**February 21, 1997**

# Real-time Signal Processing on the Ultrasparc

**Department of Electrical Engineering and Computer Science**

**University of California**

**Berkeley, California 94720**

**William Chen**

**Research Project**

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science in Engineering, Plan II.

Approval for the Report and Comprehensive Examination:

**Committee:**

_____

Professor E.A. Lee
Research Advisor

_____

Date

_____

Professor J. Rabaey
Second Reader

_____

Date

# 1.0 Abstract

With the convergence of audio, video, and the Internet, multimedia applications increasingly demand real-time processing from general purpose CPUs. Traditionally, computer architects have increased the performance of CPUs in incremental steps, increasing the number of cache levels, increasing the complexity of I/O controllers, increasing the length of the pipeline, increasing the number of instructions issued per cycle, and reducing the overall chip size. But the constraint of real-time processing is a hard problem, and computer architects must take a different approach to keep up.

One approach is to support the signal processing functions that are common to these applications by introducing native signal processing (NSP) instructions into the microprocessor. In this report, I describe the architectural features of the UltraSparc processor and the signal processing functionality of the Visual Instruction Set (VIS). I also measure the performance of a number of real-time applications and signal processing kernels that use the Visual Instruction Set and discuss the implementation of the VIS code generation domain within the Ptolemy environment.

## 2.0 Acknowledgements

I would like to thank my advisor, Professor Edward A. Lee, for his guidance and patience as I tackled this project and climbed a steep learning curve. I would also like to thank Dr. John Reekie for leading this project, Professor Brian Evans for answering my many questions, and the rest of the Ptolemy team for assisting me. Finally, I would like to thank my parents and my friends for their support and encouragement.

# 3.0 Introduction

## 3.1 Motivation

Multimedia applications, such as high-quality audio, speech recognition, video conferencing, and Internet communications, all require intense signal processing. Traditionally, computer architects have made it possible to augment the signal processing capabilities of workstations by inserting DSP boards with high-performance DSP chips. Beyond rapid prototyping, this idea has not really caught on.

A competing approach, termed native signal processing (NSP), moves real-time signal processing functions onto the CPU itself. The UltraSparc processor is one of the first modern RISC chips to include instructions specialized for signal processing. In particular, the Ultrasparc Visual Instruction Set (VIS) is an instruction set that is geared to support image and video processing [1].

Any successful multimedia application, however, will eventually require some sort of audio support. Although not specifically developed for audio signal processing, the VIS is an attractive target for work in this area. The VIS supports data types that pack multiple 16 bit words and includes fixed-point instructions that operate on these multiple words in parallel.

## 3.2 Approach

Ptolemy is a graphical software tool that simulates, prototypes, and synthesizes code for signal processing and communication systems. As a rapid prototyping tool, Ptolemy can be easily extended. New stars and targets can be added to simulate signal processing systems on a number of different platforms. As a code synthesis tool, Ptolemy can stitch together C code from a block diagram in an efficient manner. In this project, I rely on the rapid prototyping and C code synthesis capabilities to develop a VIS code generation domain and to analyze its performance.

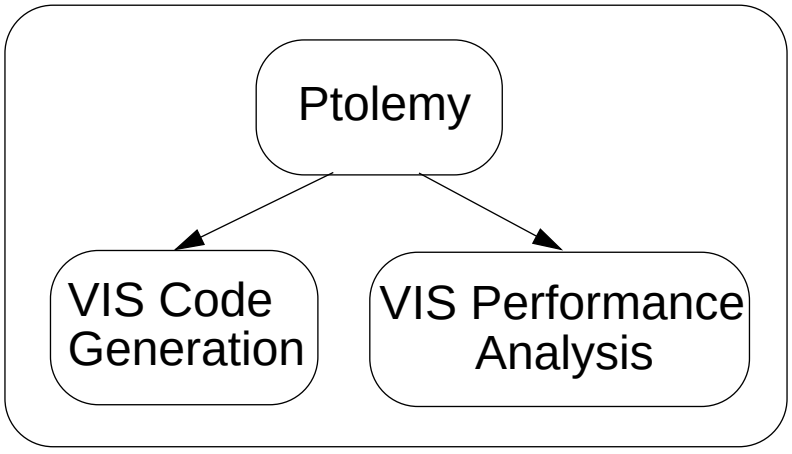


Figure 3.1 Overview of CGCVIS extension.

The VIS code generation domain is developed as an extension of the CGC domain. There are several advantages to this approach. The Ptolemy CGC domain already contains a rich library of signal processing stars. Thus any application can leverage off of this infrastructure. Furthermore, the majority of VIS instructions have a C interface via an inline mechanism [1]. As such, the CGC domain can be easily extended to generate VIS code. In this manner, applications can be developed with and without VIS so that speed and quantization performance of basic signal processing kernels can be measured. The measurement of these key benchmarks will provide a data point in judging the potential of the VIS.

From signal processing kernels, I then move onto audio applications. Much of the current infrastructure within the Ptolemy simulation and code generation environment has been developed for one dimensional signal processing. For this reason, audio applications rather than image or video applications are developed with the VIS.

## 3.3 Overview of Report.

The rest of this report is divided into four sections. Section 3 gives an overview of the Ultrasparc processor and Visual Instruction Set. Emphasis is placed on the instructions and architecture that support one-D signal processing. Section 4 overviews the development of the VIS domain within the CGC domain. New stars, targets, makefiles, and Tcl/Tk interfaces are introduced. Section 5 analyzes both the quantization and speedup performance of the VIS. Performance results compare the VIS code to the integer and floating point C code of the Ultrasparc processor. Finally, Section 6 presents the development of a real time audio application with the VIS.

# 4.0 Ultrasparc Processor and VIS

## 4.1 Overview

The Ultrasparc processor is a superscalar processor that implements a 64 bit RISC architecture [2]. The significant features of the architecture are its ability to issue four instructions per cycle and the enhancement of its floating point unit to support the Visual Instruction Set. The enhancement is provided by a partitioned adder that can add four 16 bit words in parallel and a partitioned multiplier that can multiply four 8x16 bit words in parallel. The partitioned multiplication and addition provides up to four times speedup in signal processing applications.

## 4.2 UltraSparc Architecture

### 4.2.1 Functional Units

Figure 4.1 shows a high level diagram of the five main functional units that make up the Ultrasparc processor.

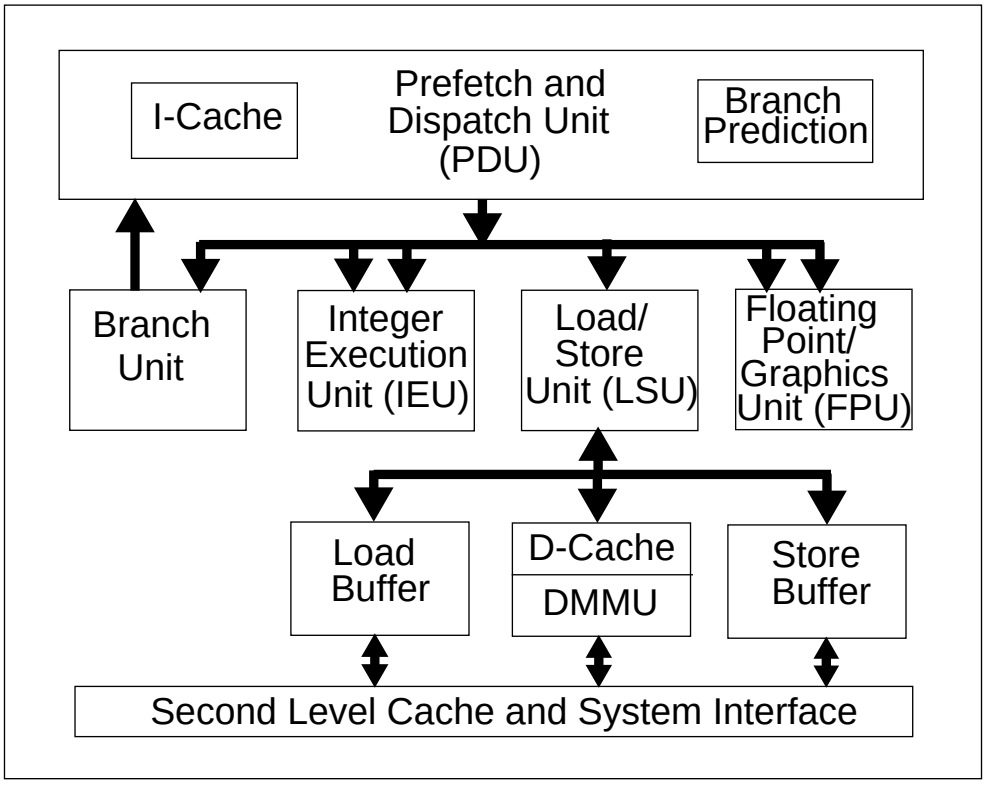


Figure 4.1 Simplified Diagram of the UltraSparc processor

- **Prefetch and Dispatch Unit**

  The prefetch and dispatch unit (PDU) prefetches and dispatches up to four instructions per cycle. Four instructions are prefetched from the 16 kbyte I-Cache. A dynamic branch prediction mechanism is used to tag each pair of instructions and set the prefetch pointer to the next four instructions. A prediction rate of 90% has been reported on typical database applications. The instructions are then partially decoded and stored in a 12-deep instruction buffer. The grouping logic accesses the instruction buffer and issues up to four instructions based on resource availability and dependencies. A grouping, for example, of 2 floating/graphics, 1 integer, and 1 load/store instruction per cycle is possible.

- **Integer Execution Unit**

  The integer execution unit (IEU) performs all the integer arithmetic and logical operations. Two dual 64 bit ALUs perform addition, subtraction, multiplication, and division. A separate 64 adder is provided for virtual address additions for memory instructions.

- **Floating-Point / Graphics Unit**

  The floating-point / graphics unit (FGU) performs all the floating-point arithmetic operations and the VIS fixed-point arithmetic operations. In all, FGU contains 5 separate units: floating-point adder, multiplier, and divider and graphics adder and multiplier. The graphics adder performs single cycle partitioned add and subtract. The graphics multiplier performs three cycle partitioned multiplication and compare.

- **Load Store Unit**

  The load store unit (LSU) executes all the memory operations between the FGU and IEU and the external memory hierarchy. The LSU consists of the data cache, load buffer, store buffer, and data memory management unit.

- **External Cache**

  The external cache services misses from the I-Cache of the PDU and the D-Cache of the LSU.

### 4.2.2 Processor Pipeline

The functional units of the Ultrasparc processor carry out their operations in a dual 9-stage pipeline. Figure 4.2 shows the stages of the pipeline. The prefetch and dispatch unit make up the first three stages of the pipeline. The pipeline then splits into dual four stage integer and floating point pipelines. The pipelines then join together again to resolve traps and write the output.

