DOI: 10.13009/EUCASS2019-790

DOI:

Ensuring Numerical Reproducibility for Model-Based Software Engineering

Andoni Arregi⋆, Fabian Schriever⋆, Carlos Arias⋆, and Andreas Jung⋆⋆
⋆GTD GmbH
GTD GmbH, Ravensburger Str. 30a, 88677 Markdorf, Germany
andoni.arregi@gtd-gmbh.de · fabian.schriever@gtd-gmbh.de · carlos.arias@gtd-gmbh.de
⋆⋆ESA
ESTEC, Keplerlaan 1, 2200 AG Noordwijk, Netherlands
andreas.jung@esa.int

2019-06-18

Abstract

Model based algorithm and space software development only obtains partial benefits if the validation has to be carried out on the target embedded-system. This especially applies to numerical computation algorithms. Standards such as the IEEE-754 for floating-point arithmetic clarify the situation but numerical reproducibility issues between systems used for development and embedded target systems impede an early validation on the model. We analysed the numerical issues when using Simulink for algorithm development on PCs and the implications of auto-code generation when running on host and target platforms. This evaluation has been based on the use of the Mathematical Library for Flight Software as a solution for numerical reproducibility. We produced guidelines which yield reproducible results on Model In the Loop, Software In the Loop, and Processor In the Loop executions. This study has been performed in the frame of the ESA technology program with Contract No. 4000122343/17/NL/FE/as.

1. Introduction

There is great interest in the space-software community for model based algorithm and software development but only partial benefits are obtained if the verification and validation has to be carried out on the *target* embedded system, as the development environment is not representative enough of it. This especially applies to the development of numerical computation algorithms, such as the ones used for GNC/AOCS (Guidance Navigation and Control, Attitude and Orbit Control System) systems as well as for scientific payload algorithms.

Numerical reproducibility issues among different hardware and software environments have accompanied the development of numerical software since its historical beginnings and despite the advent of standards such as the IEEE-754 for floating-point arithmetic and other standards like the ISO C99 and POSIX have greatly clarified the situation, numerical reproducibility issues between the *host* systems used for development (normal PCs) and the embedded *target* systems (on-board processors) impede a proper and early validation on host, as well as the investigation of problems observed during AIT (Assembly, Integration, and Test) phases on target systems. These numerical reproducibility issues affect the accuracy and the error signaling behaviour, including exceptions and special-value generation e.g., NaN (Not a Number) and INF (Infinity).

We analysed the numerical reproducibility issues when using Mathworks Simulink as a model based development environment for algorithm development on PC platforms and the implications of auto-code generation when running the results on host and target platforms. As host platforms we analysed the differences among the x86-64 processors and for the target side especially the SPARC V8 (Scalable Processor Architecture) LEON processor family and their FPUs (Floating Point Unit).

This evaluation has been based on the use of the space qualified `MLFS`[1] (Mathematical Library for Flight Software) provided by ESA (European Space Agency) as a possible solution for numerical reproducibility of elementary mathematical procedures and its test suite BLTS (Basic Library Test Suite) to assess the exceptions behaviour.

[1] Download at https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software

DOI: 10.13009/EUCASS2019-790

We have been able to produce guidelines which - when applied - yield reproducible numerical results (to the last bit) on MIL (Model In the Loop - host), SIL (Software In the Loop - host), and PIL (Processor In the Loop - target) executions of a non-linear system including elementary mathematical procedures as well as elementary arithmetical operations. Although we cannot prove our guidelines as universally applicable, understanding and applying them can imply a significant win in advancing on model based numerical algorithms development for the space industry.

1.1 Model Based Space On-Board Software Development

Figure 1 shows what we envision as a typical Model Based SW Engineering approach based on Matlab/Simulink including MIL, SIL, and PIL simulation modes.

