Programming Tips

Table of Contents

About This Manual

This is the Runtime Library Release 2.4 version of the *Programming Tips* manual.

Changes Since Last Release

New.

Related Documentation

The *Creating and Playing Movie Data* manual describes stream data, explains how to create stream data and lists the streaming-related libraries.

Note: the Developer Support Web site 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.

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In North America:	*In North America:*
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: PS2_Support@playstation.sony.com Web: http://www.devnet.scea.com/ Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees in Europe only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
In Europe:	*In Europe:*
Attn: Production Coordinator Sony Computer Entertainment Europe 30 Golden Square London W1F 9LD, U.K. Tel: +44 (0) 20 7859-5000	E-mail: ps2_support@scee.net Web: https://www.ps2-pro.com/ Developer Support Hotline: +44 (0) 20 7859-5777 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT)

Overview

This document describes a variety of programming techniques for the PlayStation 2. In particular, it describes methods that can be used to optimize the performance of many applications, and emphasizes those areas that are most prone to misunderstanding, or are often overlooked. In conjunction with other documentation, this document can help make development more efficient.

Cache and Scratchpad

Appropriate use of the cache is a key factor in optimizing the performance of EE Core programs. The data cache (D$), scratchpad (SPR) and instruction cache (I$) are provided to compensate for the difference in speed between main memory and the EE core. However, they need to be used effectively or a large cache-miss penalty and performance hit will be incurred. This document discusses the structure and features of the two types of caches and the scratchpad, and gives guidelines for using them effectively.

- How to avoid cache misses
- How to keep data that will not be reused out of the cache
- How to perform scratchpad processing and DMA transfers in parallel

This document also explains common cache-related problems.

Multimedia Instructions

One of the special features of the EE core is that it provides multimedia instructions for processing 128-bit data in parallel. This document provides a basic introduction to these instructions.

Compiler Characteristics

While ee-gcc can usually be used to produce suitably efficient code, there are some special cases where this might not be the case. This document discusses some of the more common problems you might encounter.

VU Programming

This document gives an overview of double-buffering (the basis for VU1 programming) and the tuning of microprograms. It also covers common problems you need to be aware of regarding VU microinstructions.

GS Characteristics

With the GS architecture, you can suffer a speed hit depending on the textures and polygon structures you use. This document gives a basic overview of graphics and how to work within these constraints.

Effective Use of the Data Cache

Characteristics of the Data Cache

The data cache is a high-speed memory that buffers data exchanged between main memory and the EE core, as a result of executing load and store instructions. To put it very simply, because it takes time to read and write data from/to main memory, some data is placed in the cache, and as long as reading/writing the cache is sufficient, main memory does not need to be accessed at all. When a cache miss occurs, that is, when the desired data isn't present in the cache, main memory is automatically accessed.

The data cache is used even if programmers aren't especially aware of it when coding, and there are many cases where it can be used quite easily and effectively. However, if you fail to keep time-consuming memory accesses in mind, there can be a large impact on the performance of the program. To get the best performance, you need to write code that keeps cache misses to a minimum.

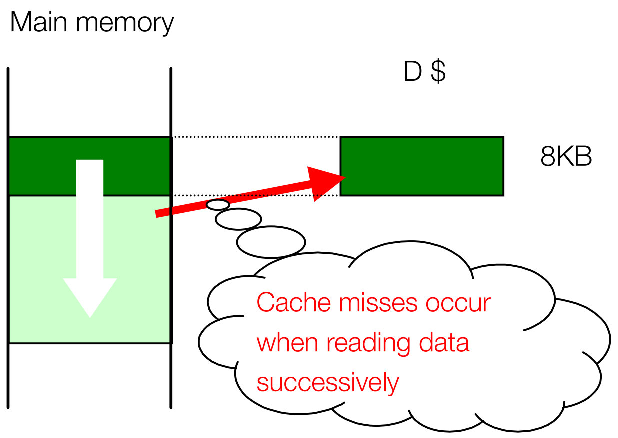
Data Suitable for the Data Cache

The data cache provides the best results when it contains the following kinds of data:

- Data that is frequently read or written repeatedly
- Data with a high degree of locality

Automatic variables on the stack are a good example of this type of data. Conversely, there's no point in using the cache for data that gets used only once. And using the cache for big chunks of data that are larger than the cache size (8 KB) will always result in cache misses.

Figure 1: Large data chunks cause cache misses



How to Reduce Data Cache Misses

The basic technique for reducing cache misses is to avoid putting unnecessary data in the cache.

With data that will only be read or written once, it's best to perform a non-cached access that doesn't use the cache. Non-cached accesses are described later in this document.

As Figure 1 shows, when processing large chunks of data successively, it's better to use the scratchpad. The scratchpad is described below.

Using the prefetch instruction (PREF) will explicitly load a particular piece of data into the cache. Using the prefetch instruction a few steps before it is actually needed will cause a cache miss when it is used, but will not have an overall impact on performance.

How to Measure Data Cache Misses

The performance counter library (libpc) can be used to measure data cache misses. The following example program illustrates this.

```
int control, count;
control = SCE_PC0_NO_EVENT | (SCE_PC_U0|SCE_PC_S0|SCE_PC_K0| SCE_PC_EXL0);
control |= SCE_PC1_DCACHE_MISS | (SCE_PC_U1);
control |= SCE_PC_CTE;
(code omitted)
scePcStart( control, 0, 0); // Start measurement
(Routine where cache misses are to be measured)
count = scePcGetCounter1(); // Get measured value
scePcStop();
```

Strictly speaking, this does not give you a precise count of cache misses, rather, it represents the number of times data was not loaded in the cache during a load or store instruction. For example, in non-blocking load mode, if there is a cache miss, the data is loaded into the cache from main memory, and if you attempt to load adjacent data (on the same cache line) during that time, the second load will not result in a main memory access but will nevertheless be counted as a data cache miss. Also, a load/store operation of uncached data will also be counted as a data cache miss.

Precautions Regarding DMA and the Data Cache

Memory accesses by DMA are performed independently of memory accesses from the EE core. In the EE core, because there is no cache snooping function that forces the contents of main memory that would be visible as they passed through the data cache from the EE core, to match the contents of main memory as viewed directly by the DMAC, you must be aware of whether data being transferred by DMA is resident in the data cache or main memory.

The EE Kernel library overview document gives more details on this, but in brief, you need to follow the procedures described below to ensure proper DMA transfers:

EE core -> main memory -> peripheral:

1. Write data to the buffer (write to the data cache)
2. Writeback the data cache (write from the data cache to main memory)
3. Perform the DMA transfer

Peripheral -> main memory -> EE core:

1. Invalidate the cache corresponding to the buffer
2. Perform the DMA transfer
3. Read in buffer (load into the data cache)

When executing a non-cached access for the buffer, there is no need to writeback the data cache or to invalidate it.

Precautions Regarding the Data Cache and Alignment

The data cache has a 64-byte line size, which means that when data is exchanged with main memory, writebacks, and invalidations are all done with 64-byte aligned blocks.

One point you need to bear in mind is that when the writeback data size is not a multiple of 64 bytes, or is not aligned on 64 bytes, nearby data will be written back along with it. Since those areas could use variables from different processes, you might have a problem where data is unexpectedly written back.

How to specify alignment

When specifying variable alignment, add the "aligned" attribute to the variable definition. For example, if you want to specify 16-byte alignment, do the following:

a. Initialized static/global variable (data section)

```
unsigned char data[] __attribute__ ((aligned(16))) = { 'a', 'b', 'c' };
```

b. Uninitialized static/global variable (bss section)

```
unsigned char data[1*1024*1024] __attribute__ ((aligned(16)));
```

It is invalid to specify alignment for automatic variables.

The data section and bss section differ as to whether or not they can be written as data in an elf file. Specifying the -fno-common option when compiling will allocate a data section with uninitialized variables, and the elf file can grow by that amount.

Effective Use of the Scratchpad (SPR)

Characteristics of the Scratchpad

The scratchpad contains 16 KB of high-speed memory. Scratchpad memory can be accessed from the EE core as memory mapped to the range 0x70000000-0x70003fff. Data can be transferred between the scratchpad and main memory using DMA.

The scratchpad differs from the data cache in that the contents of the scratchpad and transfer timing are entirely under the control of the programmer. Accordingly, the scratchpad is ineffective unless the programmer actively uses it.

