VM LABS

NUON™ Troubleshooting Guide

Oct. 19, 2001

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Copyright notice

Contents

1. Compile Time Problems

1.1 Building Programs

1.1.1 What flags should I use?

The C compiler generally requires the `-O` flag in order to generate good code, and also to do the code analysis necessary for producing effective warnings. So always use `-O` (or a higher level of optimization, such as `-O2` or, even better, `-Os`, which is the same as `-O2` but optimizes for code size rather than number of instructions).

During development, it makes sense to use the lowest reasonable level of optimization (`-O`) to make code generation faster. You should also use `-g` to get debugging information into the code, and `-Wall` to produce the maximum level of warnings from the compiler – this will help to catch errors such as uninitialized variables, undeclared functions, and so forth. Be sure to correct warnings about undefined prototypes for any functions that take a variable number of arguments (for example, `printf`), because `mgcc` cannot correctly call such a varargs function unless a prototype is in scope. For example, to provide a prototype for `printf`, make sure you include the `stdio.h` header file.

For production code, it's worthwhile to bump the optimization level up a bit to `-Os`, and to add assembler optimization with `-mreopt`. Note that the `-mreopt` option makes some cache bugs (section 2.2.1) more likely to occur, so be sure that all variables in local RAM (as opposed to those accessed via the cache) are read with the macro `_GetLocalVar` from the `libmutil` package.

Performance on the NUON chip is often dominated by the cache, so it's usually best to optimize for space (`-Os`). However, for small functions which are called a lot, it may make sense to optimize for number of cycles, even if this creates bigger code. In this case, `-O3` and `-mreopt` will be good. If debugging is no longer a concern, `-fomit-frame-pointer` will also produce smaller and faster code, but it will be very difficult to debug.

1.1.2 What do -mreopt and -mreopt-more do?

These `mgcc` flags instruct the assembler to do instruction packing and re-arranging. This can significantly improve the generated code, but has the drawback that debugging becomes very difficult (because instructions have been extensively re-arranged). It also makes a cache bug in the beta hardware (section 2.2.1) more likely to occur unless you've been careful to use the `_GetLocalVar` macro for all local memory references.

`-mreopt` corresponds to the LLAMA flag `-O`, and `-mreopt-more` corresponds to the LLAMA flag `-O2`. `-mreopt-more` is generally much slower than `-mreopt`, and gives only a marginal improvement in the resulting code, so it probably isn't worth using – just stick with `-mreopt` unless you're trying to squeeze out every possible cycle.

1.1.3 How can I change the address where my C program runs?

The linker's `-B` flag allows you to set the default load address. If you are invoking the linker directly from the makefile, you can pass an argument like `-B=0x80200000` to load your program at offset 2MB to the start of system RAM. If the linker is being invoked indirectly through `mgcc`, you'll have to use its `-Xlinker` option, *e.g.* `-Xlinker -B=0x80200000`.

1.1.4 How should I track down compiler problems?

If you're getting some weird compile time problem that isn't covered above, try giving the `-v` (for verbose) flag to `mgcc`. This will cause it to print a detailed listing of what's going on. This is often helpful in diagnosing a problem.

1.2 Code Generation

1.2.1 How can I see the assembly language generated by the compiler?

Use the `-S` switch to generate an assembly language file instead of a COFF object file. For example, use:

```
mgcc -O -Wall -o foo.s -S foo.c
```

to generate the assembly language code for `foo.c`.

Note that the effect of using the `-mreopt` flags will not be shown, since `-S` only runs the compiler, not the assembler, and it is the assembler that does `-mreopt`. To see the effect of `-mreopt`, try:

```
mgcc -O -Wall -o foo.s -S foo.c
llama -fasm -O -c -b -o foo.opt foo.s
```

1.2.2 Why is the generated code so big and slow?

By default, the C compiler does no optimization at all. Needless to say, this means that the code it generates is big and slow. You should always use the `-O` flag to the compiler – this will dramatically reduce the size of the generated code. See "What flags should I use?" (section 1.1.1).

Also, you should remember that a C compiler will never generate code as good as that produced by a good human programmer. We're continuing to work on improvements for the compiler, but for the really time critical inner loops you may want to write the code in assembly language by hand.

1.3 Warning and Error Messages

1.3.1 Unable to find previous instruction packet for padding

This warning message from the assembler is harmless. Some instructions must be aligned in particular ways; for example, no instruction can cross a cache line boundary. The assembler forces alignment by inserting padding into instruction packets. This padding uses space, but does not take any time to execute. The operation of padding is normally transparent, but there are some times when the assembler needs to insert padding to force alignment but is unable to find a packet to insert the padding into. For example, this can happen if some data has been inserted in the middle of code. In these circumstances, the assembler is forced to insert a `nop` instruction. The warning informs the user that this has happened. It's useful for an assembly language programmer, since it can be a problem in a carefully crafted inner loop; but the C programmer can safely ignore this message, and indeed future versions of the LLAMA assembler will not output it when run on compiler generated code.

