# PlayStation®2 VU Command Line™ Preprocessor Release 1.4x

## User's Manual

Publication date and document version: May 2004, Version 2.0

Sony Computer Entertainment Inc.
1-1, Akasaka 7-chome, Minato-ku
Tokyo 107-0052, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

# Table of Contents


About This Manual vii
- Changes Since Last Release vii
- Related Documentation vii
- Typographic Conventions vii

Overview 1
- What is the VU Command Line™ (VCL) Preprocessor? 1
- Merging of Upper and Lower Instructions in One Code Stream 1
- Syntax Simplification 1
- Variable Naming and Registers Allocation 1
- Instruction Scheduling 1
- Macro Usage 2

Syntax Simplification 3
- Merging of Upper and Lower Instructions 3
- Variable Naming and Registers Allocation 3
- Number Literals Peculiarities 4
- Register Availability 4
- Instruction Simplification 4
- Floating-Point Register Fields Specification 6
- Broadcast Instructions 6

Instruction Scheduling and Data Tracking 7
- Instruction Scheduling 7
- Loop Unrolling 7
- Instructions Ordering 9
- Branch Delay Slots 10
- Code Removal 10
- E, D, and T Bits 10
- Load and Store Offsets 11
- Data Tracking 12
- Set Before Use 12

Branching 13
- Labels 13
- Calls to Functions 13
- Functions Calling Sub-Functions 13
- Jump Tables 14
- Recursive Functions 15

Integration of VSM Code Within the VCL Code 16
- .vsm / .endvsm and .raw / .endraw 16
- .rawloop / .endrawloop 17

Macros and Other Preprocessor Usages 18
- Using the C Preprocessor 18
- Using Macros with the C Preprocessor 18
- Using gasp 18
- Using Macros with gasp 19
- Issues with gasp 19
- Examples of Preprocessor Usage 19

Command-Line Parameters 20
- Command-Line Syntax 20
- –c 20
- -C 20
- –d 20
- -e 20

-f 20
–g 20
–g+ 20
–G 20
–h 20
–I<includefilepath> 21
<inputfilename> 21
-j<outputfilename.s> 21
-K 21
–L 21
–m 21
–M 21
–n 21
–o<outputfilename.vsm> 21
–P 21
-q 21
-s 21
-S 22
–t<seconds> 22
-u<string> 22
–Z 22
Keywords 23
.global symbolname 23
.init_vi VIxx <, VIxx …> 23
.init_vf VFxx <, VFxx …> 23
.rem_vi VIxx <, VIxx …> 23
.rem_vf VFxx <, VFxx …> 23
.init_vi_all 23
.init_vf_all 23
.mpg vucodeoffset 23
.name progname 24
.raw / .endraw 24
.rawloop / .endrawloop 24
.syntax old | new 24
.vsm / .endvsm 24
--barrier 24
--cont 24
--enter / --endenter 24
in_vi varname (VIxx) 24
in_vf varname (VFxx) 25
in_hw_acc acc / in_hw_clip clip / in_hw_i i / in_hw_p p / in_hw_q q / in_hw_r r / in_hw_status status 25
--exit / --endexit 25
--exitm macroname / --endexit 25
out_vi varname (VIxx) 25
out_vf varname (VFxx) 25
out_hw_acc acc / out_hw_clip clip / out_hw_i i / out_hw_p p / out_hw_q q / out_hw_r r25
--LoopCS n,m 25
--LoopExtra n 26
--LoopAbs n 26
Appendix A: Macro Examples 27
Appendix B: Detailed Information Regarding Loops in the VCL Preprocessor 43
Pipelining and VCL 43
"--LoopCS n,m" Directive 45

Appendix C: VCL Tips and Common Mistakes 49
Preprocessor Errors 49
Reordering of Instructions 49
Working Registers 49
Input and Output Registers 49
Entry Points 49
Exit Points in Code 49
Conditional Branching and Loop Unrolling 49
Number Literals 49
Memory Management 50
Variable Names 50
Broadcast Instructions and Variable Names 50
Long Dependency chains 50
Typos and Instruction Pruning 50
EFU Instructions Usage 51
Register with 1s 51
Dot Product (Inner Product) 51

This page intentionally left blank.

# About This Manual

This User's Manual provides a description of the various functionalities of the VU Command Line™ (VCL) preprocessor (1.4x).

## Changes Since Last Release

The following sections were updated or added:

- Syntax Simplification, (updated) on page 3.
- Number Literals Peculiarities, (updated) on page 4.
- Register Availability, (updated) on page 4.
- Broadcast Instructions, (updated) on page 6.
- Loop Unrolling, (updated) on page 8.
- Clip Instruction, (added) on page 9.
- E, D and T Bits, (updated) on page 11.
- Data Tracking, (updated) on page 12.
- Set Before Use, (updated) on page 13.
- Jump Tables, (rewritten) on page 15.
- Recursive Functions, (rewritten) on page 15.
- Using Macros with the C Preprocessor, (rewritten) on page 19.

Subsections -f, -g+, -j, -q and -s of section Command-Line Parameters were either modified or added, on pages 21–23.

Subsections .init_vi, .init_vf, .rem_vi, .rem_vf, .syntax, in_hw_*, --exit, out_hw_*, --LoopExtra and LoopAbs of section Keywords were either modified or added, on pages 24–27.

Appendix C: VCL Tips and Common Mistakes was added.

## Related Documentation

**Note:** the Developer Support Website posts current developments regarding the Libraries and also provides notice of future documentation releases and upgrades.

## Typographic Conventions

Certain Typographic Conventions are used throughout this manual to clarify the meaning of the text:

| Convention | Meaning |
|---|---|
| `courier` | Indicates literal program code. |
| *italic* | Indicates names of arguments and structure members (in structure/function definitions only). |
| **medium bold** | Indicates data types and structure/function names (in structure/function definitions only). |
| blue | Indicates a hyperlink. |

This page intentionally left blank.

# Overview

## What is the VU Command Line™ (VCL) Preprocessor?

The VCL preprocessor is an application that was developed to simplify some of the complex and tedious tasks associated with assembly-level programming of the VU processor. These tasks include:

- Dual pipeline processing
- Loop unrolling
- Register allocation
- Instruction scheduling

The VCL preprocessor outputs a standard VSM/DSM file (that can be compiled using dvpasm). It is available for both the Linux and Win32 platforms.

## Merging of Upper and Lower Instructions in One Code Stream

The VCL preprocessor simplifies VU programming by merging upper and lower instructions. Pairing of instructions is no longer required.

## Syntax Simplification

In some cases, standard VU programming requires the programmer to specify which register will be used as a parameter in instructions such as MULQ. The VCL preprocessor performs the proper instruction assignment, and only requires the programmer to specify the more generic MUL. The rest is deduced from parameters attached to the instruction in question.

## Variable Naming and Registers Allocation

The VCL preprocessor provides for variable naming. Self-explanatory variable names like 'vertexptr' and 'vertexcolor' are permissible, instead of using standard register names such as '`vi02`' or '`vf06`'.

## Instruction Scheduling

Assembly language programming for high-performance applications requires intimate knowledge of instruction timing (throughput and latency). Because of this, instruction scheduling is often very time consuming.

Data tracking is a tedious task because error messages regarding suspicious variable-related issues (use before set, for example) are not provided. High-performance code can appear to be confusing, especially if two blocks of code not otherwise related are reorganized into one block, for speed purposes.

While knowledge of instructions timing is still highly recommended, use of the VCL preprocessor will simplify instruction scheduling, keeping the code in a logical order, without compromising performance. The VCL preprocessor is aware of timing and dependencies, and in most cases will generate code that rivals hand-tuned code.

One of the most powerful features of the VCL preprocessor is related to loop unrolling. (See "Loop Unrolling" on page 7 for more details.)

## Macro Usage

With C-preprocessor or gasp, macro usage has always been possible, even without the VCL preprocessor. Use is limited however, as instructions must be paired, and that this pairing may not be broken down. However, because VCL programming is single-streamed, it reinforces the power offered by macros.

# Syntax Simplification

The VCL preprocessor offers two syntax schemes, simply referred to as “old” and “new”. The “old” scheme is standard VU programming, where instructions must be specified using the full name, and where fields must be specified in instructions, as well as the registers to which they belong.

The “new” scheme attempts to simplify coding and code readability. To enable its use, the following must be added to the source file before any instructions:

```
.syntax new
```

An alternative way of enabling it is to specify “-n” as a command-line argument to the VCL preprocessor.

At any given point in a VCL file, it is possible to go back to the old syntax by specifying:

```
.syntax old
```

Note that merging of upper and lower instructions, as well as register naming, happens with either syntax. This section describes aspects of the “new” syntax.

## Merging of Upper and Lower Instructions

Because the VCL preprocessor manages instruction scheduling, it is no longer necessary to pair instructions. All instructions (from either the upper or lower pipeline) can now be ordered sequentially, in a single stream.

## Variable Naming and Registers Allocation

The use of named variables is permissible and encouraged. Besides making code more readable, named variables allow the VCL preprocessor to manage register allocation.

It is allowable to tie a specific register to a named variable and use register names directly. (See “Instruction Scheduling” on page 7 for more details.) However, this limits the register allocation process.

The following are acceptable examples of variable naming usage:

```
IADDIU      inputptr, vi00, 32
LQ          vertex, 0(inputptr)
MAX         vector1111, vf00, vf00[w]
```

The VCL preprocessor checks to see if a variable is used before being set, and will output an error message if that is the case.

### Floating-Point and Integer Variable Naming

The VCL preprocessor tracks floating-point and integer variable names separately. Although not recommended, it is still possible to use the same variable name for both the floating point and the integer. Note that each will have to be initialized separately prior to use.

### Special Variable Name “i”

To ease the porting of C code to the VCL preprocessor, the variable name “i” is permitted for an integer register. Assuming such a named variable isn’t used with upper instructions, the VCL preprocessor will be able to discern between the hardware register I, and the variable. However, use of this variable name is discouraged, as it inhibits one of the VCL preprocessor’s primary functions, which is to make code more readable.

## Number Literals Peculiarities

In cases where a number literal must be specified, it is possible to specify a string instead, which will be assumed to be a defined value, and will therefore be ported as-is to the output file. Care must be taken, as the VCL preprocessor might not always be able to differentiate a typo from valid code. For example, if you want to write:

```
LOI                 0x3F
```

but instead write:

```
LOI                 x3F
```

The VCL preprocessor will accept it, only to have dvpasm reject it, with what could look like a cryptic message. Also note that any number literal that is not preceded by "0x" will be parsed by VCL as a float value.

## Register Availability

It is necessary to let the VCL preprocessor know which registers are available for it to use. This is accomplished by using the following keywords:

```
.init_vi            VIxx <,VIxx>
.init_vf            VFxx <,VFxx>
.init_vi            VIxx-Vixx
.init_vf            VFxx-VFxx
.init_vi_all
.init_vf_all
```

"`.init_vi`" and "`.init_vf`" are used to specify, respectively, which integer and floating-point registers are available. A register range may be specified by using a dash. Specifying "`vi00`" or "`vf00`" is illegal, and will result in an error message. "`.init_vi_all`" and "`.init_vf_all`" may be alternatively used in lieu of specifying every single register by hand. Having both "`.init_vi`" and "`.init_vi_all`", for example, is illegal.

Similarly, it is also possible to make a register unavailable by using the following kewords:

```
.rem_vi             VIxx <,VIxx>
.rem_vf             VFxx <,VFxx>
.rem_vi             VIxx-Vixx
.rem_vf             VFxx-VFxx
```

## Instruction Simplification

Many VU instructions stem from a single, more generic instruction. Such is the case for `ADD`, `ADDi`, `ADDq`, `ADDbc`, `ADDA`, `ADDAi`, `ADDAbc`. In this case, the stem instruction would be `ADD`.