Data Suitable for theScratchpad

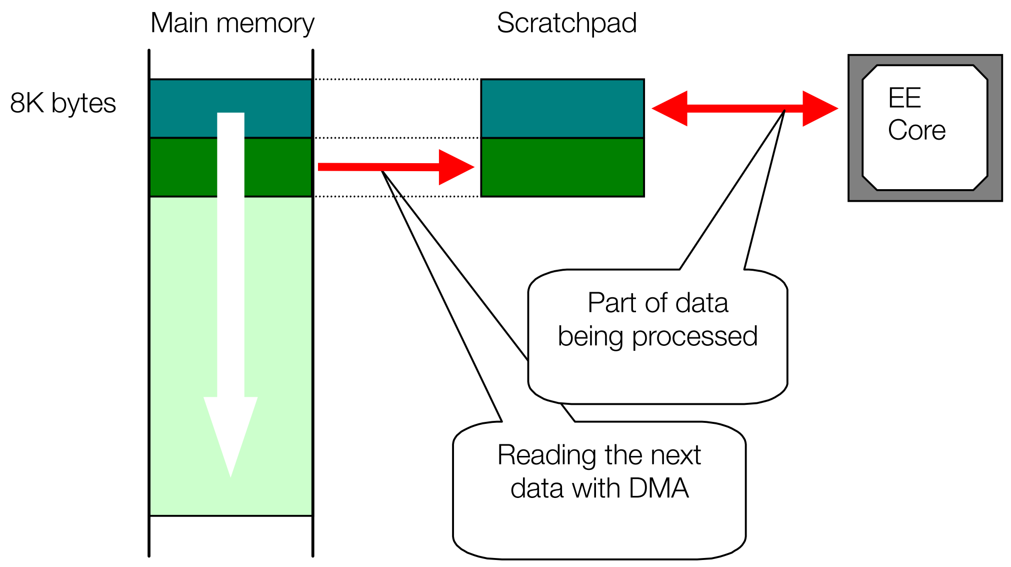
Any data that will be accessed frequently, and where speed is a priority, is suitable for the scratchpad. Placing data in the scratchpad will not cause a cache miss, and will not consume space in the data cache.

In addition, when sequential processing is performed with large amounts of data broken into small chunks, the scratchpad can be used effectively as a double buffer.

Processing Large Amounts of Data With the Scratchpad

When dealing with more data than can fit in the scratchpad, if that data can be divided into 8 KB blocks and processed sequentially, data can be DMA transferred to the scratchpad while at the same time, the data in the scratchpad is processed in parallel, for faster throughput.

Figure 2: Processing large amounts of data with the scratchpad



Non-Cached Accesses

Characteristics of Non-Cached Accesses

In some situations, it is preferable to access main memory directly, bypassing the cache entirely. The EE core provides three cache modes to support this.

- Cached:
  Accesses are performed through the data cache
- Uncached:
  Main memory is accessed directly, bypassing the data cache
- Uncached-accelerated:
  Main memory is accessed through the 128-byte UCAB (uncached acceleration buffer), bypassing the data cache. Uncached-accelerated is faster than uncached mode when reading data continuously

Uncached and uncached-accelerated modes are referred to collectively as non-cached accesses. Non-cached accesses mainly have the following differences when compared with cached accesses.

- Slower than cached accesses in situations that would cause a cache hit
- Faster for store operations in situations that would cause a cache miss
- Memory is written directly, so the cache need not be written back

Store operations are comparatively faster in situations that would cause a cache miss as the loading of data corresponding to the so-called write allocate operation, before the data is written, is not performed. There is an advantage to non-cached accesses to the degree that this load is not performed.

Data Suitable for Non-Cached Accesses

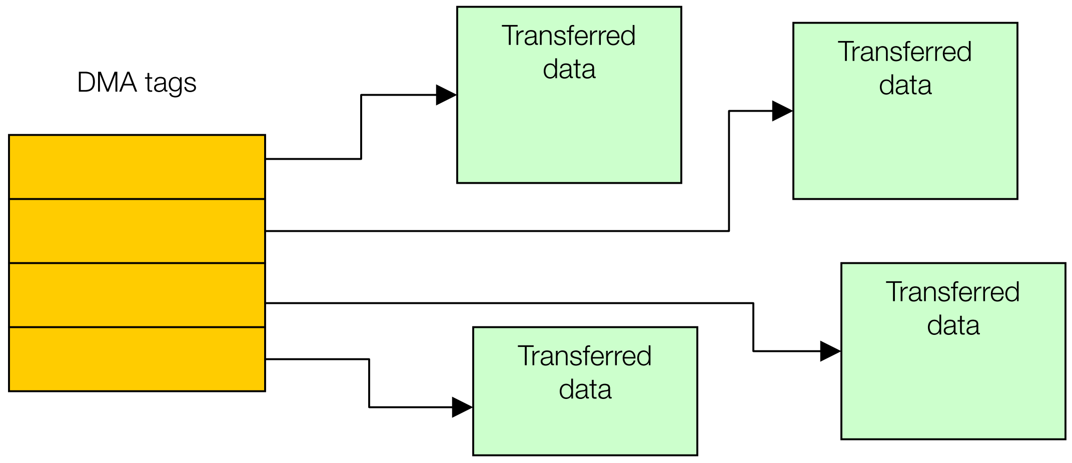
Use non-cached accesses for the following kinds of data.

- Data that is only written
- Data that is read once and never reused
- Data that you want to write to memory immediately

Because of the limited capacity of the data cache, data that meets these conditions is best handled through non-cached accesses, which will avoid dirtying the data cache.

For example, a non-cached access is suitable for the process which sets the DMA tag. Since the DMA tag, once written, is never read, it would be pointless to leave it in the cache. The only additional effort would be the necessary writeback before performing the DMA transfer. Since this amounts to only 2 words, there is no need to worry about the speed of the memory access.

Figure 3: DMA tags written using non-cached accesses



Using Non-Cached Accesses

The EE kernel's memory mapping allocates the three virtual addresses corresponding to cached, uncached, and uncached-accelerated access to the same physical addresses.

Table 1

Cache mode	Virtual address
Cached	0x00100000 ~ 0x01ffffff
Uncached	0x20100000 ~ 0x21ffffff
Uncached-accelerated	0x30100000 ~ 0x31ffffff

As such, accessing an address resulting from the logical OR of a given variable's address and 0x20000000 will cause an uncached access to be performed to fetch that variable. Take a look at the following.

```
int buff[1024];
int *p = (int*)((u_int)buff | 0x20000000);
for (i = 0; i < 1024; i++) {
   p[i] = 0;
}
```

Similarly, accessing an address resulting from the logical OR of a given variable's address and 0x30000000 will cause an uncached-accelerated access to be performed to fetch that variable.

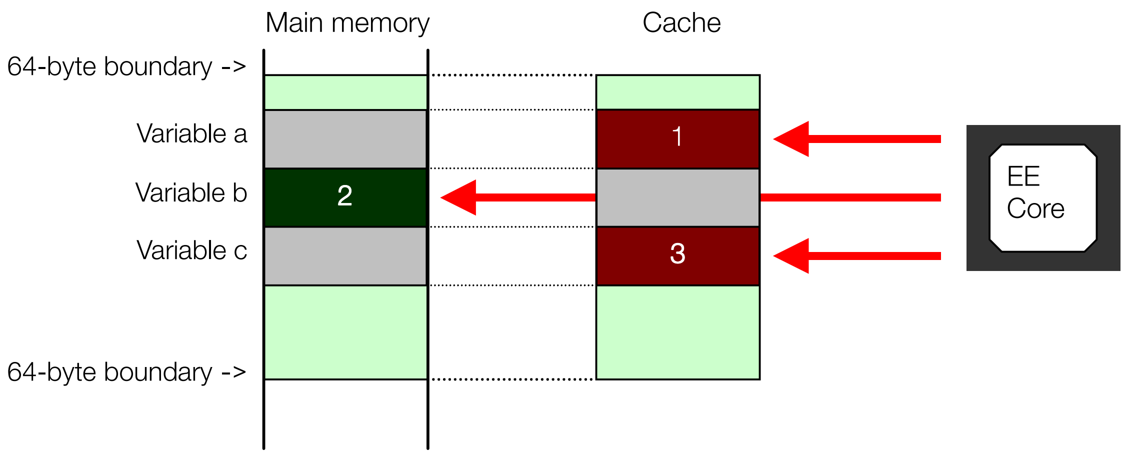
Precautions on Mixing Cached and Non-Cached Accesses

Since the data cache has a line size of 64 bytes, exchanges between it and main memory should be in block increments that are 64-byte aligned. When the same block contains adjacent variables that use both cached and non-cached accesses, the variables will be written back at the same time, which could cause problems. Take a look at the following sample code.

```
int a, b, c;
 (omitted)
   int *ub = (int*)((u_int)&b | 0x2000000);
   a = 1;
   *ub = 2;
   c = 3;
```

Assuming the variables a, b, and c are in the same block, b and c will be loaded into the same cache line when a is accessed. When *ub is written, the contents of the cache will not be changed, but the contents of b in memory will be replaced. Writing to c subsequently will overwrite c in the cache, resulting in the following state.

Figure 4 : Example of a problem arising from a mixed cached/non-cached access



In this state, when the cache is written back, the value written to *ub will be overwritten with the garbage value in the cache. Also, reading in the value for b will read a garbage value in the cache for b. So, writing to *ub will be essentially ignored. To avoid this problem, define variables with attributes as shown below. This ensures that variables using cached and non-cached accesses are divided at 64-byte boundaries.

```
int a __attribute__((aligned(64)));
int b __attribute__((aligned(64)));
int c __attribute__((aligned(64)));
```

Precautions Concerning Write Buffer Latency

Non-cached accesses write data immediately to memory, but you still must consider the effect of the write buffer, which sits between the EE core and main bus.

When a store instruction writes data from the EE core to a device register (which is mapped to the main memory address space) or other similar types of memory, the data always passes through the write buffer. Because data to be written waits in the write buffer when the main bus is busy, there can be a significant lag from the time the EE core executes the store instruction to the time the data actually reaches the device.

To ensure that the write data reaches the device properly, use one of the following two techniques.

