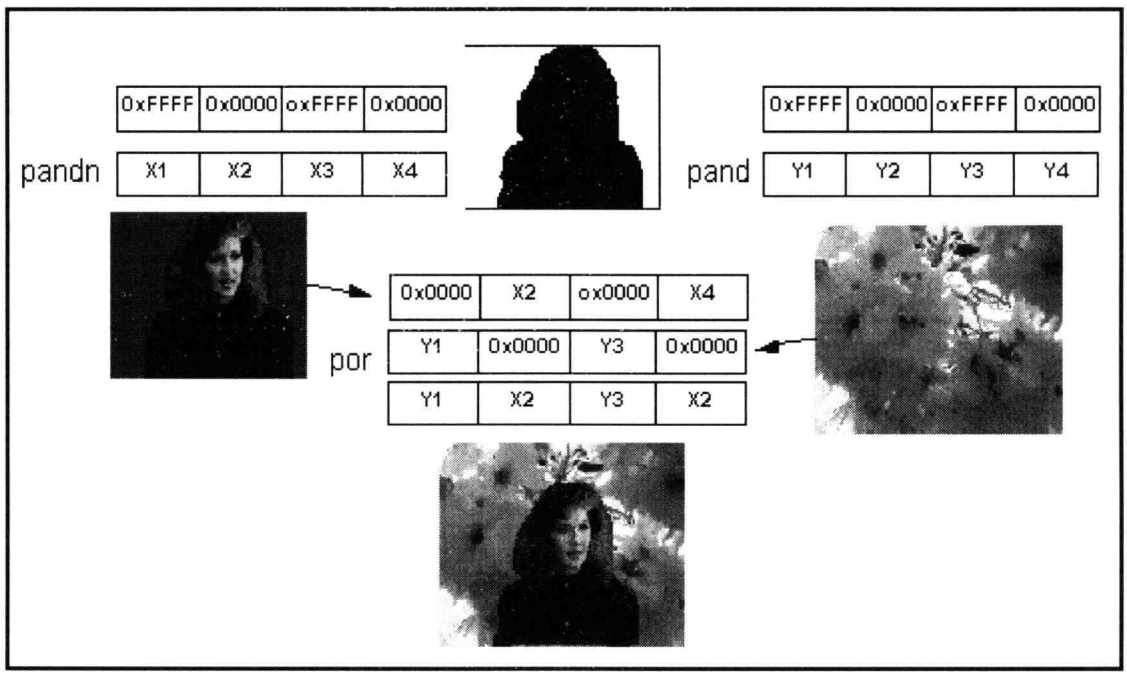


Figure 3.7: Overlapping a Woman on the Spring Blossom [11].

3.3.3 Sum of Absolute Differences in Motion Compensation

As already discussed in Chapter 2.2.2, the MPEG is the standard for moving motion compression technique. Among the stages of MPEG compression, calculating the motion vector, which is called motion compensation, is the most computing intensive stage. The motion compensation requires the calculation of an absolute sum of the differences between pixel values of two different 16×16 blocks of the frames, and finds the best match.

Figure 3.8 shows the main loop to compute the sum of differences with C code. The src1, src2, src1Ptr, and src2Ptr are pointers to data in the two frames. The processor

spends most of the time in accum += abs(*src1 - *src2), which accumulates the absolute sum of differences. This code assumes the input frames to be 8-bit gray scale.

```
accum = 0;
for (j =0; j < 16; j++){
        for (i = 0; i < 16; i++){
                accum += abs (*src1 - *src2);
                src1++; src2++;
        }
        src1Ptr = src1 = src1Ptr + stride1;
        src2Ptr = src2 = src2Ptr + stride2;
}
```

Figure 3.8: C Code of 16×16 Sum of Absolute Differences [13].

Figure 3.9 shows the same loop code when using VIS instructions. A single VIS instruction, vis_pdist, performs the sum of absolute differences. Each vis_pdist instruction replaces approximately 48 regular C instructions, and the 32 vis_pdist instructions, which are required for a 16×16 pixel block, replace approximately 1,500 conventional instructions [13]. The VIS code in Figure 3.9 provides a 5.5 times speedup over the C code. In terms of cycles, the VIS code required only 441 cycles, compared to the C code which required 2,429 cycles. The vis_pdist instruction will be discussed further in the next Chapter.

```
for (j = 0; j < 16; j++){
        for(i = 0; i < 2; i++){
                /* load 8 bytes of source data from frame1 */
                /* load 8 bytes of source data from frame2 */
                accum = vis_pdist (sd1, sd2, accum);
                sa1 += 8; /* next 8 bytes from frame1 */
                sa2 += 8; /* next 8 bytes from frame2 */
        }
}
```

Figure 3.9: Sum of Absolute Differences using VIS [13].

4 THE ARCHITECTURE OF ON-CHIP MULTIMEDIA ENHANCED MULTIPROCESSOR "RAPTOR"

Raptor is an experimental architecture proposed by ETRI shown in Figure 4.1. It is an on-chip multiprocessor, which has quad 64-bit processors that share a common GCU. The External Interface (EI) and four GP blocks are being implemented, and the simulator, called Rapsim, which simulates Raptor except GCU, is being developed by ETRI. I have designed GCU architecture and defined GCU ISA, and implemented a simulator to simulate GCU.

Chapter 4.1 discusses the organization of Raptor, and Chapter 4.2 shows how to resolve dependencies between GP and GCU. GCU ISA and GCU architecture are discussed in Chapter 4.3 and Chapter 4.4, and five stage pipeline of GCU is explained in Chapter 4.5. The simulation study of GCU will be discussed in detail in Chapter 5.

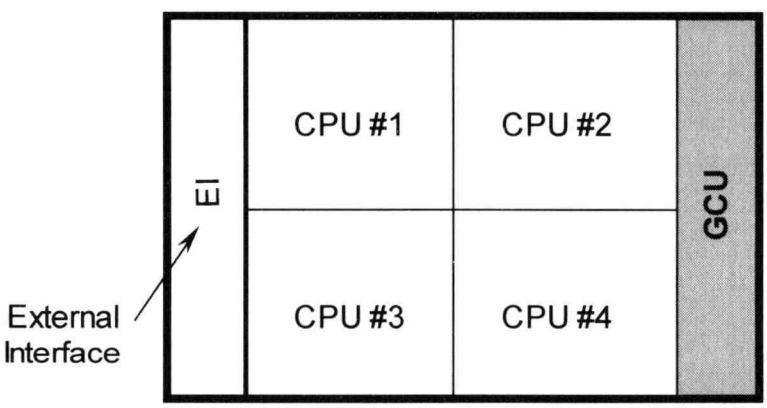


Figure 4.1: Organization of the Raptor

4.1 Organization of Raptor

Raptor has quad General Processors (GP), each having the same structure as UltraSPARC^{TM} of Sun Microsystems except the Floating Point Unit (FPU). Figure 4.2 shows the overall organization of the on-chip multimedia multiprocessor, Raptor.

The FPU of UltraSPARC has the Graphics Unit consisting of the Graphics Adder and Graphics Multiplier to execute VIS instructions. In Raptor, the four GPs share a single GCU. The basic design of GCU consists of four Instruction Buffers (IBs), four Graphics Register Files (GRFs), and four Functional Units (FUs). It executes a variant of VIS instruction set, which will be defined in the latter part of this Chapter.

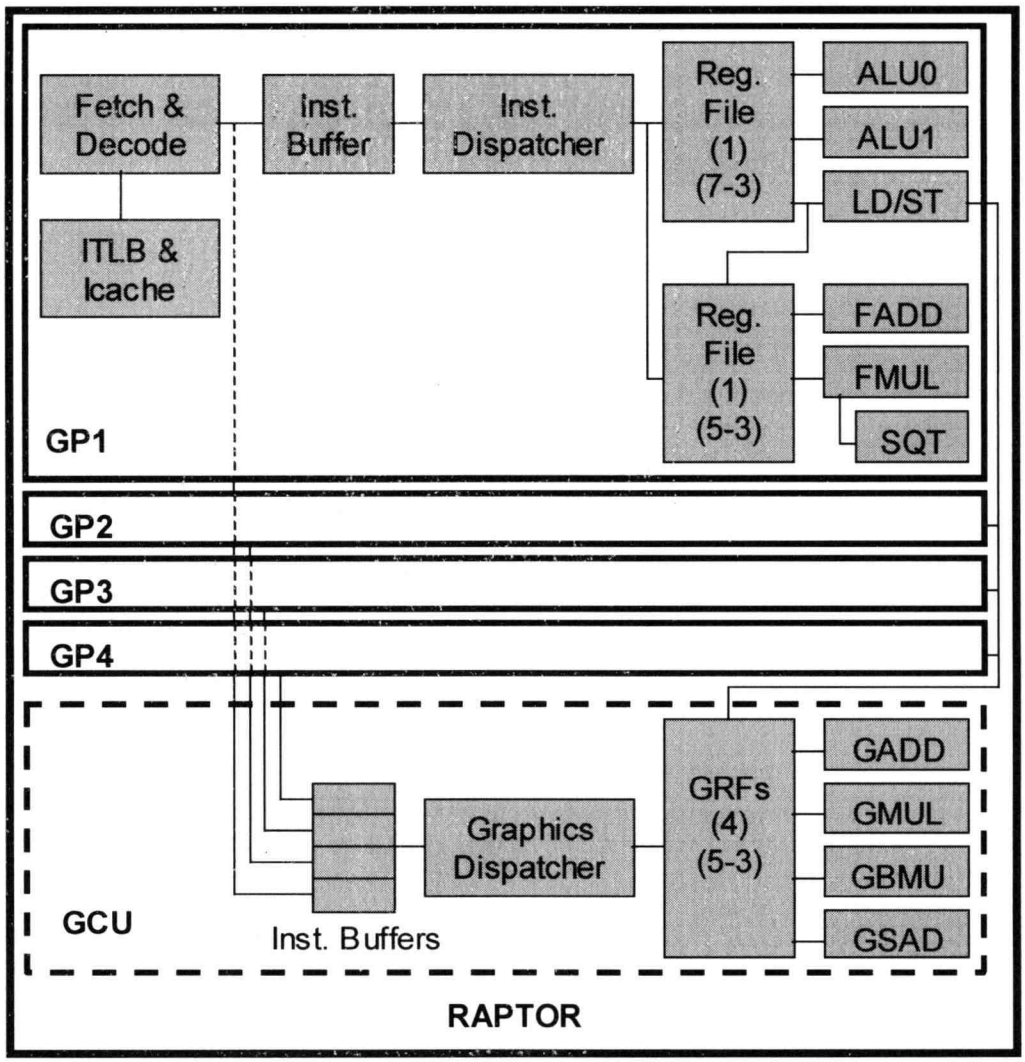


Figure 4.2: Block Diagram of Raptor

The GP pre-decodes instructions and identifies the GCU instructions and passes them to IBs of GCU. From the IBs, multiple instructions that have no dependencies are issued to the FUs. The GCU has four sets of Reservation Stations (RSs), GRF, and Reorder Buffers (ROBs) to support four independent instruction streams from GPs. However, the multiple threads share a common set of FUs. The Load/Store Unit of a GP is responsible for loading and storing graphics data to/from the GRFs, so all the data that GCU needs are obtained only through GRFs.

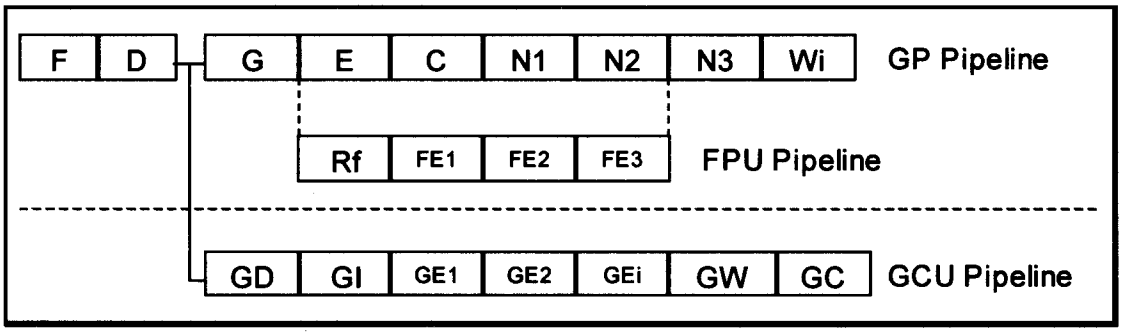


Figure 4.3: Pipeline of Raptor.