The VCL preprocessor accepts the replacement of the specific instructions by the stem instruction, as the specific instruction may be deduced by the parameters attached to the instruction. Therefore, a program could use:

```
ADD           vertexcolor, vertexcolor, q
```

The VCL preprocessor would convert this (possibly) to:

```
ADDq.xyzw     vf03xyzw, vf03xyzw, q
```

The following table lists all instructions affected by the syntax simplification:

**Table 1**

| Original Instruction | Simplified Stem Instruction |
|---|---|
| ADD | ADD |
| ADDi | |
| ADDq | |
| ADDbc | |
| ADDA | |
| ADDAi | |
| ADDAq | |
| ADDAbc | |
| SUB | SUB |
| SUBi | |
| SUBq | |
| SUBbc | |
| SUBA | |
| SUBAi | |
| SUBAq | |
| SUBAbc | |
| MUL | MUL |
| MULi | |
| MULq | |
| MULbc | |
| MULA | |
| MULAi | |
| MULAq | |
| MULAbc | |
| MADD | MADD |
| MADDi | |
| MADDq | |
| MADDbc | |
| MADDA | |
| MADDAi | |
| MADDAq | |
| MADDAbc | |

| Original Instruction | Simplified Stem Instruction |
| --- | --- |
| MSUB | MSUB |
| MSUBi | |
| MSUBq | |
| MSUBbc | |
| MSUBA | |
| MSUBAi | |
| MSUBAq | |
| MSUBAbc | |
| MAX | MAX |
| MAXi | |
| MAXbc | |
| MINI | MINI |
| MINIi | |
| MINIbc | |

## Floating-Point Register Fields Specification

The VCL preprocessor only requires that a floating-point register field be specified next to the instruction itself, as opposed to specifying it on the instruction and the register that it belongs to. Therefore, the following would be used:

```
ADD.xyz       newvertexposition, vertexposition, translation
```

Instead of:

```
ADD.xyz       vf04xyz, vf03xyz, vf02xyz
```

Specifying none is understood to be the same as specifying all (`xyzw`). Therefore, the following 2 cases would be equivalent:

```
MAX           color, color, vector0000
MAX.xyzw      color, color, vector0000
```

## Broadcast Instructions

When a field used as a broadcast is specified in the instruction (such as `ADDbc` or `MADDbc`), the VCL preprocessor requires the specified field to be next to the associated register. The following example is a typical vertex multiplication by a matrix.

```
MUL    acc,         matrix0, inputvertex[x]
MADD   acc,         matrix1, inputvertex[y]
MADD   acc,         matrix2, inputvertex[z]
MADD   finalvertex, matrix3, inputvertex[w]
```

This syntax is only recognized as such with instructions that accept broadcasts. With other type of instructions, the bracket would become part of the register name itself. For example:

```
FTOI0  result, value[y]
```

Because FTOI0 instructions do not support broadcasts, “value[y]” will be taken as a simple atomic variable name, and not as meaning “the ‘y’ element of variable ‘value’”.

# Instruction Scheduling and Data Tracking

## Instruction Scheduling

Another useful feature of the VCL preprocessor is instruction rescheduling, which maximizes execution speed and code compactness. It tracks timing for each instruction (throughput and latency), and will try to reorder instructions to minimize stalls and maximize efficiency. In some rare cases (where possible), it will also try to move instructions from the upper to lower pipeline, and vice-versa.

The VCL preprocessor also tracks the `I`, `ACC`, `Q` and `P` registers, as well as the CLIP flags, and will generate proper delays between related instructions to insure valid code generation.

If, for some reason, you want to avoid instruction rescheduling past a certain point, simply insert the keyword:

```
--barrier
```

**Important:** Under certain circumstances, using the following line of code in a VSM/DSM file causes a problem:

```
NOP[E]        XGKICK        VIxx
```

To avoid the problem, placing these two instructions on two different lines:

```
NOP           XGKICK        VIxx
NOP[E]        NOP
```

To ensure such code isn't generated, follow an XGKICK instruction with "`--barrier`".

## Loop Unrolling

Instead of repeating the same code many times in a row, the VCL preprocessor allows loop unrolling by specifying the following, at the beginning of a loop:

```
--LoopCS      n,m
```

The loop ends when a branch instruction to the beginning of the loop is encountered. The following would constitute a valid loop:

```
LoopStart:
--LoopCS 3,3

LQI           inputvertex, 0(inputptr++)

MUL           acc,         matrix0, inputvertex[x]
MADD          acc,         matrix1, inputvertex[y]
MADD          acc,         matrix2, inputvertex[z]
MADD          finalvertex, matrix3, inputvertex[w]

SQI           finalvertex, 0(outputptr++)

IBNE          inputptr, endbuffer, LoopStart
```

At this time, loop unrolling is limited to simple loops without conditional branching. While the VCL preprocessor will not fail or give any error message if a branch is encountered inside the loop, the code generated will be less than optimal.

If the loop in question is performing clipping operations, it is possible to set the `ADC` bit without actually doing any conditional branch:

```
CLIPW.xyz     clipvtx, ClipData    ; Trigger clip calculations
FCAND         vi01, 0x3FFFF        ; Set if any of previous 3 vtx is clipped
IADDIU        adc_bit, vi01, 0x7FFF
ISW.w         adc_bit, outvl_xyzf2(output_buffer)     ; Set if clipped
```

Loop unrolling almost always results in a prologue followed by the main loop body, then concluded by an epilogue. The size of the prologue and epilogue depends partly on the parameters given to "`--LoopCS`", but also depends on how well the code can be rescheduled.

The "`--LoopCS`" keyword takes two parameters: `n` (minimum number of loops) and `m` (slop count).

### n (Minimum Number of Loops)

As it unrolls the loop, the VCL preprocessor needs to know the loop's minimum iteration count. This will allow it to potentially move instructions with side effects (stores, for example) higher in the execution pipeline, sometimes even before any conditional branch is executed. This translates into tighter (hence faster) code.

### m (Slop Count)

The slop count describes how many output iterations can be done without overwriting data past the end of an output array. A value of 1 would indicate that it is safe to execute an output instruction one iteration ahead of the current iteration, allowing for better instructions scheduling in some cases.

If the input is 30 vertices and "`--LoopCS 3,3`" is specified, a 33-vertex output buffer will be required. If the input is less than 30 vertices, specify "`--LoopCS 3,0`" instead. Note that the VCL preprocessor may not process vertices ahead. In this case it will not matter if m is equal to 3, or even 500. It will behave the same as if m is 0.

**Tip:** The number of loop iterations is often based on a counter, which is decremented once per iteration. The loop is repeated until that counter reaches 0. A vertex counter is a good example of this. If the loop happens to be running through a given array, it is worth nothing that instead of using a counter, the address of the buffer ending may be calculated ahead of the loop. Then, instead of comparing the counter to 0, compare the current array pointer to the end pointer for equality. The end result is a saved instruction (the decrementing of the counter), tighter loop, and therefore faster code.

Refer to Appendix B: Detailed Information Regarding Loops in the VCL Preprocessor for more information.

Depending on the complexity of a loop, automatic loop unrolling might take a considerable amount of time. One of the reasons for this is VCL first attempts to fit a loop's instructions within n cycles, where "n" is the loop's theoretical minimum cycle count. If it fails, it ups the cycle count by 1, and repeats the process until it finds a solution. If for some reason you know that a loop will not fit within a given amount of cycles, you can let VCL know by using either one of the following keywords within the loop:

```
--LoopExtra   n

--LoopAbs     n
```

If "`--LoopExtra n`" is specified, VCL will add "n" to the theoretical minimum cycle count. If "`--LoopAbs n`" is specified, VCL will take "n" as the theoretical minimum cycle count.

## Instructions Ordering

Care must be taken to place instructions in the order in which they are to take place. For example, standard VU programming would allow a `DIV` instruction right above an `MULQ`. The result of the previous DIV would be used, as is it known that the result of a DIV instruction isn't available for seven cycles. With the VCL preprocessor, the `MULQ` must be placed before DIV. It will reposition them as it sees fit, to maximize execution speed and code compactness.

### Memory Aliasing and Instructions Reordering

The VCL preprocessor reorders instructions while preserving logical order. Instructions modifying a given variable will always appear in the same relative order to each other as they are in the input file. However, the VCL preprocessor has no explicit knowledge of memory, and more specifically, of memory aliasing.

Memory aliasing occurs when there are two different pointers potentially pointing to the same area in memory. In such a case, it is important that instructions reading and writing to both pointers' memory be kept in the same relative order.

Steps must be taken to let the VCL preprocessor know that two pointers are potentially referring to the same memory. By simply appending a suffix to all instructions related to the aliased pointers, it will know to preserve the relative order. The following is an example for "`ptr1`" and "`ptr2`":

```
SQ     var1, 0(ptr1):memgroup1
LQ     var2, 3(ptr2):memgroup1
```

Here, "`memgroup1`" could be any valid string. It ensures that the store (`SQ`) always takes precedence to the load (`LQ`).

Within one program, more than one group may be used.

### Peculiarities with XGKick

Moving XGKick instructions could potentially result in hazardous code. For example, the XGKick could be moved ahead of stores to the buffer to be XGKicked, To avoid any such cases, XGKick will never be moved before or after a store instruction.

### Clip Instruction

When a standard clipw and related instructions (fcand, fcor, etc.) are used, VCL must follow dependencies with previous clip results, and this prohibits it from reordering such instructions. If clip test instructions like fcand and fcor only need the last clip instruction's results, cliplw may be used instead of clipw, and in this case VCL will know it is allowed more freedom when scheduling and ordering instructions. For example, with the following code:

```
CLIPW.xyz     vertex1, vertex1[w]
FCAND         VI01, 0x3f
...
CLIPW.xyz     vertex2, vertex2[w]
FCAND         VI01, 0x3f
```

VCL has to preserve the instructions' relative order, whereas in the following case:

```
CLIPLW.xyz    vertex1, vertex1[w]
FCAND         VI01, 0x3f
...
CLIPLW.xyz    vertex2, vertex2[w]
FCAND         VI01, 0x3f
```

VCL is free to reorder the 2 cliplw/fcand pairs. Keep in mind that VCL does not interpret the literal integer's parameter of instructions like fcand, so if cliplw is followed by fcand which has a literal value like 0x500 (second-to-last clip results), then results will be unpredictable.

## Branch Delay Slots

The VCL preprocessor handles branch delay slots independently. Placing any instruction immediately after a branch will cause the instruction in question to be executed only if the branch is not taken (in the case of a non-returning branch) or when the program counter comes back from a sub-routine (in the case of a function call such as `BAL`).

## Code Removal

The VCL preprocessor will recognize if a code block isn't reachable and will remove it. This saves VU micro memory in the process.

Empty lines (`NOP NOP`) are also removed, and will be replaced if necessary by either a `WAITQ` or `WAITP` instruction. Any stall introduction by instruction removal is noted by the VCL preprocessor as a comment in the output file.

### Floating-Point Field Pruning

The VCL preprocessor keeps track of which fields of a floating-point register are used, and will prune any field that does not need to be used. This step allows it to more effectively schedule instructions under some circumstances. The following is an example of pruning:

```
LQ          color, 2(inputptr)
ADD         color, color, ambientcolor
MINI        color, color, vector1111
SQ.xyz      color, 2(outputptr)
```

Assuming that "`color`" isn't used later in the code, it will prune the w field in the three first instructions. The code effectively becomes the same as:

```
LQ.xyz      color, 2(inputptr)
ADD.xyz     color, color, ambientcolor
MINI.xyz    color, color, vector1111
SQ.xyz      color, 2(outputptr)
```

## E, D, and T Bits

Putting an [`E`], [`D`] and/or [`T`] bit on an instruction is valid for the VCL preprocessor. However, as it is rescheduled, the bit will move around with the instruction it is attached to. Also, if the instruction it is attached to is duplicated (in such cases as loop unrolling), the bit will be duplicated as well.

Instead of using the [`E`] bit, it is recommended to use the VCL keyword:

```
--cont
```

"`--cont`", which stands for "continue", lets the VCL preprocessor know the program will restart from this point. Effectively, it inserts a "`NOP[E] NOP`" line. The main advantage of using this keyword over inserting an explicit [`E`] bit is that it removes any danger of the VCL preprocessor moving the [`E`] bit somewhere unexpected. Another solution to this problem is to attach the bit to a label such as:

```
Label:[D]
```

Also note, as with "`--barrier`", instructions are not rescheduled beyond a "`--cont`", but data dependency checks across are still performed.

## Load and Store Offsets

When a loop containing load and store instructions `LQ` and `SQ` is unrolled, the VCL preprocessor will most likely alter the offset parameter to at least some of the instructions. This is a result of moving the instruction past the increment of the pointer the offset relates to. The following is an example:

```
LQ          vertex, 0(inputptr)
LQ          stq,    1(inputptr)
LQ          rgb,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
```

A possible (and simplistic) unrolling of such an instruction sequence would be:

```
LQ          vertex1, 0(inputptr)
LQ          stq1,    1(inputptr)
LQ          rgb1,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
LQ          vertex2, 0(inputptr)
LQ          stq2,    1(inputptr)
LQ          rgb2,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
LQ          vertex3, 0(inputptr)
LQ          stq3,    1(inputptr)
LQ          rgb3,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
```

However, the VCL preprocessor might instead unroll a sequence as follows, which produces the same result, but under certain conditions provides better instruction rescheduling:

```
LQ          vertex1, 0(inputptr)
LQ          stq1,    1(inputptr)
IADDIU      inputptr, inputptr, 3
LQ          rgb1,    -1(inputptr)
...
LQ          vertex2, 0(inputptr)
LQ          rgb2,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
LQ          stq2,    -2(inputptr)
LQ          vertex3, 0(inputptr)
LQ          stq3,    1(inputptr)
LQ          rgb3,    2(inputptr)
IADDIU      inputptr, inputptr, 3
...
```

Note that using `LQI`, `LQD`, `SQI`, and `SQD` will prevent the possibility of such optimizations. However, their use has proven to be more effective under certain conditions such as very tight loops.