- After writing, do an empty read of that register at the same address.
- After writing, execute the SYNC.L instruction. This would be coded as follows:

```
    __asm__  __volatile__(" sync.l");
```

Precautions Concerning Uncached-Accelerated Accesses

The UCAB (uncached acceleration buffer), which is used for uncached-accelerated memory accesses, is invalidated in each of the following cases:

- Load operations that do not hit in the UCAB
- Store operations
- SYNC or SYNC.L instructions
- Exceptions

Note that in situations where the UCAB will be invalidated frequently, uncached-accelerated accesses are actually slower than plain uncached accesses.

Effective Use of the Instruction Cache (I$)

Characteristics of the Instruction Cache

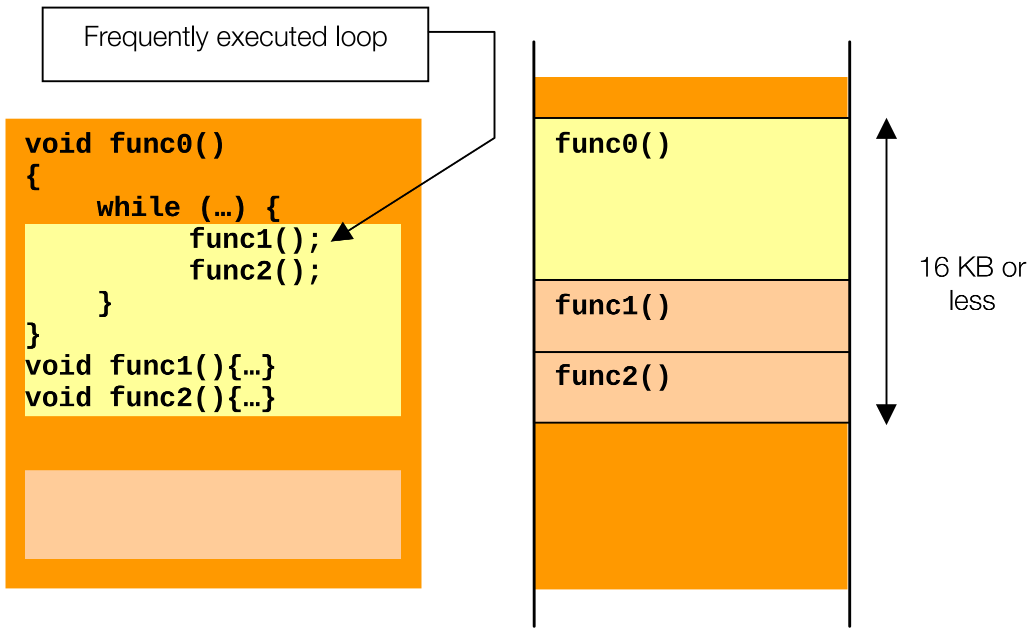
The instruction cache provides 16 KB of buffer memory, and is used when reading program code from memory. Program code, such as read-only instruction data, and loops and functions that get called repeatedly, is stored in the instruction cache, allowing comparatively faster execution.

Like the data cache, the instruction cache provides benefits more or less automatically, although programs can be made to run faster by minimizing instruction cache misses.

How to Reduce Instruction Cache Misses

Unfortunately, the techniques for reducing instruction cache misses are not as clear-cut as they are for the data cache. Two techniques are appropriate for use with loops that are executed frequently and functions which are called from inside those loops. These techniques are to reduce the size of the code and place it at addresses that are sufficiently close together so as to fit within the 16 KB instruction cache. This permits the entire loop to be executed within the cache.

Figure 5: Reducing the size of the loop and functions called from within the loop



```
void func0()
{
    while (…) {
          func1();
          func2();
    }
}
void func1(){…}
void func2(){…}
```

You can use the following techniques to reduce your code size:

- Rewrite the program in assembly code
- Don't overuse macros and inline functions
- Try compiling with the -Os option

Macros and inline functions that are called from more than one place can be replaced by normal functions to reduce the code size. Since function calls do incur some overhead, and can also cause instruction cache misses, you will need to determine whether these replacements are really an improvement on a case-by-case basis. Likewise with the -Os compiler option, sometimes this will tend to actually slow the execution speed of the code itself, rather than reduce the code size, so again, you will need to analyze this on a case-by-case basis.

The following techniques can be used to adjust the placement of addresses for each routine.

- Specify the order of object files when linking
- Modify the link script file (*.cmd)

Either of these can be used to arrange routines at the object level. While it is a typical approach to encapsulate each function as an object, you may find that you need to arrange things differently, depending on the frequency of execution and function calls.

Precautions on Functions Called Implicitly by the Compiler

When minimizing functions called from within loops, you must be aware of not only library functions but also functions that are called implicitly by the compiler.

For example, executing the dptopf() function (for conversion from double to float), or the fptosi() function (for conversion from float to int) can cause extra instruction cache misses, in addition to the overhead they cause on their own. In most cases, you can use intelligent coding techniques, such as defining the types of constants and variables appropriately, to improve your code. You can make sure these calls are not being issued from speed-critical sections of code by checking the assembler source and map files.

How to Measure Instruction Cache Misses

The performance counter library (libpc) can be used to measure instruction-cache misses. The following example program illustrates this.

```
int control, count;
control = SCE_PC0_ICACHE_MISS | (SCE_PC_U1);
control |= SCE_PC1_NO_EVENT | (SCE_PC_U0|SCE_PC_S0|SCE_PC_K0| SCE_PC_EXL0);
control |= SCE_PC_CTE;
 (code omitted)
scePcStart( control, 0, 0); // Start measurement
(Routine where cache misses are to be measured)
count = scePcGetCounter0(); // Get measured value
scePcStop();
```

Multimedia Instructions

With its 128-bit wide registers, the EE core supports multimedia instructions that can perform parallel calculations on 16 single bytes of data, 8 short words, or 4 words.

You should consider using multimedia instructions in the following cases:

- When performing integer calculations that are amenable to parallelization, such as image and sound data processing
- When processing data in 128-bit chunks (logical operations, loads, stores)

Thoughtful use of parallelization can result in a several-fold performance improvement. Note, however, that the compiler doesn't support instructions beyond load and store: you will need to write your code in assembly. This may require you to change your algorithms and data structures.

A useful reference work on parallel computation based on Intel's MMX instruction set is "Optimization Techniques for MMX Technology," by Eiichi Kowashi (ASCII Press, 1997).

128-bit Integer Declarations

128-bit integer variables are declared as follows:

```
long128  a;       // Signed 128-bit integer
u_long128  a; // Unsigned 128-bit integer
```

long128 and u_long128 are defined as follows in eetypes.h:

```
typedef int long128 __attribute__ ((mode (TI)));
typedef unsigned int u_long128 __attribute__ ((mode (TI)));
```

long128 and u_long128 are guaranteed to be aligned on a 128-bit boundary.

Assignments involving either type are coded the same as normal types. The multimedia instructions lq/sq are generated during the compilation process. Operations other than assignments, such as arithmetic operations, cannot be used and will result in a compilation error.

Characteristics of the Compiler

double and float

The EE core provides hardware support only for single-precision floating-point calculations; double-precision floating-point calculations are handled through software. In terms of execution speed, there is an overwhelming advantage to limiting your floating-point calculations to single-precision.

However, because the C language uses double-precision floating-point (double) by default, if you don't make a conscious effort, your programs will be littered with double-precision floating-point arithmetic, resulting in an unforeseen performance hit. You should be especially careful with constants. Take a look at the following:

```
float addone(float a)
{
  return a + 1.0;
}
```

In this example, "1.0" is interpreted as double, so the argument a is first converted through the function fptodp() to double, then the two double values are added using the function dpadd(), and the results converted back to float using the function dptofp(). That's a huge amount of wasted processor time: two unnecessary type conversions, slow execution because the addition is performed in software, and wasted use of the instruction cache. To avoid all that, add "f" to the constant to explicitly define it as float:

```
float addone(float a)
{
  return a + 1.0f;
}
```

Alternatively, you could use a cast such as "(float) 1.0" or specify the -fshort-double option.