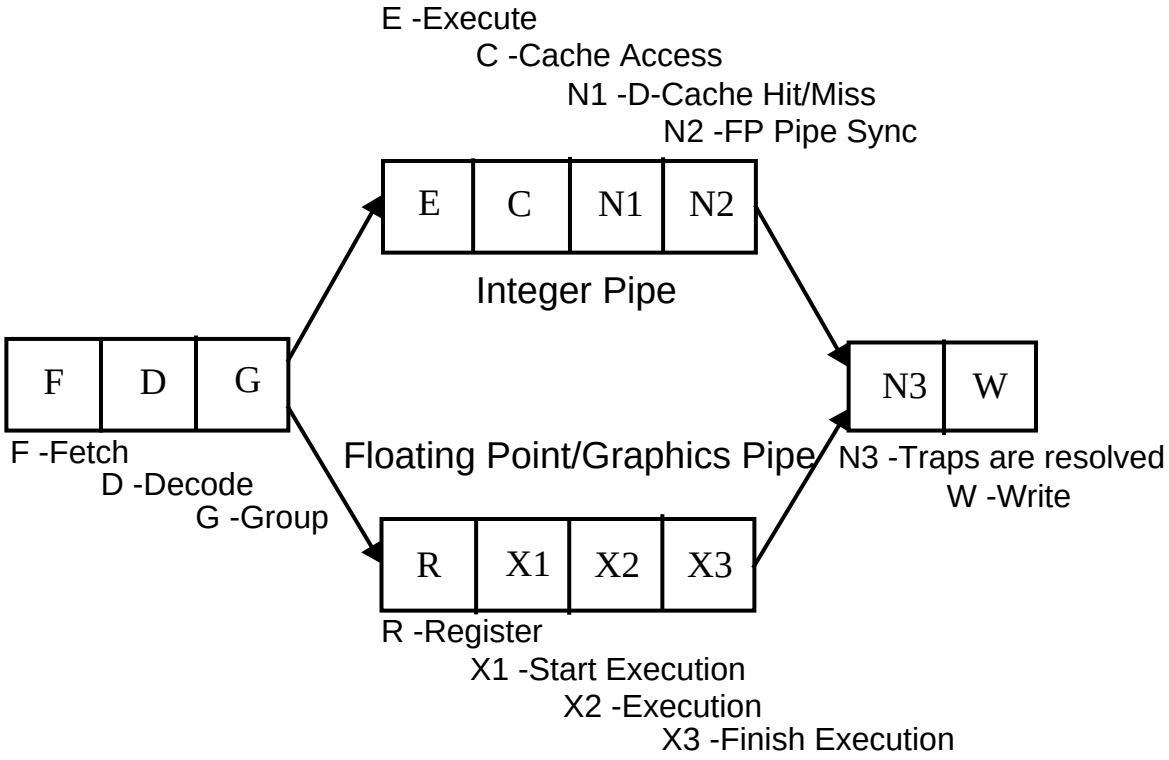


Figure 4.2 Ultrasparc Dual Pipeline

**Table 1. Stages in the Ultrasparc pipeline**

| Stage | Description |
|---|---|
| Fetch (F) | • Up to four instructions are fetched from the I-Cache, along with branch prediction information. |
| Decode (D) | • The fetched instructions are pre-decoded to speed up grouping in the next stage.<br>• The decoded instructions are sent to the Instruction Buffer. |
| Group (G) | • The instructions in the buffer are grouped and dispatched.<br>• Up to four valid instructions are dispatched to the IEU, FPU, and LSU functional units. |
| Execution (E) | • Data from the integer register file is sent to the two integer ALUs.<br>• Results are available by the next cycle. |
| Register (R) | • Corresponding to the E stage of the integer unit, the data from the floating point register file is accessed and further decoded. |

**Table 1. Stages in the Ultrasparc pipeline**

| Stage | Description |
|---|---|
| Cache Access (C) | • Virtual addresses calculated in the E stage are sent to determine whether the access is a hit or a miss in the D-Cache. |
| X1 | • Floating point instructions start their execution. |
| N1 | • A data cache miss/hit or a TLB miss/hit is determined. |
| X2 | • For most floating point instructions, this is the second execution stage. |
| N2 | • Integer pipe waits for the floating point pipe to complete. |
| X3 | • Most floating-point and graphics instructions finish their execution in this stage. |
| N3 | • Integer and floating-point instructions converge to resolve traps. |
| Write (W) | • All results are written to the register files. |

## 4.3 VIS Data Types

The Ultrasparc is a 64 bit processor. It can process (add, multiply, etc.) data types up to 64 bits long. As Figure 4.3 shows, the VIS Data types use the normal 8 bit byte, 16 bit short, 32 bit integer, 32 bit float, and 64 bit double data types. It does not create a data type with a new bit length. However, the novelty is the way the VIS partitions the old data types. In particular, the VIS introduces four new partitioned data types, partitioning the 32 bit float and the 64 bit double so that each will represent multiple 8 and 16 bit words.

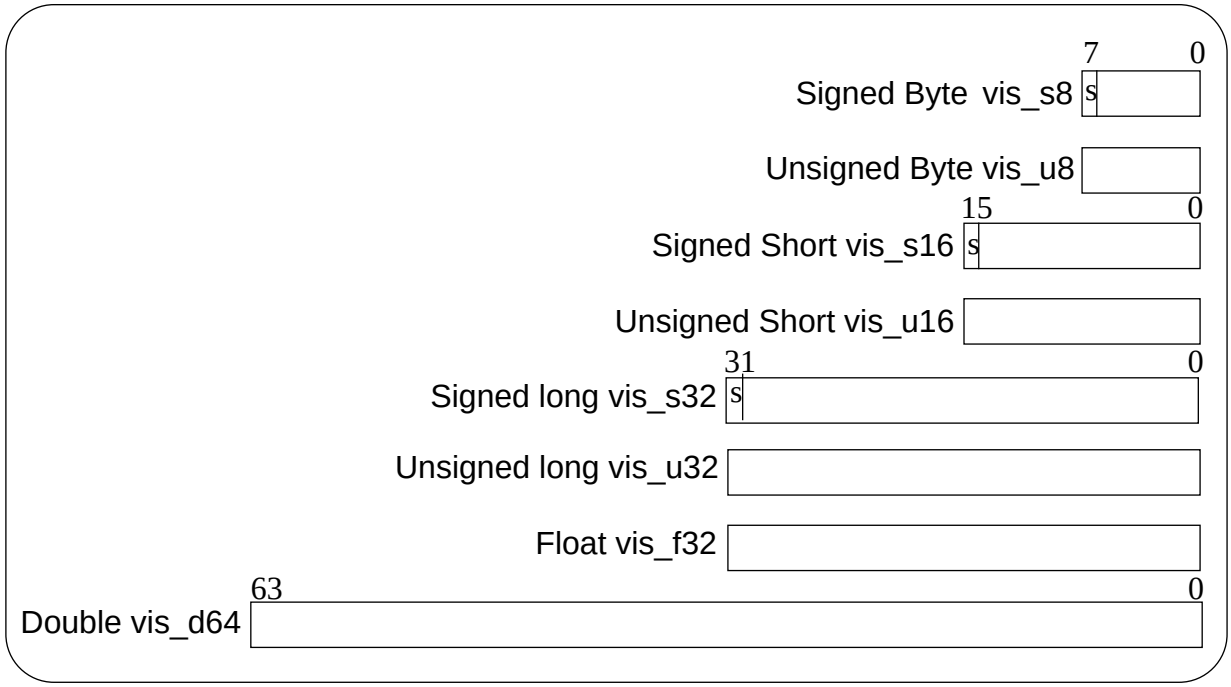


Figure 4.3 VIS Data Types Used

Audio samples come in many different formats and each format requires a different number of bits per audio sample. The VIS allocates 16 bits per audio sample. 16 bit is a convenient representation since it divides evenly into the 32 bit float and the 64 bit double. As Figure 4.4 shows, the VIS data types can pack two audio samples into a 32 bit float and four audio samples into a 64 bit double. The VIS instructions can then perform fixed point adds and multiplies on two or four audio samples at a time. The results can be kept in two intermediate formats: 16 bit fixed point or 32 bit fixed point. The 32 bit intermediate format ensures 16 bit quality audio. The trade-off, however, is speed. The necessary instructions for the 32 bit intermediate format are not as highly parallel as those for the 16 bit intermediate format.

An image is composed of a pattern of pixels. Each pixel can represent shades of gray or the intensities of RGB. The VIS allocates eight bits per pixel sample, which sets the range of values from 0 - 255. Once again this is a convenient representation since it divides evenly into the 32 bit float and the 64 bit double. As Figure 4.4 shows, the VIS data types can pack four pixels into a 32 bit float and eight pixels into a 64 bit double. The VIS instructions can then process four or eight pixels at a time. The results are kept in a 16 bit intermediate format, which is sufficient for most image processing applications.

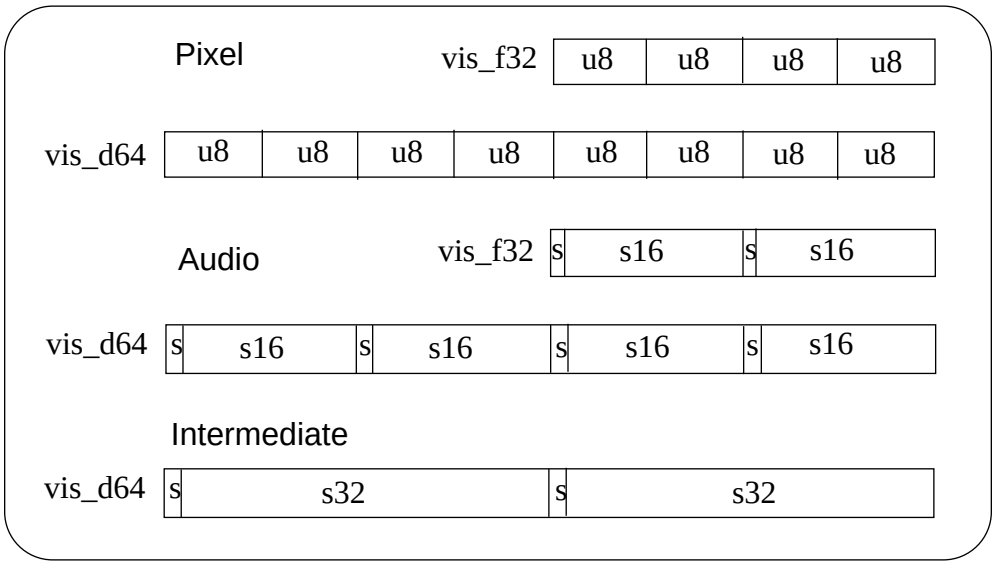


Figure 4.4 VIS partitioned data types.

## 4.4 VIS Instructions

The Visual Instruction Set adds over 50 new instructions to increase the performance of signal processing applications. The instructions can be divided into utility inlines, logical instructions, arithmetic instructions, packing instructions, and array instructions. The following sections present a descriptive overview of the VIS instructions used in one-D signal processing.

### 4.4.1 VIS Utility Inlines

The VIS utility instructions are not actually part of the VIS extension. They are inlined utilities that are generally useful in setting the fields of the graphics status register and performing VIS register to register transfers.