## Data Tracking

The VCL preprocessor tracks data usage for many reasons. If it determines that a variable is set but not used afterwards, it will remove the setting instructions. If these settings need to remain in the code for the purpose of eventually passing the variable as a parameter out of the code block, use the following keyword:

```
out_vi intvarname (VIxx)
out_vf floatvarname (VFxx)
```

"`VIxx`" and "`VFxx`" are an integer and a float register, respectively. The names "`intvarname`" and "`floatvarname`" correspond to these registers. The VCL preprocessor will make sure the specified register will contain the named variable value. These keywords must appear between the two keywords "`--exit`" and "`--endexit`", or between "`--exitm`" and "`--endexit`". (See "`--exit / --endexit`" on page 26 for more details.)

Note that similar instructions exist for `ACC` (`out_hw_acc acc`), `I` (`out_hw_i i`), `P` (`out_hw_p p`), R (out_hw_r r), `Q` (`out_hw_q q`) registers, and the CLIP flags (`out_hw_clip clip`).

## Set Before Use

If a variable is found to be used before it is even initialized, the VCL preprocessor will output an error message. If it is necessary to pass in a variable as a parameter from a different block, or from standard VSM/DSM code, use the following keywords:

```
in_vi intvarname (VIxx)
in_vf floatvarname (VFxx)
```

These keywords must appear between the two keywords "`--enter`" and "`--endenter`". (See "`--enter/ --endenter`" on page 25 for more details.)

Specifying "`in_vi`" and "`in_vf`" does not automatically mean the link between the register and the variable name will remain for the rest of the program, even if the same link is specified using "`out_vi`" or "`out_vf`". VCL reserves the right to break that link at any point.

Note that similar instructions exist for `ACC` (`in_hw_acc acc`), `I` (`in_hw_i i`), `P` (`in_hw_p p`), `Q` (`in_hw_q q`) and R (in_hw_r r) registers, as well as the FPU results flags (`in_hw_status status`) and the CLIP flags (`in_hw_clip clip`).

An alternative to using "`in_hw_clip clip`" is to have the following instruction before any clipping-related instruction:

```
FCSET  0
```

This will let the VCL preprocessor know all clipping flags are initialized.

The VCL preprocessor does not keep track of all flags. It is aware of MAC flags, but not about sticky bits.

To initialize a variable to 0, it is illegal to use the following:

```
SUB    varname, varname, varname
```

Instead, use one of the two following methods:

```
SUB    varname, vf00, vf00
MFIR   varname, vi00 ; Flags are left untouched!
```

# Branching

## Labels

With the VCL preprocessor, labels are defined in the same way as under VSM/DSM. However, for a label to remain in the output file, it must be positioned between the "`--enter`"/"`--endenter`" and "`--exit`"/"`--endexit`" keywords (or "`--exitm`"/"`--endexit`"). If "`--exit`"/"`--endexit`" is omitted, the label can be positioned after "--enter"/"--endenter".

## Calls to Functions

The VCL preprocessor currently does not support far function calls (calls to external functions). Therefore, any functions called must be included in the same file as the caller, either directly or via a header file (for `c-preprocessor`) or include file (for `gasp`).

Within the VCL preprocessor, a function is treated like any branch. There is no special method of passing parameters in and out. The following code is an example:

```
        IADDIU       vertexptr, vi00, 64
        IADDIU       endptr, vertexptr, 20

        BAL          retaddress, TransformVerticesInPlace

        ...

TransformVerticesInPlace:
        LQ           vertex, 0(vertexptr)

        ;*** Matrix used without being initialized! ***
        MatrixMultiplyVertex  vertex, matrix, vertex

        SQI          vertex, 0(vertexptr++)
        IBNE         vertexptr, endptr

        JR           retaddress
```

In this case (for register scope), the function "`TransformVerticesInPlace`" behaves the same as if it were in-lined. Assuming the matrix hasn't been initialized before the function call, an error would be given to this effect.

## Functions Calling Sub-Functions

It is valid for a function to call a sub-function. However, be aware that returning addresses conflicts will result in an infinite loop.

## Jump Tables

Starting with 1.4x, the VCL preprocessor supports jump tables. It can be implemented using either a JR or JALR instruction, and the jump instruction must be suffixed by a semi-colon (":") followed by a semi-colon-separated list of possible destination labels. Here is an example:

```
        --enter
        --endenter

        ILW.x     JumpAddress, 0(vi00)
        ILW.y     Param1, 0(vi00)
        ILW.z     Param2, 0(vi00)

        JALR      RetAddress, JumpAddress:Function1:Function2

        ISW.y    Param1, 0(vi00)
        ISW.z    Param2, 0(vi00)

        --exit
        --endexit

    Function1:
        IADDIU   Param1, Param2, 0
        JR       RetAddress

    Function2:
        IADDIU   Param2, Param1, 0
        JR       RetAddress
```

VCL does not support the calling of functions that are located in a different file from the caller.

## Recursive Functions

The VCL preprocessor does not support recursive functions natively. However, it is possible for a program to maintain its own stack, push the return address register before calling the function recursively, pop the address on return, and eventually return, as is shown in the example below. Note also that some jump and branching destinations will have to be specified on some instructions, as specified in the section on jump tables:

```
        .init_vf_all
        .init_vi_all
        .syntax new

        --enter
        --endenter

        ILW.x    JumpAddress, 0(vi00)
        ILW.y    Param1, 0(vi00)
        ILW.z    Param2, 0(vi00)
        IADDIU  StackPtr, vi00, 1023

        JALR     RetAddress, JumpAddress:Function1:Function2

    MainRet:
        ISW.y    Param1, 0(vi00)
        ISW.z    Param2, 0(vi00)

        --exit
        --endexit

    ;-------------------------------------------------
    Function1:
        IADDIU   Param1, Param2, 0
        JR       RetAddress

    ;-------------------------------------------------
    Function2:
        IBNE    Param2, vi00, F2_1

        IADDIU   Param2, Param1, 0
        JR       RetAddress

    F2_1:
        ; Do something here...

        ISW.x   RetAddress, 0(StackPtr)   ; Push return address on stack
        ISUBIU  StackPtr, StackPtr, 1     ;

        ISUBIU  Param2, Param2, 1         ; Call function recursively
        BAL     RetAddress, Function2     ;

    F2_2:
        IADDIU  StackPtr, StackPtr, 1     ; Pop return address from stack
        ILW.x   RetAddress, 0(StackPtr)   ;

        JR      RetAddress:MainRet:F2_2   ; Return to caller
```

# Integration of VSM Code Within the VCL Code

Sometimes it may be necessary to incorporate traditional VSM/DSM code within VCL code. This section introduces two methods to do so, and explains how they are used.

## .vsm / .endvsm and .raw / .endraw

This is used for inserting pre-formatted, pre-ordered VSM code. Instructions therefore have to be organized in the original 2 stream (upper and lower). However, variables may still be named, and use the same ones as non-VSM code. Checks for data use before set will still be performed.

"`.raw`" / "`.endraw`" is simply an alternative spelling for "`.vsm`" / "`.endvsm`".

Note that code within a VSM block isn't rescheduled. The block acts the same way as "`--barrier`", in that instruction rescheduling is not performed across the block. You must consider whether or not the use of such a block contributes to better overall performance.

Also note that because such a block is meant for code only, it must appear between "`--enter`" / "`--endenter`" and "`--exit`" / "`--endexit`" (or "`--exitm`" / "`--endexit`". It must not be thought of as a black box that is ported as-is to the output file. For the same reason, specifying keywords like "`.equ`" inside such a block is invalid.

The following is an example of a `.vsm / .endvsm` block:

```
IADDUI       matrixptr, vi00, 0
MatrixLoad   mat1, 0, (matrixptr)
MatrixLoad   mat2, 4, (matrixptr)

; Swap matrices
.vsm
max mat1[0], mat2[0], mat2[0]     move mat2[0], mat1[0]
max mat1[1], mat2[1], mat2[1]     move mat2[1], mat1[1]
max mat1[2], mat2[2], mat2[2]     move mat2[2], mat1[2]
max mat1[3], mat2[3], mat2[3]     move mat2[3], mat1[3]
.endvsm

MatrixSave   mat1, 0, (matrixptr)
MatrixSave   mat2, 4, (matrixptr)
```

## .rawloop / .endrawloop

These directives are used to enclose a pre-formatted VSM code block to be unrolled. More control is permitted as far as prologue and epilogue creation goes. There following is an example:

```
        .rawloop

loop:
        --LoopCS 10,10

        5..ftoi4.xyz  ixyz, sxyz           1..lqi   vrt, (ptr++)
        4..mul.xyz    sxyz, nxfrm, q          nop
           nop                                nop
        1..mul        acc,  m[3], vf00[w]  2..move  nxfrm, xfrm
        1..madd       acc,  m[0], vrt[x]   2..div   q, vf00[w], xfrm[w]
        1..madd       acc,  m[1], vrt[y]   1..ibne  ptr, end_ptr, loop
        1..madd       xfrm, m[2], vrt[z]   5..sqi  ixyz, (optr++)

        .endrawloop
```

The "`--LoopCS`" keyword is used the same way as described in "Loop Unrolling". While building the prologue, the instructions with the lowest numbers are introduced first, and follow this order: 1 1-2 1-2-3 1-2-3-4 … etc. Then the loop itself and the epilogue are created in accordance with the prologue.

Refer to Appendix B: Detailed Information Regarding Loops in the VCL Preprocessor for more information.

# Macros and Other Preprocessor Usages

Using macros with VU code has always been possible, using tools such as the C preprocessor and `gasp`. The use of macros is also possible with the VCL preprocessor. In fact, it offers even more opportunities to use these tools, as the input is single-streamed, as opposed to the double-streamed VSM/DSM programming style.

## Using the C Preprocessor

The VCL preprocessor can automatically pipe the code through the C preprocessor by giving it the "`-G`" (uppercase G) command-line parameter.

Using it will allow the use of all preprocessor directives:

```
#if, #ifdef, #ifndef
#elif, #else
#endif
#include
#define
#undef
#line
#error
```

## Using Macros with the C Preprocessor

Because VCL does not support multiple instructions on a single line, standard C-preprocessor macro usage will not work except for basic, one-line macros. Use of macros with the C preprocessor is identical to that of C/C++.

```
#define addition(result,param1,param2)     \
    IADDIU     result, param1, param2
```

## Using gasp

Just as with the C preprocessor, the VCL preprocessor can automatically pipe the code through `gasp`, by giving it the
"`-g`" (lowercase g) command-line parameter.

Using `gasp` will allow the use, among others, of the following directives:

```
.assign
.include
.macro
.endm
.end
```

Refer to the `gasp` documentation for more information on the subject. Online documentation may be found at the following URLs:

http://www.objsw.com/docs/gasp_toc.html

http://sunsite.utk.edu/gnu/binutils/gasp_toc.html

http://case.ispras.ru/PublicScripts/cgi-bin/lib.cgi/gnu/gasp_toc.html

## Using Macros with gasp

As shown in the following example, using macros with `gasp` is simple:

```
.macro mymacro param1,param2
IADDIU     Counter, vi00, 5
mymacrolabel\@:
ISUBIU     Counter, Counter, 1
IBNE       Counter, vi00, mymacrolabel\@
.endm
```

`gasp` will replace the "`\@`" by a number, which is incremented with each instance of a macro, so the same macro may be used many times in the same source file.

## Issues with gasp

With some versions of `gasp`, including `gasp`, number literals aren't translated properly to the output file. Examples of failed conversions are:

```
0x123        (converted to)   0
0.000123     (converted to)   0.1
```

The way to fix this, for now, is to use the following syntax:

```
0x123        (switch to)      H'123
0.000123     (switch to)      1.23E-4 or (0.0001)
```

Also, any line without a space or tab in front of the first non-white character will be converted as though the first word was a label. The newly created label will be suffixed by a colon.

## Examples of Preprocessor Usage

Refer to Appendix A: Macro Examples for examples.

# Command-Line Parameters

VCL is a command-line-based preprocessor. Various parameters may be passed to it, all of which are described in this chapter.

## Command-Line Syntax

VCL must be called with parameters, following the syntax:

```
vcl    [-cCdefgGhKLmMnPSZ] [-I<includefilepath>] [-t<seconds>]
       [-o<outputfilename] [-u<string>] <inputfilename>
```

### –c

Emit nearly original source code as comments.

### -C

Disable the code reduction pass.

### –d

Dumb code is generated. For example, rescheduling of instructions isn't performed.

### -e

Disable the generation of [`E`] bits at the end of the code. Alternatively, the use of `--exitm` without an argument may be used.