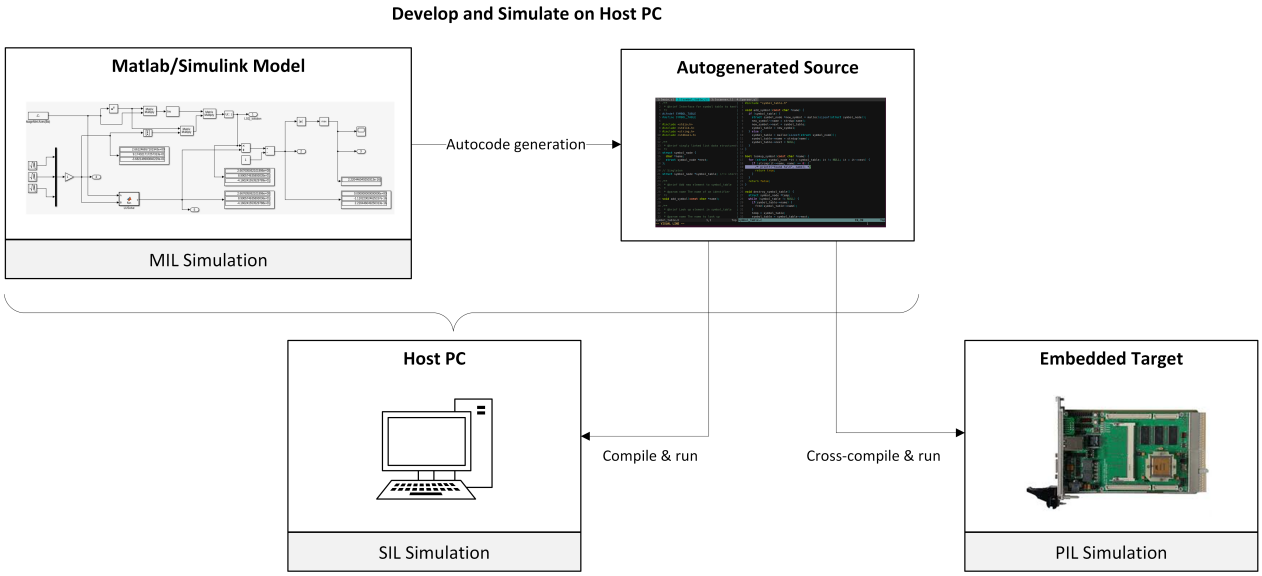

Figure 1: Model based space on-board software development

In this approach several different mathematical libraries are involved depending on the simulation mode. Matlab internally uses an `FDLIBM` (Freely Distributable Mathematical Library) based mathematical library for elementary mathematical computations (e.g., `sin`, `cos`, `exp`, `log`, `sqrt`, `pow`). This library was originally developed in the late 1980s. For SIL simulations on PCs the use of the `glibc` mathematical library is common when used together with the GCC (GNU C Compiler) tool-chain. For algebraic operations (e.g., linear system solving) other mathematical libraries such as `BLAS` (Basic Linear Algebra Subprograms) and `LAPACK` (Linear Algebra Package) are used when running Simulink simulations on PC.

The different existing simulation modes described in Figure 1 and their relation to Matlab/Simulink can be described as:

- **MIL**: Model In the Loop simulation. E.g., a native Simulink simulation, run in *normal mode* within the Simulink environment, where our model is being computed by the Simulink engine. The mathematical library is based on `FDLIBM` and BLAS/LAPACK are used for algebraic operations.
- **SIL**: Software In the Loop simulation. Here we mean a simulation where source code has been auto-generated from our model, compiled with a compiler tool-chain available on the host PC to a standalone executable, and run on that host PC without any Simulink dependency. The mathematical library used is a different one than the Matlab internal one (e.g. the `glibc` mathematical library).
- **PIL**: Processor In the Loop execution. Here we mean a run, where source code has been auto-generated from our model, cross-compiled for the embedded target processor, and run on that target.