Raptor's pipeline structure is similar to an UltraSPARC, except it has a separate pipeline for the GCU. Figure 4.3 shows the pipeline of Raptor. Each field represents one stage and their meanings are as follows:

F: Fetch Instructions.

D: Decode Instructions. Send GCU Instructions to GCU.

G: Grouping Instructions and Dispatch Integer/Load/Store Instructions.

E: Execute in Integer ALU. Calculate Address for Load/Store.

C: Access DTLB and D-Cache. Resolve Branch on Integer Operation.

Rf: Decode more in FPU.

GD: Dispatch and Schedule GCU Instructions.

GI: Issue GCU Instructions.

FEi: Execute in FPU.

GEi: Execute in GCU.

N1: Determine D-Cache hit/miss

N2: Wait for the Floating Point Pipe.

N3: Resolve Traps.

Wi: Write results to the corresponding Register File.

GW: GCU Write Back and Data Forward.

GC: GCU Commit. Write results to the corresponding GRF.

The D stage of the Integer Pipeline (IP) decodes the instructions and sends the GCU Instructions to the IB of GCU, and the G stage in the IP checks dependencies and groups instructions to be issued together. The N1 and N2 stages of IP allow more time for the FPU to complete operations, while N3 stage is used to synchronize two pipelines before the write stage.

4.2 Resolving Dependencies between GP and GCU

Raptor issues out-of-order and completes in-order. Dependencies within a GP are resolved by the Grouping stage, and the dependencies within the GCU are resolved using its own RSs and ROBs. However, there are also dependencies between GPs and the GCU. For example, in the instruction sequence shown in Figure 4.4, $g10 in the Load instruction has RAW (Read After Write) dependency with GCU instruction GADD.

```
Load  $g10, $g2, 100;                          (1)
GADD $g10, $g5, $g10;   /* Graphics ADD */     (2)
Store  $g5,  $g10, 100;                        (3)
GMUL  $g7,  $g3, $g8; /* Graphics Multiply */  (4)
```

Figure 4.4: Sample Instruction Stream.

In order to resolve RAW dependencies, the GRF has two additional fields: a *valid_gp* field and a *valid_gcu* field. When a GP dispatches the Load instruction in Figure 4.4 in the G stage, the *valid_gp* field of $g10 of GRF is set to zero, which means that the $g10 does not have a valid value. Therefore, GCU cannot issue the GADD instruction until the Load operation is completed and the *valid_gp* field is set. When the Load operation completes, *valid_gp* field of $g10 is set to one indicating the value is now available. The dependencies between the GADD and the Store instructions are checked in the G pipeline stage of GP. The G stage checks the dependencies between the GADD and the Store instructions and sets the *valid_gcu* to zero indicating the $g10 is not valid, and eliminates the entry of GADD when grouping the instruction for IP and FP Pipelines. The reason that the G stage checks this inter-unit dependency is because of timing problems. If GADD is stalled in the IB of GCU, and cannot be checked for the dependency in Dispatch stage of GCU, the Store instruction might use invalid result from the Load instruction.

4.3 GCU Instruction Set

The GCU Instruction set is based on a combination of VIS™ of SUN Microsystems and MMX™ of Intel. The VIS can be categorized into conversion instructions, arithmetic/logical instructions, address manipulation instructions, memory access instructions, and a motion compensation instruction. Among these instructions, address manipulation instructions such as ALIGNADDR operates on the integer register file, and memory access instructions such as Block Load and Block Store instructions transfers data between memory and floating-point registers of the UltraSPARC. Therefore, the GCU ISA does not include these instructions because GCU executes only a data from the GRF. MMX has several instructions that VIS does not support. The saturation mode addition/subtraction and shift instructions of MMX are useful in manipulating pixels. Therefore, some of these instructions were integrated into the GCU ISA.

4.3.1 Data Types

The most commonly used data types in multimedia applications are 8-bit and 16-bit unsigned integers. Pixels typically consist of four 8-bit unsigned integers represented within a 32-bit word. For example, a true color often consists of 8-bit values for red, green, blue, and a transparency coefficient α. To support this characteristic, GCU has four different kinds of *partitioned data types*. Figure 4.5 shows the partitioned data types. For example, partitioned byte can hold two pixels, which is composed of 8-bit R, G, B, and α components, and partitioned word can contain four 16-bit elements.

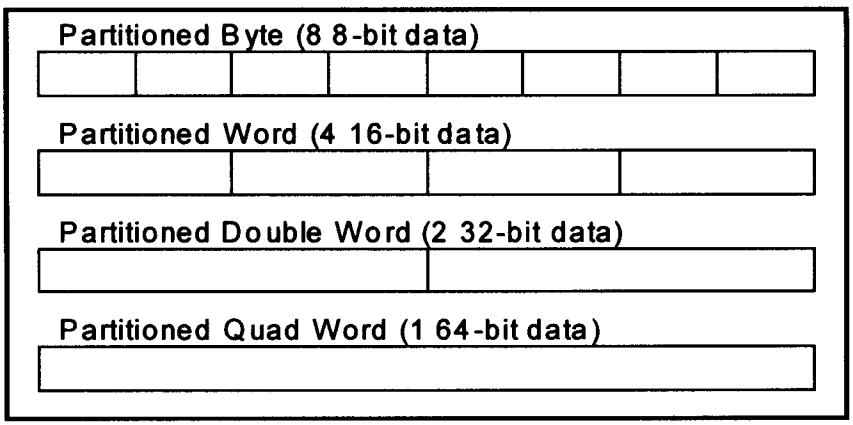


Figure 4.5: Partitioned Data Types

4.3.2 GCU Instruction Format

Figure 4.6 shows the format of GCU instructions. Each GP pre-decodes instructions and checks the *Main-op* field. If it detects GCU instructions, they are sent to the IBs of the GCU.

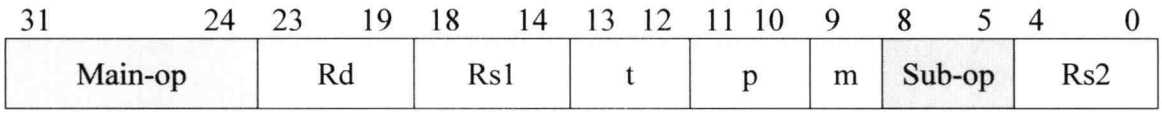


Figure 4.6: GCU Instruction Format

- *Main-op*: The field is for detecting GCU Instructions. When instructions are dispatched to GCU, the M*ain-op* field is discarded.
- *Sub-op*: This field defines what operation is to be performed in GCU.
- *Rs1*, *Rs2*, and *Rd*: These fields identify two source registers and the destination register, respectively.
- *t*: This field is used to determine which FU is required for the operation, and Table 4.1 shows the tag field and the corresponding FU.

Tag field (*t*)	FU of GCU
00	GALU
01	GMUL
10	GBMU
11	GSAD

Table 4.1: Tag Field.

- *m*: The mode field shows the wrap-around/saturation, shift right/left, greater than/less than, double precision/single precision, and high/low of the result of the computation. For example, we can distinguish GMULH (Graphics Multiply High) and GMULL (Graphics Multiply Low) according their mode field.

- *p*: This field shows the partitioned data types, and Table 4.2 shows the contents of each field.

p field	Partitioned Data Type
00	Bytes
01	Words
10	Double Words
11	Quad Words

Table 4.2: *p* Field.

4.3.3 GCU Instruction Set Category

GCU instructions are grouped based FU categories. GCU instruction uses only registers for the source operands. Figure 4.7 shows how instruction names are composed, and in what follows, how some of the GCU instructions operate will be illustrated. A detailed description of VIS ISA can be found in VIS User's Guide, Sun Microsystems, 1997 [10].

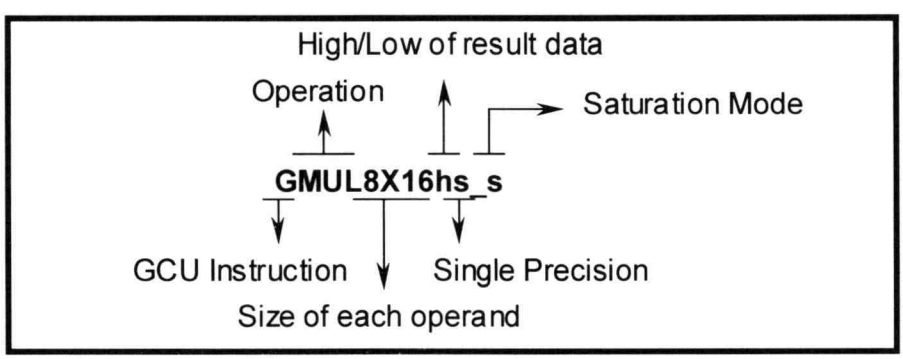


Figure 4.7: Composition of GCU Instruction Name

- GMUL (Graphics Multiply)

Table 4.3 shows the GMUL instruction set. The issue latency of GMUL instructions is one cycle and the operation latency is three cycles.

Mnemonic	SubOp	t	m	Operation	Description
GMUL8×16	0000	01		Rd = (Rs1 × Rs2)	Multiply 8×16
GMUL8×16h	0001	01	0	$Rd = (Rs1 \times Rs2)_H$	Multiply 8×16 High
GMUL8×16l	0001	01	1	$Rd = (Rs1 \times Rs2)_L$	Multiply 8×16 Low
GMUL8×16h_half	0001	01	1	$Rd = (Rs1 \times Rs2)_{Half}$	Multiply 8×16 High Half
GMUL8×16l_half	0001	01	1	$Rd = (Rs1 \times Rs2)_{Half}$	Multiply 8×16 Low Half
GMUL8x16hs	0001	01	1	$Rd = (Rs1 \times Rs2)_L$	Multiply 8×16 High Single Precision
GMUL8x16ls	0001	01	1	$Rd = (Rs1 \times Rs2)_L$	Multiply 8×16 Low Single Precision

Table 4.3: GMUL Instruction Set [10]

GMUL8×16 shown in Figure 4.8 multiplies each unsigned 8-bit component within the 32-bit pixel by the corresponding signed 16-bit fixed-point component within the scale and returns the upper 16-bits of the 24-bit product as a result. The 8^{th} bit is added by one and shifted eight bits to right to scale down 24-bit to 16-bit.

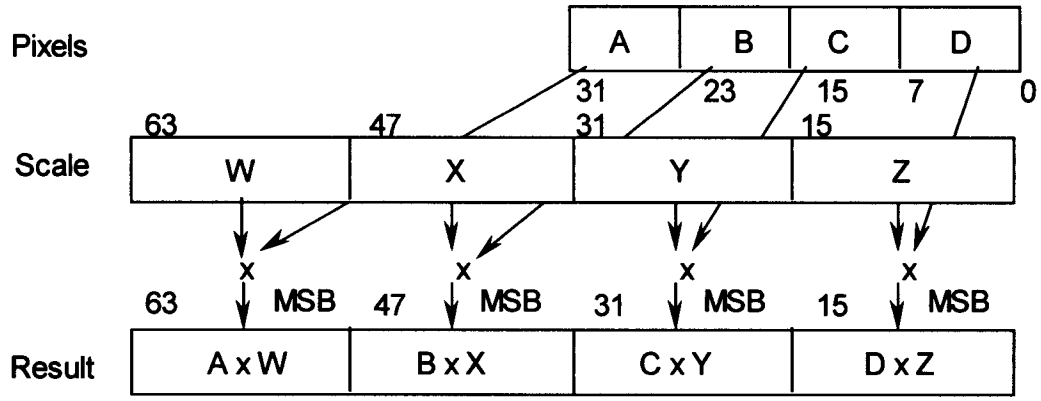


Figure 4.8: GMUL8×16 Operation [10]

Both GMUL8×16h_half and GMUL8×16l_half perform half a multiplication. GMUL8×16h_half multiplies the signed upper 8-bit of each 16-bit signed component by the corresponding 16-bit fixed-point signed component. The upper 16-bit of the 24-bit product is returned in a 16-bit partitioned result. This is the same as GMUL8×16. This operation is illustrated in Figure 4.9. GMUL8×16l_half on the other hand multiplies the unsigned lower 8-bits of each 16-bit element by the corresponding 16-bit element. Each 24-bit product is rounded at 16^{th} bit and 16-bits are shifted to right with sign-extended and the upper 16-bits of the sign extended value are returned.