### -f

Disable the generation of alignment directives (`.align n`). The inclusion of an alignment directive causes problems when many VCL output files are included in a single DSM file, so the use of this switch will correct such problem.

### –g

Run gasp on the input before any VCL-specific task is done. `gasp` is called with the following parameter string: "`–p –s –c ';'`". "`–I`" is also passed if specified.

### –g+

Run gasp on the input before any VCL-specific task is done, with alternate macro mode. `gasp` is called with the following parameter string: "`–a -p –s –c ';'`". "`–I`" is also passed if specified.

### –G

Run the C preprocessor on the input before any VCL-specific task is done.

### –h

Print out the command-line help.

### –I<includefilepath>

To be used with "`–g`"; tells `gasp` where to find include files.

### <inputfilename>

Specify the name of the VCL source file. If it is not specified, VCL will read from the standard input.

### -j<outputfilename.s>

ASM output file name. Specifying the same name as the source is invalid. Create a file specifying all labels and their address.

### -K

The temporary files created by the pre-processors are not deleted. The file locations being OS-dependant, refer to the VCL output to find them.

### –L

Globally disable loop code generation.

### –m

Generate "`.mpg`" and DMA tags automatically. This may be used as an alternative to "`.mpg`" within the VCL source file.

### –M

VCL retains the relative order of load and store instructions (known as the timid memory access mode). **Note:** `–M` will be deprecated in future versions.

### –n

Enable the new syntax. This may be used as an alternative to ".syntax new" within the VCL source file. (See "Syntax Simplification" on page 1 for more details.)

### –o<outputfilename.vsm>

VSM output file name. Specifying the same name as the source is invalid. If not specified, the result is outputted to the standard output.

### –P

Disable the removal of unused instructions and pruned fields pass. (See "Code Removal" on page 10 for more details.) **Note:** `–P` will be deprecated in future versions.

### -q

Enable quiet mode. Disable all but warnings and error messages.

### -s

Emit symbolic variable names instead of register names. This is to help with debugging.

### -S

The content of loops starting with "`--LoopCS`" will be reorganized to stagger the read and write instructions, and to facilitate memory access by the VIF and GIF.

### –t<seconds>

Specify the optimizer timeout. `<second>` must be 1 or higher. Default is `4`. Note that this has the potential side-effect of generating different code on different computers, as the processor speed is not taken into account.

### -u<string>

`<string>` is used as a unique string for label generation, instead of the file name. Useful if the filename is especially long.

### –Z

Disable the immediate field fix up pass.

**Note:** `–Z` will be deprecated in future versions.

# Keywords

## .global symbolname

The directive “`.global`”, along with “`symbolname`”, is ported as-is to the output file. “`symbolname`” is therefore only assumed to exist and be valid.

## .init_vi VIxx <, VIxx …>

Inform VCL that the specified integer registers are available for use. A register range may be specified by using a dash character. Specifying VI00 is illegal, as it is always considered available. Specifying “`.init_vi`” and “`.init_vi_all`” in the same file is illegal. Note that VCL might fail to process code if not given enough registers.

## .init_vf VFxx <, VFxx …>

Inform VCL that the specified float registers are available for use. A register range may be specified by using a dash character. Specifying VF00 is illegal, as it is always considered available. Specifying “`.init_vf`” and “`.init_vf_all`” in the same file is illegal. Note that VCL might fail to process code if not given enough registers.

## .rem_vi VIxx <, VIxx …>

Inform VCL that the specified integer registers are not available for use. A register range may be specified by using a dash character. Specifying VI00 is illegal, as it is always considered available. Note that VCL might fail to process code if not given enough registers.

## .rem_vf VFxx <, VFxx …>

Inform VCL that the specified float registers are not available for use. A register range may be specified by using a dash character. Specifying VF00 is illegal, as it is always considered available. Note that VCL might fail to process code if not given enough registers.

## .init_vi_all

Inform VCL that all integer registers are available for use. Specifying “`.init_vi`” and “`.init_vi_all`” in the same file is illegal.

## .init_vf_all

Inform VCL that all float registers are available for use. Specifying “`.init_vf`” and “`.init_vf_all`” in the same file is illegal.

## .mpg vucodeoffset

Add “`ret`” DMA tags around the code generated by VCL, for better integration of VCL code with original VSM/DSM code. If “`.name`” is also specified, two labels will be added, following the syntax “`(progname)_DmaTag`” and “`(progname)_DmaEnd`”. “`vucodeoffset`” is assumed to be a valid address.

### .name progname

Add two labels, one before the code generated by VCL and the other after, following the syntax "`(progname)_CodeStart`" and "`(progname)_CodeEnd`". For better integration of VCL code with original VSM/DSM code. The labels created are also made available globally, via the directive "`.global`".

### .raw / .endraw

Enclose pre-formatted, original VSM-style code. Same as "`vsm /.endvsm`". (See ".vsm / .endvsm and .raw / .endraw" on page 17 for more details.)

### .rawloop / .endrawloop

Enclose a pre-formatted, original VSM-style code loop, to be unrolled. (See ".rawloop / .endrawloop" on page 18 for more details.)

### .syntax old | new

If "`old`" is specified, the syntax is the same as original VSM/DSM code. "`new`" specifies the new and simplified syntax. It may be specified many times throughout the file. (See "Syntax Simplification" on page 1 for more details.)

### .vsm / .endvsm

Enclose pre-formatted, original VSM-style code. Same as "`.raw`" / "`.endraw`". (See ".vsm / .endvsm and .raw / .endraw" on page 17 for more details.)

### --barrier

Prevent the rescheduling of instruction to go across this line. (See "Instruction Scheduling" on page 1 for more details.)

### --cont

Mark a point where a program temporarily stops, and may be restarted from, via a MSCNT. A [`E`] flag is inserted at this point. (See "E, D and T Bits" on page 11 for more details.)

### --enter / --endenter

Specify an entry point to VCL code. Any file must have at least one entry point, but may have more than one.

### in_vi varname (VIxx)

Must be specified between "`--enter`" and "`--endenter`". Bind a specific integer to a specific variable name at entry. The register is considered pre-initialized, presumably by standard VSM/DSM code. Such binding is not guaranteed to persist for the duration of the code block. (See "Set Before Use" on page 12 for more details.)

### in_vf varname (VFxx)

Must be specified between "`--enter`" and "`--endenter`". Bind a specific float to a specific variable name at entry. The register is considered pre-initialized, presumably by standard VSM/DSM code. Such binding is not guaranteed to persist for the duration of the code block. (See "Set Before Use" on page 12 for more details.)

### in_hw_acc acc / in_hw_clip clip / in_hw_i i / in_hw_p p / in_hw_q q / in_hw_r r / in_hw_status status

Must be specified between "`--enter`" and "`--endenter`". The specified register is considered pre-initialized, presumably by standard VSM/DSM code. (See "Set Before Use" on page 12 for more details.)

### --exit / --endexit

Specify an exit point to VCL code. Its use is mandatory only if outputting parameters is necessary, or if an explicit separation must be made between 2 portions of the code. A [`E`] flag is inserted at this point. (See "E, D and T Bits" on page 11 for more details.)

### --exitm macroname / --endexit

Specify an exit point to VCL code. Its use is mandatory only if outputting parameters is necessary. Unlike "`--exit`", "`--exitm`" lets you specify a macro name, which will be sent as-is to the output file, therefore permitting custom ending code.

### out_vi varname (VIxx)

Must be specified between "`--exit`" and "`--endexit`" or between "`--exitm`" and "`--endexit`". Bind a specific integer to a specific variable name at exit. The variable may then be passed to other VCL blocks, or to original VSM/DSM code. Such binding is not guaranteed to persist for the duration of the code block. (See "Set Before Use" on page 12 for more details.)

### out_vf varname (VFxx)

Must be specified between "`--exit`" and "`--endexit`" or between "`--exitm`" and "`--endexit`". Bind a specific float to a specific variable name at exit. The variable may then be passed to other VCL blocks, or to original VSM/DSM code. Such binding is not guaranteed to persist for the duration of the code block. (See "Set Before Use" on page 12 for more details.)

### out_hw_acc acc / out_hw_clip clip / out_hw_i i / out_hw_p p / out_hw_q q / out_hw_r r

Must be specified between "`--exit`" and "`--endexit`", or between "`--exitm`" and "`--endexit`". The specified register may then be passed to other VCL blocks, or to original VSM/DSM code. (See "Data Tracking" on page 12 for more details.)

### --LoopCS n,m

Mark a portion of code as being a loop, and instruct VCL to unroll it. "`n`" is the minimum iteration of the loop, and "`m`" (slop count) is the amount of output iterations that can be done without overwriting data past the end of an output array. (See "Loop Unrolling" on page 7 for more details.)

## --LoopExtra n

This can only be used in conjunction with "`--LoopCS`", and cannot be used with "`--LoopAbs`". This keyword instructs VCL to attempt to fit an unrolled loop in theoretical minimum cycle count + "n" and over. (See "Loop Unrolling" on page 7 for more details.)

## --LoopAbs n

This can only be used in conjunction with "`--LoopCS`", and cannot be used with "`--LoopExtra`". This keyword instructs VCL to attempt to fit an unrolled loop in "n" cycles and over. (See "Loop Unrolling" on page 7 for more details.)