**Table 2. VIS utility inlines**

| Instruction | Description |
|---|---|
| vis_write_gsr()<br>vis_read_gsr() | Assign a value to the Graphics Status Register (GSR) and read the GSR. |
| vis_read_hi()<br>vis_read_lo()<br>vis_write_hi()<br>vis_write_lo() | Read and write the upper and lower 32 bits of a vis_d64 variable. |
| vis_freg_pair() | Join two vis_f32 variables into one single vis_d64 variable. |

**Table 2. VIS utility inlines**

| Instruction | Description |
|---|---|
| vis_to_float() | Place the bit pattern of a vis_u32 variable into a vis_f32 variable without performing floating point conversion. |
| vis_to_double()<br>vis_to_double_dup() | vis_to_double() places two vis_u32 variables into the upper and lower 32 bits of a vis_d64 variable. vis_to_double_dup() places the same vis_u32 variable into the upper and lower 32 bits of a vis_d64 variable. |

In general, the multiplication of two fixed point numbers forms a product whose binary point is no longer aligned with the multiplicands. Thus it may be necessary to normalize the product by bit shifting. The graphics status register (GSR) is used to control the amount of shifting needed to normalize the result of VIS partitioned multiplications. The GSR contains two fields: a 4 bit scale factor and a 3 bit alignaddr offset. The value of the scale factor determines how much to shift each of the products. Figure 4.5 shows the GSR and an example of how to load the scale factor.

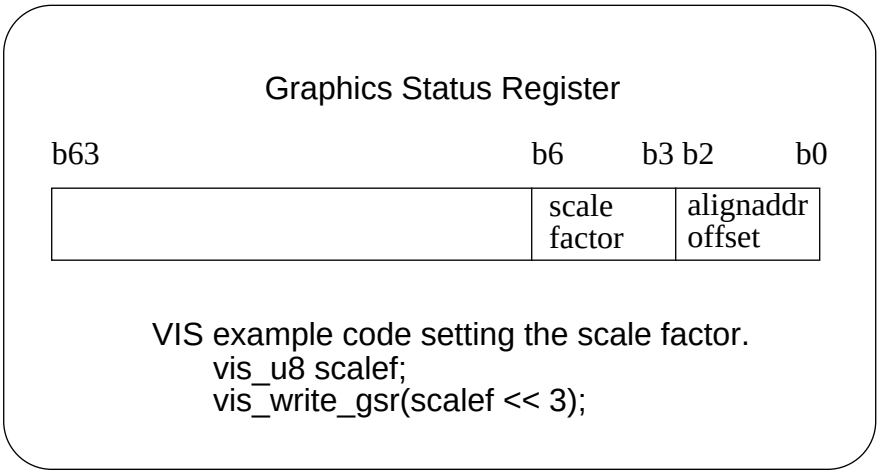


Figure 4.5 Graphics Status Register

The other inline utilities are used for VIS register to register transfers. One set of instructions loads 32 bit words from a vis_d64 register to a vis_f32 register. The vis_read_hi(), for example, extracts the upper 32 bits from a vis_d64 register and places them into a vis_f32 register. The other set of instructions stores 32 bit words from vis_f32 registers to a vis_d64 register. The vis_freg_pair(), for example, joins together data from two vis_f32 registers to pack a vis_d64 register. It is interesting to note here that while there are instructions to perform register to register transfers between vis_f32 and vis_d64 registers, there are no instructions that perform register to register transfers between vis_s16 and vis_f32 registers. Thus, bit shifting is used to load 16 bit audio samples into vis_f32 registers.

### 4.4.2 VIS Logical Instructions

These instructions perform a full suite of partitioned logical operations between VIS variables. There are two versions of the eight basic logical operations (or, and, xor, nor, nand, xnor, ornot, and not). The 32 bit version, marked with the suffix s, operates on the vis_f32 variables, while the 64 bit version operates on the vis_d64 variables. There are also two versions of the six basic compare operations, which generates a bit mask as the result. The 16 bit version, explicitly labeled with the suffix 16, compares two 16 bit partitioned vis_d64 variables, while the 32 bit version, explicitly labeled with the suffix 32, compares two 32 bit partitioned vis_d64 variables.

**Table 3. VIS Logical Instructions**

| Instructions | Description |
|---|---|
| vis_fzero()<br>vis_fone()<br>vis_fzeros()<br>vis_fones() | Set the vis_f32 or the vis_d64 to all ones or all zeros. |
| vis_fsrc()<br>vis_fnot()<br>vis_fsrcs()<br>vis_fnots() | Copy a vis_f32 or a vis_d64 value or its complement. |
| vis_f[or,and,xor,nor,nand,xnor,ornot,not]()<br>vis_f[or,and,xor,nor,nand,xnor,ornot,not]s() | Perform a logical operation between two 32 bit vis_f32 or two 64 bit vis_d64 variables. |
| vis_fcmp[gt,le,eq,ne,lt,ge]16()<br>vis_fcmp[gt,le,eq,ne,lt,ge]32() | Perform a comparison between two 16 bit partitioned or 32 bit partitioned vis_d64 variables and generates a bit mask as the result. |

### 4.4.3 VIS Arithmetic Instructions

The VIS provides instructions to perform parallel addition and multiplication. These instructions are the workhorse responsible for providing most of the speedup in signal processing algorithms.

**Table 4. VIS Arithmetic Instructions**

| Instructions | Description |
|---|---|
| vis_fpadd[16,16s,32,32s]()<br>vis_fpsub[16,16s,32,32s]() | Perform addition or subtraction on two 16 bit partitioned vis_f32 variables (16s), two 16 bit partitioned vis_d64 variables (16), two 32 bit vis_f32 variables (32s), or two 32 bit partitioned vis_d64 variables (32). |

**Table 4. VIS Arithmetic Instructions**

| Instructions | Description |
|---|---|
| vis_fmul8x16() | Perform multiplication between the elements of an 8 bit partitioned vis_f32 variable and those of a 16 bit partitioned vis_d64 variable to produce a 16 bit partitioned vis_d64 result. |
| vis_fmul8x16au()<br>vis_fmul8x16al() | Perform multiplication between the elements of an 8 bit partitioned vis_f32 variable and the upper (au) or lower (al) 16 bits of a vis_f32 variable to produce a 16 bit partitioned vis_d64 result. |
| vis_fmul8sux16()<br>vis_fmul8ulx16() | Perform multiplication between the elements of a 16 bit partitioned vis_d64 variable and those of another 16 bit partitioned vis_d64 variable to produce a 16 bit partitioned vis_d64 result. |
| vis_fmuld8sux16()<br>vis_fmuld8ulx16() | Perform multiplication between the elements of a 16 bit partitioned vis_f32 variable and those of another 16 bit partitioned vis_f32 variable to produce a 32 bit partitioned vis_d64 result. |

The VIS can perform parallel fixed point addition and subtraction on two 16 bit partitioned vis_f32 variables, two 16 bit partitioned vis_d64 variables, or two 32 bit partitioned vis_d64 variables. In fixed point addition, the decimal point is assumed to be aligned, and overflow will result in wraparound. Figure 4.6 diagrams an example of the VIS fixed point addition on two 16 bit partitioned vis_d64 variables.

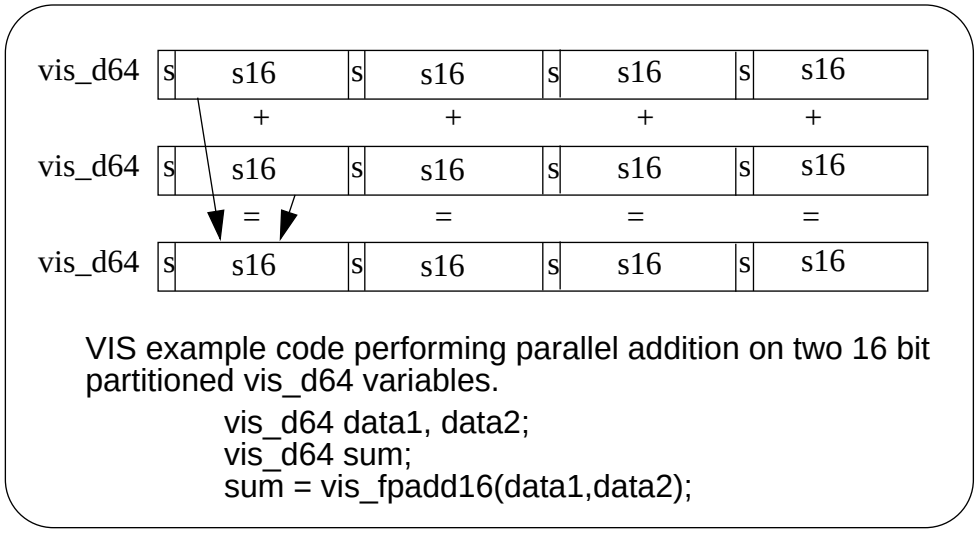


Figure 4.6 16 bit partitioned vis_d64 addition.

The VIS can perform partitioned multiplication on two 16 bit partitioned vis_f32 variables or two 16 bit partitioned vis_d64 variables. The two different partitioned multiplications allows the programmer to keep two different intermediate formats. The 16 bit partitioned vis_f32 multiplication keeps its results in a 32 bit intermediate format, which provides a dynamic range that is good enough for 16 bit quality audio processing. The 16 bit partitioned vis_d64 multiplication keeps its results in a 16 bit intermediate format, which provides greater parallelism and a dynamic range that is good enough for non critical audio processing. Also, as mentioned earlier, there does not exist a parallel 16x16 bit multiplier but only a parallel 8x16 bit multiplier. Therefore, in both cases, the 16 bit partitioned multiplications are performed in three stages.

Figure 4.7 shows the three stages of the 16 bit partitioned vis_d64 multiplication. The first stage multiplies the lower 8 bits of each 16 bit partitioned element in *op1* by the corresponding 16 bit partitioned element in *op2*. Each 24 bit product is signed extended to 32 bits and the upper 16 bits are returned in a 16 bit partitioned *lowerproduct*. The second stage multiplies the upper 8 bits of each 16 bit partitioned element in *op1* by the corresponding 16 bit partitioned element in *op2*. The upper 16 bits of each product are returned in a 16 bit partitioned *upperproduct*. The final stage performs a 16 bit partitioned add between the *upperproduct* and the *lowerproduct* to produce a 16 bit partitioned *result*.

Figure 4.8 shows the three stages of the 16 bit partitioned vis_f32 multiplication. The first stage multiplies the lower 8 bits of each 16 bit partitioned element in *op1* by the correspond in 16 bit partitioned element in *op2*. Each 24 bit product is signed extended to 32 bits, and the entire 32 bits are returned in a 32 bit partitioned *lowerproduct*. The second stage multi plies the upper 8 bits of each 16 bit partitioned element in *op1* by the corresponding 16 bit partitioned element in *op2*. The entire 32 bits are returned in a 32 bit partitioned *upperproduct*. The final stage performs a 32 bit partitioned add between the *upperproduct* and the *lowerproduct* to produce a 32 bit partitioned *result*.