2. Motivation and Problem Formulation

Algorithms developed for space-software and especially for GNC/AOCS systems base on numerical computations including among others:

DOI: 10.13009/EUCASS2019-790

- Elementary mathematical functions (trigonometric functions, logarithms, exponentials, &c)
- Other arithmetic/algebraic operations:
  - Matrix multiplications (based on the *dot product* operation)
  - Matrix inversions
  - Linear system solving
  - &c

To standardize such numerical computations and their error condition behaviour several international standards have been defined over decades:

- the IEEE-754 Floating-point arithmetic standard (see [3]),
- the ISO C programming language standard(see [1]), and
- the POSIX (Portable Operating System Interface) standard (see [2])

But the numerical reproducibility problem throughout hardware/software development environments is still not solved because of the common non-compliance to these standards and opacity of the simulation tools used. Common non-compliances of these standards are, among others:

- the lack of subnormal floating-point support on several FPUs such as the GRFPU (Cobham Gaisler High Performance FPU, see [4]) commonly used on LEON3/4 processors,
- the lack of FMA (Fused Multiply Add, $fma(x, y, z) = x \times y + z$) operation support (e.g., on SPARC V8 processor architectures), and
- compiler and tool-chain library non-compliances e.g., non compliances in the tool-chain inherent mathematical library (for example *fmax(0, -0)* returns the false zero in certain libraries), non compliances in exception behaviour and NaN propagation (for example when using relational operators)

The numerical differences in computations we analysed show absolute differences of 5×10^{-5} for 64 bit double-precision computations between the model based development environment and the embedded target and even a single dot product of dimension 3 vectors can show an absolute difference of 2.22×10^{-16} (see [8]).

2.1 Floating-Point Numbers as a Model for Real Numbers

The most widely used model for real numbers in computations is the floating-point number model defined in the IEEE-754 standard. Figure 2 shows the complexity of the complete range of 32 bit (single-)precision floating-point numbers.

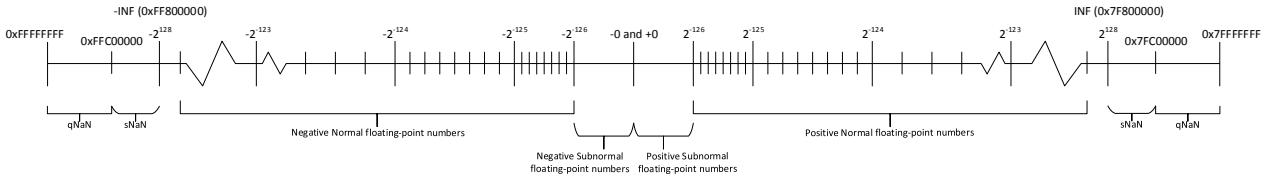

Figure 2: Floating-point number range in 32 bit precision

This model of the real numbers presents many tricky behaviours that might not be expected by software engineers (see [5] for what any software engineer should know about floating-point computations). For example the rational number $\frac{1}{10} = 0.1$ has no exact representation in floating-point (not even in 64 bit double-precision).

```
32 bits single:          0x3DCCCCCC 9.99999940395355224609375E-2
64 bits double: 0x3FB999999999999A 1.00000000000000005551115E-1
```

DOI: 10.13009/EUCASS2019-790

Other curios non-intuitive phenomena that affect the computation with floating-point numbers, of which the software programmer shall be aware of, include Rump's example (see the original article [10] and how to reproduce it on IEEE 754 arithmetic in [9]):

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b^2} \tag{1}$$

```
a = 77617
b = 33096
32 bits evaluation f = +1.172604
64 bits evaluation f = +1.1726039400531786
correct value      f = −0.827396059946821368141165095479 82
```

Within this function a so called *catastrophic cancellation* phenomenon is hidden, which produces a loss of many significant digits and leads to the erroneous result.

3. Methodology

Within our study we analysed the following aspects regarding numerical and error condition handling reproducibility:

- Implications of the use of Matlab/Simulink (*normal*, *accelerator*, *rapid accelerator*, and SIL modes) and auto-code generation
- Implications of the use of elementary mathematical functions (those provided by `math.h`)
- Implications of the use of other arithmetic/algebraic operations (dot product, matrix multiplication, &c)
- Implications of the differences in FPU architectures (e.g., availability of FMA instructions, subnormal support)
- Implications of the use of parallel/multicore computing and the use of Graphics Processing Units (GPU) (in examples given by Intel the same binary can give different results even on the same processor in successive runs, see [12])
- Implications of the used compilers and different tool-chains (i.e., GCC, Clang, and Intel C compiler)
- Implications of exception generation and NaN handling

These aspects are studied on several example applications of numerical computations of which we present two representative applications: an altered logistic map simulation model and the evaluation of polynomials.