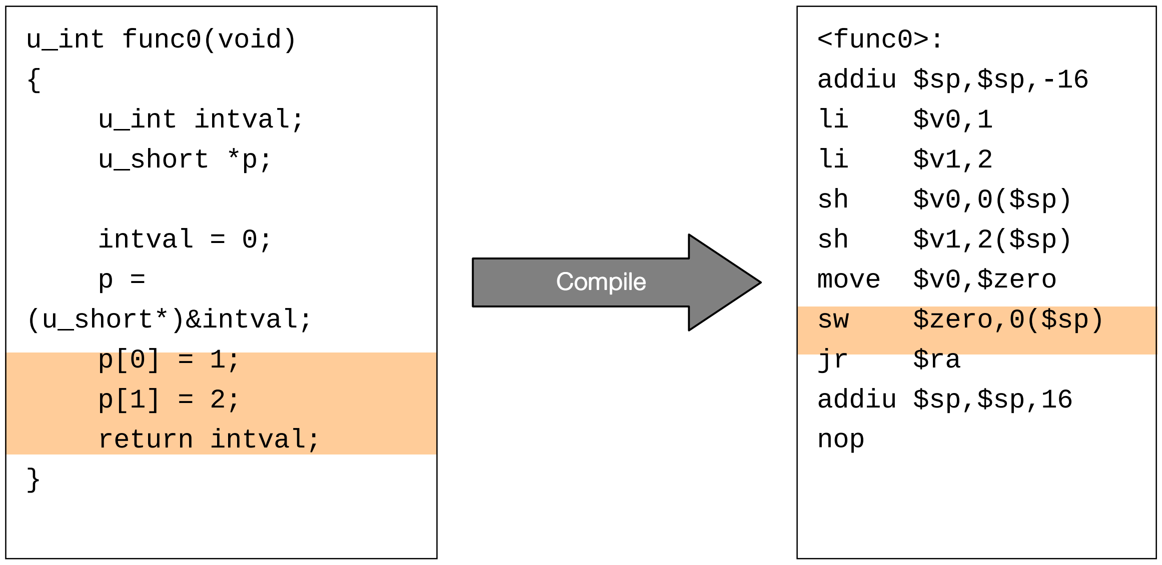
Using the -fshort-double option with ee-gcc defines the size of double to be the same as float, so using the constant "1.0" without a suffix will cause it to be handled as a single-precision number, just like float. However, when linking object files that were compiled using this option with object files that weren't, be careful not to use double for data that is exchanged between the objects, e.g. global variables, and parameters and return values of functions.

Pay particular attention to the use of standard libraries. For example, when passing a floating-point parameter to printf(), it will always be converted to double, but if you use the -fshort-double option, it will be passed as single-precision. But, if you don't compile with that option, you should expect it to be passed as a double-precision number, and you may not get the correct result.

Data Access with Different Pointer Types

Depending on the variable, when different pointer types are used to access the same data, results may be different than anticipated. Take a look at the following:

Figure 6



```
u_int func0(void)
{
     u_int intval;
     u_short *p;

     intval = 0;
     p =
(u_short*)&intval;
     p[0] = 1;
     p[1] = 2;
     return intval;
}
```

```
<func0>:
addiu $sp,$sp,-16
li    $v0,1
li    $v1,2
sh    $v0,0($sp)
sh    $v1,2($sp)
move  $v0,$zero
sw    $zero,0($sp)
jr    $ra
addiu $sp,$sp,16
nop
```

0x00020001 should be returned… …but… …0 is returned instead

The value is assigned to the variable intval through the pointer p, so the value you'd expect to be returned by func0 is 0x00020001, but in compilation, the assignment order is reversed in the assembly code, so 0 is returned.

This is not a compiler bug. The possibility that the compiler can determine whether or not the entity indicated by the pointer is the same as another variable is in fact, considered to be generally impossible, so the C language specification restricts the combinations of types that can be used when accessing a variable. In the code shown above, this restriction has been violated, inevitably resulting in unexpected code.

When it is necessary to access a given variable through a different type, the correct technique is to define a union as shown here. Otherwise, you can use the -fno-strict-aliasing option, but that will not provide adequate optimization.

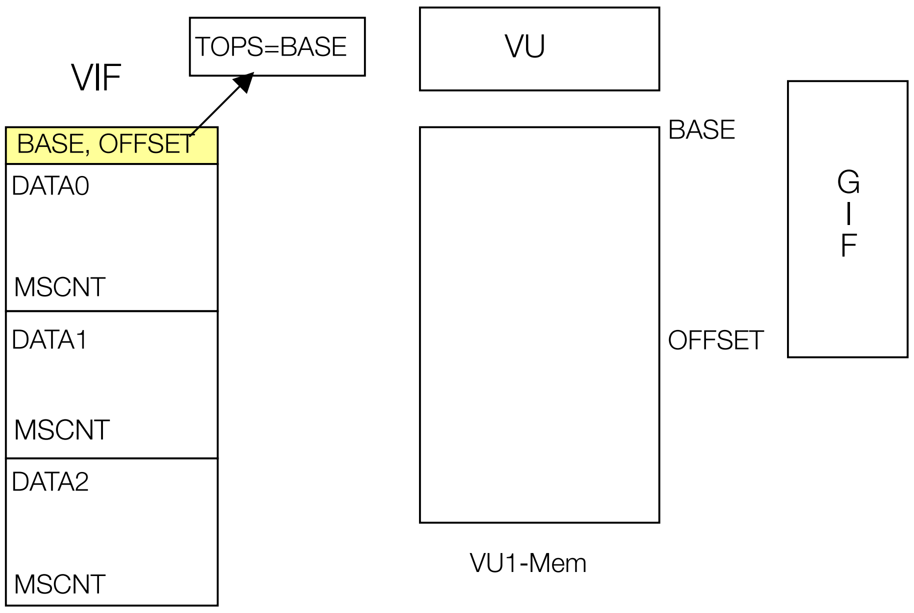
```
u_int func1(void)
{
  union {
          u_int intval;
          u_short p[2];
  } u;

  u.intval = 0;
  u.p[0] = 1;
  u.p[1] = 2;
  return u.intval;
}
```

Using VU1

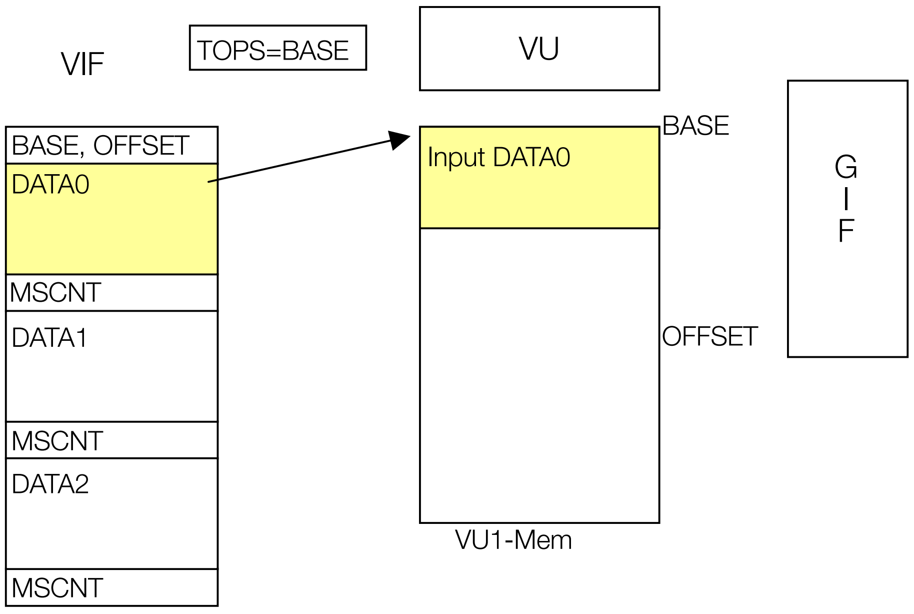
The Basic VU1 Processing Model

VU1 processing involves using VU memory as a double buffer, with the general idea being that reading data from the VIF, performing calculations, and sending data to the GIF can all be done in parallel. First, the basic model will be described.