The 16 bit partitioned vis_d64 multiplication has a high degree of parallelism, performing four 16x16 bit multiplies. However, there is a loss of precision in both the first and second stages since all intermediate formats are clipped to 16 bits. Also there are no instructions that can perform a scale change on each of the four products in the *result*. The 16 bit partitioned vis_f32 multiplication, on the other hand, has a lower degree of parallelism, performing only two 16x16 bit multiplies. However, there is no loss of precision since the intermediate formats keep all 32 bits. Furthermore the vis_fpack16 instruction performs a scale change and then packs the 32 bit partitioned *result* into a 16 bit partitioned *packed_result*.

```
16 bit partitioned vis_d64 multiplication
vis_d64 Op1, Op2, product;
vis_d64 UpperProd, LowerProd;
LowerProd = vis_fpmul8ul16(Op1,Op2);
UpperProd = vis_fpmul8sux16(Op1,Op2);
product = vis_fpadd16(LowerProd,UpperProd);
```

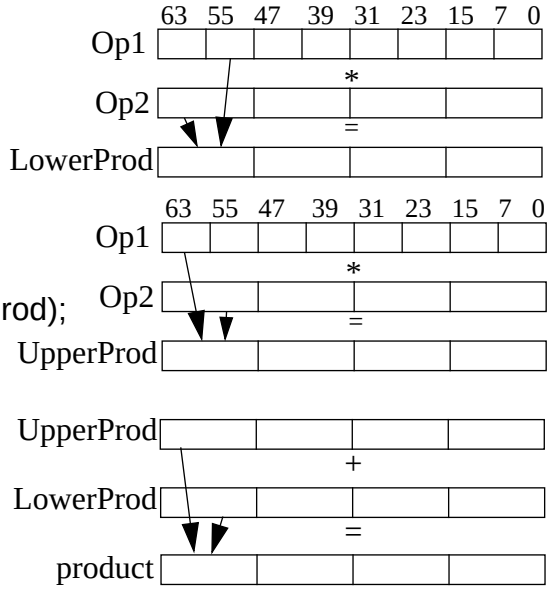


Figure 4.7 16 bit partitioned vis_d64 multiplication.

```
16 bit partitioned vis_f32 multiplication
vis_f32 Op1, Op2, packed;
vis_d64 UpperProd, LowerProd, product;
vis_write_gsr(1<<3);
LowerProd = vis_fpmuld8ul16(Op1,Op2);
UpperProd = vis_fpmuld8sux16(Op1,Op2);
product = vis_fpadd32(LowerProd,UpperProd);
packed = vis_fpack16(product);
```

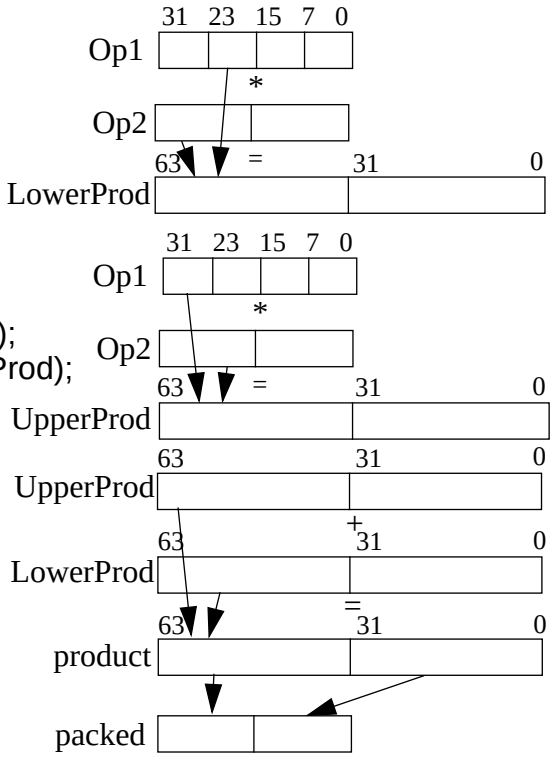


Figure 4.8 16 bit partitioned vis_f32 multiplication.

### 4.4.4 VIS Data Formatting Instructions

The VIS data formatting instructions are used for packing VIS data and for performing VIS register to memory transfers. The packing instructions convert 16 or 32 bit data to lower precision fixed point data. Input values are shifted by the amount in the GSR and then clipped to fit within the dynamic range of the output.

**Table 5. VIS Data Formatting Instructions**

| Instructions | Description |
|---|---|
| vis_fpack16() | Clips four 16 bit fixed values to four unsigned 8 bit pixel values. |
| vis_fpack32() | Clips two 32 bit fixed values to two unsigned 8 bit pixel values. |
| vis_fpackfix() | Converts a 32 bit partitioned vis_d64 variable to a 16 bit partitioned vis_f32 variable. |
| vis_fexpand() | Converts four unsigned 8 bit pixel values to four 16 bit fixed values. |
| vis_fpmerge() | Merges two 8 bit partitioned vis_u32 variables into a single 8 bit partitioned vis_d64 variable by selecting bytes in an alternating fashion. |
| vis_falignaddr()<br>vis_faligndata() | Calculate 8 byte aligned address and extract an arbitrary 8 bytes from two 8 byte aligned addresses. |
| vis_edge[8,16,32]() | Compute a mask used for partial storage at an arbitrarily aligned start or stop address. |
| vis_pst_[8,16.32]() | Write mask enabled 8, 16, 32 bit components from a vis_d64 value to memory. |
| vis_st_u[8,8_i,16,16_i]()<br>vis_ld_u[8,8_i,16,16_i]() | Perform 8 and 16 bit loads and stores to and from floating point registers. |
| vis_pdist() | Compute the absolute value of the difference between the two pixel pairs. |

## 4.5 Fixed Point Arithmetic

The VIS performs fixed point arithmetic. Fixed point arithmetic generally requires less decoding than floating point and is therefore faster. However, the effects of overflow and loss of precision are greater for fixed point arithmetic, and therefore the programmer must take greater care in adding and multiplying fixed point numbers.

As Figure 4.9 illustrates, a 16 bit fixed point number is just a binary representation of a number. Bits to the left of the decimal point represent the sign and integer part of the number, while those to the right represent the fraction part of the number. There is no physical decimal point. The position of the decimal point must be implicitly defined by the user. Once it is defined, the dynamic range of the fixed point number is set. In Figure 4.9, the decimal point is

located at bit 14 so that the dynamic range of the fixed point numbers, assuming two's complement representation, is from -one to one.

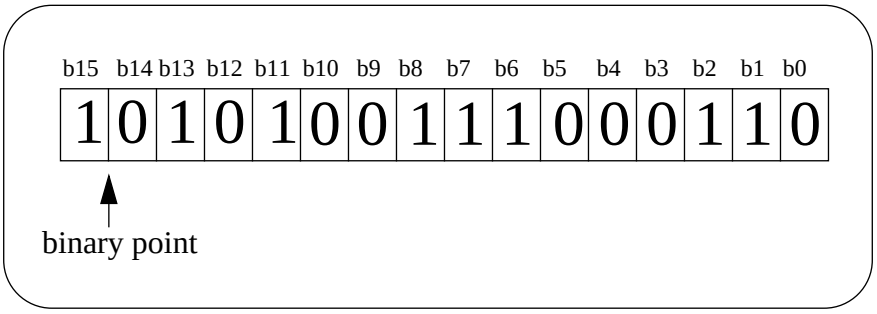


Figure 4.9 16 bit fixed point number.

When computing with fixed point numbers, the programmer should minimize overflow and minimize the loss of precision. The loss of precision is introduced by quantization effects from two sources: A/D and D/A conversions and multiplication [3]. The former is discussed in section 5.4 Quantization Performance, and the second, multiplication, is presented below.

When two fixed point numbers are multiplied, there can be a loss of precision through quantization and a scale change. Figure 4.10 shows the multiplication of two fixed point numbers of length N. Notice that the product is of length 2N. If only N bits of the product are returned, the product is quantized by either dropping or rounding up the least significant N bits. Also notice that the binary point of the product has shifted to the right by one. The product no longer has the same dynamic range as that of the two inputs. It is up to the user to keep track of the binary point and shift the result back if necessary.

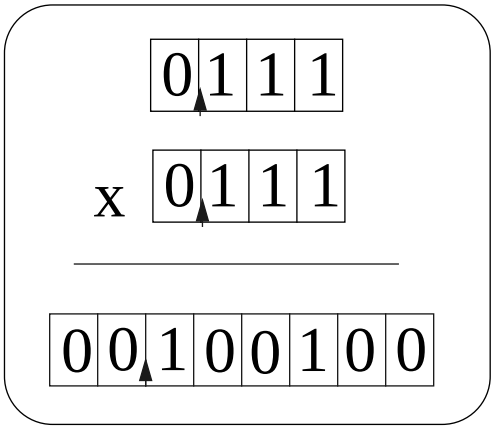


Figure 4.10 Fixed Point Multiplication

Overflow occurs when an arithmetic operation produces a result that is too large to fit within the original fixed point representation. There are several ways of dealing with overflow.

- **Saturation Arithmetic**

When overflow occurs, saturation arithmetic sets the result to the highest possible value. Saturation is often used since it has a predictable behavior, setting all overflow values to the maximum value.

- **Wraparound Arithmetic**

  When overflow occurs, wraparound arithmetic discards the overflow bits and uses the remaining bits “as is”. The final result therefore wraps around and can take on values of the opposite sign. For example, Figure 4.11 shows the addition of two sine waves that results in wraparound. Wraparound is used because it is simple to implement. Also the accumulation of several overflows can still lead to the correct value even if intermediate ones wraparound [4].

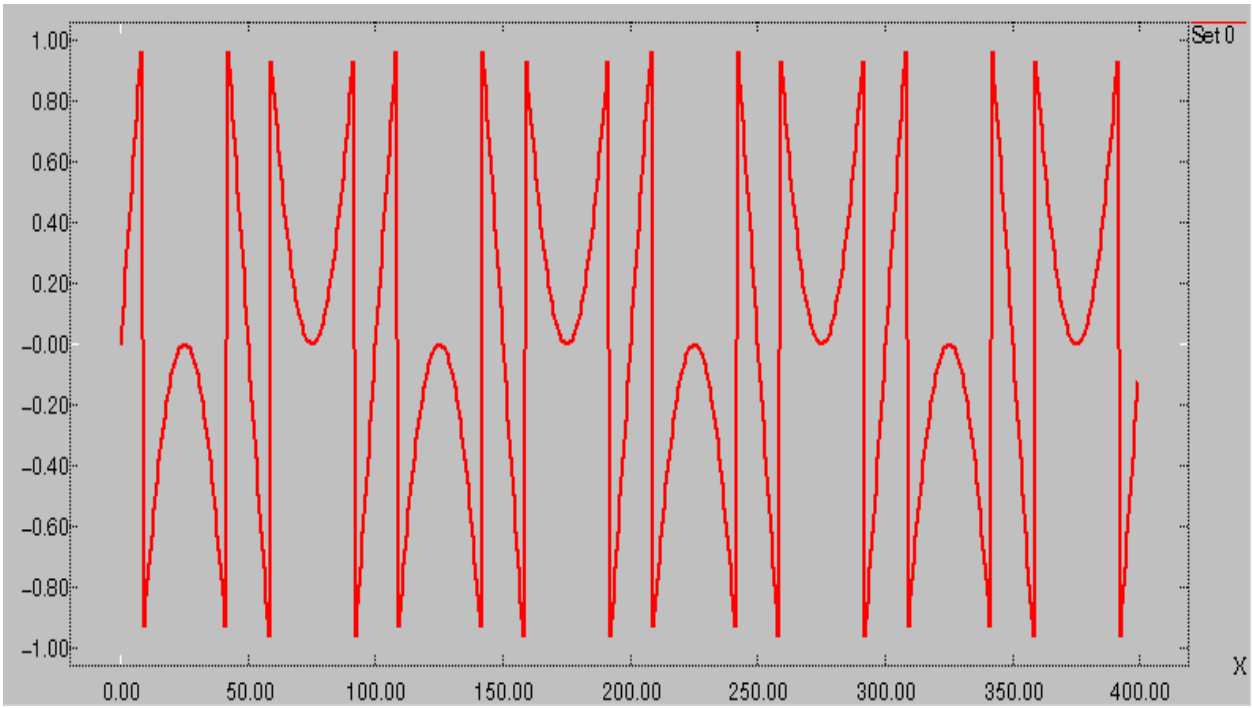


Figure 4.11 Wraparound in adding two sine waves.