3.1 The Altered Logistic Map Simulation Model

As an artificial simulation created for the specific purpose to test numerical reproducibility aspects within the workflow described above, we used the logistic map function to create an aperiodic sequence[2][3] with an *alteration* introduced in the feed-back loop to modify the generated sequence:

$$x_{n+1} = r \cdot \arcsin(\sin(x_n)) \cdot (1 - \arcsin(\sin(x_n))) \tag{2}$$

This simulation model, once with standard Simulink blocks for the mathematical functions (see Figure 3) and once with Matlab function blocks calling `MLFS` MEX (Matlab Executable) functions (see Figure 4), has been run as described in the Appendix with an initial value of $x = 0.5$ and for 482 070 iterations in each case.

When comparing MIL simulations (normal, accelerator and rapid accelerator modes) with SIL simulations the auto-code generation process and compilation may introduce the following alterations:

- differences in order of arithmetical operations (e.g., altered execution order due to parallelization with multi-threading),

[2]The logistic map is in a chaotic regime for $r = 3.9375$ thus, it will produce an aperiodic sequence which will diverge in case of slight numerical discrepancies (its Lyapunov exponent for these conditions is $\lambda \approx 0.531$).

[3]https://www.wolframalpha.com/input/?i=logistic+map&assumption={"F",+"LogisticMap",+"r"}+->"3.9375"&assumption={"F",+"LogisticMap",+"x0"}+->"0.5"&assumption={"C",+"logistic+map"}+->+{"Formula"}

DOI: 10.13009/EUCASS2019-790

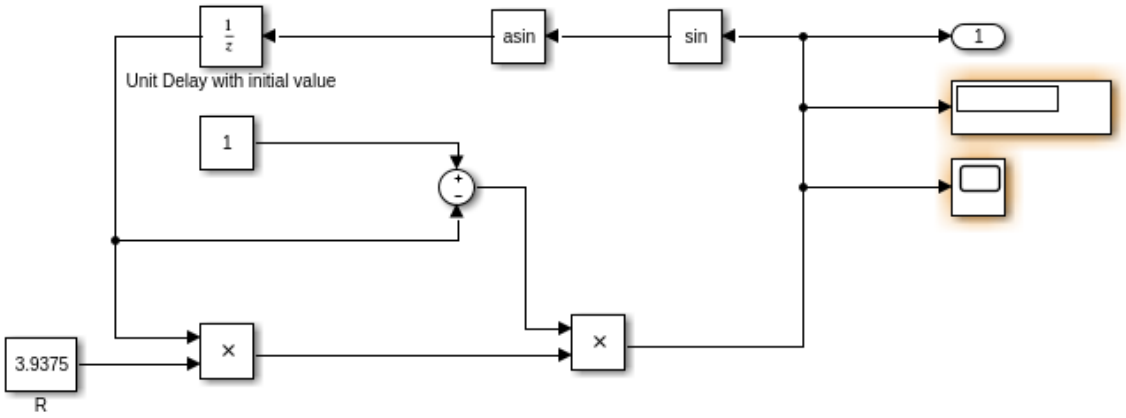

Figure 3: Logistic map model in Simulink with alteration in feedback loop

- differences in the implementation of elementary mathematical functions by linking to a different mathematical library, and
- differences in the executed machine instructions (e.g, presence or absence of FMA)