# Appendix A: Macro Examples

All macros in this appendix may be found in the file VCL_SML.i, included with the VCL preprocessor distribution. All can be used as-is with `gasp` (via the "`-g`" command-line parameter), but all can easily be converted to be used by the C preprocessor.

```
;//--------------------------------------------------------------------
;// MatrixLoad - Load "matrix" from VU mem location "vumemlocation" +
;// "offset"
;//--------------------------------------------------------------------
   .macro   MatrixLoad  matrix,offset,vumemlocation
   lq             \matrix[0], \offset+0(\vumemlocation)
   lq             \matrix[1], \offset+1(\vumemlocation)
   lq             \matrix[2], \offset+2(\vumemlocation)
   lq             \matrix[3], \offset+3(\vumemlocation)
   .endm

;//--------------------------------------------------------------------
;// MatrixSave - Save "matrix" to VU mem location "vumemlocation" +
;// "offset"
;//--------------------------------------------------------------------
   .macro   MatrixSave  matrix,offset,vumemlocation
   sq             \matrix[0], \offset+0(\vumemlocation)
   sq             \matrix[1], \offset+1(\vumemlocation)
   sq             \matrix[2], \offset+2(\vumemlocation)
   sq             \matrix[3], \offset+3(\vumemlocation)
   .endm

;//--------------------------------------------------------------------
;// MatrixIdentity - Set "matrix" to be an identity matrix
;// Thanks to Colin Hughes (SCEE) for that one
;//--------------------------------------------------------------------
   .macro   MatrixIdentity matrix
   add.x          \matrix[0], vf00, vf00[w]
   mfir.yzw       \matrix[0], vi00

   mfir.xzw       \matrix[1], vi00
   add.y          \matrix[1], vf00, vf00[w]

   mr32           \matrix[2], vf00

   max            \matrix[3], vf00, vf00
   .endm

;//--------------------------------------------------------------------
;// MatrixCopy - Copy "matrixsrc" to "matrixdest"
;// Thanks to Colin Hughes (SCEE) for that one
;//--------------------------------------------------------------------
   .macro   MatrixCopy matrixdest,matrixsrc
   max            \matrixdest[0], \matrixsrc[0], \matrixsrc[0]
   move           \matrixdest[1], \matrixsrc[1]
   max            \matrixdest[2], \matrixsrc[2], \matrixsrc[2]
   move           \matrixdest[3], \matrixsrc[3]
   .endm
```

```
;//-----------------------------------------------------------------------
;// MatrixSwap - Swap the content of "matrix1" and "matrix2"
;// The implementation seems lame, but VCL will convert moves to maxes
;// if it sees fit
;//-----------------------------------------------------------------------
   .macro   MatrixSwap matrix1,matrix2
   move           vclsmlftemp, \matrix1[0]
   move           \matrix1[0], \matrix2[0]
   move           \matrix2[0], vclsmlftemp

   move           vclsmlftemp, \matrix1[1]
   move           \matrix1[1], \matrix2[1]
   move           \matrix2[1], vclsmlftemp

   move           vclsmlftemp, \matrix1[2]
   move           \matrix1[2], \matrix2[2]
   move           \matrix2[2], vclsmlftemp

   move           vclsmlftemp, \matrix1[3]
   move           \matrix1[3], \matrix2[3]
   move           \matrix2[3], vclsmlftemp
   .endm

;//-----------------------------------------------------------------------
;// MatrixTranspose - Transpose "matrixsrc" to "matresult".  It is safe
;// for "matrixsrc" and "matresult" to be the same.
;// Thanks to Colin Hughes (SCEE) for that one
;//-----------------------------------------------------------------------
   .macro   MatrixTranspose matresult,matrixsrc
   mr32.y         vclsmlftemp,   \matrixsrc[1]
   add.z          \matresult[1], vf00, \matrixsrc[2][y]
   move.y         \matresult[2], vclsmlftemp
   mr32.y         vclsmlftemp,   \matrixsrc[0]
   add.z          \matresult[0], vf00, \matrixsrc[2][x]
   mr32.z         vclsmlftemp,   \matrixsrc[1]
   mul.w          \matresult[1], vf00, \matrixsrc[3][y]
   mr32.x         vclsmlftemp,   \matrixsrc[0]
   add.y          \matresult[0], vf00, \matrixsrc[1][x]
   move.x         \matresult[1], vclsmlftemp
   mul.w          vclsmlftemp,   vf00, \matrixsrc[3][z]
   mr32.z         \matresult[3], \matrixsrc[2]
   move.w         \matresult[2], vclsmlftemp
   mr32.w         vclsmlftemp,   \matrixsrc[3]
   add.x          \matresult[3], vf00, \matrixsrc[0][w]
   move.w         \matresult[0], vclsmlftemp
   mr32.y         \matresult[3], vclsmlftemp
   add.x          \matresult[2], vf00, vclsmlftemp[y]

   move.x         \matresult[0], \matrixsrc[0]        ;// These 4
instructions will be
   move.y         \matresult[1], \matrixsrc[1]        ;// removed if
"matrixsrc" and
   move.z         \matresult[2], \matrixsrc[2]        ;// "matresult" are
the same
   move.w         \matresult[3], \matrixsrc[3]        ;//
   .endm
```

```
;//--------------------------------------------------------------------
;// MatrixMultiply - Multiply 2 matrices, "matleft" and "matright", and
;// output the result in "matresult".  Dont forget matrix multipli-
;// cations arent commutative, i.e. left X right wont give you the
;// same result as right X left.
;//
;// Note: ACC register is modified
;//--------------------------------------------------------------------
   .macro   MatrixMultiply   matresult,matleft,matright
   mul            acc,           \matright[0], \matleft[0][x]
   madd           acc,           \matright[1], \matleft[0][y]
   madd           acc,           \matright[2], \matleft[0][z]
   madd           \matresult[0], \matright[3], \matleft[0][w]

   mul            acc,           \matright[0], \matleft[1][x]
   madd           acc,           \matright[1], \matleft[1][y]
   madd           acc,           \matright[2], \matleft[1][z]
   madd           \matresult[1], \matright[3], \matleft[1][w]

   mul            acc,           \matright[0], \matleft[2][x]
   madd           acc,           \matright[1], \matleft[2][y]
   madd           acc,           \matright[2], \matleft[2][z]
   madd           \matresult[2], \matright[3], \matleft[2][w]

   mul            acc,           \matright[0], \matleft[3][x]
   madd           acc,           \matright[1], \matleft[3][y]
   madd           acc,           \matright[2], \matleft[3][z]
   madd           \matresult[3], \matright[3], \matleft[3][w]
   .endm

;//--------------------------------------------------------------------
;// LocalizeLightMatrix - Transform the light matrix "lightmatrix" into
;// local space, as described by "matrix", and output the result in
;// "locallightmatrix"
;//
;// Note: ACC register is modified
;//--------------------------------------------------------------------
   .macro   LocalizeLightMatrix locallightmatrix,matrix,lightmatrix
   mul            acc,                 \lightmatrix[0], \matrix[0][x]
   madd           acc,                 \lightmatrix[1], \matrix[0][y]
   madd           acc,                 \lightmatrix[2], \matrix[0][z]
   madd           \locallightmatrix[0], \lightmatrix[3], \matrix[0][w]

   mul            acc,                 \lightmatrix[0], \matrix[1][x]
   madd           acc,                 \lightmatrix[1], \matrix[1][y]
   madd           acc,                 \lightmatrix[2], \matrix[1][z]
   madd           \locallightmatrix[1], \lightmatrix[3], \matrix[1][w]

   mul            acc,                 \lightmatrix[0], \matrix[2][x]
   madd           acc,                 \lightmatrix[1], \matrix[2][y]
   madd           acc,                 \lightmatrix[2], \matrix[2][z]
   madd           \locallightmatrix[2], \lightmatrix[3], \matrix[2][w]

   move           \locallightmatrix[3], \lightmatrix[3]
   .endm
```

```
;//-----------------------------------------------------------------------
;// MatrixMultiplyVertex - Multiply "matrix" by "vertex", and output
;// the result in "vertexresult"
;//
;// Note: Apply rotation, scale and translation
;// Note: ACC register is modified
;//-----------------------------------------------------------------------
   .macro   MatrixMultiplyVertex vertexresult,matrix,vertex
   mul            acc,           \matrix[0], \vertex[x]
   madd           acc,           \matrix[1], \vertex[y]
   madd           acc,           \matrix[2], \vertex[z]
   madd           \vertexresult, \matrix[3], \vertex[w]
   .endm

;//-----------------------------------------------------------------------
;// MatrixMultiplyVertex - Multiply "matrix" by "vertex", and output
;// the result in "vertexresult"
;//
;// Note: Apply rotation, scale and translation
;// Note: ACC register is modified
;//-----------------------------------------------------------------------
   .macro   MatrixMultiplyVertexXYZ1 vertexresult,matrix,vertex
   mul            acc,           \matrix[0], \vertex[x]
   madd           acc,           \matrix[1], \vertex[y]
   madd           acc,           \matrix[2], \vertex[z]
   madd           \vertexresult, \matrix[3], vf00[w]
   .endm

;//-----------------------------------------------------------------------
;// MatrixMultiplyVector - Multiply "matrix" by "vector", and output
;// the result in "vectorresult"
;//
;// Note: Apply rotation and scale, but no translation
;// Note: ACC register is modified
;//-----------------------------------------------------------------------
   .macro   MatrixMultiplyVector vectorresult,matrix,vector
   mul            acc,           \matrix[0], \vector[x]
   madd           acc,           \matrix[1], \vector[y]
   madd           \vectorresult, \matrix[2], \vector[z]
   .endm

;//-----------------------------------------------------------------------
;// VectorLoad - Load "vector" from VU mem location "vumemlocation" +
;// "offset"
;//-----------------------------------------------------------------------
   .macro   VectorLoad  vector,offset,vumemlocation
   lq             \vector, \offset(\vumemlocation)
   .endm

;//-----------------------------------------------------------------------
;// VectorSave - Save "vector" to VU mem location "vumemlocation" +
;// "offset"
;//-----------------------------------------------------------------------
   .macro   VectorSave  vector,offset,vumemlocation
   sq             \vector, \offset(\vumemlocation)
   .endm
```

```
;//-------------------------------------------------------------------
;// VectorAdd - Add 2 vectors, "vector1" and "vector2" and output the
;// result in "vectorresult"
;//-------------------------------------------------------------------
   .macro   VectorAdd   vectorresult,vector1,vector2
   add            \vectorresult, \vector1, \vector2
   .endm

;//-------------------------------------------------------------------
;// VectorSub - Subtract "vector2" from "vector1", and output the
;// result in "vectorresult"
;//-------------------------------------------------------------------
   .macro   VectorSub   vectorresult,vector1,vector2
   sub            \vectorresult, \vector1, \vector2
   .endm

;//-------------------------------------------------------------------
;// VertexLoad - Load "vertex" from VU mem location "vumemlocation" +
;// "offset"
;//-------------------------------------------------------------------
   .macro   VertexLoad  vertex,offset,vumemlocation
   lq             \vertex, \offset(\vumemlocation)
   .endm

;//-------------------------------------------------------------------
;// VertexSave - Save "vertex" to VU mem location "vumemlocation" +
;// "offset"
;//-------------------------------------------------------------------
   .macro   VertexSave  vertex,offset,vumemlocation
   sq             \vertex, \offset(\vumemlocation)
   .endm

;//-------------------------------------------------------------------
;// VertexPersCorr - Apply perspective correction onto "vertex" and
;// output the result in "vertexoutput"
;//
;// Note: Q register is modified
;//-------------------------------------------------------------------
   .macro   VertexPersCorr vertexoutput,vertex
   div            q, vf00[w], \vertex[w]
   mul            \vertexoutput, \vertex, q
   .endm

;//-------------------------------------------------------------------
;// VertexPersCorrST - Apply perspective correction onto "vertex" and
;// "st", and output the result in "vertexoutput" and "stoutput"
;//
;// Note: Q register is modified
;//-------------------------------------------------------------------
   .macro   VertexPersCorrST vertexoutput,stoutput,vertex,st
   div            q,             vf00[w], \vertex[w]
   mul.xyz        \vertexoutput, \vertex, q
   move.w         \vertexoutput, \vertex
   mul            \stoutput,     \st,     q
   .endm
```

```
;//--------------------------------------------------------------------
;// VertexFPtoGsXYZ2 - Convert an XYZW, floating-point vertex to GS
;// XYZ2 format (ADC bit isnt set)
;//--------------------------------------------------------------------
   .macro   VertexFpToGsXYZ2  outputxyz,vertex
   ftoi4.xy       \outputxyz, \vertex
   ftoi0.z        \outputxyz, \vertex
   mfir.w         \outputxyz, vi00
   .endm

;//--------------------------------------------------------------------
;// VertexFPtoGsXYZ2Adc - Convert an XYZW, floating-point vertex to GS
;// XYZ2 format (ADC bit is set)
;//--------------------------------------------------------------------
   .macro   VertexFpToGsXYZ2Adc  outputxyz,vertex
   ftoi4.xy       \outputxyz, \vertex
   ftoi0.z        \outputxyz, \vertex
   ftoi15.w       \outputxyz, vf00
   .endm

;//--------------------------------------------------------------------
;// VertexFpToGsXYZF2 - Convert an XYZF, floating-point vertex to GS
;// XYZF2 format (ADC bit isnt set)
;//--------------------------------------------------------------------
   .macro   VertexFpToGsXYZF2 outputxyz,vertex
   ftoi4          \outputxyz, \vertex
   .endm

;//--------------------------------------------------------------------
;// VertexFpToGsXYZF2Adc - Convert an XYZF, floating-point vertex to GS
;// XYZF2 format (ADC bit is set)
;//--------------------------------------------------------------------
   .macro   VertexFpToGsXYZF2Adc outputxyz,vertex
   ftoi4          \outputxyz,  \vertex
   mtir           vclsmlitemp, \outputxyz[w]
   iaddiu         vclsmlitemp, 0x7FFF
   iaddi          vclsmlitemp, 1
   mfir.w         \outputxyz,  vclsmlitemp
   .endm

;//--------------------------------------------------------------------
;// ColorFPtoGsRGBAQ - Convert an RGBA, floating-point color to GS
;// RGBAQ format
;//--------------------------------------------------------------------
   .macro   ColorFPtoGsRGBAQ  outputrgba,color
   ftoi0          \outputrgba, \color
   .endm

;//--------------------------------------------------------------------
;// ColorGsRGBAQtoFP - Convert an RGBA, GS RGBAQ format to floating-
;// point color
;//--------------------------------------------------------------------
   .macro   ColorGsRGBAQtoFP  outputrgba,color
   itof0          \outputrgba, \color
   .endm
```

```
;//-----------------------------------------------------------------------
;// CreateGsPRIM - Create a GS-packed-format PRIM command, according to
;// a specified immediate value "prim"
;//
;// Note: Meant more for debugging purposes than for a final solution
;//-----------------------------------------------------------------------
   .macro   CreateGsPRIM   outputprim,prim
   iaddiu         vclsmlitemp, vi00, \prim
   mfir           \outputprim, vclsmlitemp
   .endm

;//-----------------------------------------------------------------------
;// CreateGsRGBA - Create a GS-packed-format RGBA command, according to
;// specified immediate values "r", "g", "b" and "a" (integer 0-255)
;//
;// Note: Meant more for debugging purposes than for a final solution
;//-----------------------------------------------------------------------
   .macro   CreateGsRGBA   outputrgba,r,g,b,a
   iaddiu         vclsmlitemp, vi00, \r
   mfir.x         \outputrgba, vclsmlitemp
   iaddiu         vclsmlitemp, vi00, \g
   mfir.y         \outputrgba, vclsmlitemp
   iaddiu         vclsmlitemp, vi00, \b
   mfir.z         \outputrgba, vclsmlitemp
   iaddiu         vclsmlitemp, vi00, \a
   mfir.w         \outputrgba, vclsmlitemp
   .endm

;//-----------------------------------------------------------------------
;// CreateGsSTQ - Create a GS-packed-format STQ command, according to
;// specified immediate values "s", "t" and "q" (floats)
;//
;// Note: I register is modified
;// Note: Meant more for debugging purposes than for a final solution
;//-----------------------------------------------------------------------
   .macro   CreateGsSTQ   outputstq,s,t,q
   loi            \s
   add.x          \outputstq, vf00, i
   loi            \t
   add.y          \outputstq, vf00, i
   loi            \q
   add.z          \outputstq, vf00, i
   .endm

;//-----------------------------------------------------------------------
;// CreateGsUV - Create a GS-packed-format VU command, according to
;// specified immediate values "u" and "v" (integer -32768 - 32768,
;// with 4 LSB as precision)
;//
;// Note: Meant more for debugging purposes than for a final solution
;//-----------------------------------------------------------------------
   .macro   CreateGsUV   outputuv,u,v
   iaddiu         vclsmlitemp, vi00, \u
   mfir.x         \outputuv, vclsmlitemp
   iaddiu         vclsmlitemp, vi00, \v
   mfir.y         \outputuv, vclsmlitemp
   .endm
```

```
;//---------------------------------------------------------------------
;// CreateGsRGBA - Create a GS-packed-format RGBA command, according to
;// a specified immediate value "fog" (integer 0-255)
;//
;// Note: Meant more for debugging purposes than for a final solution
;//---------------------------------------------------------------------
   .macro   CreateGsFOG   outputfog,fog
   iaddiu         vclsmlitemp, vi00, \fog * 16
   mfir.w         \outputfog, vclsmlitemp
   .endm

;//---------------------------------------------------------------------
;// VectorDotProduct - Calculate the dot product of "vector1" and
;// "vector2", and output to "dotproduct"[x]
;//---------------------------------------------------------------------
   .macro   VectorDotProduct  dotproduct,vector1,vector2
   mul.xyz        \dotproduct, \vector1,    \vector2
   add.x          \dotproduct, \dotproduct, \dotproduct[y]
   add.x          \dotproduct, \dotproduct, \dotproduct[z]
   .endm

;//---------------------------------------------------------------------
;// VectorDotProductACC - Calculate the dot product of "vector1" and
;// "vector2", and output to "dotproduct"[x].  This one does it using
;// the ACC register which, depending on the case, might turn out to be
;// faster or slower.
;//
;// Note: ACC register is modified
;//---------------------------------------------------------------------
   .macro   VectorDotProductACC  dotproduct,vector1,vector2
   max            Vector1111,  vf00,        vf00[w]
   mul            vclsmlftemp, \vector1,    \vector2
   add.x          acc,         vclsmlftemp, vclsmlftemp[y]
   madd.x         \dotproduct, Vector1111,  vclsmlftemp
   .endm

;//---------------------------------------------------------------------
;// VectorCrossProduct - Calculate the cross product of "vector1" and
;// "vector2", and output to "vectoroutput"
;//
;// Note: ACC register is modified
;//---------------------------------------------------------------------
   .macro   VectorCrossProduct  vectoroutput,vector1,vector2
   opmula.xyz     ACC,           \vector1, \vector2
   opmsub.xyz     \vectoroutput, \vector2, \vector1
   sub.w          \vectoroutput, vf00,     vf00