# 5.0 CGCVIS

The code generation domains in Ptolemy consist of a set of Stars, Targets, and Schedulers that generate code for a particular computational model. The CGC domain, for example, generates C code according to the synchronous dataflow (SDF) semantics. The CGCVIS domain extends the CGC domain by synthesizing C code with VIS instructions. The CGCVIS domain contains a new set of Stars, Targets, and an enhanced Tcl/Tk interface.

## 5.1 Stars

The purpose of the CGCVIS domain was to compare the performance of the Visual Instruction Set against regular C code. A parallel set of CGCVIS stars were created from the CGC library.

**Table 6. CGCVIS Stars**

| Star Name | Description |
|---|---|
| CGCVISAddSh | Add the corresponding 16-bit fixed point numbers of two 16 bit partitioned vis_d64 particles. |
| CGCVISBiquad | Implements an IIR biquad filter using the inner product method. |
| CGCVISFFTCx | Implements the radix 2 algorithm to calculate the FFT of a complex input sequence. |
| CGCVISFIR | Implements an FIR filter using a matrix vector multiplication. |
| CGCVISInterleaveIn | Reads four 16-bit audio samples at each invocation. The audio samples are read in through the left and right stereo channels and are interleaved into a single output stream. |
| CGCVISInterleaveOut | Sends out a stream of audio samples to the audio device. The stream of audio interleaves the left and right stereo channels. |
| CGCVISMpyDblSh | Multiply the corresponding 16-bit fixed point numbers of two 16 bit partitioned vis_d64 particles. The vis_fmuld8sux16() and vis_fmuld8ulx16() instructions are used which keep a 32 bit intermediate format. |
| CGCVISMpySh | Multiply the corresponding 16-bit fixed point numbers of two 16 bit partitioned vis_d64 particles. The vis_fmul8sux16() and vis_fmul8ulx16() instructions are used which keep a 16bit intermediate format. |
| CGCVISPackSh | Takes four float particles, casts them into four signed 16-bit fixed point numbers, and packs them into a single 64-bit float particle. |

**Table 6. CGCVIS Stars**

| Star Name | Description |
|---|---|
| CGCVISParametricEq | Implements a parametric biquad filter. The user supplies the parameters such as Bandwidth, Center Frequency, and Gain. The digital biquad coefficients are quickly calculated based on the procedure defined by Schpak. |
| CGCVISStereoBiquad | Implements an IIR biquad filter that assumes the input is a stream of audio samples with the left and right stereo channels interleaved. |
| CGCVISStereoIn | Reads four 16-bit audio samples at each invocation. The audio samples are read in through the left and right stereo channels and are sent out as separate streams. |
| CGCVISStereoOut | Sends out a stream of audio samples to the audio device. The stream of audio enters as separate left and right stereo channels but is interleaved before being sent to the audio driver. |
| CGCVISSubSh | Subtract the corresponding 16-bit fixed point numbers of two 16 bit partitioned vis_d64 particles. |
| CGCVISUnpackSh. | Takes a single 64-bit float particle, unpacks them into four 16-bit fixed point numbers, and casts them into four float particles. |

The CGCVIS stars use the partitioned multiplies and adds to speed up the compute intensive portions of signal processing applications. The data format is assumed to be a 4x16 bit word, which we called a “quad-word”. The CGCVISPack and CGCVISUnpack stars are used to pack a quad-word for processing and unpack a quad-word so that the results can be displayed.

## 5.2 Targets

When compiling VIS code, several compile options must be chosen. The -fast option with the optimization option level -xO[1|2|3|4|5] chooses the fastest code generation option available on the compile time hardware. Also, the vis.il file must be included in the command line to resolve VIS function calls. Finally, the -xchip-ultra specifies the target architecture and the -xarch-v8plusa identifies the instructions that the compiler can use.

The most convenient way to encapsulate this information in Ptolemy is through the CGCMakefileTarget. This particular target generates a separate makefile with the above options to compile the source code. Several other alternatives were explored, such as developing a separate CGCVISTarget or deriving all VIS stars from a VISBase star. Since just compile time options needed to be set, the former proved unnecessary causing much code duplication, and the latter created problems with multiple inheritance issues.

## 5.3 Tcl/Tk Interface

To support real time user control, a new Tcl/Tk interface was developed by John Reekie. Figure 5.1 shows a parametric biquad block diagram and its user interface. Each star and its parameters are named and can be associated with a single user control panel. The control panel can be made of predefined components in the Tcl/Tk library or custom made components. During run time, each component controls a specific parameter of a named star.

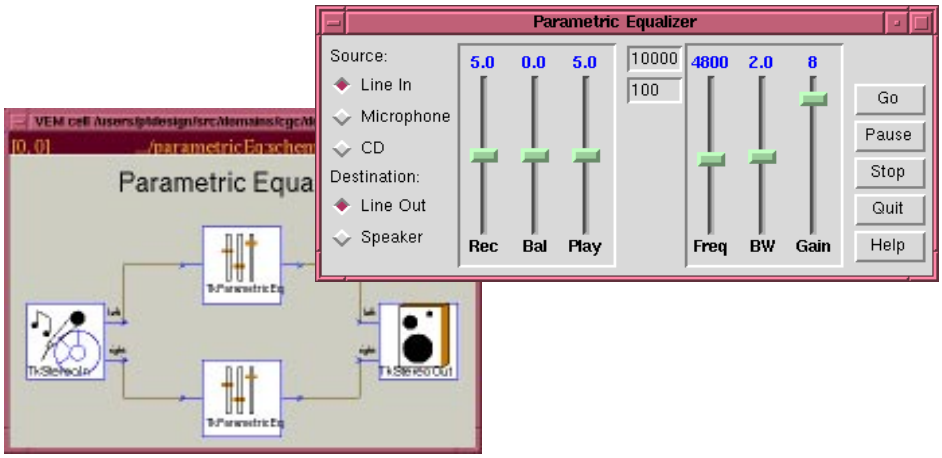


Figure 5.1 Parametric Equalizer

For example, the two parametric biquad filters in Figure 5.1 are named “*left*” and “*right*”. Previously, each star would generate its own control panels and separate controls, but the new interface allows both stars to connect to a single user control panel. The “*gain*” control is used to adjust the gain parameter in both stars.

# 6.0 Performance of Key Benchmarks

## 6.1 Overview

The goal of the Visual Instruction Set is to speed up signal processing functions. In this section, the performance advantage of the VIS is measured on three basic signal processing kernels: FIR, biquad, and FFT. Each of these kernels is selected because they are representative of other more complex signal processing kernels. For example, the FIR will characterize how much the VIS speeds up basic multiply and add routines. The biquad will demonstrate how well the VIS handles a feedback element, and the FFT will show how well the VIS handles nested for loops.

## 6.2 Matrix-Vector Approach

The question remains: Given this jumble of VIS instructions and data types, how can one speed up basic signal processing kernels? The signal processing kernels are linear time invariant systems that can be rewritten as matrix-vector operations. The matrix-vector operations can then be implemented with the Visual Instruction Set. The partitioned additions and multiplications of the VIS allows a systems to process a vector of inputs and produce a vector of outputs in the same amount of time as that of a single input/single output system. The VIS therefore uses these partitioned additions and multiplications to reduce the overall number of adds and multiplies and speed up basic signal processing operations.

### 6.2.1 FIR

The convolution of an input sequence and a real finite impulse response filter is an order N operation. For every output, N multiplications and N-1 additions are used in the convolution. Figure 6.1 shows how this problem can be written as a matrix-vector multiplication. The matrix-vector approach basically unrolls the convolution operation, multiplying an entire row of the coefficient matrix with the input vector.

The structure of the coefficient matrix is known as a circulant matrix. Each row is just a shifted copy of the previous one. Since the VIS has implemented 16 bit partitioned adders and multipliers, optimal performance is achieved if the input, coefficient matrix, and output are 8-byte aligned. The VIS partitioned instructions can then be used to reduce the number of multiplications to N/4 and the number of additions to N/4-1 to provide a four times speed up.

$$y[n] = \sum_{k=0}^{n} h[k] \times x[n-k]$$

$$\begin{bmatrix} y[11] \\ y[10] \\ y[9] \\ y[8] \end{bmatrix} = \begin{bmatrix} h[0] & h[1] & h[2] & h[3] & h[4] & h[5] & h[6] & h[7] & h[8] & h[9] & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & h[0] & h[1] & h[2] & h[3] & h[4] & h[5] & h[6] & h[7] & h[8] & h[9] & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h[0] & h[1] & h[2] & h[3] & h[4] & h[5] & h[6] & h[7] & h[8] & h[9] & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & h[0] & h[1] & h[2] & h[3] & h[4] & h[5] & h[6] & h[7] & h[8] & h[9] & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x[11] \\ x[10] \\ x[9] \\ x[8] \\ \cdots \\ x[1] \\ x[0] \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 6.1 Two versions of the FIR filter.

### 6.2.2 Biquad

The biquad filter is an infinite impulse response filter so it cannot be directly implemented by convolution. Instead, a recursive computational algorithm can be developed from its second order difference equation. Each output, y[n], depends on the previous two outputs, {y[n-1], y[n-2]}, the current input, x[n], and previous input samples, x[n-1] and x[n-2]. However, there are an infinite number of computational algorithms that might prove more efficient. Figure 6.2 shows the difference equation reframed into a classic state space form. In fact, the biquad can be put into a canonical controllable form, where all the states are updated by a matrix-vector multiplication. The output is then a function of the state vector and the current input vector.

Notice however that the size of the matrix is not optimal. Remember the optimal VIS multiplications and additions are 4x4. Also Figure 6.2 shows that the state vector must be updated for every new output. Another approach is to eliminate this feedback dependency This approach, shown in Figure 6.3, is called the inner product method [5].