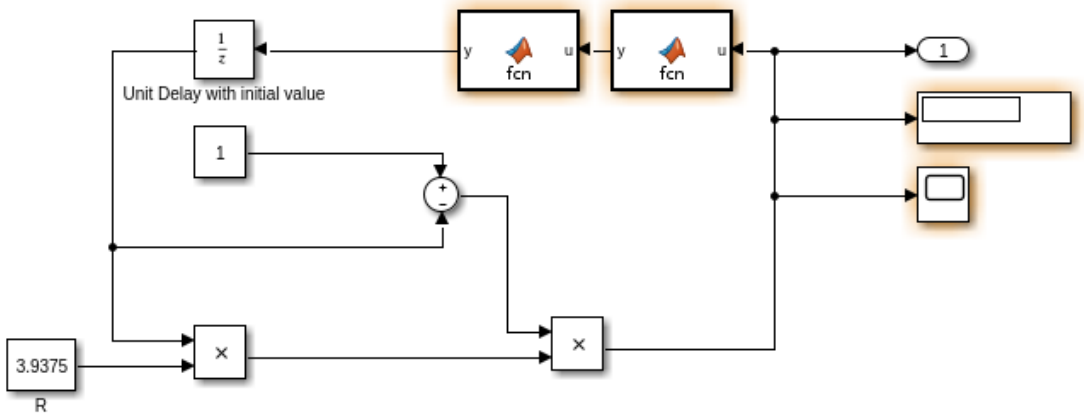

Figure 4: Logistic map model in Simulink with alteration in feedback loop and Matlab function blocks calling `MLFS` MEX functions instead of Simulink base blocks

3.1.1 Impact on Polynomial Evaluation

Polynomial evaluation is needed in many numerical computations. For example the most convenient way to evaluate several elementary mathematical functions are polynomial approximations (e.g., trigonometric functions). Furthermore, one of the most widely used algorithms to evaluate polynomials is the *Horner scheme* (see [7]), where a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{3}$$

can be rewritten as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + a_n x)))) \tag{4}$$

For this method of evaluation no power procedure is needed and the resulting code is more efficient than a *naive* evaluation of the first version.

DOI: 10.13009/EUCASS2019-790

The computations of this scheme will differ numerically in their results depending on the concrete implementation (e.g., *canonical* or *compensated*[4] implementation of Horner scheme) and on the availability or not of an FMA instruction (see [6]).

For our analysis we took[5] a difficult to evaluate polynomial, a *Wilkinson's polynomial* (see [11], §2.9):

$$w(x) = \prod_{i=1}^{20}(x-i) = (x-1)(x-2)(x-3)\cdots(x-20) \tag{5}$$

but truncated it to degree 12:

$$w_{truncated}(x) = \prod_{i=1}^{12}(x-i) = (x-1)(x-2)(x-3)\cdots(x-12) \tag{6}$$

which has the expanded form:

$$\begin{aligned} w_{truncated}(x) = x^{12} - 78x^{11} + 2\,717x^{10} - 55\,770x^9 + 749\,463x^8 - 6\,926\,634x^7 + 44\,990\,231x^6 \\ -206\,070\,150x^5 + 657\,206\,836x^4 - 1\,414\,014\,888x^3 + 1\,931\,559\,552x^2 - 1\,486\,442\,880x + 479\,001\,600 \end{aligned} \tag{7}$$

We analysed the numerical impact and accuracy of different methods to evaluate this difficult polynomial for $x = 12.001$.

4. Results

4.1 The Altered Logistic Map Simulation Model

Tests show no numerical differences between the three Simulink MIL modes, normal, accelerator and rapid accelerator, since in all cases the same implementations of the mathematical functions are used.

Differences show in the numerical results with this model, when comparing MIL simulations (normal, accelerator and rapid accelerator modes) with SIL and PIL simulations.

Figure 5 shows the numerical differences obtained when running the logistic map model described above in different modes. The numerical experiments described in the Appendix produce three numerically different results, grouped as follows (refer to the Appendix for the definition of the notation used for the identifiers of the experiments):

1. Simulink MIL simulations in normal, accelerator and rapid accelerator modes using the standard Simulink blocks for mathematical functions.
   Experiments: Host1(a), Host1(b), and Host1(c)

2. Simulink SIL simulations and running the Simulink auto-code generator result on a PC using the `glibc` implementations for the mathematical functions.
   Experiments: Host1(d) and Host2(d)

3. Simulink MIL simulations in all modes, SIL simulation, and target PIL runs when using the `MLFS` library implementations for the mathematical functions.
   Experiments: Host1(I), Host1(II), Host1(III), Host1(IV), and Target(V)

Figure 5 shows the differences between the groups mentioned above the following way:

- between 1. and 2. in grey,
- between 1. and 3. in blue, and
- between 2. and 3. in purple

The yellow line shows as an example that there are no differences between a Simulink MIL simulation in normal mode and a PIL run on the target LEON2 processor when using the `MLFS` library for the mathematical functions.