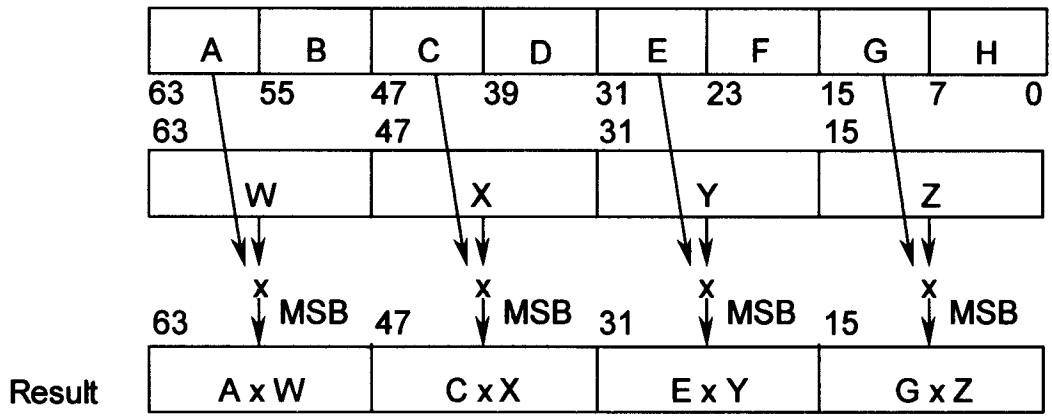


Figure 4.9: GMUL8×16h_half Operation [10]

- GALU (Graphics Arithmetic and Logical Unit)

Table 4.4 shows the GALU instruction set. The arithmetic operation supports saturation and wrap-around mode, and the issue and operation latency is one cycle. Saturation means that if an addition results in an overflow or a subtraction results in an underflow, the result is clamped to the largest or the smallest representable value. For an unsigned, 16-bit word, the largest and the smallest representable values are 0xFFFFh and 0x0000h, respectively. For a signed word, the largest and the smallest representable values are 0x7FFFh and 0x8000h, respectively. This saturation function is important for pixel calculations to prevent a black pixel from unexpectedly turning to a white pixel.

Mnemonic	SubOp	t	m	p	Operation	Description
GADD16	0000	00	0	01	Rd = Rs1 + Rs2	Addition Partitioned 16-bit with Wraparound
GSADD32	0000	00	0	10	Rd = Rs1 + Rs2	Addition Partitioned 32-bit with Wraparound
GADD16_s	0000	00	1	01	Rd = Rs1 + Rs2	Addition Partitioned 16-bit with Saturation
GADD32_s	0000	00	1	10	Rd = Rs1 + Rs2	Addition Partitioned 32-bit with Saturation
GADD16s	0001	00	0	01	Rd = Rs1 + Rs2	Addition Partitioned 16-bit with Wraparound, Single Precision
GADD32s	0001	00	0	10	Rd = Rs1 + Rs2	Addition Partitioned 32-bit with Wraparound, Single Precision

Table 4.4: GALU Instruction Set [10]

Mnemonic	SubOp	t	m	p	Operation	Description
GSUB16	0010	00	0	01	Rd = Rs1 - Rs2	Subtraction Partitioned 16-bit with Wraparound
GSUB32	0010	00	0	10	Rd = Rs1 – Rs2	Subtraction Partitioned 32-bit with Wraparound
GADD16s_s	0001	00	1	01	Rd = Rs1 + Rs2	Addition Partitioned 16-bit with Saturation, Single Precision
GADD32s_s	0001	00	1	10	Rd = Rs1 + Rs2	Addition Partitioned 32-bit with Saturation, Single Precision
GSUB16_s	0010	00	1	01	Rd = Rs1 - Rs2	Subtraction Partitioned 16-bit with Saturation
GSUB32_s	0010	00	1	10	Rd = Rs1 - Rs2	Subtraction Partitioned 32-bit with Saturation
GSUB16s	0011	00	0	01	Rd = Rs1 - Rs2	Subtraction Partitioned 16-bit with Wraparound, Single Precision
GSUB32s	0011	00	0	10	Rd = Rs1 - Rs2	Subtraction Partitioned 32-bit with Wraparound, Single Precision
GSUB16s_s	0011	00	1	01	Rd = Rs1 - Rs2	Subtraction Partitioned 16-bit with Saturation, Single Precision
GSUB32s_s	0011	00	1	10	Rd = Rs1 - Rs2	Subtraction Partitioned 16-bit with Saturation, Single Precision
GCMPEQ8	0100	00	0	00	Rd = Rs1 :: Rs2	Compare Equal Partitioned 8-bit

Table 4.4: GALU Instruction Set (Continued) [10]

Mnemonic	SubOp	t	m	p	Operation	Description
GCMPEQ16	0100	00	0	01	Rd = Rs1 :: Rs2	Compare Equal Partitioned 16-bit
GCMPEQ32	0100	00	0	10	Rd = Rs1 :: Rs2	Compare Equal Partitioned 32-bit
GCMPGT8	0101	00	0	00	Rd = Rs1 > Rs2	Compare Greater Than Partitioned 8-bit
GCMPGT16	0101	00	0	01	Rd = Rs1 > Rs2	Compare Greater Than Partitioned 16-bit
GCMPGT32	0101	00	0	10	Rd = Rs1 > Rs2	Compare Greater Than Partitioned 32-bit
GCMPLE8	0101	00	1	00	Rd = Rs1 < Rs2	Compare Less Than Partitioned 8-bit
GCMPLE16	0101	00	1	01	Rd = Rs1 < Rs2	Compare Less Than Partitioned 16-bit
GCMPLE32	0101	00	1	10	Rd = Rs1 < Rs2	Compare Less Than Partitioned 32-bit
GAND32	0110	00	0	10	Rd = Rs1 & Rs2	Bit wise Logical AND
GAND32s	0110	00	1	10	Rd = Rs1 & Rs2	Bit wise Logical AND Single Precision
GANDnot32	0111	00	0	10	Rd = Rs1 & ~Rs2	Bit wise Logical AND NOT
GANDnot32s	0111	00	1	10	Rd = Rs1 & ~Rs2	Bit wise Logical AND NOT Single Precision
GOR32	1000	00	0	10	Rd = Rs1 | Rs2	Bit wise Logical OR
GOR32s	1000	00	1	10	Rd = Rs1 | Rs2	Bit wise Logical OR Single Precision

Table 4.4: GALU Instruction Set (Continued) [10]

Mnemonic	SubOp	t	m	p	Operation	Description
GORnot32	1001	00	0	10	Rd = Rs1 | ~Rs2	Bit wise Logical OR NOT
GORnot32s	1001	00	1	10	Rd = Rs1 | ~Rs2	Bit wise Logical OR NOT Single Precision
GXOR32	1010	00	0	10	Rd = Rs1 ^ Rs2	Bit wise Logical XOR
GXOR32s	1010	00	1	10	Rd = Rs1 ^ Rs2	Bit wise Logical XOR Single Precision
GNOT32	1011	00	0	10	Rd = ~Rs1	Bit wise Logical NOT
GNOT32s	1011	00	1	10	Rd = ~Rs1	Bit wise Logical NOT Single Precision

Table 4.4: GALU Instruction Set (Continued) [10]

GADD16 and GSUB16 perform partitioned addition and subtraction between two 64-bit partitioned variables. These instructions are illustrated in Figure 4.10.

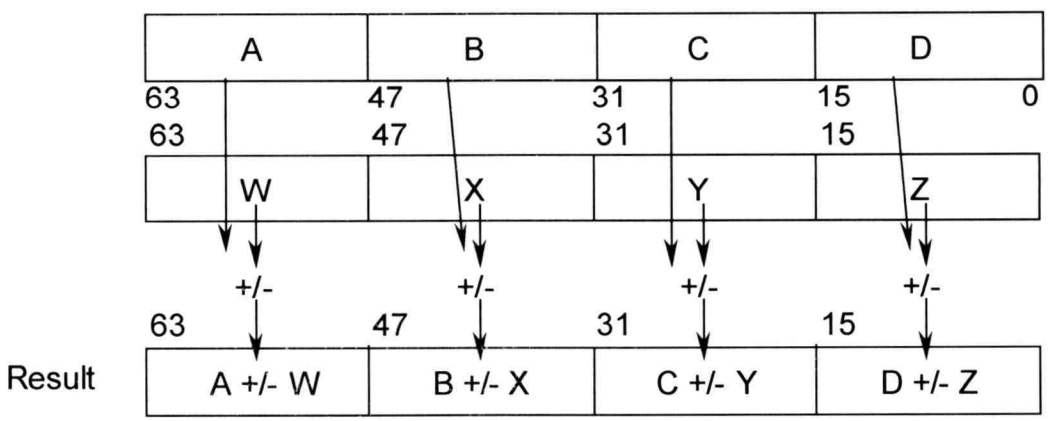


Figure 4.10: GADD16 and GSUB16 Operations [10]

- GSAD (Graphics Sum of Absolute Difference)

Table 4.5 shows the GSAD instruction, which is the only instruction for GSAD FU. Its issue latency is one cycle and the operation latency is three cycles.

Mnemonic	SubOp	t	m	Operation	Description
GSAD	0000	11	-	Rd = GSAD (Rs1, Rs2, Rd)	Pixel Sum of Absolute Difference

Table 4.5: GSAD Instruction Set [10]

Figure 4.11 illustrates the GSAD operation. GSAD takes three double-precision arguments rs1, rs2, and rd. rs1 and rs2 contain 8 pixels each in raw format. The pixels are subtracted from one another, pair-wise, and the absolute values of the differences are accumulated into rd. Note that the destination register rd contains an integral value. The GSAD instruction is used for accelerating the motion compensation to support real-time video compression in such applications as MPEG-1, MPEG-2, and H.320.

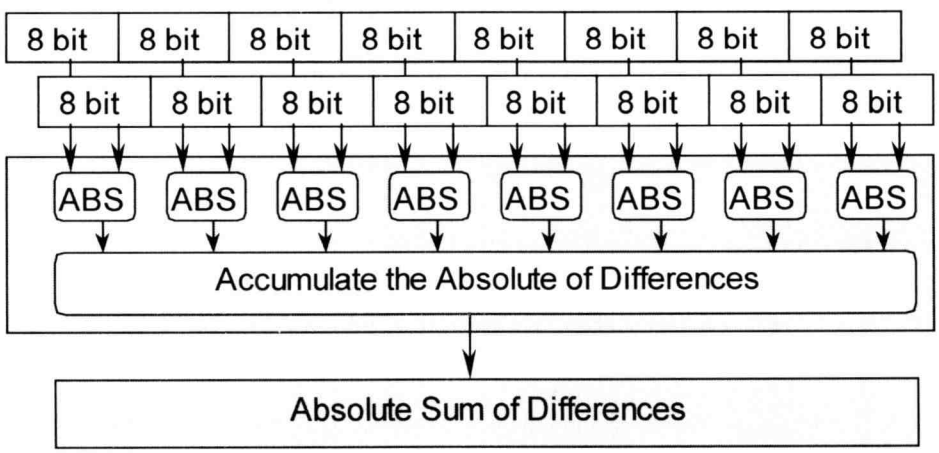


Figure 4.11: GSAD Operation

- GBMU (Graphics Bit Manipulation)

Table 4.6 shows GBMU instruction set. GBMU manipulates the result of other instructions as demanded by the application. Its issue and operation latency is one cycle each.

Mnemonic	SubOp	t	m	p	Operation	Description
GLSHL16	0000	10	0	01	Rd = Rs1 <<< Rs2	Logical Shift Left filled with zeros
GLSHL32	0000	10	0	10	Rd = Rs1 <<< Rs2	Logical Shift Left filled with zeros
GLSHL64	0000	10	0	11	Rd = Rs1 <<< Rs2	Logical Shift Left filled with zeros
GLSHR16	0000	10	1	01	Rd = Rs1 <<< Rs2	Logical Shift Right filled with zeros
GLSHR32	0000	10	1	10	Rd = Rs1 <<< Rs2	Logical Shift Right filled with zeros
GLSHR64	0000	10	1	11	Rd = Rs1 <<< Rs2	Logical Shift Right filled with zeros
GASHR16	0001	10	1	01	Rd = Rs1 >> Rs2	Arithmetic Shift Right filled with the sign bit
GASHR32	0001	10	1	10	Rd = Rs1 >> Rs2	Arithmetic Shift Right filled with the sign bit
GASHR64	0001	10	1	11	Rd = Rs1 >> Rs2	Arithmetic Shift Right filled with the sign bit

Table 4.6: GBMU Instruction Set (Continued) [10]

Mnemonic	SubOp	t	m	p	Operation	Description
GSHFH8	0010	10	0	-	Rd = shuffle (Rs1, Rs2)	Pixel Shuffle High
GSHFL8	0010	10	1	-	Rd = shuffle (Rs1, Rs2)	Pixel Shuffle Low
GWCHG8	0011	10	-	-	Rd = gwchange (Rs1)	Pixel Word Change
GREGPAIR32	0100	10	-	-	Rd = gregpair(Rs1, Rs2)	Join Two Single Precision Operands to Double Precision
GPACK16	0101	10	-	01	Rd = pack16 (Rs1)	Pixel Truncate, insert and Pack
GPACK32	0101	10	-	10	Rd = pack32 (Rs1, Rs2)	Pixel Truncate, insert and Pack
GPACK16_s	0110	10	-	01	Rd = pack16_s(Rs1, Rs2)	Pixel Truncate, insert and Pack
GPACK32_s	0110	10	-	10	Rd = pack32 _s(Rs1, Rs2)	Pixel Truncate, insert and Pack
GEXPAND8	0111	10	-	-	Rd = gexpand8 (Rs1)	Pixel Unpack
GEXPAND8h	0110	10	0	-	Rd = gexpand16h (Rs1)	Pixel Unpack High
GEXPAND8l	0110	10	1	-	Rd = gexpand16l (Rs1)	Pixel Unpack Low

Table 4.6: GBMU Instruction Set (Continued) [10]

Figure 4.12 shows the operation of the GREGPAIR32 instruction. GREGPAIR32 instruction joins two 32-bit variables into a single 64-bit variable.