The inner product method eliminates the dependency of the feedback by recursively expanding the difference equation. The entire set of outputs {y[n], y[n+1], y[n+2], y[n+3]} depends on the same previous outputs {y[n-1] and y[n-2]}. Hence the VIS partitioned instructions can be used to calculate all the outputs at the same time with no direct dependencies. However, the number of multiplies and adds greatly increases. Even with the VIS partitioned multiplication and addition, the final number of multiplies equals that of a regular biquad. Thus there is little performance gain here.

$$y[n] + b_0*y[n-1] + b_1*y[n-2] = a_0*u[n] + a_1*u[n-1] + a_2*u[n-2]$$

$$\begin{bmatrix} x_1[n+1] \\ x_2[n+1] \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -b_1 & -b_0 \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u[n]$$

$$y[n] = \begin{bmatrix} a_2 - a_0b_1 & a_1-a_0b_0 \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} a_0 \end{bmatrix} u[n]$$

Figure 6.2 State space formulation of Biquad Filter

$$y[n] = - b_0*y[n-1] - b_1*y[n-2] + a_0*u[n] + a_1*u[n-1] + a_2*u[n-2]$$

$$\begin{aligned} y[n+1] &= - b_0*y[n] - b_1*y[n-1] + a_0*u[n+1] + a_1*u[n] + a_2*u[n-1] \\ &= (b_0^2-b_1)*y[n-1] + b_0*b_1*y[n-2] + a_0*u[n+1] + (a_1-a_0*b_0)*u[n] \\ &\quad + (a_2-a_1*b_0)*u[n-1] - a_2*b_0*u[n-2] \end{aligned}$$

$$\begin{aligned} y[n+2] &= - b_0*y[n+1] - b_1*y[n] + a_0*u[n+2] + a_1*u[n+1] + a_2*u[n] \\ &= (2*b_0*b_1-b_0^3)*y[n-1] + (b_1^2-b_0^2*b_1)*y[n-2] + a_0*u[n+2] \\ &\quad + (a_1-a_0*b_0)*u[n+1] + (a_0*b_0^2-a_1*b_0-a_0*b_1+a_2)*u[n] \\ &\quad + (a_1*b_0^2-a_2*b_0-a_1*b_1)*u[n-1] + (b_0^2*a_2-a_2*b_1)*u[n-2] \end{aligned}$$

$$\begin{aligned} y[n+3] &= - b_0*y[n+2] - b_1*y[n+1] + a_0*u[n+3] + a_1*u[n+2] + a_2*u[n+1] \\ &= (b_0^4-3*b_0^2*b_1+b_0^2)*y[n-1] + (b_0^3*b_1-2*b_0*b_2)*y[n-2] + a_0*u[n+3] \\ &\quad + (a_1-a_0*b_0)*u[n+2] + (a_2-2*a_1*b_0+a_0*b_0^2)*u[n+1] \\ &\quad + (a_1*b_0^2-a_0*b_0^3-a_1*b_1-a_2*b_0+2*a_0*b_0*b_1)*u[n] \\ &\quad + (a_2*b_0^2-a_2*b_1-a_1*b_0^3+2*a_1*b_0*b_1)*u[n-1] \\ &\quad + (2*a_2*b_0*b_1-a_2*b_0^3)*u[n-2] \end{aligned}$$

Figure 6.3 Inner Product Method

### 6.2.3 FFT

The implementation of the FFT was straight forward. Figure 6.4 shows the flow graph of a complete decimation in time decomposition of an 8 point FFT. The VIS flow graph is similar, except that at each node in the flow graph, four multiplications and four additions take place. This takes advantage of the VIS partitioned instructions and reduces the overall number of multiplies by a factor of four.

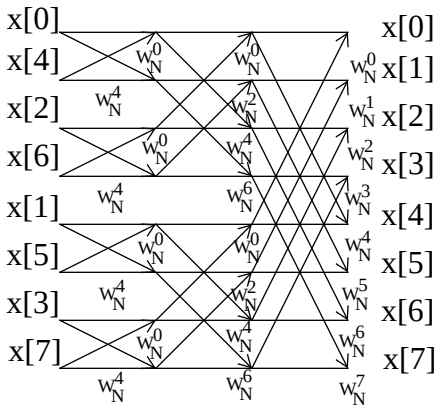


Figure 6.4 Flow graph of 8 point FFT decimation in time

### 6.2.4 FIR in depth

The FIR can be reformulated as a matrix-vector multiplication and implemented with VIS partitioned multiplications and additions. The implementation within the Ptolemy environment is divided into three sections: setup, go, and wrapup.

In the setup method, memory is allocated for the coefficient matrix. Most of the VIS partitioned instructions operate on 8-byte aligned data. Non-aligned data can be accessed with vis_alignaddr() and vis_faligndata() instructions, but these are very cycle expensive operations. Therefore, to optimize the VIS instructions, the coefficient matrix and the vector of current and past input data (x) are aligned along an 8-byte boundary.

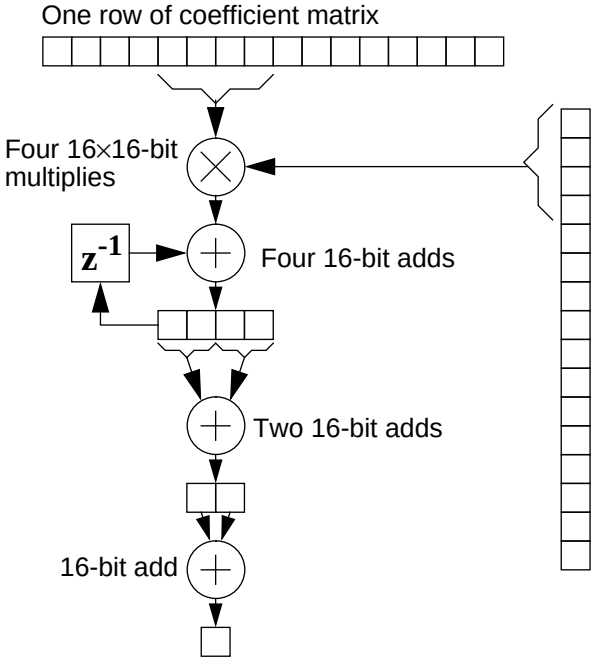


Figure 6.4 FIR filter calculation (courtesy of John Reekie)

In the go method, the matrix-vector multiplication is performed. Figure 6.4 illustrates the multiplication of one row of the coefficient matrix by the input vector. The first multiplication and addition accumulates four partial sums in a vis_d64 register. The code is:

```
for (outerloop = 0; outerloop < n; outerloop++) {
        data = src[nminusk];
        tapvalue = *tapptr0++;
        /* take product */
        pairhi = vis_fmul8sux16(data, tapvalue);
        pairlo = vis_fmul8ulx16(data, tapvalue);
        pair = vis_fpadd16(pairhi,pairlo);
        /* accumulate */
        accum0 = vis_fpadd16(accum0,pair);
        ...
}
```

Notice that the 16 bit partitioned vis_d64 multiplications are used. This multiplication keeps all the results in a 16 bit intermediate format. Therefore, if audio samples were being filtered here, the output would not be 16 bit quality audio.

In order to produce a single output, the four partial sums need to be unpacked from the vis_d64 register and summed together, as illustrated by the lower adders in Figure 6.4. This is performed by taking the lower pair and the upper pair and adding them (vis_fpadd16s),

transferring the result into integer registers (the cast to vis_u32), bit-shifting to separate the two 16-bit words, and summing:

```
splithi = vis_read_hi(accum0);
splitlo = vis_read_lo(accum0);
split = vis_fpadd16s(splithi, splitlo);
accum0u = *((vis_u32*) &split);

splithihi = (short)(accum0u>>16);
splitlolo = (short)(accum0u&0xffff);
y11 = splithihi + splitlolo;
...
```

This piece of code illustrates some of the awkwardness associated with manipulating multiple datawords. As mentioned earlier, there are no vis_d64 to vis_s16 register to register transfer instructions so bit shifting is necessary. The matrix-vector multiplication is repeated four times for each row of the coefficient matrix, and the 16-bit results obtained from each row are then combined into a single vis_d64 register.

## 6.3 Timing Performance

The VIS can potentially provide a four times speedup to signal processing applications. In fact, Sun has reported a speedup from 1.5 to 10 times on basic image processing functions.

**Table 7. Timing Results**

| Kernels | VIS/Float | VIS/ Integer |
|---|---|---|
| FIR | 3.43 | 6.33 |
| 256 Pt FFT | 1.28 | N/A |
| Biquad | 0.99 | 2.76 |

Table 6 shows the performance of three basic signal processing kernels: a single sample FIR filter, a single sample biquad filter, and a 256 point complex FFT. Both VIS and C versions of each kernel function were written: all versions are functionally equivalent but exercise different computational units of the Ultrasparc, such as the integer unit, the floating point unit, and the VIS unit. Timing results are obtained on an Ultrasparc-I workstation running at 143 MHz. A high resolution Unix timer, gethrtime(), measures the execution speeds of the functions. Several iterations were run to obtain a mean time:

```
start = gethrtime();

for (j = 0; j < NTIME; j ++) {
```

```
            firvis_filter()
    }
    end = gethrtime();
    time = (float) (end - start)/1000000/NTIME;
```

The speedup of the VIS code over the floating point C code and the integer code were calculated. The results range from no speedup to just over 2 times speedup for the VIS vs the float ing point unit and 2 times speedup to over 6 times speedup for the VIS vs integer unit. One immediate observation is that the VIS does not outperform its floating point unit by the same amount as that of the VIS and integer unit.

The FIR filter proved to be consistently faster when written with the VIS code over both the floating and integer units. There are two reasons. First the algorithm is simple. The FIR is just a straight inner product between the filter taps and the inputs. There is no introduction of a feedback element or nested for loops. Second, the convolution operation is easy to rewrite with VIS code. The convolution operation can be rewritten as a matrix vector multiplication, and the VIS partitioned instructions can be applied to reduce the overall number of adds and multiplies.

In implementing the FIR filter, two design choices were made. First, the 16 bit partitioned vis_d64 multiplications are used, which keeps a 16 bit intermediate format. For higher precision calculations, a 32 bit intermediate format can be kept, but the VIS/Float speedup is reduced from 3.43 to 2.00. Second, 16 bits are used for the input and the filter coefficients. This requires 16x16 bit multiplies, which are carried out in three stages. For many applications, greater quantization effects are tolerated so that 8 bits are sufficient for the filter coefficients. All partitioned multiplications can therefore be carried out in a single stage, which should lead to a greater speedup.

The biquad filter showed little or no gain when implemented with VIS. One of the limitations of the matrix vector approach is its ability to handle feedback. The feedback introduces a loop such that every output depends on the previous output. Hence the outputs can no longer be computed separately or in parallel. Several methods, such as the state space and inner product method, were used to eliminate the feedback loop. But more multiplies and adds were generated so that the VIS partitioned instructions showed little gain.

The FFT is implemented here with a standard radix 2 algorithm. Three nested for loops are used to calculate all the nodes of the butterfly. However the algorithm is modified so that, at each node, four multiplies and additions take place. The speedup here is modest since more shuffling and unpacking of data is required.

## 6.4 Quantization Analysis

When implementing a digital filter or algorithm, the designer must minimize the effects of quantization errors. Luckily, there are many choices available to minimize quantization errors, such as the number of bits to allocate in the A/D conversion of the inputs, the representation of

the coefficients in the digital filter, the finite-precision arithmetic of the hardware architecture, and the actual implementation structure of the digital filter. In the following sections, we discuss the implementation of a digital filter for audio processing using the VIS architecture. At each step of the way, we examine the design choices - the VIS architecture dictated many of these choices- and the related quantization effects on the system.