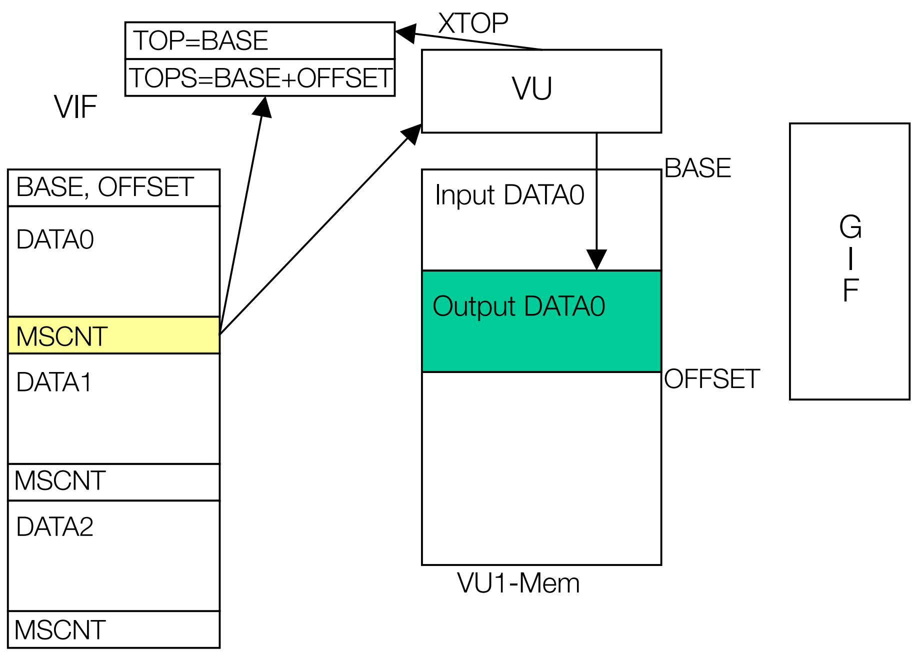
1. Set up the double buffer



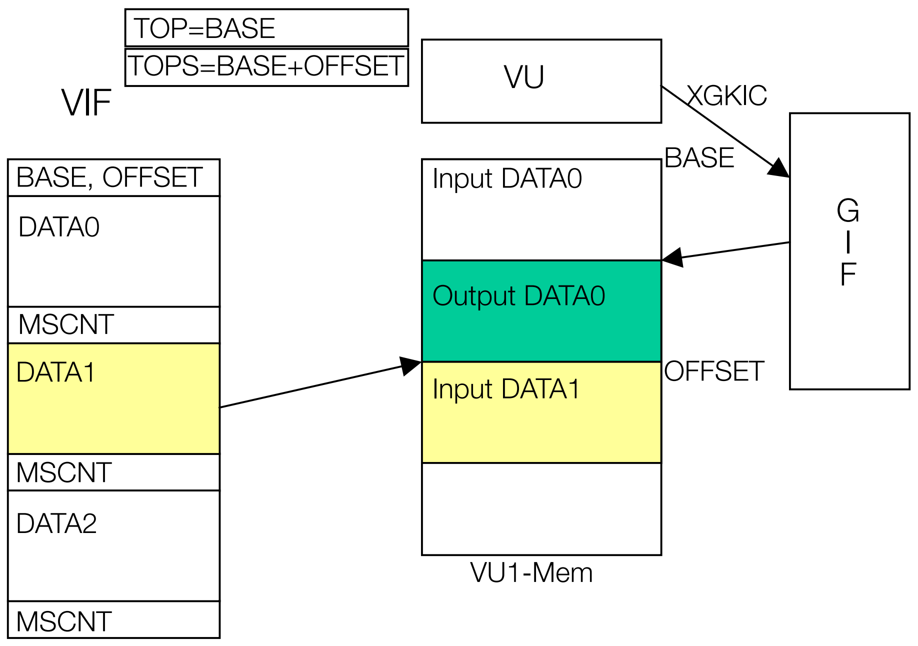
2. Fetch data for buffer 0



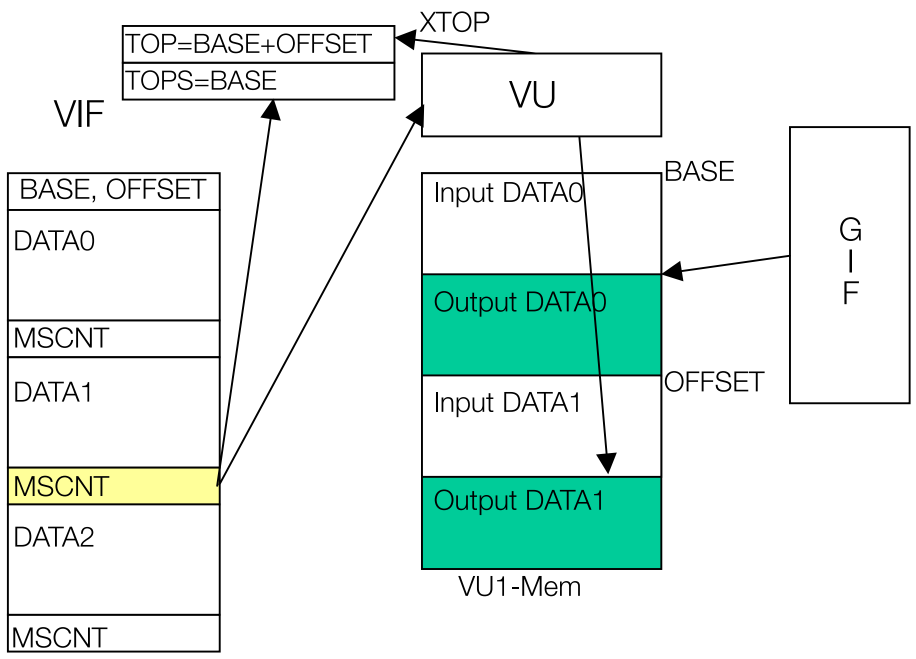
3. Perform calculations for buffer 0



4. Swap double buffers



5. Perform calculations for buffer 1



Tuning Microprograms

The conventional way to speed up VU execution is to optimize microprogram loops. Take a look at the following:

Sample program before tuning:

```
LOOP1:    NOP                             LQI.xyzw VF30, (VI06++)
          MULAx.xyzw ACC, VF04, VF30x     NOP
          MADDAy.xyzw ACC, VF05, VF30y    NOP
          MADDAz.xyzw ACC, VF06, VF30z    NOP
          MADDw.xyzw VF31, VF07, VF30w    NOP
          NOP                             DIV Q, VF00w, VF31w
          MULq.xyzw VF28, VF31, Q         NOP
          FTOI4.xyzw VF27, VF28           NOP
          NOP                             IADDI VI11, VI11, -1
          NOP                             SQ.xyzw VF27, 0(VI12)
          NOP                             IBNE VI11, VI00, LOOP1
          NOP                             IADDI VI12, VI12, 2
```

This sample program shows only the substantive processes, but because of stalls induced by resource hazards and data hazards, one iteration through the loop takes 29 cycles.

Sample program after tuning:

```
LOOP:     NOP                             DIV Q, VF00w, VF31w
          MULAx.xyzw ACC, VF04, VF30x     SQ.xyzw VF27, 0(VI12)
          MADDAy.xyzw ACC, VF05, VF30y    NOP
          MADDAz.xyzw ACC, VF06, VF30z    IADDI VI12, VI12, 2
          MADDw.xyzw VF31, VF07, VF30w    IADDI VI11, VI11, -1
          FTOI4.xyzw VF27, VF28           LQI.xyzw VF30, (VI06++)
          NOP                             IBNE VI11, VI00, LOOP1
          MULq.xyzw VF28, VF31, Q         NOP
```

Thanks to tuning, one iteration of the loop has been shortened by up to 8 cycles. Let's go over the optimization process in more detail.

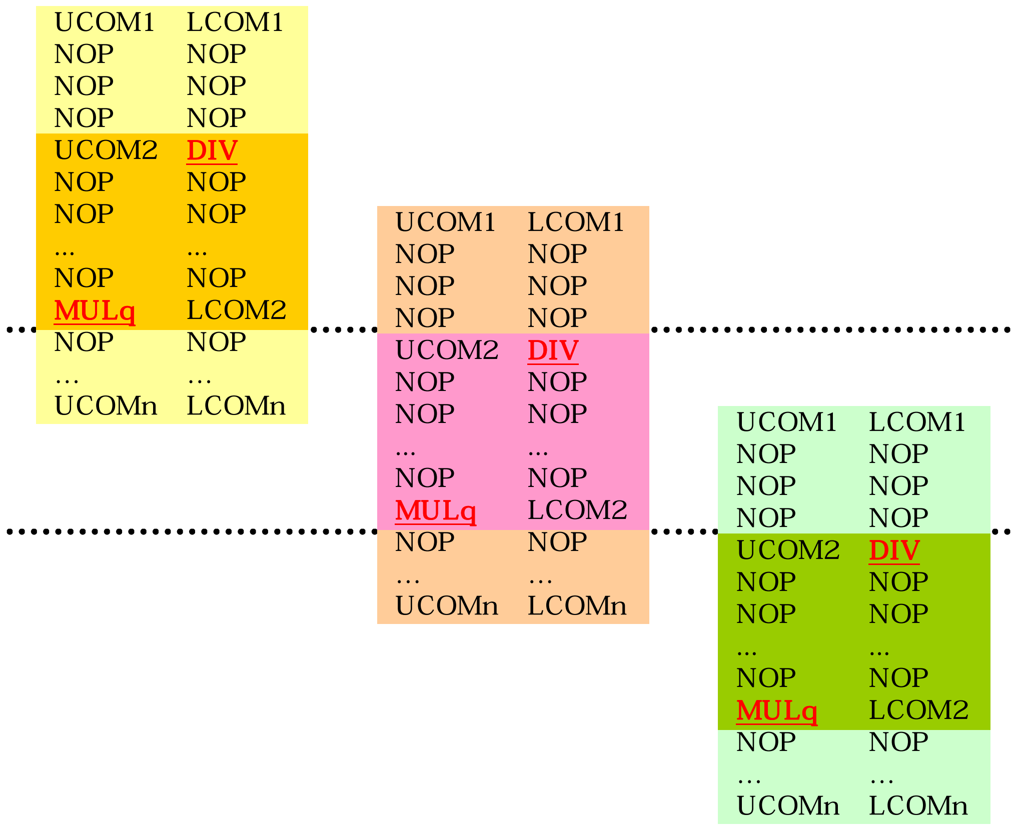
1. For loop processes, write out hazards explicitly using NOPs

2. Look for high-latency processes within the loop

```
LOOP:   UCOM1       LCOM1
        NOP         NOP
        NOP         NOP
        NOP         NOP
        UCOM2       DIV
        NOP         NOP
        NOP         NOP
        ...         ...
        NOP         NOP
        MULq        LCOM2
        NOP         NOP
        ...         ...
        UCOMn       LCOMn
```

```
LOOP:   NOP         LQI...
        NOP         NOP
        NOP         NOP
        NOP         NOP
        MULAx...    NOP
        MADDAy...   NOP
        MADDAz...   NOP
        MADDAw...   NOP
        NOP         NOP
        NOP         NOP
        NOP         NOP
        NOP         DIV
        NOP         NOP
        NOP         NOP
        NOP         NOP
        NOP         NOP
        NOP         NOP
        NOP         NOP
        MULq        NOP
        NOP         NOP
        NOP         NOP
        NOP         NOP
        FTOI...     NOP
        NOP         NOP
        NOP         NOP
        NOP         IADDI...
        NOP         SQ...
        NOP         IBNE...
        NOP         IADDI...
```