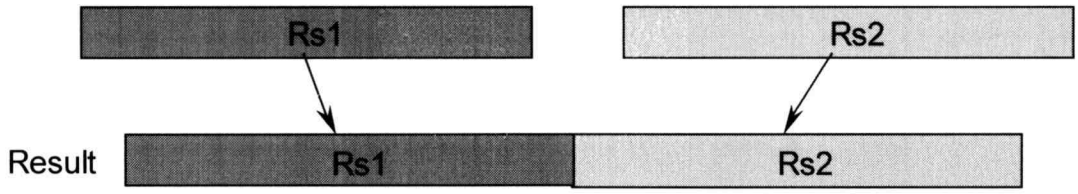


Figure 4.12: GREGPAIR32 Operation [10]

Figure 4.13 illustrates GSHFH/L8 and GWCHG8 instructions. These instructions support geometric transform and transformation from band sequential data to band interleaved data. Band interleaved data format stores the data such as RGBαRGBα RGBαRGBα, and band sequential data format stores the data such as RRRRGGGG BBBBαααα.

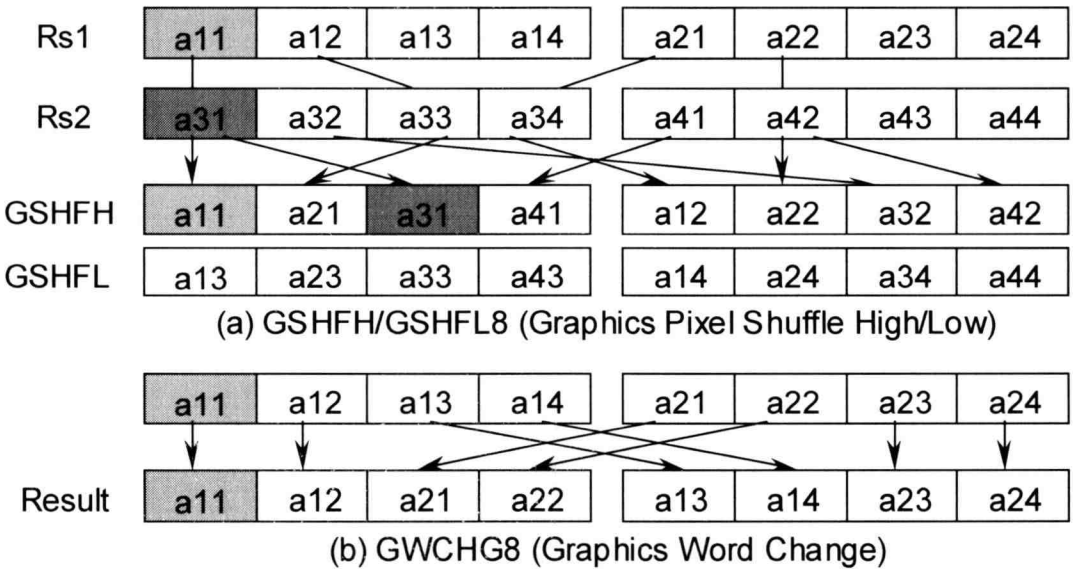


Figure 4.13: GSHFH/GSHFL8, GWCHG8 Operations.

4.4 GCU Architecture

This subsection discusses the microarchitecture of the GCU. GCU's microarchitecture is basically identical to that of a superscalar architecture except GCU has four independent IBs, RSs, ROBs, and GRFs. Figure 4.14 shows the organization of the GCU. Each component of the GCU is discussed below:

- Instruction Buffer (IB)

Four independent IBs are used to store instructions pre-decoded from the GPs and the number of entries in the IB is four.

- `Rd: The Destination Register Number.`
- `Rs1/2:` The two Source Registers Number.
- `Op:` The Opcode of the Instruction.
- `t:` Indicate which FU is required for this Instruction.
- `m:` The Mode Field.
- `p:` Indicate the Partitioned Data Types.
- `PC:` Program Counter.

- Reservation Station (RS)

The GCU has *Distributed Instruction Window* to issue instructions out-of-order. Each FU has its own set of RS and together with the corresponding ROB checks the RAW dependencies. The RS holds instructions dispatched by the t field of an instruction. Each RS entry has ten fields:

- `Op:` The Opcode of the Instruction.
- `T1/2:` The ROB entry number or the tag that will produce the corresponding Source Operand.
- `Src1/2:` The Value of the Source Operands.
- `Busy:` Indicate that this RS is Full.

- `Tag:` The ROB entry number, which is the Destination for the Result.
- `m:` The Mode Field.
- `p:` Indicates the Packed Data Types.
- `GPID:` Indicate from which GP this Instruction comes and used for determining the Destination ROB.

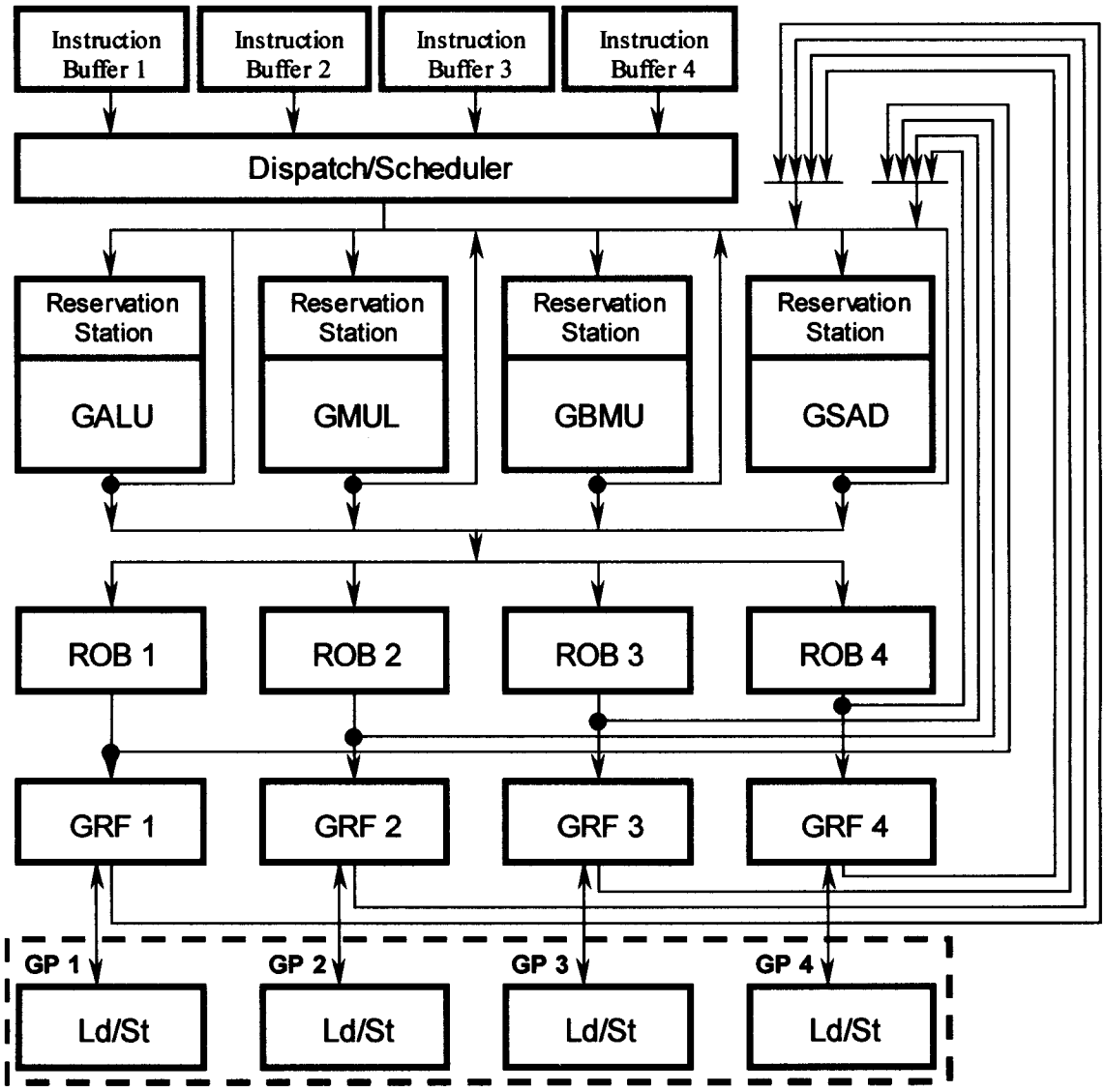


Figure 4.14: The Architecture of the GCU

- Functional Units (FU)

FUs execute the instructions by copying Op, Src1/2, Tag, m, p, and GPID fields from the RS. FUs are composed of GALU, GMUL, GBMU, and GSAD. The GALU is organized as 4 independent 16-bit adders named A, B, C, and D as shown in Figure 4.15

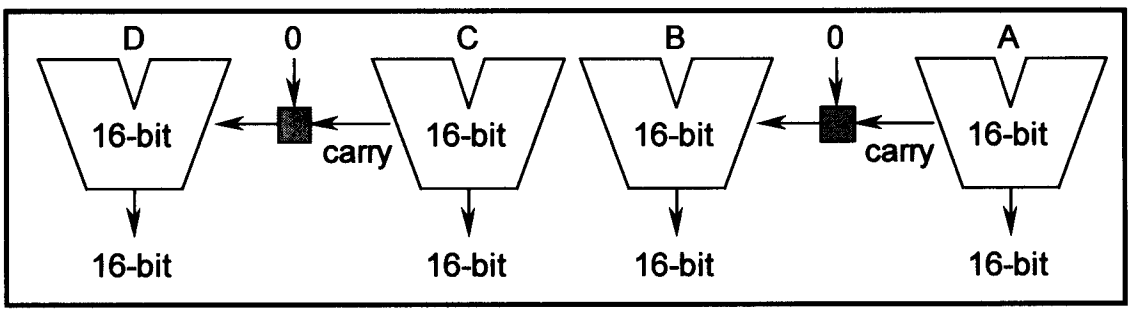


Figure 4.15: Structure of the GALU [14]

These adders do not propagate carries between them. In addition, for 32-bit operations, a carry is generated by adder A and C and is used to select one of two results computed by adders B and D. One result assumes that the incoming carry is zero and the other result assumes that the incoming carry is one. Selection of the correct results is based on the value of the carry bits of adders A and C. This scheme is analogous to a *conditional sum adder* [15]. GMUL is composed of four 8×16 multipliers.

One of the characteristics of GCU design is GSAD unit, which is dedicated to motion compensation calculation. This is the most expensive operation involving motion video compression. Motion compensation involves searching a reference frame for the closest match to a 16×16 block of pixels in the target frame. The closest match is determined by finding the block with smallest absolute difference between the target and the reference frames. Therefore, the GSAD should be able to accumulate back-to-back distances in consecutive cycles. As shown in Figure 4.16, the GSAD has three 4:2 counter, two 11-bit carry propagate adder, and a 53-bit incrementer.