   .endm
```

```
;//--------------------------------------------------------------------
;// VectorNormalize - Bring the length of "vector" to 1.f, and output
;// it to "vectoroutput"
;//
;// Note: Q register is modified
;//--------------------------------------------------------------------
   .macro   VectorNormalize   vecoutput,vector
   mul.xyz        vclsmlftemp, \vector,     \vector
   add.x          vclsmlftemp, vclsmlftemp, vclsmlftemp[y]
   add.x          vclsmlftemp, vclsmlftemp, vclsmlftemp[z]
   rsqrt          q,           vf00[w],     vclsmlftemp[x]
   sub.w          \vecoutput,  vf00,        vf00
   mul.xyz        \vecoutput,  \vector,     q
   .endm

;//--------------------------------------------------------------------
;// VectorNormalizeXYZ - Bring the length of "vector" to 1.f, and out-
;// put it to "vectoroutput".  The "w" field isn't transfered.
;//
;// Note: Q register is modified
;//--------------------------------------------------------------------
   .macro   VectorNormalizeXYZ   vecoutput,vector
   mul.xyz        vclsmlftemp, \vector,     \vector
   add.x          vclsmlftemp, vclsmlftemp, vclsmlftemp[y]
   add.x          vclsmlftemp, vclsmlftemp, vclsmlftemp[z]
   rsqrt          q,           vf00[w],     vclsmlftemp[x]
   mul.xyz        \vecoutput,  \vector,     q
   .endm

;//--------------------------------------------------------------------
;// VertexLightAmb - Apply ambient lighting "ambientrgba" to a vertex
;// of color "vertexrgba", and output the result in "outputrgba"
;//--------------------------------------------------------------------
   .macro   VertexLightAmb rgbaout,vertexrgba,ambientrgba
   mul            \rgbaout, \vertexrgba, \ambientrgba
   .endm

;//--------------------------------------------------------------------
;// VertexLightDir3 - Apply up to 3 directional lights contained in a
;// light matrix "lightmatrix" to a vertex of color "vertexrgba" and
;// having a normal "vertexnormal", and output the result in
;// "outputrgba"
;//
;// Note: ACC register is modified
;//--------------------------------------------------------------------
   .macro   VertexLightDir3
rgbaout,vertexrgba,vertexnormal,lightcolors,lightnormals
   mul            acc,      \lightnormals[0], \vertexnormal[x]
   madd           acc,      \lightnormals[1], \vertexnormal[y]
   madd           acc,      \lightnormals[2], \vertexnormal[z]
   madd           \rgbaout, \lightnormals[3], \vertexnormal[w] ;// Here
"rgbaout" is the dot product for the 3 lights
   max            \rgbaout, \rgbaout,         vf00[x]          ;// Here
"rgbaout" is the dot product for the 3 lights
   mul            acc,      \lightcolors[0],  \rgbaout[x]
   madd           acc,      \lightcolors[1],  \rgbaout[y]
   madd           \rgbaout, \lightcolors[2],  \rgbaout[z]      ;// Here
"rgbaout" is the light applied on the vertex
   mul            \rgbaout, \vertexrgba,      \rgbaout         ;// Here
"rgbaout" is the amount of light reflected by the vertex
   .endm
```

```
;//---------------------------------------------------------------------
;// VertexLightDir3Amb - Apply up to 3 directional lights, plus an
;// ambient light contained in a light matrix "lightmatrix" to a vertex
;// of color "vertexrgba" and having a normal "vertexnormal", and
;// output the result in "outputrgba"
;//
;// Note: ACC register is modified
;//---------------------------------------------------------------------
   .macro   VertexLightDir3Amb
rgbaout,vertexrgba,vertexnormal,lightcolors,lightnormals
   mul            acc,      \lightnormals[0], \vertexnormal[x]
   madd           acc,      \lightnormals[1], \vertexnormal[y]
   madd           acc,      \lightnormals[2], \vertexnormal[z]
   madd           \rgbaout, \lightnormals[3], \vertexnormal[w] ;// Here
"rgbaout" is the dot product for the 3 lights
   max            \rgbaout, \rgbaout,         vf00[x]          ;// Here
"rgbaout" is the dot product for the 3 lights
   mul            acc,      \lightcolors[0],  \rgbaout[x]
   madd           acc,      \lightcolors[1],  \rgbaout[y]
   madd           acc,      \lightcolors[2],  \rgbaout[z]
   madd           \rgbaout, \lightcolors[3],  \rgbaout[w]      ;// Here
"rgbaout" is the light applied on the vertex
   mul.xyz        \rgbaout, \vertexrgba,      \rgbaout         ;// Here
"rgbaout" is the amount of light reflected by the vertex
   .endm

;//---------------------------------------------------------------------
;// FogSetup - Set up fog "fogparams", by specifying "nearfog" and
;// "farfog".  "fogparams" will afterward be ready to be used by fog-
;// related macros, like "VertexFogLinear" for example.
;//
;// Note: I register is modified
;//---------------------------------------------------------------------
   .macro   FogSetup fogparams,nearfogz,farfogz
   sub            \fogparams, vf00,          vf00           ;// Set XYZW to
0
   loi            \farfogz                                  ;//
   add.w          \fogparams, \fogparams,    i              ;// fogparam[w]
is farfogz
   loi            \nearfogz
   add.z          \fogparams, \fogparams,    \fogparams[w]
   sub.z          \fogparams, \fogparams,    i
   loi            255.0
   add.xy         \fogparams, \fogparams,    i              ;// fogparam[y]
is 255.0
   sub.x          \fogparams, \fogparams,    vf00[w]        ;// fogparam[x]
is 254.0
   div            q,          \fogparams[y], \fogparams[z]
   sub.z          \fogparams, \fogparams,    \fogparams
   add.z          \fogparams, \fogparams, q;// fogparam[z] is 255.f /
(farfogz - nearfogz)
   .endm
```

```
;//---------------------------------------------------------------------
;// VertexFogLinear - Apply fog "fogparams" to a vertex "xyzw", and
;// output the result in "xyzfoutput". "xyzw" [w] is assumed to be
;// the distance from the camera.  "fogparams" must contain farfogz in
;// [w], and (255.f / (farfogz - nearfogz)) in [z]. "xyzfoutputf" [w]
;// will contain a float value between 0.0 and 255.0, inclusively.
;//---------------------------------------------------------------------
   .macro   VertexFogLinear   xyzfoutput,xyzw,fogparams
   move.xyz       \xyzfoutput, \xyzw                    ;// XYZ part won't
be modified
   sub.w          \xyzfoutput, \fogparams,  \xyzw[w]    ;// fog = (farfogz -
z) * 255.0 /
   mul.w          \xyzfoutput, \xyzfoutput, \fogparams[z];//      (farfogz -
nearfogz)
   max.w          \xyzfoutput, \xyzfoutput, vf00[x]    ;// Clamp fog values
outside the
   mini.w         \xyzfoutput, \xyzfoutput, \fogparams[y];// range 0.0-255.0
   .endm

;//---------------------------------------------------------------------
;// VertexFogRemove - Remove any effect of fog to "xyzf".  "fogparams"
;// [x] must be set to 254.0.  "xyzf" will be modified directly.
;//---------------------------------------------------------------------
   .macro   VertexFogRemove   xyzf,fogparams
   add.w          \xyzf, vf00, \fogparams[x]  ;// xyzw[w] = 1.0 + 254.0 =
255.0 = no fog
   .endm

;//---------------------------------------------------------------------
;// PushInteger1 - Push "integer1" on "stackptr"
;//
;// Note: "stackptr" is updated
;//---------------------------------------------------------------------
   .macro   PushInteger1   stackptr,integer1
   isubiu         \stackptr, \stackptr, 1
   iswr.x         \integer1, (\stackptr):VCLSML_STACK
   .endm

;//---------------------------------------------------------------------
;// PushInteger2 - Push "integer1" and "integer2" on "stackptr"
;//
;// Note: "stackptr" is updated
;//---------------------------------------------------------------------
   .macro   PushInteger2   stackptr,integer1,integer2
   isubiu         \stackptr, \stackptr, 1
   iswr.x         \integer1, (\stackptr):VCLSML_STACK
   iswr.y         \integer2, (\stackptr):VCLSML_STACK
   .endm

;//---------------------------------------------------------------------
;// PushInteger3 - Push "integer1", "integer2" and "integer3" on
;// "stackptr"
;//
;// Note: "stackptr" is updated
;//---------------------------------------------------------------------
   .macro   PushInteger3   stackptr,integer1,integer2,integer3
   isubiu         \stackptr, \stackptr, 1
   iswr.x         \integer1, (\stackptr):VCLSML_STACK
   iswr.y         \integer2, (\stackptr):VCLSML_STACK
   iswr.z         \integer3, (\stackptr):VCLSML_STACK
   .endm
```

```
;//------------------------------------------------------------------
;// PushInteger4 - Push "integer1", "integer2", "integer3" and
;// "integer4" on "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------
   .macro   PushInteger4   stackptr,integer1,integer2,integer3,integer4
   isubiu         \stackptr, \stackptr, 1
   iswr.x         \integer1, (\stackptr):VCLSML_STACK
   iswr.y         \integer2, (\stackptr):VCLSML_STACK
   iswr.z         \integer3, (\stackptr):VCLSML_STACK
   iswr.w         \integer4, (\stackptr):VCLSML_STACK
   .endm

;//------------------------------------------------------------------
;// PopInteger1 - Pop "integer1" on "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------
   .macro   PopInteger1   stackptr,integer1
   ilwr.x         \integer1, (\stackptr):VCLSML_STACK
   iaddiu         \stackptr, \stackptr, 1
   .endm

;//------------------------------------------------------------------
;// PopInteger2 - Pop "integer1" and "integer2" on "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------
   .macro   PopInteger2   stackptr,integer1,integer2
   ilwr.y         \integer2, (\stackptr):VCLSML_STACK
   ilwr.x         \integer1, (\stackptr):VCLSML_STACK
   iaddiu         \stackptr, \stackptr, 1
   .endm

;//------------------------------------------------------------------
;// PopInteger3 - Pop "integer1", "integer2" and "integer3" on
;// "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------
   .macro   PopInteger3   stackptr,integer1,integer2,integer3
   ilwr.z         \integer3, (\stackptr):VCLSML_STACK
   ilwr.y         \integer2, (\stackptr):VCLSML_STACK
   ilwr.x         \integer1, (\stackptr):VCLSML_STACK
   iaddiu         \stackptr, \stackptr, 1
   .endm
```

```
;//------------------------------------------------------------------------
;// PopInteger4 - Pop "integer1", "integer2", "integer3" and
;// "integer4" on "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------------
   .macro   PopInteger4   stackptr,integer1,integer2,integer3,integer4
   ilwr.w         \integer4, (\stackptr):VCLSML_STACK
   ilwr.z         \integer3, (\stackptr):VCLSML_STACK
   ilwr.y         \integer2, (\stackptr):VCLSML_STACK
   ilwr.x         \integer1, (\stackptr):VCLSML_STACK
   iaddiu         \stackptr, \stackptr, 1
   .endm

;//------------------------------------------------------------------------
;// PushMatrix - Push "matrix" onto the "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------------
   .macro   PushMatrix  stackptr,matrix
   sq             \matrix[0], -1(\stackptr):VCLSML_STACK
   sq             \matrix[1], -2(\stackptr):VCLSML_STACK
   sq             \matrix[2], -3(\stackptr):VCLSML_STACK
   sq             \matrix[3], -4(\stackptr):VCLSML_STACK
   iaddi          \stackptr, \stackptr, -4
   .endm

;//------------------------------------------------------------------------
;// PopMatrix - Pop "matrix" out of the "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------------
   .macro   PopMatrix   stackptr,matrix
   lq             \matrix[0], 0(\stackptr):VCLSML_STACK
   lq             \matrix[1], 1(\stackptr):VCLSML_STACK
   lq             \matrix[2], 2(\stackptr):VCLSML_STACK
   lq             \matrix[3], 3(\stackptr):VCLSML_STACK
   iaddi          \stackptr, \stackptr, 4
   .endm

;//------------------------------------------------------------------------
;// PushVector - Push "vector" onto the "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------------
   .macro   PushVector  stackptr,vector
   sqd            \vector, (--\stackptr):VCLSML_STACK
   .endm

;//------------------------------------------------------------------------
;// PopVector - Pop "vector" out of the "stackptr"
;//
;// Note: "stackptr" is updated
;//------------------------------------------------------------------------
   .macro   PopVector   stackptr,vector
   lqi            \vector, (\stackptr++):VCLSML_STACK
   .endm
```

```
;//--------------------------------------------------------------------
;// PushVertex - Push "vector" onto the "stackptr"
;//
;// Note: "stackptr" is updated
;//--------------------------------------------------------------------
   .macro   PushVertex  stackptr,vertex
   sqd            \vertex, (--\stackptr):VCLSML_STACK
   .endm

;//--------------------------------------------------------------------
;// PopVertex - Pop "vertex" out of the "stackptr"
;//
;// Note: "stackptr" is updated
;//--------------------------------------------------------------------
   .macro   PopVertex   stackptr,vertex
   lqi            \vertex, (\stackptr++):VCLSML_STACK
   .endm

;//--------------------------------------------------------------------
;// AngleSinCos - Returns the sin and cos of up to 2 angles, which must
;// be contained in the X and Z elements of "angle".  The sin/cos pair
;// will be contained in the X/Y elements of "sincos" for the first
;// angle, and Z/W for the second one.
;// Thanks to Colin Hughes (SCEE) for that one
;//
;// Note: ACC and I registers are modified, and a bunch of temporary
;//       variables are created... Maybe bad for VCL register pressure
;//--------------------------------------------------------------------
   .macro   AngleSinCos    angle,sincos
   move.xz        \sincos, \angle              ; To avoid modifying the
original angles...

   mul.w          \sincos, vf00, \sincos[z]    ; Copy angle from z to w
   add.y          \sincos, vf00, \sincos[x]    ; Copy angle from x to y

   loi            1.570796                     ; Phase difference for sin as
cos ( PI/2 )
   sub.xz         \sincos, \sincos, I          ;

   abs            \sincos, \sincos                           ; Mirror cos
around zero

   max            Vector1111, vf00, vf00[w]                  ; Initialise all
1s

   loi            -0.159155      ; Scale so single cycle is range 0 to -1 (
*-1/2PI )
   mul            ACC,  \sincos, I                           ;

   loi            12582912.0                    ; Apply bias to remove
fractional part
   msub           ACC,  Vector1111, I                        ;
   madd           ACC,  Vector1111, I           ; Remove bias to leave
original int part

   loi            -0.159155         ; Apply original number to leave
fraction range only
   msub           ACC,  \sincos, I                           ;

   loi            0.5                                        ; Ajust range: -
0.5 to +0.5
```

```
   msub           \sincos, Vector1111, I                    ;

   abs            \sincos, \sincos                          ; Clamp: 0 to
+0.5

   loi            0.25                                      ; Ajust range: -
0.25 to +0.25
   sub            \sincos, \sincos, I                       ;

   mul            anglepower2, \sincos, \sincos             ; a^2

   loi            -76.574959                                ;
   mul            k4angle, \sincos, I                       ; k4 a

   loi            -41.341675                                ;
   mul            k2angle, \sincos, I                       ; k2 a

   loi            81.602226                                 ;
   mul            k3angle, \sincos, I                       ; k3 a

   mul            anglepower4, anglepower2, anglepower2     ; a^4
   mul            k4angle, k4angle, anglepower2             ; k4 a^3
   mul            ACC,  k2angle, anglepower2                ; + k2 a^3

   loi            39.710659                                 ; k5 a
   mul            k2angle, \sincos, I                       ;

   mul            anglepower8, anglepower4, anglepower4     ; a^8
   madd           ACC,  k4angle, anglepower4                ; + k4 a^7
   madd           ACC,  k3angle, anglepower4                ; + k3 a^5
   loi            6.283185                                  ;
   madd           ACC,  \sincos, I                          ; + k1 a
   madd           \sincos, k2angle, anglepower8             ; + k5 a^9
   .endm

;//---------------------------------------------------------------------
;// QuaternionToMatrix - Converts a quaternion rotation to a matrix
;// Thanks to Colin Hughes (SCEE) for that one
;//
;// Note: ACC and I registers are modified
;//---------------------------------------------------------------------
   .macro   QuaternionToMatrix   matresult,quaternion

   mula.xyz       ACC,  \quaternion, \quaternion        ; xx yy zz

   loi            1.414213562
   muli           vclsmlftemp, \quaternion, I           ; x sqrt2 y sqrt2 z
sqrt2 w sqrt2

   mr32.w         \matresult[0], vf00                   ; Set rhs matrix
line 0 to 0
   mr32.w         \matresult[1], vf00                   ;
   mr32.w         \matresult[2], vf00                   ; Set rhs matrix
   move           \matresult[3], vf00                   ; Set bottom line to
0 0 0 1

   madd.xyz       vcl_2qq, \quaternion, \quaternion     ; 2xx       2yy
2zz
   addw.xyz       Vector111, vf00, vf00                 ; 1         1
1         -
```

```
   opmula.xyz     ACC,  vclsmlftemp, vclsmlftemp        ; 2yz       2xz
2xy       -
   msubw.xyz      vclsmlftemp2, vclsmlftemp, vclsmlftemp; 2yz-2xw   2xz-2yz
2xy-2zw   -
   maddw.xyz      vclsmlftemp3, vclsmlftemp, vclsmlftemp; 2yz+2xw   2xz+2yz
2xy+2zw   -
   addaw.xyz      ACC,  vf00, vf00                      ; 1         1
1         -
   msubax.yz      ACC,  Vector111, vcl_2qq              ; 1         1-2xx
1-2xx

   msuby.z        \matresult[2], Vector111, vcl_2qq     ; -         -
1-2xx-2yy -
   msubay.x       ACC, Vector111, vcl_2qq               ; 1-2yy     1-2xx
1-2xx-2yy -
   msubz.y        \matresult[1], Vector111, vcl_2qq     ; -         1-2xx-
2zz -         -
   mr32.y         \matresult[0], vclsmlftemp2
   msubz.x        \matresult[0], Vector111, vcl_2qq     ; 1-2yy-2zz -
-         -
   mr32.x         \matresult[2], vclsmlftemp2
   addy.z         \matresult[0], vf00, vclsmlftemp3
   mr32.w         vclsmlftemp, vclsmlftemp2
   mr32.z         \matresult[1], vclsmlftemp
   addx.y         \matresult[2], vf00, vclsmlftemp3
   mr32.y         vclsmlftemp3, vclsmlftemp3
   mr32.x         \matresult[1], vclsmlftemp3

   .endm

;//-----------------------------------------------------------------------
;// QuaternionMultiply - Multiplies "quaternion1" and "quaternion2",
;// and puts the result in "quatresult".
;// Thanks to Colin Hughes (SCEE) for that one
;//
;// Note: ACC register is modified
;//-----------------------------------------------------------------------
   .macro   QuaternionMultiply   quatresult,quaternion1,quaternion2
   mul            vclsmlftemp, \quaternion1, \quaternion2   ; xx yy zz ww

   opmula.xyz     ACC,         \quaternion1, \quaternion2   ; Start
Outerproduct
   madd.xyz       ACC,         \quaternion1, \quaternion2[w]; Add w2.xyz1
   madd.xyz       ACC,         \quaternion2, \quaternion1[w]; Add w1.xyz2
   opmsub.xyz     \quatresult, \quaternion2, \quaternion1   ; Finish
Outerproduct

   sub.w          ACC,         vclsmlftemp,  vclsmlftemp[z] ; ww - zz
   msub.w         ACC,         vf00,         vclsmlftemp[y] ; ww - zz - yy
   msub.w         \quatresult, vf00,         vclsmlftemp[x] ; ww - zz - yy -
xx
   .endm
```