1.3.2 Cache stall may cause repeated read/write to register

This is a warning about a bug in the beta hardware (section 2.2.2) that can cause problems with accesses to certain volatile registers. If the instruction that causes this warning is in a branch delay slot, move it out of the delay slot. If it is in a large packet, try moving it to a smaller packet or make it an instruction all on its own. If all else fails, insert one or two `nop` instructions before the offending instruction.

1.3.3 Obsolete instruction form

The syntax for the `addr` changed in order to accomodate some new instruction semantics which became possible late in the design of the chip. The old syntax took

```
addr #1,rx
```

to mean "add 1 in 16.16 fixed point format to `rx`". However, it is in fact possible to add an arbitrary 32 bit constant using `addr`, and so an ambiguity arose; how could we specify adding small literal constants? As of revision 20 of the instruction syntax, the `addr` instruction always takes a 32 bit constant, so the example above should become:

```
addr #1<<16,rx
```

1.3.4 Obsolete shift

There are several instructions (for example, `mul_sv`) which operate on small vectors, which are the upper 16 bits of each of four consecutive registers. Because only 16 bits were involved in the operation, the original assembly language syntax treated all shifts as being "16 bit", that is, considering only the bits involved in the operation. However, since the small vectors occupy the *upper* 16 bits of registers, this can be confusing. Other multiply operations have shift counts relative to the full 32 bit registers. For consistency's sake, and to allow for future expansion of the instruction set, the assembly syntax has been changed for revision 20 of the instruction set to make the small vector operations use full 32 bit shifts. During the switch over the assembler will accept old forms but issue warnings. It is important to correct the warnings, because there is one ambiguity (the old shift of 0 will become 16, which conflicts with the old shift of 16 which has become 32). Follow the assembler's instructions and your code should be OK.

1.3.5 Unrecognized storage class 0 (assuming debugging)

This message comes from some of the object file manipulation tools (such as `vmnm` and `vmar`). It is an artifact of the fact that these tools were ported from a different environment, and that the NUON COFF object file format has been extended with some new symbol types. The message is harmless, as the assumption (that the symbols are for debugging) is correct.

2. Run Time Problems

2.1 Crashes

2.1.1 What are the exceptions, and what do they mean?

Whenever an MPE detects an erroneous condition, it will raise an exception which halts that processor. Which exception occurred will be indicated in the `excepsrc` register, and will be reported by the debugger.

The most common exceptions, and their codes, are:

halt (0x01) This isn't really an abnormal condition; it is the exception raised by the `halt` instruction. Most often this is caused by a call to the C `exit` function. If you're using `exit` to signal errors, you probably should pass a unique code to `exit` for each erroneous condition. This code is passed to `exit` in register `r0`, so it will be very easy for you to figure out from the contents of that register what went wrong.

bad data address (0x80) This exception is raised when the processor detects a data address that is not a valid address. Usually this means a pointer that's out of range (for example, a NULL pointer). If you have compiled your program with the `-g` flag, the debugger should be able to show you the offending C source line. If for some reason the source code isn't available (for example, the error occurred in library code, or in some assembly language you've written) then you can try to diagnose the problem from the code. Because instructions are pipelined, the program counter is probably *not* pointing at the offending instruction any more. Most likely the instruction that caused the exception is a load, store, push, or pop that is two instructions before the one where the exception was detected. If it was a `push` or `pop` instruction, check the hardware stack pointer `sp` for underflow or overflow. If it was a `ld_s`, `st_s`, or similar instruction, check the register used for the indirect address to see where it is pointing.

bad instruction address (0x100) This exception is due to the program counter being set to a bad address via an indirect `jmp` or `jsr`, or by an `rts` instruction when the `rz` register has a bad value. This exception is raised when `pcfetch` is detected to be invalid; at this point the `pcexec` program counter (the one that the debugger normally uses) is one instruction past the `jmp`, `jsr`, or `rts` which caused the problem. If the offending instruction is an `rts`, then it's quite possible that the `rz` register has been restored incorrectly because the stack has become corrupted. Check the hardware stack pointer `sp` if the problem occurred in an assembly language function, or the C stack pointer `r31` if it was inside a C function.

dma exception (0x200) This exception is raised by the main bus DMA engine when it detects a write to the `mdmacptr` register while the PENDING bit is set, or if an illegal address is written to `mdmacptr`. Because the error occurs outside of the processor, many cycles can pass between the offending write to `mdmacptr` and the raising of this exception. When you get this exception, check the value of `mdmacptr` displayed by the debugger; the value there may give you a clue as to what happened. Also, try to figure out from the current value of the program counter when the last write to `mdmacptr` occurred. Check there to make sure that code there checks the PENDING bit (bit 5 of `mdmactl`).