The main feature shown is that the `MLFS` based MIL, SIL, and PIL executions provide exactly the same numerical results.

[4] A *compensated* implementation means that the algorithm includes a separate running accumulation of small errors to compensate the final result.
[5] The example has been taken from the Python bug tracker issue 29 282 as it suits our needs.

DOI: 10.13009/EUCASS2019-790

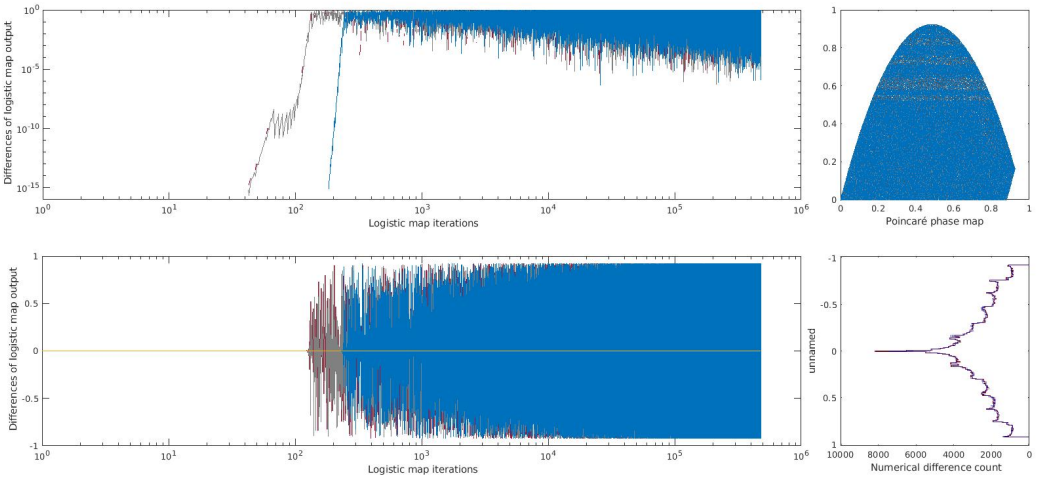

Figure 5: Numerical differences comparison over logistic map iterations

4.2 Impact on Polynomial Evaluation

Evaluating this polynomial with a *canonical* Horner scheme implementation, a *canonical* implementation with FMA, and a *compensated* implementation with FMA produces the following different results for $x = 12.001$ when computed with 64 bit floating-point numbers on an Intel Core i5-3360M without FMA support (the same result is obtained with GCC 6.3.0, clang 3.8.1-24 and icc 18.0):

```
Horner (standard): w_t(12.001) = 4.00375245919823646545 4E+04 0X1.38CB0C9752P+15
Horner (fma):      w_t(12.001) = 4.00374882163976799347 4E+04 0X1.38CAF9F77FEADP+15
Horner (comp.):    w_t(12.001) = 4.00374948632578307297 1E+04 0X1.38CAFD5EB788CP+15
```

The numerical result obtained with Maple is:

$$4.00374948632800850721729154556719250660 01 \times 10^4 \tag{8}$$

Which means that the *canonical* Horner scheme implementation is correct to 5 digits (relative error of 7.43×10^{-7}), the *canonical* version with FMA support to 6 digits (relative error of 1.66×10^{-7}), and the *compensated* implementation with FMA support to 12 digits (relative error of 5.56×10^{-13}).

Although this is just an example, it shows the potential of the availability of FMA instructions to obtain more accurate results and how the presence or absence, together with different numerical implementations will produce varying numerical results with relative errors differing in orders of magnitude.

5. Conclusions and Resulting Guidelines

5.1 Guidelines to Improve Reproducibility and Portability of Numerical Computations

1. Always use the same mathematical library (e.g., `MLFS`) on all systems (host and target) to assure that the starting point of numerical and exception behaviour will be the same on all those platforms (the compilation and the hardware itself will still have an impact though, which we will try to solve with the following guidelines).

