Handling denormalized numbers with the GRFPU

Application note 2015-11-26

Doc. No GRLIB-AN-0007

Issue 1.0

CHANGE RECORD

Issue	Date	Section / Page	Description
1.0	2015-11-26	All	First issue

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Scope of the document

This document describes issues relating to handling of denormalized floating-point numbers on systems using the LEON processors together with the GRFPU high-performance floating-point unit.

The work has been performed by Cobham Gaisler AB, Göteborg, Sweden.

1.2 Reference documents

[RD1] "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Std 754-1985

[RD2] GRLIB IP Core User's Manual, Cobham Gaisler AB, http://www.gaisler.com/products/grlib/grip.pdf

2 ABBREVIATIONS

FPU	Floating Point Unit
TBC	To Be Confirmed
TBD	To Be Defined

3 DENORMALIZED NUMBER OVERVIEW

This section provides some background on the floating-point format. Please see the original IEEE-754 standard [RD1] for more detail.

Floating-point numbers consist of a sign bit, an exponent part and a fraction (or mantissa) part. In the normal case, the fraction represents a real number between 0.5 and 1.0 (excluding the exact value 1). The exponent then allows this to be scaled with a factor 2^n, and the sign bit allows negation. Numbers defined this way are in normalized format.

For the lowest supported exponent, the IEEE standard allows a linear range towards zero. This is done by interpreting the lowest exponent value (binary 0) as a special case. The numbers in this extended range are called denormal or subnormal (except for the special case of absolute zero).

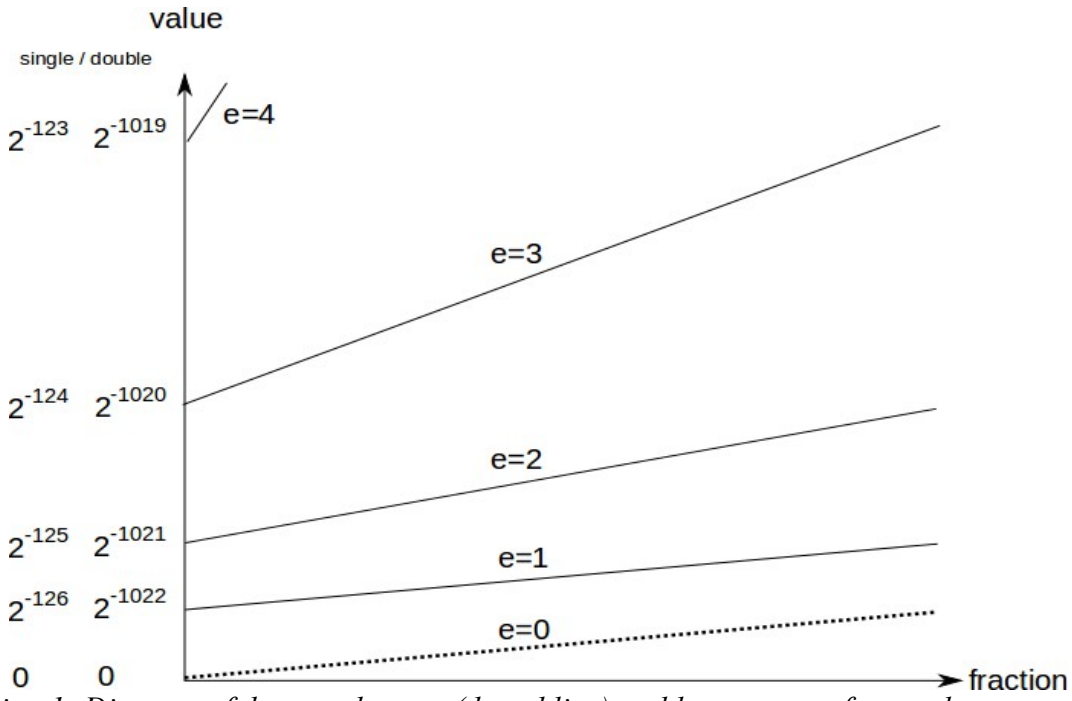

Illustration 1: Diagram of denormal range (dotted line) and lowest part of normal range

The minimum values in the normal and denormal ranges are tabulated below.

Format	Single precision	Double precision
Bits	1 sign, 8 exponent, 23 fraction	1 sign, 11 exp, 52 fraction
Minimum normal	2^{-126}	2^{-1022}
Minimum denormal	2^{-149}	2^{-1074}

4 BEHAVIOR OF LEON SYSTEM COMPONENTS

4.1 GRFPU behavior

As specified in the GRFPU documentation, the GRFPU hardware does not support denormalized numbers as input to operations. The FPU will generate a floating-point exception with the floating-point trap type field set to *unfinished_Fpop* whenever an operation on a denormal is attempted. This provides a "hook" where software can implement emulation of denormalized calculations if desired. However this requires understanding of the intricate details in the SPARC standard regarding the FPU deferred floating-point queue.

The GRFPU can be set to a mode where denormalized numbers are treated as if they were zero. This is done by setting the NS bit in the FSR register to enable non-standard mode. The only effect of non-standard mode is the treatment of denormalized numbers when given as input to the FPU.

The GRFPU never generates denormalized numbers as output. Results in the denormalized range are rounded to a normal number or to zero and the underflow bit is set.

Note that GRFPU will only generate *unfinished_FPop* on input of denormalized numbers. No other sources for this exception exist.