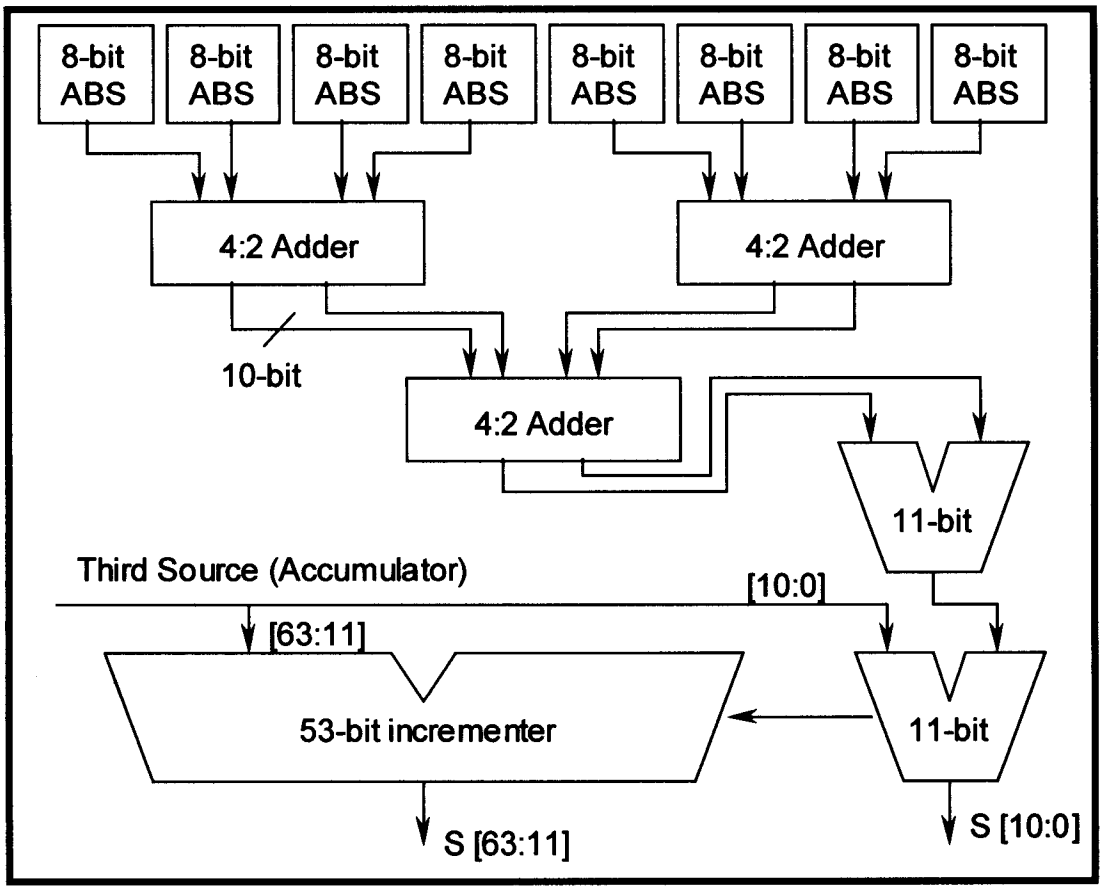


Figure 4.16: GSAD Logical Implementation [14]

The 4:2 adder receives four inputs (actually it's five because of the carry-in of previous column) and the outputs are Carry (i + 1), Sum (i), and C_{out} (i + 1) as shown in Figure 4.17. This adder reduces four rows of 8-bit absolute values to two rows of 10-bit, and the final 11-bit result of 4:2 adder is added by 11-bit adder. In addition, the 12-bit result is added to the corresponding 12-bit of the accumulator to accumulate the absolute sum of differences. The third source operand comes only from the destination register or accumulator because the operation is done using **Rd = GSAD (Rs1, Rs2, Rd)**.

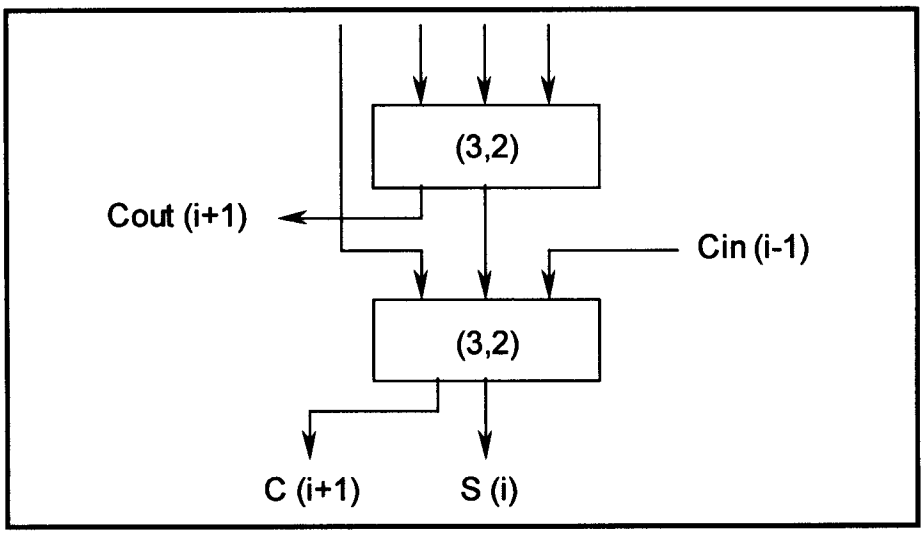


Figure 4.17: A (4:2) Adder [15]

- Reorder Buffer (ROB)

The GCU has four independent ROBs. Each ROB is hard-wired with the corresponding IBs to keep the correct program sequences from each GPs. ROB holds either the pending or the completed results of instructions that may have finished execution but have not yet committed. Each field is composed of four fields.

- `D:` Destination Register Number.
- `Dst:` Result Value.
- `V`: When Set, signifies that Dst field holds a Valid Result.
- `PC:` Address of the instruction.

- Graphics Register File (GRF)

The GCU has four 32×64 bits Register Files. Each GRF has 3 reads/2 writes ports to supply the data efficiently. Two reads ports are for source operands and one read port is for the load instruction of GP. One write port is needed for the result and the other

write port is for store instruction from GP. In addition, as discussed in Chapter 4.2, GRF has two extra fields, which are *valid_gp* and *valid_gcu* fields, to synchronize data between the GCU and the GPs.

Implementing GCU with separate Register Files has several advantages. First, it leaves GPs free for address and branch computation. Second, GCU instructions do not affect the main integer pipeline. Instruction operands in the GRF can refer to either single or double precision registers. Many of the instructions have formats that use either a 32-bit single precision or a 64-bit double precision register as an operand. This is because the Sparc V9 architecture defines 32 single and 32 double precision registers [16], and various algorithms implemented with VIS instruction use this functionality. Figure 4.18 shows the layout of the GRF.

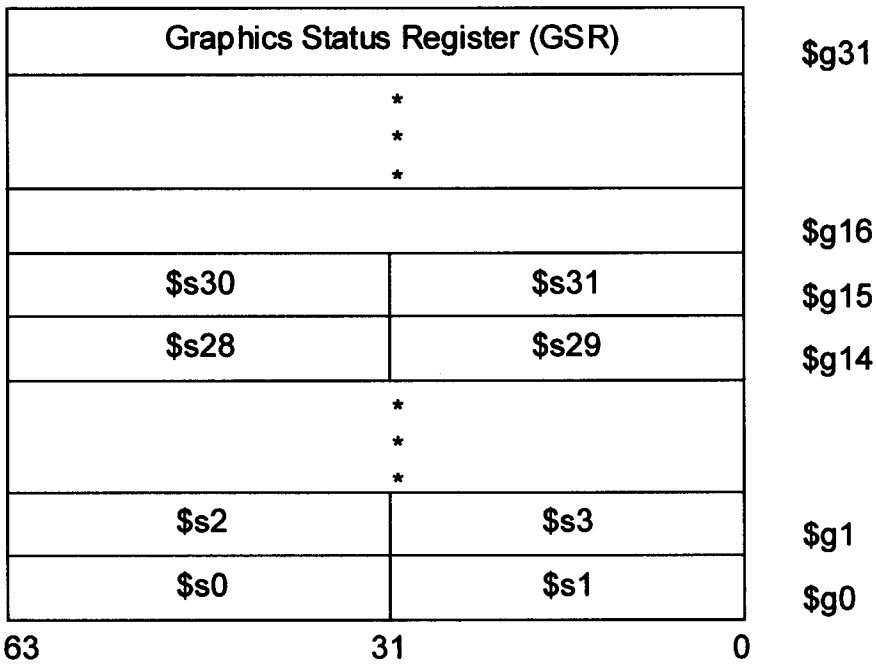


Figure 4.18: The Layout of Graphics Register File [13]

The 32 single precision registers corresponded to the first 16 double precision registers, and single precision operands cannot access the last 16 double precision

registers. This arrangement allows the GCU to access and update the high and low 32-bit halves of a 64-bit double precision register. As shown in Figure 4.18 the last register ($g31) is reserved for a special register, which is called Graphics Status Register (GSR). The GSR is used for conversion between formats and for memory alignment. The fields of the GSR are shown in Figure 4.19 [10].

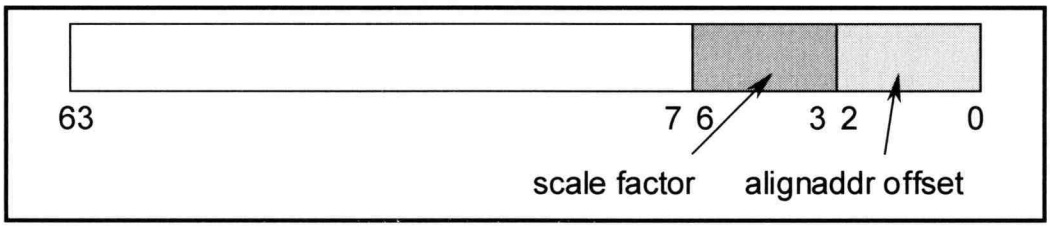


Figure 4.19: Graphics Status Register [10].

4.5 Five-Stage Pipeline of the GCU

The GCU has five-stage pipeline, which are Dispatch, Issue, Execution, Write-back and Commit stages.

- Dispatch stage

Figure 4.20 depicts the Dispatch stage. If there are empty slots in the RS and in the ROB, the Dispatcher dispatches the instructions. Dispatching is based on the *t* field of the instruction, which indicates which FU is required for the instruction. The Dispatcher dispatches one instruction each from IB1 to IB4, as a basic scheme. For each instruction dispatched, the following operations are performed:

1. Pop an instruction from Instruction Buffer and assign it to an RS entry.

2. Allocate the next entry in the ROB. Place the Tag value pointing to this entry into the Tag field of the allocated RS entry, and the number of this field is used to tag the result when the execution is done.
3. For each of the source registers s1 and s2, perform an associative search in the appropriate ROB for the latest entry where the destination Register D field matches the source Register number.
   a) If found and the result value Dst is valid, update the appropriate source fields (Src1 or Src2) with the value from the Dst field. If the result value Dst is not valid, update the tag fields T1 and T2 with the Tags of the entries in the ROB.
   b) If not found, read the matched Register File for the source register.

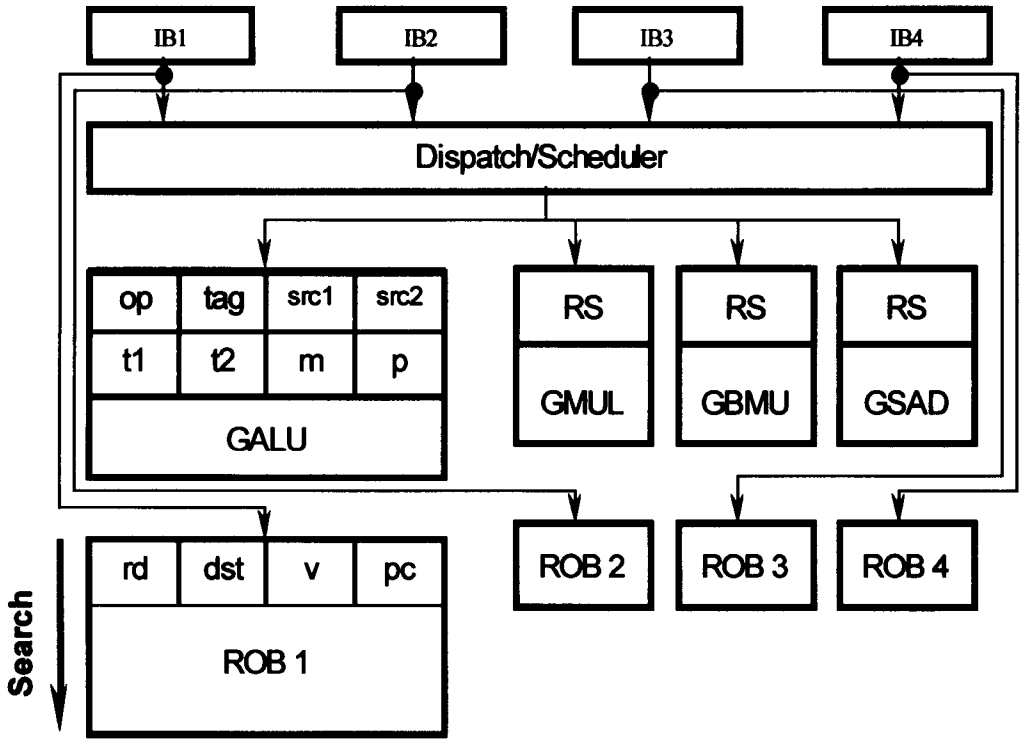


Figure 4.20: Dispatch Stage of the GCU

- Issue Stage

An instruction can be issued to a FU when all the operands are ready in the RS. This is done by

- Copy Op, Src1/2, Tag, m, p, and GPID from the RS to the FU and set the busy field of the FU.
- Free the entry of RS.