### 6.4.1 Input Quantization and Scaling

At the input, quantization occurs to transform the continuous audio signal to a finite set of values. This operation can be mathematically represented as:

$$\hat{x} = Q_B[x] = X_m\left(-b_o + \sum_{i=1}^{B} b_i 2^{-i}\right) \quad \text{Equation 1}$$

Two parameters here need to be defined: the number of bits, B, to allocate for the input and the scaling of the signal, $X_m$, to prevent overflow. For example, in section 6, the graphic equalizer reads in audio samples from a CD that have been linearly encoded into 16 bits. This fits nicely within the VIS framework. The VIS allocates 16 bits per audio sample. If no scale conversion is necessary, the 16 bit audio samples are directly stored into the VIS variables and $X_m$ is set to $2^{15}$. The quantization error is given by Eq (2) and can be used to model the effects of the quantizer, $Q_B$. The effects of $Q_B$ are essentially replaced as an additive white noise signal and an overall metric, known as the signal-to-noise ratio, is used to quantify its effect. For 16 bit CD quality audio, the signal-to-noise ratio is around 90-96 dB and is given by Eq (3).

$$-\frac{(X_m 2^{-B})}{2} < e \leq \frac{(X_m 2^{-B})}{2} \quad \text{Equation 2}$$

$$SNR = 6.02B + 10.8 - 20\log\left(\frac{X_m}{\sigma_x}\right) \quad \text{Equation 3}$$

However, if the audio samples are sent to a filter with a positive gain, overflow may occur. In the graphic equalizer pictured in Figure 7.1, the user can adjust the volume control to scale down the amplitude of the incoming audio stream. This minimizes the occurrence of overflow but simultaneously increases $X_m$. Eq (2) shows that increasing $X_m$ increases the size of the quantization errors, and Eq (3) shows that increasing $X_m$ reduces the signal-to-noise ratio.

### 6.4.2 Coefficient Quantization

Coefficient quantization represents another source of quantization error. In the VIS implementation, the coefficients are quantized to 8 or 16 bit accuracy. The finite word length here

determines the density of points at which the poles and zeros of a transfer function can be placed. A pole (zero) grid is helpful in visualizing this [5]. It displays the z-plane with the allowed pole (zero) positions. Two quantization effects are evident. First, the poles (zeros) of the desired transfer function are shifted to new positions. Second, there exists the possibility that poles near (but within) the unit circle are moved outside, making the transfer function unstable.

The graphic equalizer in section 6 is composed of a cascade of parametric biquad filters. Two implementations, one in floating point C code and one in VIS code, were implemented. The floating point C version uses the full 64 bit word to represent the filter coefficients while the VIS version uses 16 bit accuracy. The audible effect of coefficient quantization was greater in the latter.

One approach to analyzing the effects of coefficient quantization is by designing a series of filters with different wordlengths and measuring the error in the frequency responses [5]. For instance, to analyze the effect of coefficient quantization on the passband region, the relative error in the passband is measured between the quantized transfer function and infinite precision transfer function:

$$RE = \begin{cases} \dfrac{H_{MAX} - H_{MIN} - A_M}{A_M} & H_{MAX} - H_{MIN} > A_M \\ 0 & H_{MAX} - H_{MIN} \le A_M \end{cases} \quad \text{Equation 4}$$

$H_{MAX}$ and $H_{MIN}$ represent the maximum and minimun values (in dB) of the passband response in the quantized transfer function, and $A_M$ represents the ripple (in dB) in the infinite precision transfer function. Plotting the relative error (RE) for several different finite wordlengths can show general trends as to how sensitive the particular filter design is to coefficient quantization.

### 6.4.3 Roundoff Noise

Another important source of quantization errors is the finite precision arithmetic. Section 3 described the operations of the VIS fixed point arithmetic. All VIS partitioned multiplications round the results up before quantization. When these partitioned instructions are used in implementing biquad filters, the quantization effects can be modeled in a similar manner as the input quantization. That is, the quantization resulting from rounding up fixed point multiplication can be modeled as additive white noise.

Figure 6.5 shows a second order direct form IIR filter with a quantizer in the feedback loop to model the quantization effects of the finite arithmetic. The quantizer can actually be place in any one of several positions. Even two quantizers can be used. However, since the overall results do not vary much, the simplest scheme is kept. Furthermore, the quantizer is replaced with an additive noise source, e[n], to model the round off noise.

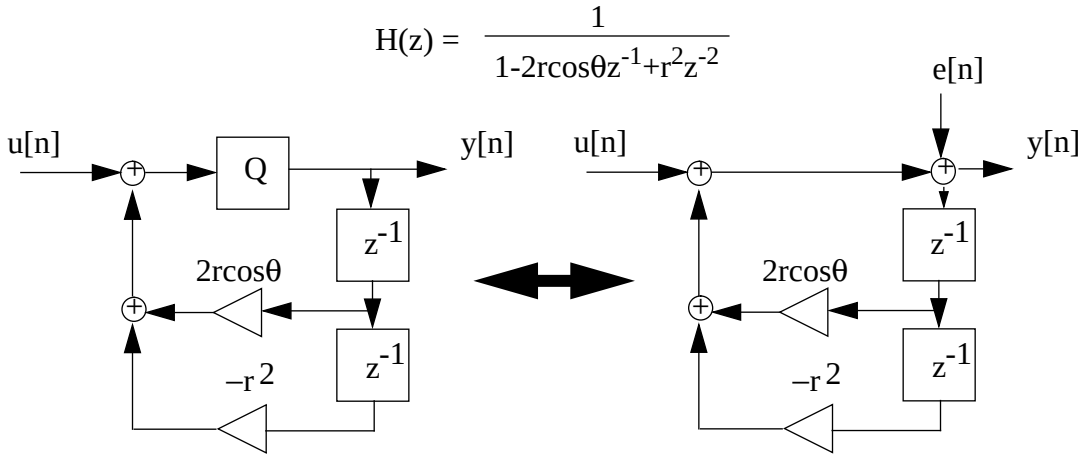


Figure 6.5 Roundoff Noise Model

One way to analyze this is to look at the noise transfer function, G(z), which just happens to be the same as the filter transfer function, H(z). The noise gain can then be calculated:

$$\sum_{n=0}^{\infty} h^2[n] = \left[\frac{1+r^2}{1-r^2}\right]\left[\frac{1}{r^4 + 2r^2\cos 2\Theta + 1}\right] \quad \text{Equation 5}$$

Not surprisingly, the noise gain increases dramatically as the poles of the system are pushed near the unit circle. This is consistent with the behavior of the graphic equalizer in section 6 and the quantization effects from the input and filter coefficients.

# 7.0 An Audio Application: Parametric Equalization

A 10 band graphic equalizer was implemented in the CGCVIS domain. Code was generated using the VIS instructions, and the application was run in real time on the Ultrasparc processor. For the implementation, we developed a parametric biquad and enhanced the Ptolemy audio I/O capabilities and Tcl/Tk interface.

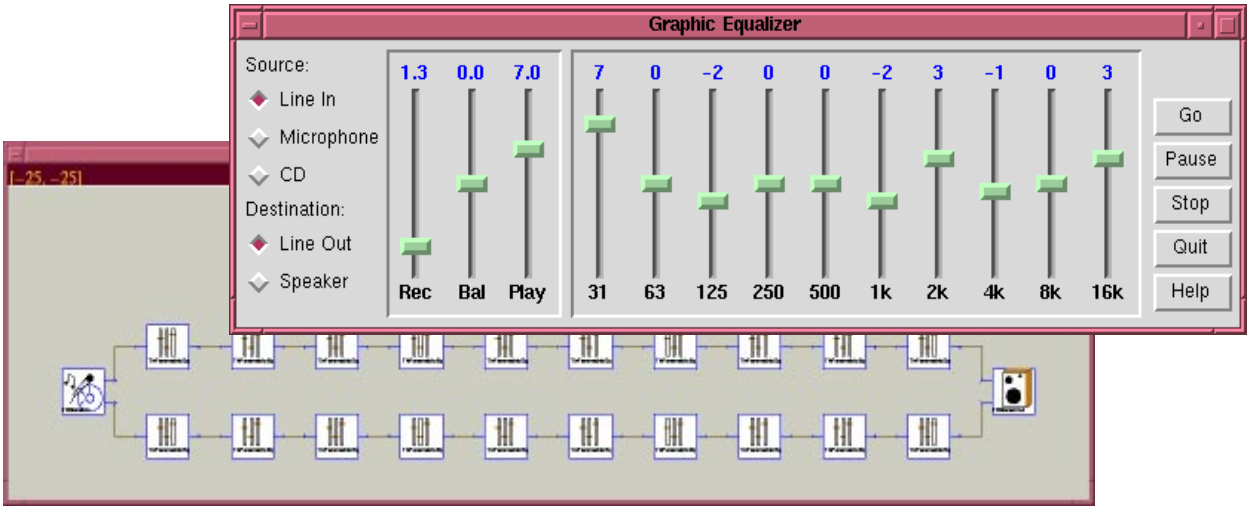


Figure 7.1 Ten-band Graphic Equalizer

Figure 7.1 above shows a ten-band graphic equalizer that is simulated in the Ptolemy environment and implemented on the UltraSparc processor. 16 bit audio samples are read into a FIFO buffer at 44.1 kHz. In the graphical user interface, the user specifies whether to gain or attenuate the audio signal in ten preset frequency bands. The specifications are then mapped down into parameters for a bank of twenty parametric biquad filters. Each biquad filter must then filter the buffered audio stream and output the results to the speaker in real time.

In audio processing, it is useful to control the gain of an audio signal in specified frequency bands. A parametric filter allows the user to control the gain by mapping a set of parameters that describe a specific frequency response into filter coefficients. To make the parametric filter truly interactive, it must update the filter coefficients in real time. The design of such parametric filters is given by Schpak [6].

## 7.1 Parametric filter design

There are many ways to design a parametric filter, but in order to meet the real time constraint, three design choices are made. First, IIR filters are chosen. Although FIR filters are stable and have linear phases, they require high filter orders. IIR filters require lower filter orders, but the

phase response is highly nonlinear. In particular, second order biquadratic sections are used since only moderate filter selectivity is required. Second, closed-form analytical methods are chosen over iterative optimization methods. Optimization methods are compute intensive and degrades the real time response. Finally, a cascade implementation is used so that when the characteristics of one frequency band is changed, only the coefficients for that particular biquad section is recomputed. Given the design choices, the graphic equalizer is implemented as a cascade of biquadratric filter sections.

In the graphic equalizer above, the user chooses either a lowpass, bandpass, or highpass filter and inputs the sampling frequency, desired gain and center frequency and bandwidth of the bandpass frequency response or passband and stopband of the lowpass and highpass frequency response. The following algorithm is used to map those “user-friendly” parameters into digital filter coefficients in real time:

- Pre-warp the analog frequency specifications using Newton’s method. This is necessary since a bilinear mapping is used later to transform the analog filter coefficients to digital filter coefficients.
- Solve the filter coefficients of an analog biquadratic filter section $\{\alpha, \beta, Q_z, Q_p\}$.

Equation 1

$$H(s) = \frac{\alpha s^2 + \frac{s\Omega_c}{Q_z} + \beta\Omega_C{}^2}{s^2 + \frac{s\Omega_C}{Q_P} + \Omega_C{}^2}$$

- Apply bilinear transformation to find digital filter coefficients.