4.2 Operating system runtime behavior

The only operating system that handles the unfinished_FPop trap and emulates the operation is Linux. Other operating systems and runtimes (BCC, RTEMS, VxWorks) will treat the floating point trap as a fatal error.
Neither of the ports as provided by Cobham Gaisler enable nonstandard mode in the FPU.

It is in theory possible to also emulate denormalized output generation by trapping on every FPU underflow and emulating the operation. However no operating system implements this.

4.3 Compiler behavior

GCC can produce floating-point constant in the denormal range if specified directly in the code. Constants in the denormal range can also be introduced indirectly, if the compiler can optimize out a computation that has known input values and a denormalized result. The compiler does the computation in software on the host, and does not know of this limitation of the runtime system.

Floating point parsing routines in the C library (such as scanf, strtod) may produce denormals if the string value is in that range, depending on how the specific C library implementation.

It is possible to modify GCC and define the floating-point format for LEON so that the compiler knows that the hardware does not support denormalized numbers. Precomputed values produced by the compiler would in this case be zero instead of a denormalized number. This has not been done for the toolchains provided by Cobham Gaisler since this change could hide poorly scaled floating-point arithmetic. In most cases the use of denormalized numbers is an indication of a bug or design error in software. Users who have analyzed their application and can accept treatmeant of denormalized numbers as zero can instead enable non-standard mode for GRFPU.

5 SOURCES OF DENORMALIZED NUMBERS

Since the GRFPU does not produce floating point numbers, any denormalized numbers that come into the application must come from the outside world.

Possible sources of reading in denormals are:

- Software bugs, reading variables that have not been initialized correctly before use or invalid pointers to random data that happens to be denormal.
- Reading binary floating point data from an outside source that contains denormal numbers.
- Constructing manually denormalized floating-point numbers using (integer) code and then reading it in.
- Floating point parsing routines in the C library (such as scanf, strtod) may produce denormals if the string value is in that range.
- Denormal floating point constants directly in the C code causing the compiler to generate denormal constants in the binary code.

A special corner case can happen if the code contains calculations with known input values that can be precomputed by the compiler during optimization, where the input values are in the normal range but the results are not. If the compiler optimizes the code, the compiler will precompute the denormal value and store that in the binary code, leading to triggering the *unfinished_fpop* trap. If the code is not optimized, the computation is done using the FPU hardware and will not trigger the trap, but the output will underflow instead.

Another similar case can occur if double-precision constants that are in the denormal range for single precision are converted to single precision, and this conversion can be precomputed by the compiler resulting in a denormalized single-precision constant in the code if the optimization is performed. This can sometimes be difficult to see since casts can be implicit, for example when passing arguments to functions.

6 RECOMMENDATIONS

If you see *unfinished_fpop* traps when you run your program:

- Check first that you are not reading uninitialized variables.
- If a custom boot loader is used, make sure that the floating point register file is correctly cleared during boot.
- The GRMON commands *float* and *inst* are useful for debugging.
- Check constants used in the code so that they are not close to the denormal ranges.
- If you need to read in untrusted floating-point numbers, they can be "flushed" using an integer routine.
- As an alternative to the above, the GRFPU also supports a non-standard mode where denormalized inputs are automatically treated as zero. This is enabled by setting the NS bit in the FSR register.

7 MANUAL FLUSHING ROUTINE

Below are two small routines to manually scan an array for denormals and flush them to zero. The routines are coded in assembler with a C-compatible calling interface.

7.1 C header, flushfp.h

```
void flushfps(float *array, int nelem);
void flushfpd(double *array, int nelem);
```

7.2 Assembler source, flushfp.S

```
#define EXPMASK_SP   0x7F800000
#define MANTMASK_SP  0x007FFFFF
#define EXPMASK_DP   0x7FF00000
#define MANTMASK_DP  0x000FFFFF

       .global flushfps,flushfpd

flushfps:
       set MANTMASK_SP, %o4
       set EXPMASK_SP, %o5
2:     subcc %o1, 1, %o1
       bge,a 1f
        ld [%o0], %o2
       retl
       nop
1:     addcc %o0, 4, %o0
       andcc %o2, %o5, %g0
       bne 2b
        andcc %o2, %o4, %g0
       be 2b
        andn %o2, %o4, %o2
       b 2b
        st %o2, [%o0-4]

flushfpd:
       set MANTMASK_DP, %o4
       set EXPMASK_DP, %o5
2:     subcc %o1, 1, %o1
       bge,a 1f
        ldd [%o0], %o2
       retl
       nop
1:     addcc %o0, 8, %o0
       andcc %o2, %o5, %g0
       bne 2b
        and %o2, %o4, %g1
       orcc %g1, %o3, %g0
       bne 2b
        andn %o2, %o4, %o2
       set 0, %o3
       b 2b
        std %o2, [%o0-8]
```

7.3 Test program, flushfp_test.c

```
#include <stdio.h>
#include <string.h>
#include "flushfp.h"

float farr[]    = { 0.0f, 1.0f, 1.0e-40f, -1.0e-40f };
float fexpres[] = { 0.0f, 1.0f, 0.0f    , -0.0f };

double darr[]    = { 0.0, 1.0, 1.0e-40, 1.0e-350, -1.0e-350 };
double dexpres[] = { 0.0, 1.0, 1.0e-40, 0.0,      -0.0 };

int main(void)
{
       flushfps(farr,4);
       if (memcmp(farr,fexpres,sizeof(farr))) { puts("fail float"); return 1; }
       flushfpd(darr,5);
       if (memcmp(darr,dexpres,sizeof(darr))) { puts("fail double"); return 1; }
       puts("Success");
       return 0;
}
```