2.2 Miscellaneous Bad Behaviour

2.2.1 My program hangs in the same spot all the time!

If there's no apparent reason for the hanging, then this may be a cache bug in the beta hardware. The symptoms are that the program hangs, but clicking on STOP and then RUN again in the debugger will cause it to resume (at least until the next time this code is reached). There is no loop at the point where the code is hanging.

If a load instruction causing a cache miss is followed immediately by a load or store instruction to local memory (which includes any registers accessed via load or store), then the beta hardware will lock up. You can avoid this situation in assembly language by re-arranging your code and/or by inserting `nop` instructions to make sure that a load from cache is not immediately followed by a local memory access. In C code, you should always use the `_GetLocalVar` macro from the `<nuon/mutil.h>` header file to load any values from local memory and/or registers.

2.2.2 Why doesn't my assembly language code for sending comm bus packets or doing DMAs work?

If it seems as though comm bus packets are being sent multiple times, or DMAs are being messed up, from code that is running in a cached MPE, it may be that you're encountering another cache bug. If an instruction accessing a "volatile" register (such as the comm bus send or receive registers, or `mdmacptr` occurs near the end of a cache line or in a branch delay slot, and a cache miss occurs while fetching the next instruction, the original register may be accessed multiple times. For most registers this isn't a problem, but some hardware registers change state when accessed. The assembler will insert padding to try to avoid this bug, and issue warnings when it cannot; but it can't always recognize when this situation happens. For example, it doesn't detect indirect accesses (in which the address of the register has been loaded into a general purpose register).

2.2.3 Why does my program crash mysteriously, and none of the printf calls work?

It is very important to provide prototypes for `printf` and any other function that takes a variable number of arguments. The NUON calling convention is different for functions with a fixed number of arguments and those with a variable number of arguments, and the compiler must know which is which. Make sure you do:

```
#include <stdio.h>
```

if you have any `printf` calls.

Invoking `mgcc` with the `-O` and `-Wall` flags (section 1.1.1)mgcc flags will allow the compiler to report any functions that don't have prototypes, so that this bug is caught at compile time.

2.2.4 *My other bus DMAs aren't working right. What's wrong?*

If you're having trouble with other bus DMAs, the first thing to suspect is the beta hardware bug (fixed in production silicon) which causes problems when an other bus DMA ends just before a page boundary (in development machines this is a 2K boundary). The easiest and probably best way to avoid this bug is to make sure that all other bus transfers are vector aligned and start on a vector boundary. This can be tricky if you're using doing DMA from C data structures. GCC's `__align__` directive can help for statically allocated data structures. For dynamically allocated structures, you may have to insert some code to make sure that your structures end up aligned on 16 byte boundaries.

2.2.5 *What could cause SDRAM memory corruption?*

If you're noticing that objects in SDRAM (such as texture maps or other multimedia data) are becoming corrupted, check your frame buffer code carefully to make sure that all pixel draws are happening within the borders of the screen. Because of the way screen memory is laid out, writing to an illegal bilinear address (for example an (x,y) address where x is too large, even if y is legal) can result in a write to a memory location well beyond the boundaries of the frame buffer.

The MML3D library has some problems with front plane clipping, and so can trigger this bug. For safety's sake, always use a clipping rectangle several pixels smaller than the screen. This problem will be fixed in a future release of the MML3D library.

2.2.6 *My C program is putting data into memory, but other MPEs can't read it. What's wrong?*

The cache is not write-through; that is, data in the cache doesn't get written back to memory until either the cache line is re-used or an explicit flush operation is performed. You should call the `_DCacheSync` function whenever another MPE may want to read data from MPE 0. The `_DCacheFlush` function does the same thing as `_DCacheSync`, but it also causes the cache to be invalidated so that if the C program tries to access memory the copy in cache will not be used.

2.2.7 *I have some data written to memory by another MPE, but my C program isn't reading it correctly.*

This is most likely a cache coherency bug. The C program should call the `_DCacheFlush` function to flush the cache before trying to read from external memory that another

MPE is writing to. Note that this is an expensive operation, so try to minimize the number of times your program does this. For example, it's a good idea to group all inter-process communication using memory into a small section of code.

2.2.8 Why does the video output look funny?

Oz hardware has some problems with scaling video. Unless you set the video up to be 360 or 720 pixels across, you may see some vertical stripes or other interpolation artifacts. This bug is fixed in Aries.

Another Oz bug (also fixed in Aries) is that fully saturated 16 bit per pixel colors can be distorted and come out looking wrong. This is even more of a problem if the two tap vertical filter has been turned on; in that case, the whole screen is likely to take on a greenish cast.

2.2.9 Can I use a 4bpp or 8bpp frame buffer?

Yes, if you use the overlay channel. The main video channel does not support CLUT based video modes. See the BIOS documentation for information on setting up video modes.

Index