## 7.2 Analog Bandpass Filter

For an analog bandpass filter, set the parameters $\alpha = 1$ and $\beta = 1$. The user further specifies:

**Table 8. User parameters for bandpass filter**

| Parameters | Symbols |
|---|---|
| Sampling rate | $T_s$ |
| Center Frequency | $f_c$ |
| Bandwidth | B |
| Gain | k |

Substituting $s = j\Omega$ into Eq. (1) and solving for the squared magnitude gives:

$$\text{Equation 2}\quad |H(j\Omega)|^2 = \frac{Q_p^2}{Q_z^2}\frac{Q_z^2(\Omega_c^4+\Omega^4)+\Omega_C^2\Omega^2(1-2Q_Z^2)}{Q_p^2(\Omega_c^4+\Omega^4)+\Omega_C^2\Omega^2(1-2Q_P^{2s})}$$

and at $\Omega_c$, the peak gain is k, which further reduces Eq. (2):

$$\text{Equation 3}\quad \left|H(j\Omega_c)\right| = \frac{Q_p}{Q_z} = k$$

Substituting Eq. (3) into Eq. (2) the magnitude can be expressed in terms of the peak gain k.

$$|H(j\Omega)|^2 = \frac{Q_p^2(\Omega_c^2+\Omega^2)+k^2\Omega_c^2\Omega^2}{Q(\Omega_c^2-\Omega^2)^2+\Omega_c^2\Omega^2}\ \text{Equation 4}$$

Now at the lower frequency point, $\Omega_1$, where the bandwidth is specified, the magnitude is set to the root peak gain. Solving for $|H(\Omega_1)| = \sqrt{k}$ yields,

$$Qp = \frac{\sqrt{k}\Omega_c\Omega_1}{\Omega_c^2-\Omega_1^2}\text{Equation 5}$$

The magnitude at $\Omega_1$ could also have been set to -3 dB. However, the above formulation ensures a smooth transition from one biquad filter response to the next.

## 7.3 Analog Lowpass Filter

For an analog lowpass filter, set the parameters $\alpha = 1$ and $\beta$ = gain. The user further specifies:

**Table 9. User parameters for lowpass filter.**

| Parameters | Symbols |
|---|---|
| Sample rate | $T_s$ |
| Passband Frequency | $f_c$ |
| Stopband Frequency | $f_s$ |
| Gain | k |

Substituting $s = j\Omega$ into Eq. (1) and solving for the squared magnitude gives:

$$|H(j\Omega)|^2 = \frac{Q_p^2}{Q_z^2}\frac{Q_z^2(\beta^2\Omega_c^4 + \Omega^4) + \Omega_C^2\Omega^2(1 - 2\beta Q_Z^2)}{Q_p^2(\Omega_c^4 + \Omega^4) + \Omega_C^2\Omega^2(1 - 2Q_P^{2s})}$$ Equation 6

Substituting $\Omega = \Omega_c$ in Eq. (6) yields:

$$\left|H(j\Omega_c)\right| = \frac{Q_p^2}{Q_z^2}[Q_Z^2(\beta - 1)^2 + 1]$$ Equation 7

Setting the magnitude $|H(\Omega_c)|^2 = \beta^2$ yields:

$$Qp^2 = \frac{\beta^2}{(\beta - 1)^2 - Q_Z^{-2}}$$ Equation 8

and substituting this value of $Q_p$ into Eq (6) gives:

$$|H(j\Omega)|^2 = \frac{\beta^2[Q_z^2(\beta^2\Omega_c^4 + \Omega^4) + \Omega_C^2\Omega^2(1 - 2\beta Q_Z^2)]}{\beta^2 Q_z^2(\Omega_c^4 + \Omega^4) + \Omega^2\Omega_c^2[Q(\beta - 1) + 1] - 2\beta^2 Q_z\Omega_c^2\Omega^2}$$ Equation 9

At the stopband frequency $\Omega_s$, the magnitude is set to the root peak gain. $Q_z$ equals:

$$Qz^2 = \frac{\Omega_s^2\Omega_c^2}{\Omega_s^4\beta + \Omega_s^2\Omega_c^2(\beta - 1) - \Omega_c^4\beta^2}$$ Equation 10

## 7.4 Analog Hipass Filter

For an analog highpass filter, the resulting filter design equations can be obtained in a similar manner. Set the parameters $\alpha$ = gain and $\beta = 1$. The user further specifies:

**Table 10. User parameters for lowpass filter.**

| Parameters | Symbols |
|---|---|
| Sample rate | $T_s$ |
| Passband Frequency | $f_c$ |
| Stopband Frequency | $f_s$ |
| Gain | k |

The design equations are given below:

$$Qp^2 = \frac{\alpha^2}{(\alpha-1)^2 - Q_Z^{-2}} \quad \text{Equation 11}$$

$$Qz^2 = \frac{\Omega_s^2\Omega_c^2}{-\Omega_s^4\alpha^2 + \Omega_s^2\Omega_c^2(\alpha-1) - \Omega_c^4\alpha} \quad \text{Equation 12}$$

## 7.5 Digital Filter

From the user specifications, the desired digital filter specifications can be calculated. However, the digital filter specifications need to be prewarped before using them in computing the analog biquad filter coefficients $Q_p$ and $Q_z$. The bilinear transformation is then applied to biquad filter sections to obtain the digital filter coefficients. The prewarping formulas between the analog $\{\Omega_c\}$ and desired digital $\{\omega_c\}$ specifications of a bandpass filter are given below:

$$\Omega_c = \frac{2}{T}\tan\left(\frac{\omega_c T}{2}\right) \quad \text{Equation 13}$$

$$B_D = \frac{2}{T}\left[\text{atan}\left(\frac{\gamma\Omega_c T}{2}\right) - \text{atan}\left(\frac{\Omega_c T}{2\gamma}\right)\right] \quad \text{Equation 14}$$

$$\gamma = \frac{\Omega_c}{\Omega_1} = \frac{\Omega_2}{\Omega_c} \quad \text{Equation 15}$$

In order to preserve the center frequency $\Omega_c$ and the bandwidth $B_D$, it is necessary to apply Newton's method to solve for $\Omega_1$, since Eq (14) cannot be solved analytically. Only five iterations are used since Newton's method degrades the real time performance.

Once prewarping is finished, the required $Q_p$ and $Q_z$ are computed as before. Then bilinear transformation is applied to the digital transfer function to yield:

$$\text{Equation 15 } F(z) = H_0\frac{z^2 + a_1 z + a_0}{z^2 + b_1 z + b_0}$$

$$\text{Equation 16 } a_2 = Q_p[Q_z(4\alpha + \beta\omega_c^2 T^2) + 2\omega_c T]$$

$$\text{Equation 17 } a_1 = \frac{2Q_pQ_z(-4\alpha + \beta\Omega_c{}^2T^2)}{a_2}$$

$$\text{Equation 18 } a_0 = \frac{Q_p(Q_z(4\alpha + \beta\Omega_c^2T^2) - 2\Omega_cT)}{a_2}$$

$$\text{Equation 19 } b_2 = Q_z[Q_p(4 + \Omega_c^2T^2) + 2\Omega_cT]$$

$$\text{Equation 20 } b_1 = \frac{2Q_pQ_z(-4 + \Omega_c{}^2T^2)}{b_2}$$

$$\text{Equation 21 } b_0 = \frac{Q_z(Q_p(4 + \Omega_c^2T^2) - 2\Omega_cT)}{b_2}$$

$$\text{Equation 22 } H_0 = \frac{a_2}{b_2}$$

## 7.6 Some Example Frequency Responses

In the following example, the frequency specifications of seven biquad sections are listed in table 10 and their corresponding frequency responses are plotted in Figure 7.2.

**Table 11. Biquadratic filter sections.**

| Spec. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| fc or fp | 64 | 160 | 400 | 1000 | 2500 | 6250 | 15625 |
| B | | 151.8 | 379.5 | 948.7 | 2372 | 5929 | |
| k (dB) | 6 | -10 | 6 | 3 | -3 | 6 | 3 |

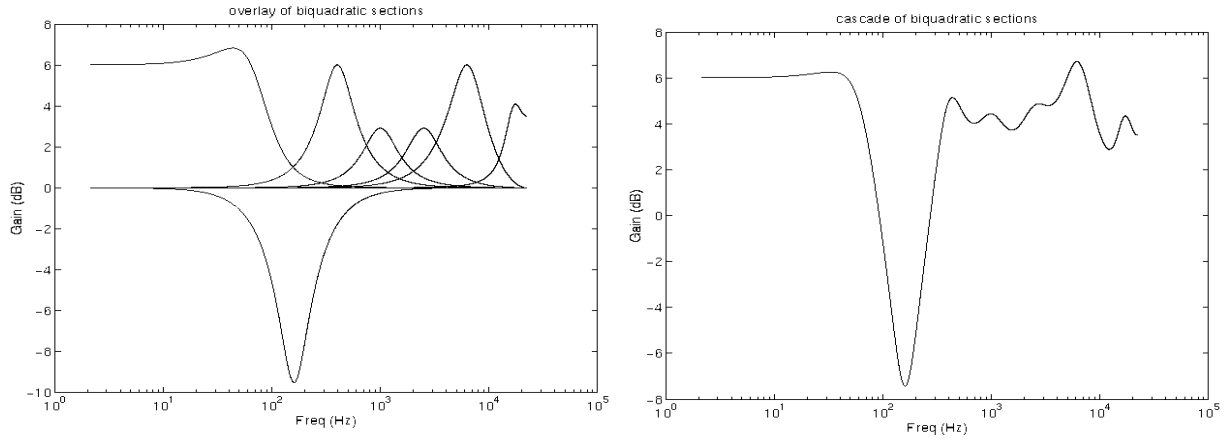


Figure 7.2 Overlay and Cascade of Biquad Filter Sections

# 8.0 Conclusions

We have implemented the CGCVIS domain in Ptolemy, compared the timing performance of several basic signal processing kernels, and implemented a real time audio application. The VIS showed modest speedups on basic signal processing kernels. While the FIR filter exhibited good speedups, the biquad and FFT showed little or none, despite intensive hand coding and several different algorithms. And as expected, the VIS was sensitive to quantization effects. However, there are “free” performance gains available to the Ptolemy workstation that are worth supporting.

Comparing the CGCVIS domain to the other DSP code generation domains, we conclude that coding in VIS is as difficult as coding in other assembly languages. Although the library of VIS arithmetic instructions was flexible enough to handle most of the audio processing, the library of VIS data formatting instructions was not rich enough to support all the packing and unpacking of data.

It is interesting to note that Intel has recently released a Pentium processor with the Multimedia extension (MMX). The MMX, like the VIS, is a set of instructions that support multimedia applications. 57 new instructions are added to enhance the performance of signal processing applications such as audio, video, imaging and 3D geometry.

Future works include extending the library of audio stars in Ptolemy to handle a range of audio capabilities, such as ambience simulation and dynamic range control, and implementing CGC matrices to handle video and image applications.

# 9.0 References

[1] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[2] David L. Weaver and Tom Germond, The Sparc Architecture Manual, Version 9, Prentice-Hall, Inc., 1994

[3] Edward A. Lee, "Programmable DSP Architectures: Part 1," IEEE ASSP Magazine, Vol. 5, No. 4, pp 4-19, October, 1988.

[4] Alan V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc., 1989.

[5] D. Mitra and J. F. Kaiser, eds., *Handbook of Digital Signal Processing*, John Wiley and Sons, 1993.

[6] Dale J. Shpak, "Analytical Design of Biquadratic Filter Sections for Parametric Filters", Journal of Audio Engineering Society, Vol. 40, No. 11, 1992.