- Execute Stage

Number of cycles required depends on the FU and the latencies can be obtained from "VIS Speed New Media Processing," IEEE Micro, 1996 by Tremblay et al. [13]. Table 4.7 shows the latencies of each FU.

FU	Issue Latency	Operation Latency
GALU	1	1
GMUL	1	3
GBMU	1	1
GSAD	1	3

Table 4.7: Latencies of each FU

- Write-back Stage

When an instruction completes, its result is posted in the ROB as well as forwarded to RS. This is done by followings.

1. Post the result in the Dst field of the entry in the ROB pointed by the Tag field and GPID filed, and set V to indicate that the result is valid.
2. Forward the result back to the RSs where the Tag field of the completed instruction matches the source Register tags T1 and T2 in the RSs. For each match, copy the result into the respective Src1/Src2 fields.

- Commit Stage

When an instruction reaches the head of the ROB and its V field is set, update the result and set the *valid_gcu* bit to indicate to GPs that the computation by the GCU is done, and free the entry in the ROB.

5 TRACE DRIVEN GCU SIMULATION

A trace driven simulator for the GCU has been implemented. This simulator accepts the input files, which are traces of GCU instructions. Because this is an experimental architecture and the ISA was quite different from either VIS or MMX, there were no real benchmark programs to run on the simulator. Therefore, instruction streams were generated from sample codes of VIS and MMX to test the simulator. In order words, VIS or MMX codes generated for sample multimedia applications were converted to GCU instructions.

This Chapter discusses the structure of the simulator, some examples of multimedia application codes for the GCU, and the results of the simulation study.

5.1 Structure of the GCU Simulator

Figure 5.1 depicts the structural overview of the Simulator. The simulator was written in C with GNU gcc library and a compiler on a Linux platform running 266 MHz Pentium- II processor. The input file is composed of four different files that represent the four GP instruction streams.

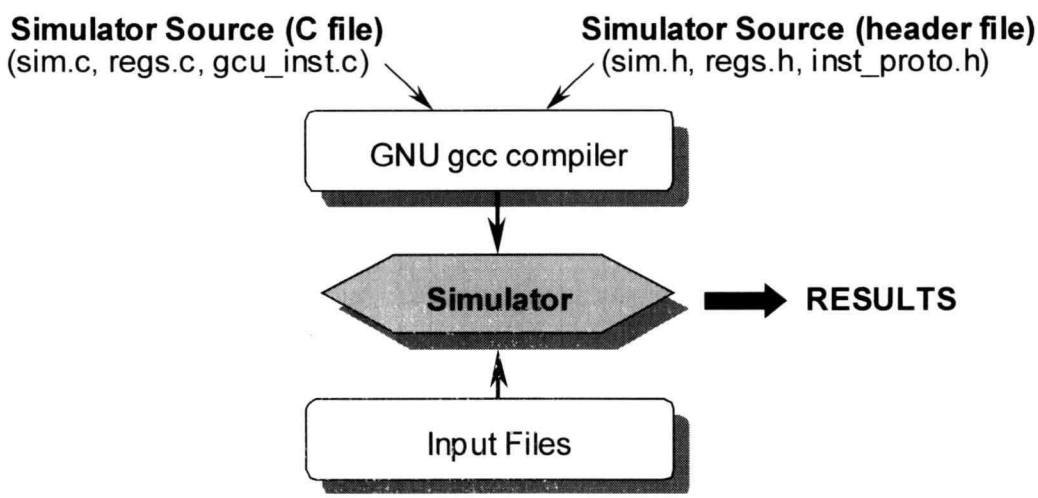


Figure 5.1: Structural Overview of Simulator

The simulator uses dynamic data structures to implement IB, RS and ROB. Figure 5.2 illustrates the *doubly linked-list* data structure, which is used to implement these components. The previous and next fields within an RS structure are linked to other RS structures and are used to search, delete, and add elements. The tag field of RS keeps the pointer to a ROB entry to write the result of the computation at the Write-back stage. The RS_LIST structure has three pointer fields and a full flag. The head field points the first RS entry and tail points the last RS entry, and the full flag will be set when the number of entries in the linked-list is equal to the maximum number of RS entries defined by the simulator. The ROB also has a similar structure to the RS structure.

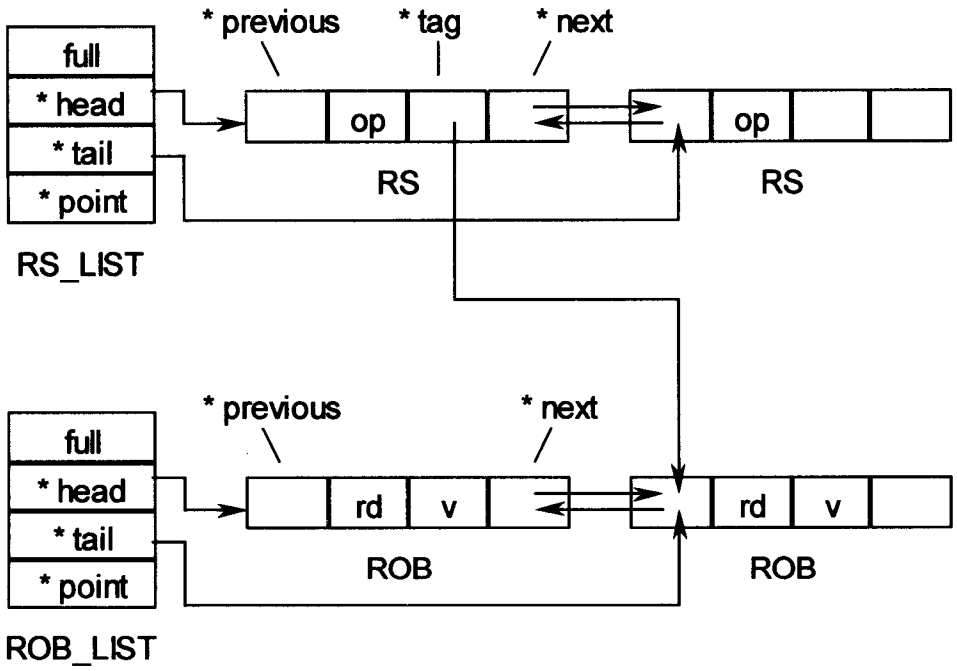


Figure 5.2: Doubly Linked List Structure for RS and ROB

- Simulator Code File Description

The following list describes the functionality of the C code files in the GCU simulator.

- gcu_insts.c: Defines all the GCU Instructions.
- inst_proto.h: Holds the prototypes of all the GCU Instructions.
- regs.c: Defines the GCU Registers.
- regs.h: Header file of the regs.c.
- sim.c: Performs all the initialization and contains the main loop.
- sim.h: Defines the data types for the simulator and the function prototypes, and contains the data structure.

- Options of the Simulator

The GCU simulator can accept several command-line options. The number of entries for IB, ROB, and RS can be changed, as well as the Fetch and Dispatch bandwidths. It can execute arbitrary number of instructions by using the argument *input_num*, and the number of FUs can be doubled. It can also print out all the status and statistical information of the pipeline stages, including information such as Instruction Per Cycle (IPC), total cycle time, number of FU busy, etc.

5.2 Out-of-order Simulation Timing

The main loop of the simulator is structured as follows:

```
sim_init( ); /* Initialize the Simulator */
sim_main( );
while{
        Issue_Execute( );
        Write_Back( );
        Commit( );
        Fetch( );
        Dispatch( );
}
```

This loop is executed once for each cycle. The Dispatch( ) stage is executed at the last step to keep the cycle count correct. At the first cycle, only the Fetch( ) and the

Dispatch( ) stages are executed. The Fetch( ) stage is used to supply instructions from an input file to the Dispatch( ) stage. For each cycle, it fetches one instruction from each of the four input files and enqueues them in the appropriate IBs.

In the Dispatch( ) stage, instructions are dispatched to RSs. This is done by adding another entry onto the RS and writes the instruction's information on to it. Moreover, it looks up the ROB to check for RAW dependencies. If dependencies exist and the value from the ROB is valid, it is copied as the source operand. If the value is not valid, the tag field is updated to point to that entry of the ROB so that the result can be written back when the computation is done. If no dependencies exist and the value of the corresponding Graphics Register is valid, which is checked by the *regs_valid_gp* field, the source operand is updated by the value of the GRF.

The Issue_Execute( ) stage tries to issue instructions out-of-order from RSs when dependencies are resolved and corresponding FU are not busy. If the operands are ready, it sends the instruction to the execution stage and the busy flag of the FU is set for the duration of the FU's latency. The Execution stage executes the instruction using the opcode, t field, m field, and p field. It has a nested case statement to execute instructions which are distinguishable by its unique combination. After the execution is completed, the result is saved to a FU_WB. The FU_WB structure keeps the result data, the entry number of the ROB, the GP's ID to write-back to the corresponding ROB, and the operation latency to decrement the latency time once in a cycle. The Write_Back( ) stage writes back the result to the corresponding ROB and forwards the result to the RSs that have their tags pointing to this ROB entry.

The Commit( ) stage commits up to four instructions starting from the head of the ROB. If the valid bit in the head of the ROB is set, the instruction can be committed, and the next instruction is examined if the valid bit is set or not, and if the instruction has a valid bit, it can be also committed. But if the instruction does not have the valid bit, it will not be committed and the next entry will not be examined at the same cycle. Therefore, the GCU can commit upto 16 instructions per cycle.

5.3 Code Examples

The followings are some code examples illustrating the applications of the GCU instructions. The main characteristic of these examples is that the core portion of the codes is short and nested within a loop. The GP executes the data alignment and load operations, and then the GCU executes the computation intensive GCU instructions. This subsection illustrates sample codes, which are Alpha Blending, Conversion, and Motion Estimation.