# Appendix B: Detailed Information Regarding Loops in the VCL Preprocessor

## Pipelining and VCL

When the "`--LoopCS`" directive is used, the VCL preprocessor will attempt to unroll and pipeline back to the start any block of code that ends with a conditional branch.

The VCL preprocessor analyzes the sequence of instructions and determines what the best size for the loop would be. A simplistic calculation for the loop size is:

```
best_size = max (number_upper_instructions, number_lower_instructions,
sum_throughput_p, sum_throughput_q)
```

The actual calculation is actually more involved, particularly due to issues like IALU instructions placing (with respect to the branch) and the possibility of circular dependency chains in the instruction sequence.

**Note:** The VCL preprocessor currently relies on the user to identify memory store/load sequences. It does so by the use of tags. (Refer to "Memory Aliasing and Instructions Reordering" on page 9 for more details.)

For a typical renderer, the usual circular chain is of the form:

```
IADDIU       ptr, ptr, sizeof
IBNE         ptr, end_ptr, loop
```

Or:

```
IADDI        count, count, -1
IBNE         count, vi00, loop
```

These cases aren't long compared to the size of the loop. (The rest of a typical renderer pulls data out of memory, processes it, then writes it back to memory. However, it does not cross a loop iteration boundary.)

For other types of code loops (such as physics dynamics calculations on strings), where the output from one loop iteration is an input to the next, and the length of the calculation is the same as the overall length of the loop, pipelining will not greatly improve performances without a modification of the algorithm. Such a modification could be to process multiple strings at once.

After analyzing the code, the VCL preprocessor tries to schedule it so that it fits into a block that is of size greater or equal to 0, but smaller than the loop size. It does so by wrapping instructions off the end and back to the start. The number of times an instruction is wrapped around the loop will determine the stage number to which it belongs.

In short:

- In linear mode (non-looped), VCL schedules in Z instructions.
- In loop mode, it schedules in Z modulo `n`, where `n` equals best_size + current optimization phase.
- Software pipelining where the branch is at the end of the critical path will not be optimized greatly by VCL.

Study the following example, which will be referred to later in this appendix:

```
vrt_loop:
      --LoopCS     10,10
      LQI          vrt, (in_p++)              ; Load vertex

      MUL          acc,  mat[0], vrt[x]              ; Transform vertex
      MADD         acc,  mat[1], vrt[y]              ;
      MADD         acc,  mat[2], vrt[z]              ;
      MADD         camv, mat[3], vf00[w]             ;

      DIV          q, vf00[w], camv[w]        ; Perspective correction
      MUL          screenv, camv, q           ;

      FTOI4        fixpt, screenv                    ; Convert to GS format

      SQI          fixpt, (out_p++)           ; Save the vertex

      IADDI        count, count, -1           ; Next vertex...
      IBNE         count, vi00, vrt_loop             ;
```

The following is a typical sequence of instructions for a loop, where the numerals 1 to 9 denote blocks of instructions that are grouped by pipeline stages. Execution in the order 1, 2, 3, 4, 5, 6, 7, 8, 9 is equivalent to one iteration of the loop.