3. Arrange high-latency processes so they don't overlap one another



```
UCOM1   LCOM1
NOP     NOP
NOP     NOP
NOP     NOP
UCOM2   DIV
NOP     NOP
NOP     NOP
...     ...
NOP     NOP
MULq    LCOM2
NOP     NOP
…       …
UCOMn   LCOMn
```

```
UCOM1   LCOM1
NOP     NOP
NOP     NOP
NOP     NOP
UCOM2   DIV
NOP     NOP
NOP     NOP
...     ...
NOP     NOP
MULq    LCOM2
NOP     NOP
…       …
UCOMn   LCOMn
```

```
UCOM1   LCOM1
NOP     NOP
NOP     NOP
NOP     NOP
UCOM2   DIV
NOP     NOP
NOP     NOP
...     ...
NOP     NOP
MULq    LCOM2
NOP     NOP
…       …
UCOMn   LCOMn
```

4. Overlap in order from the left

If instructions other than NOP are colliding with each other, move them down the left side.

```
UCOM1       LCOM1
NOP         NOP
NOP         NOP
NOP         NOP
UCOM2       DIV
NOP         NOP
UCOM1       LCOM1
NOP         NOP
NOP         NOP
NOP         NOP
MULq        LCOM2
UCOM2       DIV
NOP         NOP
UCOM1       LCOM1
NOP         NOP
NOP         NOP
NOP         NOP
MULq        LCOM2
UCOM2       DIV
UCOMn       LCOMn
UCOM1       LCOM1
NOP         NOP
NOP         NOP
NOP         NOP
MULq        LCOM2
…           …
```

5. **Include the last instruction on the left and the first instruction on the right as delimiters for the loop block**

6. **Add conditional-branching instructions to the loop block**

```
        UCOM1    LCOM1
        ...      ...
        NOP      NOP
        MULq     LCOM2
LOOP:   UCOM2    DIV
        UCOMn    LCOMn
        UCOM1    LCOM1
        …        …
        NOP      IBNE LOOP
        MULq     LCOM2
        …        …
```

7. Eliminate register duplication

Data Tuning

One effective way to speed up VU1 processing is to optimize the data you pass to VU1. Specifically, this means:

- As much as possible, avoid independent triangles, and replace them with triangle strips
- Perform the same processes together

However, these techniques won't be effective for certain data sizes. You must also consider the GS bottleneck.

Precautions Related to VU Programming

This chapter discusses some of the most frequently encountered problems with VU programming.

Dest Field Collisions

Suppose that both Upper and Lower instructions write to the same register during the same cycle, as shown below. Even if the Dest fields are different, this sequence won't work.

```
MULw.xy  VF01,VF02,VF00w  MOVE.zw VF01,VF03
```

The effect is that the values that get written to each of the fields in the VF01 register are indeterminate. In most situations, the field specified in the Lower instruction is written with the calculation result from the field used by the Upper instruction, though the reverse is also possible.

Collisions Between Integer Calculations and Branch Instructions

There is no hazard checking between the IALU, which handles integer arithmetic, and the BRU, which controls branch instructions. This means that the results of an integer calculation will not be available to an immediately following branch instruction. It is common practice to use a loop with a counter that decrements to zero, but as the following example shows, one extra cycle needs to be added.

```
NOP               IADDI   VI01, VI01, -1
NOP               NOP
NOP               IBNE    VI01, VI00, LOOP
```

XGKICK Process Termination

XGKICK is an instruction that initiates data transfer to the GIF. The XGKICK instruction itself terminates before the data transfer completes. This means that if you overwrite the data buffer after the XGKICK instruction, you may prevent data from reaching the GIF.

To make sure the data transfer has completed, you should execute the XGKICK, FLASH or FLASHA instructions.

Memory Access Collisions

No hazard checking is performed on accesses to VU memory. You should insert one empty cycle as shown below when reading a value immediately after it is written.

```
  NOP             BAL     VI15,   FUNC
  NOP             NOP
  NOP             LQ      VF04, 0(VI03)
  …               …

FUNC:     …               …
  NOP             JR      VI15
  NOP             SQ      VF01, 0(VI03)
```

Memory Access from the EE Core

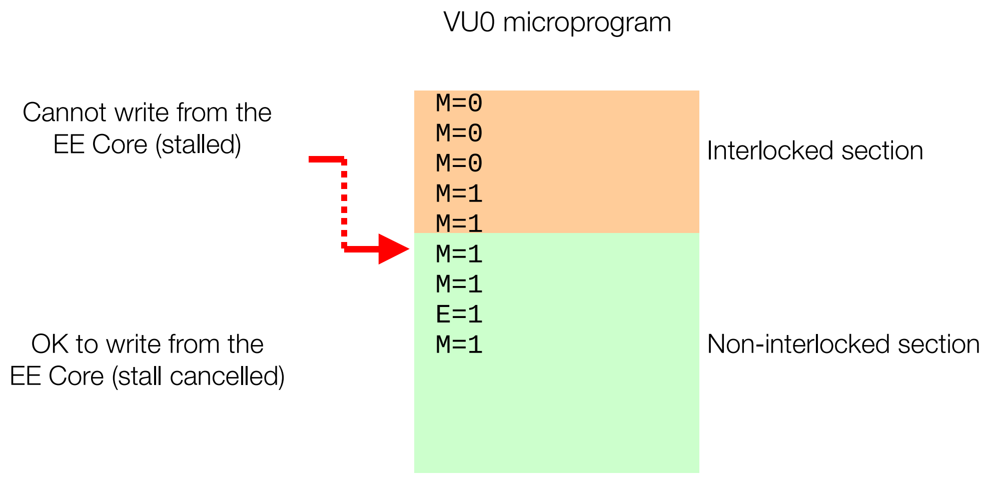
Although the EE core can directly access the VU's data memory (VU mem) because it is mapped to main memory address space, this function was really included for debugging purposes, and proper operation is not guaranteed unless both the VU and VIF are stopped.

One situation you need to be especially careful of is when data from the EE core is written to the VU mem with a store instruction, and is immediately followed by a VU macroinstruction that uses that value. Because the store instruction performs the write via the write buffer, when the following VU macroinstruction is executed, the write operation may not yet have completed. Furthermore, accessing VU mem while the VU is in the middle of an operation can result in a malfunction. When you need to perform this kind of processing, insert a SYNC.L instruction between the store and VU macroinstruction.

Interlock Using the m-bit

While it is possible to use the m-bit as an interlock to synchronize VU0 with the EE core, the only process that can guarantee synchronization is shown below, where the interlocked state is first canceled by a microprogram that is in progress.

Figure 7


VU0 microprogram
Cannot write from the EE Core (stalled)
OK to write from the EE Core (stall cancelled)
M=0
M=0
M=0
M=1
M=1
M=1
M=1
E=1
M=1
Interlocked section
Non-interlocked section

Once the m-bit is set to 1, then it must be set to 1 in all subsequent instructions. In other words, it isn't possible to synchronize more than once in the same microsubroutine.

Referencing the flag Register in Macro Mode

Suppose that a branch is performed in VU0 macro mode, and that a cfc2 instruction is used to copy a flag such as the MAC flag to an EE core register, after which an EE core branch instruction makes a branch decision. In this situation, you need to insert empty cycles between the arithmetic instruction and the cfc2 instruction, equivalent to the latency of the calculation. Be careful with the instructions you insert for the empty cycles.

```
vsuby.x vf13x, vf12x, vf15y
vnop
vnop
vnop
vnop
vnop
cfc2 $11, $vi17
```

While there's no problem using vnop instructions, it is dangerous to insert other arithmetic instructions.

```
vsuby.x vf13x, vf12x, vf15y
vnop
vnop
vnop
vnop
vaddy.x vf18x, vf12x, vf15y
cfc2 $11, $vi17
```

In the example above, a vaddy instruction is inserted before the cfc2 instruction, however, if an instruction cache miss occurs when attempting to execute the cfc2 instruction, the EE core will stall but VU0 will continue processing the vaddy instruction. This could result in the cfc2 instruction reading the MAC flag from the vaddy instruction.

Precautions when Using VCALLMS & VCALLMSR

When using VCALLMS or VCALLMSR to invoke a VU0 microsubroutine, please be aware of the following restrictions:

- The last instruction of a microsubroutine (the instruction following Ebit = 1), will not interlock because of resource hazards, data hazards, or synchronization instructions.
- If an interlock might occur during the last microinstruction (the instruction following Ebit = 1), then in the EE Core program, interpose a microinstruction after the VCALLMS/VCALLMSR instruction or a COP2 transfer instruction that interlocks, before executing the next VCALLMS/VCALLMSR instruction.

If you fail to observe these restrictions, your microinstructions may get killed if any of the following occur:

- Following a VCALLMS or VCALLMSR instruction, a stall occurs while waiting for a microsubroutine activated by the preceding VCALLMS/VCALLMSR instruction to terminate.
- The final microinstruction of a microsubroutine (the instruction following Ebit = 1) that is activated by the preceding VCALLMS/VCALLMSR instruction stalls with a resource or similar hazard.
- The following VCALLMS/VCALLMSR instruction has proceeded to the W stage because an instruction was executed that has Ebit = 1.
- An exception is detected in the W stage, causing not only the following VCALLMS/VCALLMSR instruction but also the stalled microinstruction from (2) to be killed.