- Alpha Blending

This example illustrates an application where two images are blended together. For each pair of corresponding pixels in two images 's1' and 's2', a corresponding pixel is computed from a third control image 'alpha'. The following equation is used to compute dst:

$$dst = (alpha/256) * s1 + (1-alpah/256) * s2$$
$$= (s1-s2) * (alpah/256) + s1$$

This program calculates each half of the image 's1' and 's2' and merges them together, and the code below shows the processing of one scan line.

```
$1 = Gexpand($2);          /* $2: high half of s1 */
$3 = Gexpand($4);          /* $4: high half of s2 */
$5 = Gsub16($1, $3);       /* (s1 - s2) */
$6 = Gmul8×16($7, $5);     /* (s1-s2)*alpha/256, $7: alpha/256 */
$8 = Gadd16($1, $6);       /* (s1-s2)*alpha/256 + s1 */

$10 = Gexpand($2);         /* $2: low half of s1 */
$11 = Gexpand($4);         /* $4: low half of s2 */
$12 = Gsub16($10, $11);    /* (s1 - s2) */
$13 = Gmul8×16($7, $12);   /* (s1-s2)*alpha/256, $7: alpha/256 */
$14 = Gadd16($10, $13);    /* (s1-s2)*alpha/256 + s1 */

$8 = Gpack16($8);          /* Packing 64-bit data to 32-bit data */
$14 = Gpack16($14);        /* Packing 64-bit data to 32-bit data */
$15 = Gregpair32($8, $14); /* Join two 32bit variables into one 64bit variable */
```

- Conversion 24-bit True Color to 16-bit High Color

Many applications provide RGB data with the assumption that the video display will use 24-bit true color data, which contains one byte of Red, one byte of Green, and one byte of Blue data. However an abundance of video display can only support 16-bit high color display, thus the conversion from RGB (8-bit, 8-bit, 8-bit) to RGB (5-bit, 5-bit, 5-bit) is needed. The algorithm used here is called "mask-shift-or method". This algorithm takes each 24-bit true color element stored in the three least significant bytes of a double word, masks each 8-bit color, shifts it right three bits, and ORs the result into a register. By using GCU instructions, four pixels can be processed at the same time.

```
/* $1: Two 24-bit RGB in 64-bit register.
$2: Two 24-bit RGB in 64-bit register.
$3: Blue Mask: 0xFF000000FF.
$4: Green Mask: 0xFF000000FF00.
$5: Red Mask: 0xFF000000FF0000.
$6: Shift Amount 3 to get Blue pixel.
$7: Shift Amount 6 to get Green pixel.
$8: Shift Amount 9 to get Red pixel. */

$9  = Gand32 $1, $3;          /* Mask out all but the 5 MSB Blue bits */
$10 = Gand32 $2, $3;          /* Mask out all but the 5 MSB Blue bits */
$11 = Glshr32 $9, $6;         /* Shift Blue bits to bits 0-4 */
$12 = Glshr32 $10, $6;        /* Shift Blue bits to bits 0-4 */
$13 = Gand32 $1, $4;          /* Mask out all but the 5 MSB Green bits */
$14 = Gand32 $2, $4;          /* Mask out all but the 5 MSB Green bits */
$15 = Glshr32 $13, $7;        /* Shift Blue bits to bits 5-9 */
$16 = Glshr32 $14, $7;        /* Shift Blue bits to bits 5-9 */

$17 = Gor32 $11, $15;         /* Or in the Green bits with the Blue bits */
$18 = Gor32 $12, $16;         /* Or in the Green bits with the Blue bits */

$19 = Gand32 $1, $5;          /* Mask out all but the 5 MSB Red bits */
$20 = Gand32 $2, $5;          /* Mask out all but the 5 MSB Red bits */
$21 = Glshr32 $19, $8;        /* Shift Red bits to bits 10-14 */
$22 = Glshr32 $20, $8;        /* Shift Red bits to bits 10-14 */

$23 = Gor32 $17, $21;         /* Or Blue and Green bits with Red bits */
$24 = Gor32 $18, $22;         /* Or Blue and Green bits with Red bits */
$25 = Gpack32_s $23, $24;     /* Pack the four 16-bit pixels into one quadword */
```

- Motion Estimation

This example presents a single iteration of a motion vector estimation process. A 16×16 block of pixels of frmae2 is taken and a search within a specified area in frmame1 is performed to determine if something "similar" to the 16×16 block from frame2 exists. If it does, then a motion vector is estimated from this location. The "similar" is estimated by the absolute sum of differences, i.e., "diff" between the two 16×16 blocks. The absolute sum of differences is computed in accordance with the following relationship and we can select similar block by searching minimum of "diff".

$$\text{diff} = \sum_{0}^{15}\sum_{0}^{15} | \mathit{frame}2(i,j) - \mathit{frame}1(i,j) |$$

Selected block = min(diff)

The speedup capability of GCU instruction is illustrated by processing 8 bytes at a time.

```
/* All the aligned source data are supplied by GP. Search four different blocks
simultaneously */
$1: Accumulated Sum of Difference1.
$2: Accumulated Sum of Difference2.
$3: Accumulated Sum of Difference3.
$4: Accumulated Sum of Difference4.

$5, $6: 8 Bytes of source data from frame1 and frame2.
$7, $8: 8 Bytes of source data from frame1 and frame2.
$9, $10: 8 Bytes of source data from frame1 and frame2.
$11, $12: 8 Bytes of source data from frame1 and frame2. */

/* One loop is repeated 32 times to cover 16×16 blocks. */
$1 = Gsad8 $1, $5, $6;          /* Calculate 8 bytes of row at a time */
$2 = Gsad8 $2, $7, $8;
$3 = Gsad8 $3, $9, $10;
$4 = Gsad8 $4, $11, $12;
```

5.4 Simulation Results

The simulation was performed assuming that four GPs execute different multimedia applications and the dependencies between GPs and GCU are not considered during this simulation. Four input files were generated from different applications and were fed into the simulator. GP1 executed Alpha Blending, GP2 executed Conversion, GP3 executed Convolution, and GP4 executed Motion Estimation. Each input stream consists of approximately 2 million instructions, and thus total of 8 million instructions were executed. Figure 5.3 depicts the default execution result shown by the simulator.

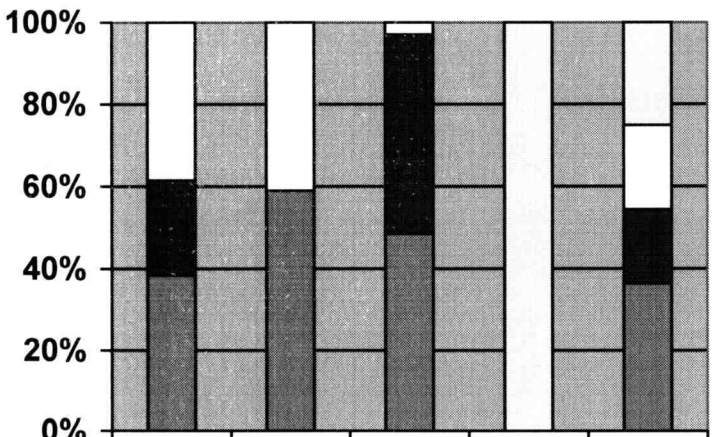
```
james:~/simulator/GCU-beta$sim -n2000000
-- Graphics Control Unit Simulator designed by Keungsik,Choi. --
-- Input Number 2000000
Simulation started @ Wed Apr 29 22:40:35 1998
====== Simulation is done!! ======
====== Statistics ========
--Fetch Stage--
IB1 FULL: 2213413  IB2 FULL: 2436671  IB3 FULL: 5237833
IB4 FULL: 5999859
--Dispatch Stage--
RS1 FULL: 1178505  RS2 FULL: 2262559  RS3 FULL: 3869814  RS4 FULL: 0
ROB1 FULL: 2060694  ROB2 FULL: 326300  ROB3 FULL: 1368022
ROB4 FULL: 5999874
IB1 EMPTY: 3787704  IB2 EMPTY: 3564444  IB3 EMPTY: 763286
IB4 EMPTY: 1248
--Issue Stage--
Operands not Ready: 4089422
GALU Busy: 2918672  GMUL Busy: 4265260  GBMU Busy: 1607909
GSAD Busy: 5999997

Total Instruction Count: 8000000
Cycle Time: 8001122
IPC:0.999860
The Execution Time is: 584.255095sec
The End Time is: Wed Apr 29 22:50:20 1998
```

Figure 5.3: Default Simulation Result shown by the Simulator

The IB FULL indicates the number of times the IB was full when the Fetch stage tried to fetch the instruction, thus instructions could not be fetched. The IB EMPTY indicates the number of times the IB was empty when the Dispatch stage tried to dispatch. The Operands not Ready indicates the number of times one of the operands was not ready thus it could not be issued. The FU BUSY indicates the number of times the instruction could not be issued because the FU was busy. The result shows that FU BUSY are highly related to what kinds of GCU instructions are used by the applications. GMUL and GSAD are suffered a lot of FU BUSY conditions because the operation latencies of those FUs are 3 cycles. The FUs are not pipelined by default.

Figure 5.4 shows the percentages of usage of each FUs. GALU and GSAD are the most frequently used FUs in this simulation. This can be varied by the simulated applications.



	GP1	GP2	GP3	GP4	TOTAL
GSAD	0	0	0	100	25
GBMU	38.5	41	3	0	20.6
GMUL	23	0	48.5	0	17.9
GALU	38.5	59	48.5	0	36.5

Figure 5.4: The Usage of FUs

Figure 5.5 shows the relative IPCs for different numbers of FUs, which are doubled and tripled, and pipelined. By adding FUs, the IPC increases linearly, and GMUL and GSAD FUs have a benefit when the FUs are pipelined as shown in Figure 5.6. GALU and GSAD are the busiest FUs even though the FUs are pipelined and doubled, because GALU and GSAD are the most frequently used ones. If the applications use all the four FUs evenly, the performance will be the best.

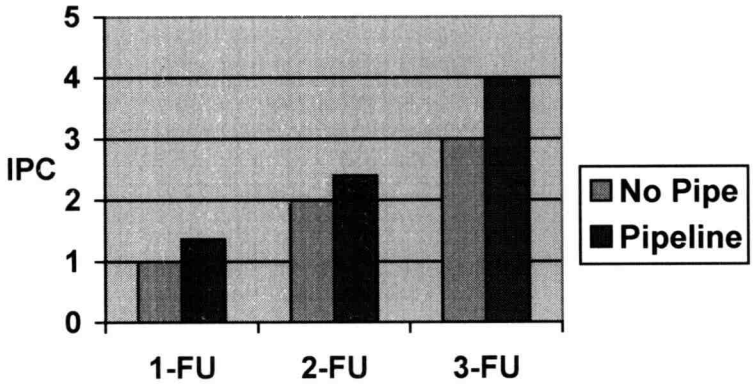


Figure 5.5: IPCs on various number of FUs

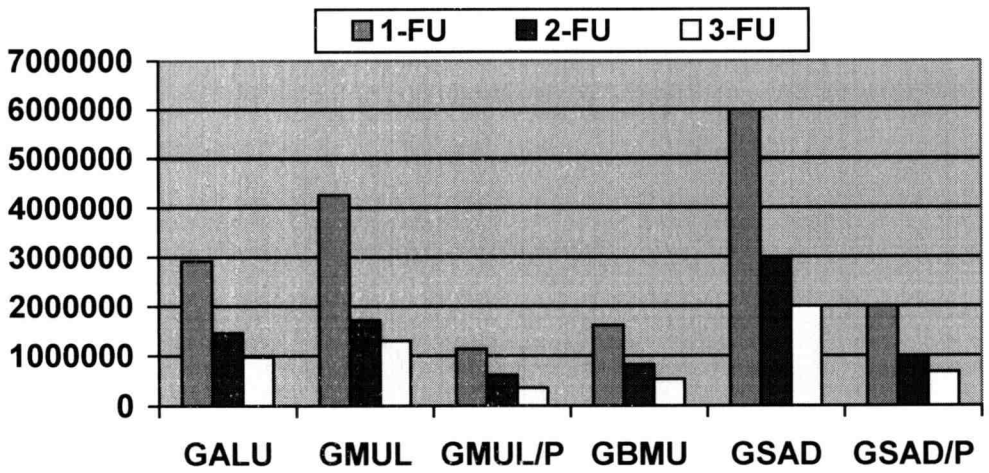


Figure 5.6: FU BUSY Conditions on Various number of FUs

As we've seen from Figure 5.5, the IPCs go up by adding more FUs, but the price and area of added FUs should be considered, and because the input streams are not real benchmark programs, the result of the simulation is not reliable enough. Therefore, the simulation is focused on exploiting all the design aspects of default configurations, rather than adds more FUs. First, several dispatch schemes are simulated, because the dependencies among the instructions and the utilization of FUs are depend on the instructions dispatched in the RS.

The sequential dispatch scheme is a default dispatch scheme, which dispatches one instruction each from IB1 to IB4. The random dispatch scheme is a dispatch scheme, which choose IBs to dispatch two or four instructions at a time from chosen IBs. Figure 5.7 shows the comparisons between these two schemes, and the numbers of slots of RS/ROB are also varied to determine the efficient size.

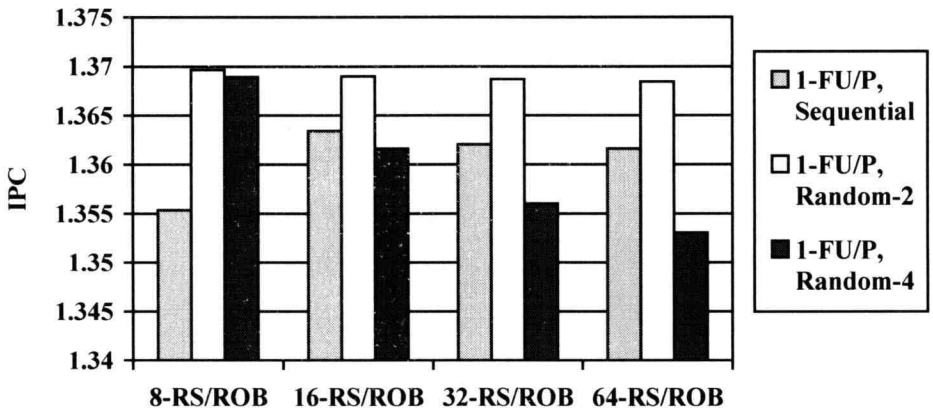


Figure 5.7: IPC Comparisons for different Dispatch Schemes

The result shows that when two IBs are chosen randomly and two instructions are dispatched from the chosen IBs, the performance is the best. Figure 5.7 also shows an interesting effect by changing the size of RS/ROB. When the size of RS/ROB is eight, the performance of random-2 dispatch scheme is the best, and the reason why the IPC goes down when the size goes bigger is that the dependencies among instructions are

increased and FU BUSY conditions are increased. Figure 5.8 shows the FU BUSY conditions when ramdon-2 dispatch scheme is used, and Figure 5.9 shows the number of Operands not Ready conditions.