**Figure 1**

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6c |
| 7 |
| 8 |
| 9m |

"c" denotes the stage for the conditional.. In the example above, it is highly likely that the conditional will be at stage 1, since it is not dependent on much else in the code. "m" is the last stage. In a typical renderer, this would often correspond to the color calculations.

As pipeline execute happens:

**Figure 2**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prologue | 1 | | | | | | | | | P1 starts |
| | 2 | 1 | | | | | | | | P2 starts, P1 is at stage 2 |
| | 3 | 2 | 1 | | | | | | | P3 starts, P2 is at stage 2, P3 is at stage 3 |
| | 4 | 3 | 2 | 1 | | | | | | Etc… |
| | 5 | 4 | 3 | 2 | 1 | | | | | |
| | 6c | 5 | 4 | 3 | 2 | 1 | | | | |
| | 7 | 6c | 5 | 4 | 3 | 2 | 1 | | | |
| | 8 | 7 | 6c | 5 | 4 | 3 | 2 | 1 | | |
| MainLoop | 9 | 8 | 7 | 6c | 5 | 4 | 3 | 2 | 1 | Main body of the loop |
| Epilogue | | 9 | 8 | 7 | | | | | | |
| | | | 9 | 8 | | | | | | |
| | | | | 9 | | | | | | |

If the conditional is at stage C (here, 6), then stages 1 to C-1 will miss-execute, i.e. in the main loop, the pipeline runs C-1 stages ahead of the conditional. But when the condition is found to be true, the VCL preprocessor will only complete the processing for valid stages that are in the graph above 9, 8-9, and 7-8-9 (epilogue part).

## "--LoopCS n,m" Directive

### n (Minimum Number of Loops)

For small loop counts, there are many instructions that are associated with pipeline stages ahead of the conditional. To get better performance for a small count (n greater or equal to 1 but smaller than M-C), once the conditional in the loop is encountered, the VCL preprocessor can jump to a special case to complete the calculations on the required pipeline stages for this count.

Following is a modified version of the above diagram:

**Figure 3**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | | | | | | | |
| | 2 | 1 | | | | | | | |
| | 3 | 2 | 1 | | | | | | |
| | 4 | 3 | 2 | 1 | | | | | |
| | 5 | 4 | 3 | 2 | 1 | | | | |
| A | 6c | 5 | 4 | 3 | 2 | 1 | | | |
| B | 7 | 6c | 5 | 4 | 3 | 2 | 1 | | |
| C | 8 | 7 | 6c | 5 | 4 | 3 | 2 | 1 | |
| MainLoop | 9 | 8 | 7 | 6c | 5 | 4 | 3 | 2 | 1 |
| | | 9 | 8 | 7 | | | | | |
| | | | 9 | 8 | | | | | |
| | | | | 9 | | | | | |

If the code has a small minimum count, such as n = 1, it is possible for the code to exit at "A" and go to a special-case epilogue (EPI_A). The steps that have already taken place are:

**Figure 4**

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1 | | | | | |
| | 2 | 1 | | | | |
| | 3 | 2 | 1 | | | |
| | 4 | 3 | 2 | 1 | | |
| | 5 | 4 | 3 | 2 | 1 | |
| A | 6c | 5 | 4 | 3 | 2 | 1 |

If the condition at stage 6 is found to be true, the following will be executed:

**Figure 5**

| | | | | | | |
|---|---|---|---|---|---|---|
| EPI_A | 7 | | | | | |
| | 8 | | | | | |
| | 9 | | | | | |

Similarly for n = 2, a special epilogue may be created for the following (EPI-B):

**Figure 6**

| | | | | | | |
|---|---|---|---|---|---|---|
| EPI_B | 8 | 7 | | | | |
| | 9 | 8 | | | | |
| | | 9 | | | | |

In the case above, having nine stages would require the creation of nine special epilogues, which is a lot of code generation. However, if the VCL preprocessor is told –via the "Minimum Number of Loops"- that there will always be, for example, three iterations, then the special case codes EPI_A and EPI_B as well as the two conditionals A and B may be removed altogether.

Up to at least version 1.3, the VCL preprocessor does not reschedule instructions across conditionals. So conditional removals by ways described above will most certainly result in better code optimization.

### m (Slop Count)

Referring to the tables above, it can be seen that, in the main loop, the VCL preprocessor will execute some stages ahead of the conditional (in the example, stages 1 to 5). If these contain instructions with side effects (like memory stores and XGKick), this could result in data corruption, since by the time the conditional takes place, such instructions would have already executed.

In the following table, "*" denotes stages containing an instruction with side-effects, and "f" denotes the first instruction containing instructions with side-effects. "c" denotes the conditional stage, and "m" the last stage.

**Figure 7**

| |
|---|
| 1 |
| 2f* |
| 3 |
| 4* |
| 5 |
| 6c |
| 7* |
| 8 |
| 9m |

If the first side effect is at stage 2, then stage 2 will have miss-executed a maximum count greater or equal to 0, but smaller than c-f (6-2=4). This is acceptable if extra padding is provided at the end of the store buffer. The number of available padding slots is specified with the Slop Count.

If, for example, m=1, then the above case would generate incorrect code, since c-f=4.

However, for the following case, the generated code would be correct, since c-f=1.

**Figure 8**

| |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5f* |
| 6c |
| 7* |
| 8 |
| 9m |

If no side-effect stages can be mis-executed, then the Slop Count must be set to 0. Note that this will, however, result in real constraints on VCL code generation.

### .rawloop / .endrawloop

For regular loops, the VCL preprocessor will analyze the loop and decide how to stage the instructions and reschedule them accordingly. Then compatible prologues and epilogues will be created around the loop's main body.

In some cases, however, you may already know what the stages are like, and simply want the VCL preprocessor to unroll them and create the prologue and epilogue. Use raw loops for this..

The regular loop shown in "Pipelining and VCL" would be similar to the following example, using raw loops:

```
.rawloop

vrt_loop:
       --LoopCS     10,10

       2..MADD    acc, mat[2], vrt[z]   ; Rotate vertex .. 2
     = 1..LQI      vrt, (in_p++)        ; Load vertex, increment pointer

       2..MADD    camv, mat[3], vf00[w] ; Translate rotated vertex
     = 3..MOVE    ocamv, camv           ; Save copy of camera space coordinate

          NOP
     = 5..SQI      fixpt, (out_p++)     ; Save out GS-format vertex

       4..FTOI4    fixpt, screenv       ; Convert screen coordinate to GS-format
     = 1..IADDI    count, count, -1     ; Decrement loop counter

       1..MUL      acc, mat[0], vrt[x]  ; Rotate vertex .. 0
     =    NOP

       1..MADD    acc, mat[0], vrt[y]   ; Rotate vertex .. 1
     = 1..IBNE    count, vi00, vrt_loop ; Reached the end?

       3..MUL      screenv, ocamv, q    ; Do perspective divide
       2..DIV      q, vf00[w], camv[w]  ; Start perspective divide calculation
.endrawloop
```

The "`<n>..`" instruction syntax tells the VCL preprocessor in which stage of the loop the instruction belongs, so it can generate proper prologues and epilogues.

In raw mode, "=" may be used as a "line continue" character, as long as it is the first non-white character on a lower-instruction line. This permits better comments placement. If not used, upper and lower instructions must be placed on the same line, much like regular VSM code.

The VCL preprocessor will create the prologue by first taking the instructions with a "1…" suffix, then the instructions with a "2…" suffix, and so on.

Prologue and epilogue instructions are not rescheduled in the case of a raw loop, as they are for regular loop unrolling. Therefore, this may result in sub-optimal code. However, this is necessary for cases where the code needs to run on VU0 in parallel with code on the EE, without synchronization.

# Appendix C: VCL Tips and Common Mistakes

This appendix is a handy compilation of common VCL mistakes often encountered by beginning VCL developers.

## Preprocessor Errors

When using either GASP or the C preprocessor, make sure you have permissions to create temporary files. Any preprocessor error will halt VCL.

## Reordering of Instructions

When converting 2-stream code to 1-stream VCL code, make sure to bring back branch-delay-slot instructions before the branch or jump instruction. Be also careful with instruction groups like divq/mulq, clip/fcand, FMAC/fsand, etc.

## Working Registers

Make sure to give VCL enough registers to work with, else the generated code might not be as optimized as it could otherwise, or worse VCL might not be able to generate code at all. (See “Register Availability” on page 4 for more details.)

## Input and Output Registers

Make sure to specify input and output variables in the code. Not specifying input will result in a VCL error, but not specifying output will simply result in instruction pruning, and hence will not output expected code. (See “Data Tracking” and “Set Before Use” on page 12 for more details.)

## Entry Points

VCL expects at least one entry point. Therefore, “--enter/--endenter” must be specified at least once.

## Exit Points in Code

Make sure to use “--exit/--endexit” and “--cont” properly. Although both will stop VU execution, “--exit/--endexit” is understood by VCL as meaning that code preceding and following are not related, so no data dependency will be performed. In contrast “--cont” can be thought of as a pause operation, and does not offer the same data dependency breaking feature.

Improperly swapping the 2 might not generate an error, but the code will not operate as intended.

## Conditional Branching and Loop Unrolling

VCL does not support conditional branching within a loop that is to be unrolled, but will not give a warning about it, so sub-optimal code might be inadvertently generated.

## Number Literals

Number literals, as used with instructions like LOI, may be either specified as float or integers. But to be recognized as an integer, it must be specified as a hexadecimal value. Values like 1 will be understood as

1.0, and therefore converted as 0x3F800000 (floating-point representation for 1.0). (See “Number Literals Peculiarities” on page 4 for more details.)

## Memory Management

VCL does not manage VU memory in any way, and does not implicitly check for memory aliasing. To avoid memory aliasing issues, related instructions must be explicitly tagged, and other common memory management issues must be managed by the programmer. (See “Memory Aliasing and Instructions Reordering” on page 9 for more details.)

## Variable Names

Because a given VU instruction either operates on an integer or floating-point register, VCL allows the same variable name to be used for an integer and a floating-point variable. This can lead to confusion for the programmer, so this feature must be used with caution. (See “Floating-Point and Integer Variable Naming” on page 3 for more details.)

## Broadcast Instructions and Variable Names

Using the new syntax, instructions accepting a broadcast field will look for a field identifier at the end of the line, as such:

```
MADD     acc, matrix1, inputvertex[y]
```

In any other cases, suffixing a variable name with “[x]”, “[y]”, “[z]” or “[w]” will not be considered a broadcast identifier, and the suffix will be merged with the name itself to form a new variable name. Therefore, in such cases, names like “color” and “color[w]” would refer to different variables.

(See “Broadcast Instructions” on page 6 for more details.)

## Long Dependency chains

Leaving long dependency chains of the form:

```
MUL      register,register,register  ; register.xyzw modified
MUL.xyz  register,register,register  ; only register.xyz modified
MUL.xyz  register,register,register  ; only register.xyz modified
MUL.xyz  register,register,register  ; only register.xyz modified
MUL.xyz  register,register,register  ; only register.xyz modified
MUL.xyz  register,register,register  ; only register.xyz modified
MUL      register,register,register  ; register w also used
MUL      register,register,register  ; register w also used
```

Using a different register at each stage is impossible, since the 'w' value will be discarded. The internal move generator is cautious at the moment, so VCL will not explicitly separate the W component from the rest.

## Typos and Instruction Pruning

Typos can be common when coding. These are easily caught if they are within an instruction name, but if they are in a variable name, finding them might prove to be tricky if they don’t generate errors like “use before set”. Sometimes, variable-name typos will result in VCL pruning otherwise valid code.

Looking carefully at the output from VCL can therefore help catch this kind of mistake.

## EFU Instructions Usage

EFU instructions are fine when used sparingly, however their long latency and throughput compared to Q-related instructions mean they will most likely be the limiting factor within loops. So they should be used with caution.

## Register with 1s

It is possible to create a register with 1s by doing:

```
MAX         Vector1111, vf00, vf00[w]
```

Or, using lower instructions:

```
RINIT       R, vf00[x]
RGET        Vector1111, R
```

## Dot Product (Inner Product)

If you have a float register containing (1.0, 1.0, 1.0, 1.0), it is possible to calculate the dot product of 2 vectors by doing the following:

```
MUL         temp, VectorA, VectorB
ADD.x       acc, temp, temp[y]
MADD.x      DotProduct, Vector1111, temp[z]
```

But because inside the body of a loop throughput, not latency, is sometimes the limiting factor, calculating a dot product in the following way might be useful:

```
MUL         temp, VectorA, VectorB
ADD.x       acc, temp, temp[y]
ADD.x       DotProduct, temp, temp[z]
```

This is the case because VCL is free to rearrange register assignment for temp.x while the dot product calculation is in flight, whereas in the former code snippet, VCL is prohibited from using ACC.x while calculations are taking place.

This page intentionally left blank.