GS-related Tuning

Avoiding Page Breaks

GS local memory is organized into 2048-word pages. Accesses within a given page can be handled quickly, but accessing different pages can be slow. This page-break penalty is fairly significant, so you need to keep page organization in mind in order to optimize performance.

Roughly speaking, one page corresponds to a rectangle which divides the screen vertically and horizontally. Since the page size is fixed at 2048 words, the size of the rectangle (no. of pixels / no. of texels) varies as a function of the data format (number of bits per pixel / texel).

Table 2

Pixel / texel / Z-value format	No. of pixels / no. of texels per page
PSMCT32 / PSMCT24 PSMT8H / PSMT4HH / PSMT4HL PSMZ32 / PSMZ24	64×32
PSMCT16 / PSMCT16S PSMZ16 / PSMZ16S	64×64
PSMCT8	128×64
PSMCT4	128×128

In any case, page breaks can be reduced if individual polygons don't straddle pages (i.e. make them so they don't extend too much on screen). In other words, large polygons can be drawn more quickly if you break them up.

The following are some actual measured values.

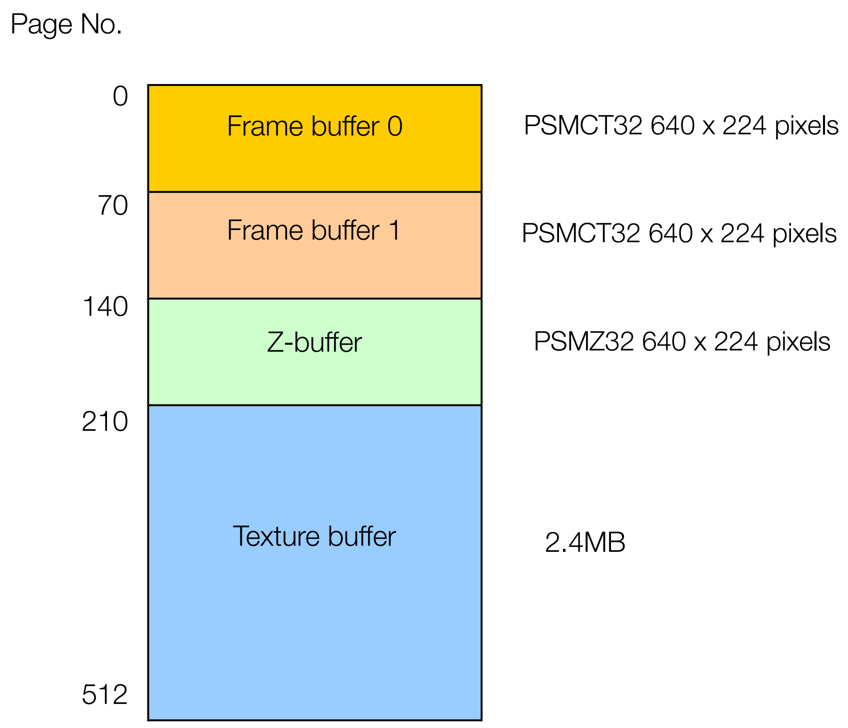
Example 1: Write penalty: 256 x 256, no texture

No divisions	770 Mpixel/sec
Broken into 8 x 8 units	1,050 Mpixel/sec

Example 2: Read penalty: 256 x 256, no texture

Broken into 64 x 64 units	1,100 Mpixel/sec
Broken into 128 x 128 units	560 Mpixel/sec

Figure 8



MIPMAP Usage

"Reduced textures" occur when pasting a large texture onto a small polygon and can cause page breaks when reading the texture data. Consequently, these textures can cause a drop in performance. The following are some actual measured values.

128×128 texture repeat 41,000 polygons

With MIPMAP	4.08(msec)
Without MIPMAP	10.46(msec)

The only way to avoid this penalty is to use MIPMAP.

Texture Transfer Processing

Texture data is loaded into GS local memory using a Host-Local transfer, but the transfer speed will vary as follows, depending on the texture format.

Table 3

Texture format	Transfer speed
RGBA32,RGB24,RGBA16	1,200MB/s
IDTEX8	900MB/s
IDTEX4	600MB/s

This being the case, sometimes it makes sense to transfer 4-bit or 8-bit textures as if they were 32-bit. Because 4-bit and 8-bit textures have a different texel ordering than 32-bit textures, you will need to reorder them. This is easily done by transferring them once to GS local memory as 4 or 8-bit textures, then reading that memory image as a 32-bit texture.

The Temporary CLUT Buffer

A common trick is to use index colors to limit texture size. Because the texture and CLUT need to be referenced when drawing the screen, there is another technique of caching the CLUT in a temporary buffer to reduce local-memory access. With a temporary buffer size of 32 bits x 256 (1 KB), the number of CLUT sets that can be placed in the temporary buffer will depend on the color format and texel format as follows.

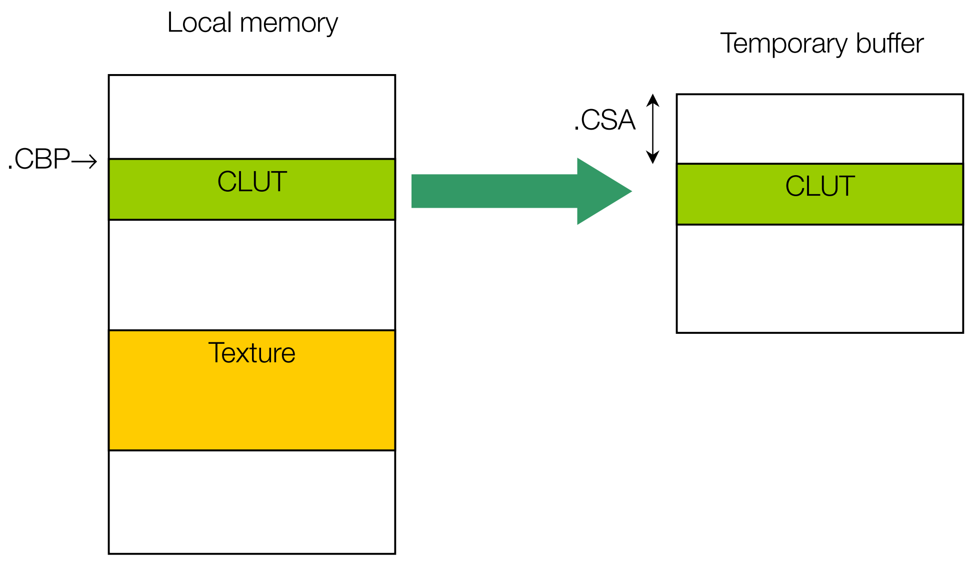
Table 4

CLUT color format	Texel format	Number of CLUT sets in temp buffer
PSMCT32	IDTEX8	1 set
PSMCT32	IDTEX4	16 sets
PSMCT16	IDTEX8	2 sets
PSMCT16	IDTEX4	32 sets

Loading from local memory to the temporary buffer occurs when setting a value in either the TEX0_1/2 register or the TEX2_1/2 register. Whether loading actually takes place or not is controlled by the value of the CLD field.

The value of the CSA field represents what part of the temporary buffer will be loaded, and at the same time, what part will be treated as the CLUT for drawing the screen. When multiple CLUT sets are loaded, the effectiveness of the temporary buffer cache can be improved even more by changing the CSA value accordingly.

Figure 9



Fundamentals of 3D Graphics

Homogeneous Coordinate Systems

Three-dimensional graphics processing doesn't so much involve a normal 3D coordinate system (x, y, z), but more often uses a homogeneous coordinate system (x, y, z, w). A homogeneous coordinate system of (x, y, z, w) (with w non-zero) corresponds to the same point in a 3D coordinate system as $(x/w, y/w, z/w)$. Using a homogeneous coordinate system has the advantage of being able to represent in a unified fashion, every combination of vertex transformation: parallel translation, expansion/contraction, and rotation, with a 4 x 4 matrix.

$$\begin{bmatrix} M_{11} & M_{12} & M_{13} & T_x \\ M_{21} & M_{22} & M_{23} & T_y \\ M_{31} & M_{32} & M_{33} & T_z \\ 0 & 0 & 0 & 1.0 \end{bmatrix} \sim \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix}$$

Because the value of w does not have a conventional meaning, it is handled by normalizing it to 1.0.

Coordinate Systems and Coordinate Conversions

To simplify modeling and rendering, several coordinate systems with different origins and units of length are used, as shown below.

• Local coordinate system:	A coordinate system that is fixed for each object
• World coordinate system:	A coordinate system of the world which contains objects
• View coordinate system:	A coordinate system fixed on a viewpoint (also known as the camera coordinate system)
• Screen coordinate system:	A coordinate system that is fixed on the screen
• GS primitive coordinate system:	A coordinate system for GS input values

The following coordinate conversions can be performed in the drawing process.