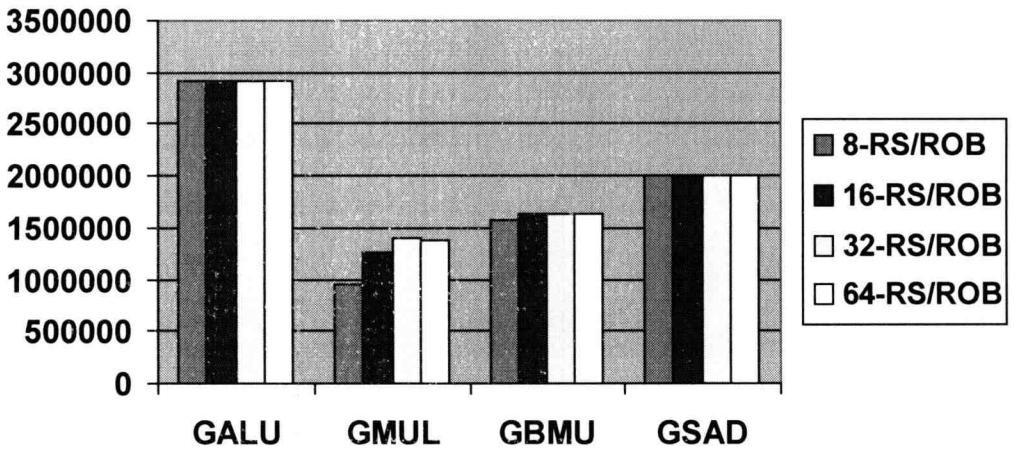


Figure 5.8: FU BUSY Conditions among varied RS/ROB size

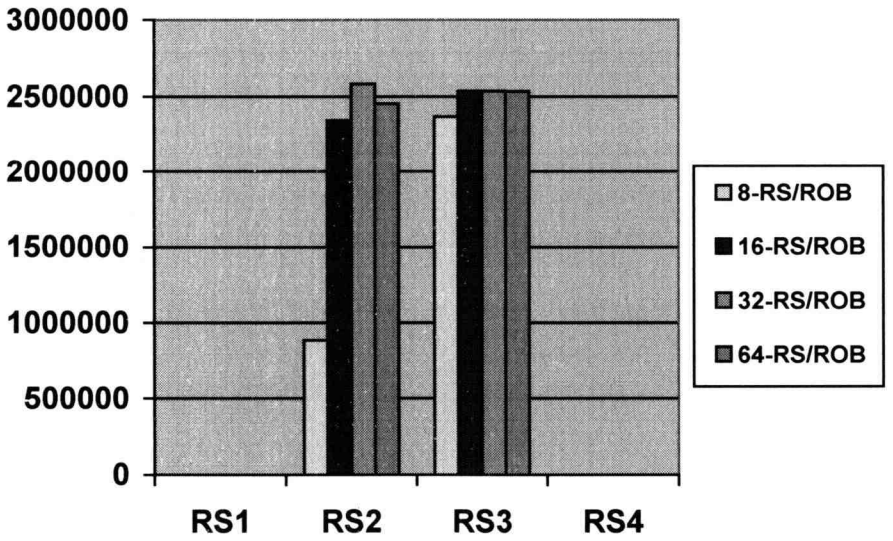


Figure 5.9: Operands not Ready Conditions among varied RS/ROB Size

As shown in Figure 5.8, the FU BUSY conditions of GMUL are increased a little bit when the size of RS/ROB is bigger. Figure 5.9 shows the Operands not Ready condition also increased when the size of RS/ROB is increased.

Figure 5.10 shows the IPCs when the FETCH/DISPATCH-bandwidths are varied. The 2-F indicates the bandwidth of FETCH is two, and 2-D indicates the bandwidth of DISPATCH is two, which means each GP fetches two GCU instructions and GCU dispatches two instructions at a time from each IB.

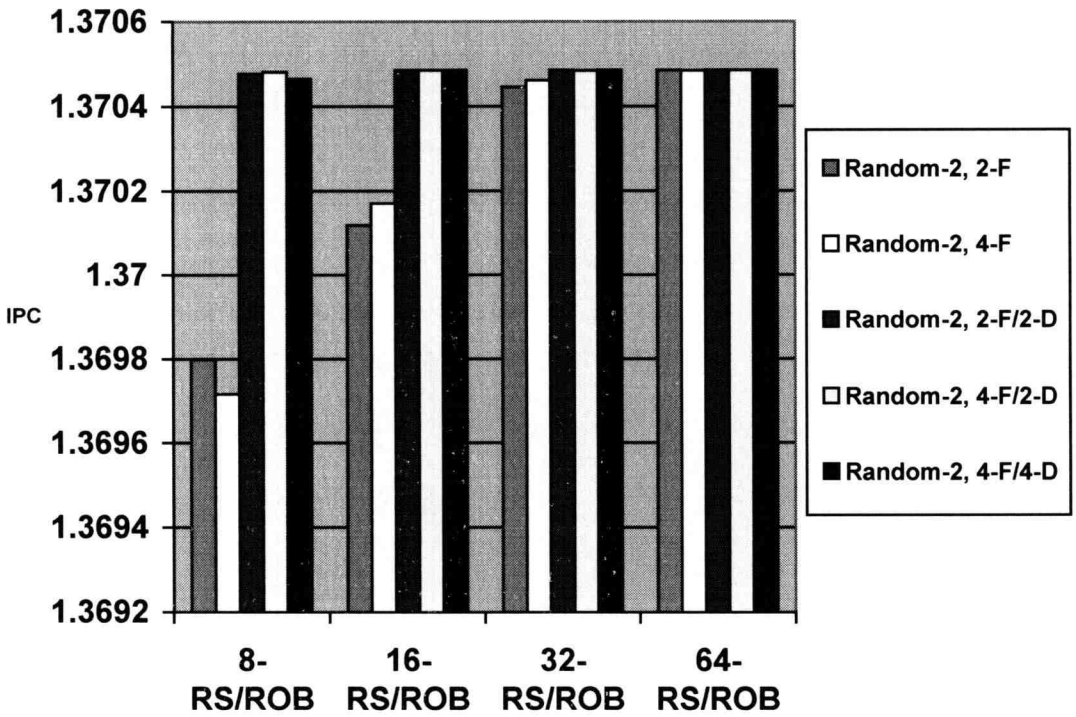


Figure 5.10: IPC Comparisons among various FETCH/DISPATCH Bandwidths

As shown in Figure 5.10, the IPCs are saturated to 1.370485. Based on this simulation, the most efficient configuration is the random-2 dispatch scheme with 2-F/2-D and 8-RS/ROB size.

Finally FUs are doubled to see how the IPCs are increased, and the same dispatch schemes are applied as the simulation of one FU shown in Figure 5.7.

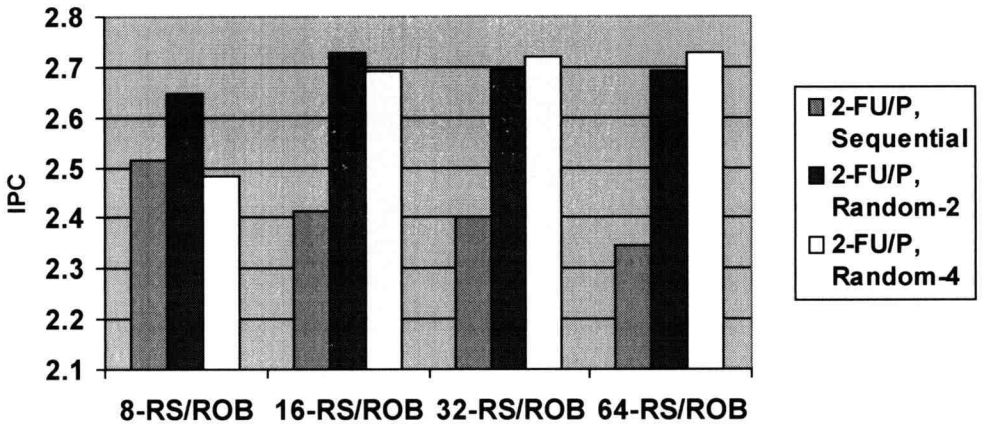


Figure 5.11: IPC Comparisons for different Dispatch Schemes when FUs are doubled

This results also show random-2 dispatch scheme utilizes the resources better than sequential dispatch scheme especially when the size of RS/ROB is small, and comparable with random-4 dispatch scheme.

Looking at the graphs shown above, the performance of GCU is dependent on the number of FUs, dispatch schemes, and FETCH/DISPATCH bandwidths. But there are two issues that have to be resolved in GCU simulation. One is the real benchmark programs. To simulate more realistic simulation, the actual instruction streams generated by the four GPs are needed. The other is the dependencies between GPs and GCU. This problem will be resolved when the Rapsim and GCUsim are integrated together.

6 CONCLUSION AND FUTURE WORK

The multimedia functionality of today's computers is becoming more and more demanding, and the applications such as video teleconferencing, 3-D visualization, animation, and image display through the network, etc., are rapidly becoming a must for the users. Until now these applications require specialized graphics hardware. But the price of separate board is expensive. Therefore, on-chip multiprocessor with GCU, Raptor was proposed by ETRI to remove the need for separate boards and to obtain better overall system price-performance by supporting these applications directly on the processor. Raptor utilized matured IC processing technology and SIMD instructions. Especially, the GCU provides a big performance advantage for computation intensive multimedia applications.

The GCU simulation was performed to verify how effective the GCU instructions are and how well GCU can be utilized by the four GPs. However, the current status of the GCU simulator is still in its infancy and therefore rather limited. First, more fine-tuning of the architecture is required. Second, the GCU simulator must be integrated into RapSim, which is still being developed by ETRI to simulate the Raptor. Without it, the real benchmarks cannot be generated to properly validate critical timing issues between the GPs and the GCU. In particular, issues such as how well the GPs can supply and schedule instructions and data to the GCU will become the critical factor that determines the overall performance. Therefore, the next stage of this project is to integrate GCU into RapSim.

BIBLIOGRAPHY

1. *Multimedia Processor: Parims2000*, available at http://www.parims.com/vrsys/vrsys0/multimedia.html

2. *Electronic Engineering TIMES*, June 10, 1996: A CMP Publication.

3. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang, *The Case for a Single-Chip Multiprocessor*, Stanford University.

4. Ralf Steinmetz and Klara Nahrstedt, *Multimedia: Computing, Communications And Applications*, Prentice Hall, 1995.

5. James D. Murray and William Vanryper, *Graphics File Formats*, O'Reilly & Associates, Inc, 1996.

6. Roy Hoffman, Data Compression In Digital Systems, ITP, 1996.

7. William B. Pennebaker, Joan L. Mitchell, *JPEG - Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

8. Prabhat K. Andleigh, Kiran Thakrar, *Multimedia Systems Design*, Prentice Hall, 1996.

9. Millind Mittal, Alex Peleg, Uri Weiser, *MMX™ Technology Architecture Overview*, Intel, 1996.

10. *Visual Instruction Set (VIS™) User's Guide*, Sun Microsystems, 1997, available at http://www.sun.com/sparc/vis/

11. *MMX™ Technology Technical Overview*, Intel, 1996, available at http://developer.intel.com/drg/mmx/manuals/overview/index.htm

12. Hearn, Donald and M. Pauline Baker, *Computer Graphics Basics*, Prentice Hall, 1996

13. Marc Tremblay, J. Michael O'Connor, Venkatesh, Liang He, *VIS Speeds New Media Processing*, IEEE Micro 1996.

14. L. Kohn et al., *The Visual instruction Set (VIS) in UltraSparc*, Proc. Compcon, IEEE CS Press, 1995, pp. 462-469.

15. Israel Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

16. David L. Weaver, Tom Germond, *The SPARC Architecture Manual*, Prentice Hall, 1994.