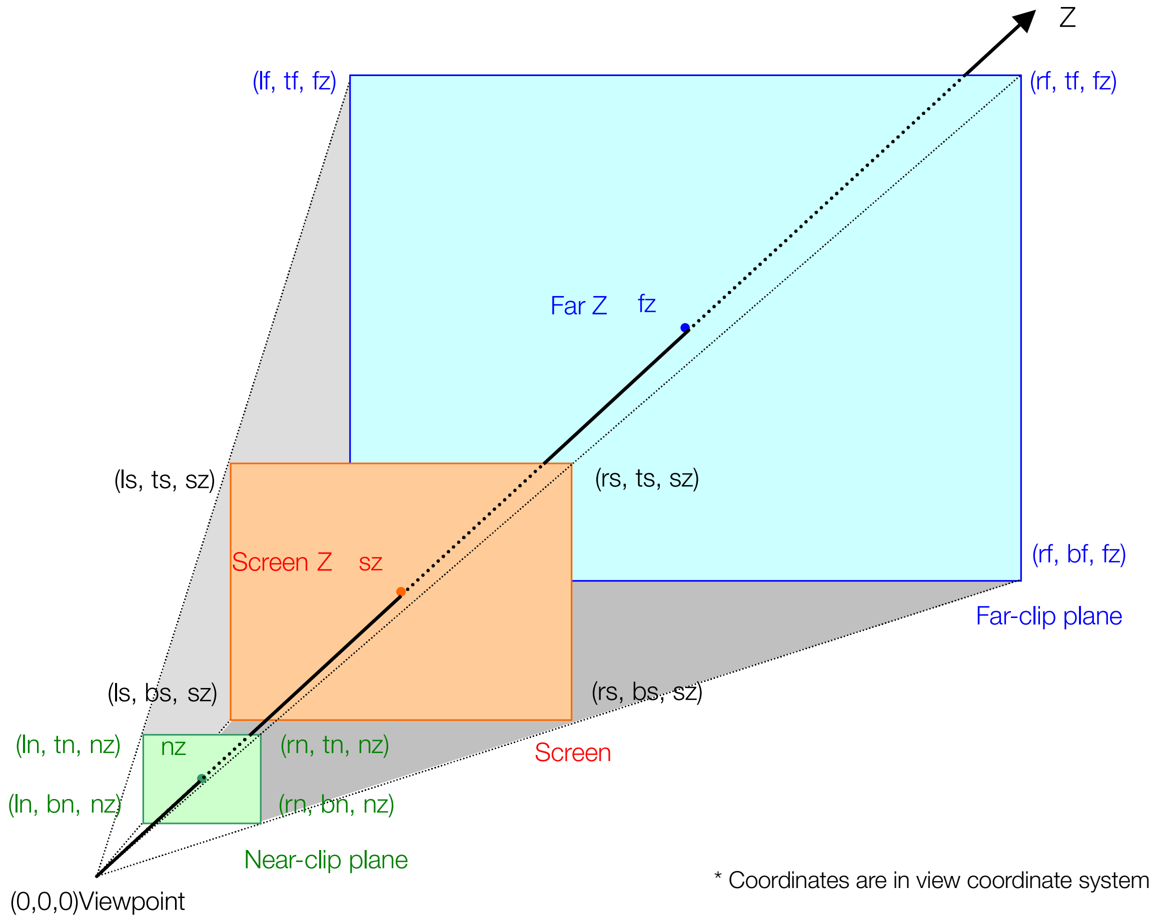
Coordinate conversion flow

```
Local coordinates ( xL , yL , zL, 1.0 )
          ↓ Apply [World/Local] Matrix
World coordinates ( xW , yW , zW, 1.0 )
          ↓ Apply [View/World] Matrix
View coordinate ( xV , yV , zV, 1.0 )
          ↓ View clipping
          ↓ Apply [Screen/View] Matrix
          | DIV  (1/W)
          | Mulq.xyz        (X/W, Y/W, Z/W)
Screen coordinates ( xS , yS , zS, 1.0 )
          ↓ FTOI4
GS primitive coordinates
          | SQ
          ↓ XGKICK
```

View Clipping

The drawing area of the screen is a pyramid with a rectangular base extending to each side of the screen, and with the top of the pyramid being the viewpoint. Some parts of the image are very close to the viewpoint, but other parts are beyond a certain distance and do not need to be drawn. This places a limit on the objects in the pyramid (the view volume) that need to be rendered. The process that excludes objects outside the view volume is known as "view clipping."

Figure 10



Screen/View Conversion and Z-Values

Every point in the view volume is projected onto the screen plane. To maintain information on the correct front-and-back relationships of overlapping objects, the z-coordinate is converted to a Z-value as shown here:

$$Z = z \times \frac{-(\mathrm{Zmax} \times \mathrm{nz} - \mathrm{Zmin} \times \mathrm{fz})}{\mathrm{fz} - \mathrm{nz}} + \frac{\mathrm{Zmax} \times \mathrm{fz} \times \mathrm{nz} - \mathrm{Zmin} \times \mathrm{fz} \times \mathrm{nz}}{\mathrm{fz} - \mathrm{nz}}$$

The Screen/View conversion matrix is as follows:

$$\begin{bmatrix} sZ & 0 & 0 & 0 \\ 0 & sZ & 0 & 0 \\ 0 & 0 & ZA & ZB \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$ZA = -\,(\,Zmax \times nz - Zmin \times fz\,)\,/\,(\,fz - nz\,)$$

$$ZB = (\,Zmax \times fz \times nz - Zmin \times fz \times nz\,)\,/\,(\,fz - nz\,)$$

Compensation for GS Primitives

The screen coordinate system and GS primitive coordinate system have different screen centers and aspect ratios.

- Aspect ratio: 640:224 = 1:0.467
- Center of screen: 2048, 2048

In the interest of performance, compensation parameters that adjust these values are multiplied in the Screen/View conversion matrix beforehand. This matrix, which includes both Screen/View conversion and GS primitive compensation, is handled as the Screen/View conversion matrix.

$$\begin{bmatrix} Ax & 0 & 0 & Ox \\ 0 & Ay & 0 & Oy \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} sZ & 0 & 0 & 0 \\ 0 & sZ & 0 & 0 \\ 0 & 0 & ZA & ZB \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} Ax \times sZ & 0 & 0 & Ox \\ 0 & Ay \times sZ & Oy & 0 \\ 0 & 0 & ZA & ZB \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Texture Coordinates and Perspective Compensation

Perspective transformation from the view coordinate system to the screen coordinate system makes near objects appear larger. Likewise, if textures are not similarly distorted when pasted, objects will look unnatural. Therefore, the texel coordinates $(\,u, v\,)$ should be compensated using the divisor W for perspective transformation. The compensated coordinates are normalized texel coordinates $(\,S, T, Q\,)$.

```
Texel coordinates ( u, v )
          ↓ TW, TH (texture width and height) normalized to 1.0
( U, V, 1.0 )
          | DIV  (1/W)
          | Mulq.xyz        (U/W, V/W, 1/W)
( S, T, Q )
          | SQ.xyz
          ↓ XGKICK
```

Z-Buffer

The Z-buffer provides an efficient means of eliminating hidden surfaces, i.e. determing the front-and-back relationships of overlapping objects, then enabling the screen to be drawn such that areas that should be visible are drawn, and those that should be invisible are not.

The Z-buffer holds the Z-value of every pixel, as well as a frame buffer that represents all the pixels on the screen. The polygon rendering process will request the color of each pixel (RBG value), plus its Z-value. Then, before the pixel's RBG value is written to the frame buffer, its Z-value is compared to the Z-value in the Z-buffer, and if the newly computed Z-value is greater than the one in the Z-buffer, the newly computed polygon is judged to be closer than the polygon already drawn at that pixel, and the newly-computed RGB value will be written to the frame buffer while the corresponding Z-value will be written to the Z-buffer.

Conversely, if the newly computed Z-value is smaller, its polygon is judged to be farther than the polygon already drawn at that pixel, meaning it is not visible, so the RGB value present in the frame buffer and the Z-value present in the Z-buffer will not be updated.

This technique is generally used for determining the visibility of each polygon on a pixel-by-pixel basis before drawing the polygons.

One advantage of using the Z-buffer is that it permits you to eliminate hidden surfaces regardless of the order in which polygons are drawn, so you can draw polygons in whatever order is most appropriate for your program. Also, because the analysis is done on a pixel-by-pixel basis, you don't need to take into account situations such as whether one polygon passes through another.

Note, however, that for alpha-blending, you need to sort in order of Z-value and draw from the farthest object forward, so it may not always be possible to have all processing done by the Z-buffer. Also, Z-values and z-coordinates (view coordinates) have a non-linear relationship, so the precision of Z-values decreases as the distance increases. In some cases, it may not be possible to accurately determine the distance of distant objects.

Miscellaneous

Specifying Sections

By adding a "section" attribute to a variable definition, you can specify the data section. For example, to place something in the data section, do the following:

```
unsigned char data[100] __attribute__((section (".data")));
```

Upper Limit on DMA Transfer Size

The maximum size that can be transferred at once using DMA in normal mode, and the maximum size of data that can be specified with one DMA tag, is up to but not including 1 MB. Because the QWC field of the DMA tag is 16 bits, this is not an obvious limit, but if you accidentally exceed this limit, you may encounter an error which will show up as a "data too big to run" phenomenon.

To transfer 1 MB or more using DMA, break the data into blocks of less than 1 MB, attach a DMA tag to each block, and use chain mode.