2. Always compile using the `-frounding-math -fsignaling-nans -fno-builtin` compiler options to obtain an IEEE-754 compliant floating-point arithmetic behavior when using GCC compilers[6], or the `-fp-model strict -fp-model source` options when using the Intel C compiler.

3. Configure the FSR register on SPARC V8 processors (see Figure 6) taking the following into account:

   - Always use *round to nearest tie to even* rounding mode (bits 30 and 31, `rd`, set to 0).

[6]Clang does not provide the `-frounding-math -fsignaling-nans` flags as of version 3.8.1-24.

DOI: 10.13009/EUCASS2019-790

- Configure the FPU to trap on the *important* exceptions (*Invalid Op.*, *Div. by 0*, and *Overflow*) while developing the software. For LEON2, LEON3, and LEON4 processors see the FSR register bits below and set the corresponding bits to 1 (`nvm` to 1, `ofm` to 1, and `dzm` to 1). The general trap enabling bit has also be set to 1 (PSR bit 5 to 1).
- Set the *nonstandard* modus on processors using the GRFPU (bit 22, `ns` set to 1), since the processor will otherwise trap on subnormal floating-point numbers. This modus will handle subnormals as 0, which is not IEEE-754 compliant but will not trap (see [4]).
- **Note**: on x86 processors the `MXCSR` register will have to be configured accordingly.

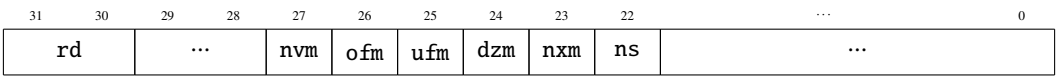

Figure 6: FSR register on SPARC V8 architecture processors

4. In case of using task preemption on RTEMS (Real-Time Executive for Multiprocessor Systems), make sure that the current floating-point status is saved and restored during context switching by setting the the `RTEMS_FLOATING_POINT` attribute flag when creating the RTEMS task with `rtems_task_create()`.

5. In case of using a processor with an FPU without subnormal support such as the GRFPU, disable subnormal support in Matlab and Simulink setting the DAZ (Denormals Are Zero) and FTZ (Flush To Zero) modes on.

6. Use the MEX function mechanism to use `MLFS` procedures within the Simulink MIL simulation[7].

7. Limit the computational threads of Matlab and limit Matlab to use only one CPU core.

   (a) Set the `-singleCompThread` command line option when starting Matlab to limit Matlab to a single computational thread[8].

   (b) Specify a given set of CPUs (cores) that are available for use by the Matlab process by setting CPU affinity provided by the operating system (This still allows Matlab to use more than one computational threads, that is why the previous option is also necessary).

   - GNU/Linux:

     ```
     % taskset -c 0 matlab -singleCompThread
     ```

   - MS-Windows (a tool like https://docs.microsoft.com/en-us/sysinternals/downloads/pstoolsPsTools can be used, otherwise this can be set in the Task Manager: Processes › Select Image name › rightclick › Set Affinity):

     ```
     > psexec -a 0 %MATLAB%\bin\win64\MATLAB.exe -singleCompThread
     ```

     where `0` is the number of the CPU (e.g., `0` for the CPU 0).

8. Always compile for SSE/AVX architecture on x86-64 platforms (e.g., using the flag `-mavx2` with GCC or Clang compilers, or `-march=core-avx2` with the Intel C compiler) to avoid x87 80bit precision intermediate registers[9]. The use of the 80 bit x87 registers would produce a more accurate results, but the results will not be reproducible on architectures not using an extended precision (e.g., the SPARC V8 architecture).

9. Always compile excluding FMA instructions using the `-mno-fma` compiler option (this is valid for the GCC and Clang compilers, use `-no-fma` for the Intel C compiler) on x86-64 platforms to ensure no FMA instructions are used when the target platform architecture does not support FMA either.

10. When using CUDA (Compute Unified Device Architecture) for GPU programming, the following compiler flags shall be used:

    - `-ftz=false`: use subnormal floating-point numbers (do not flush them to zero)
    - `-prec-div=true`: compute division to the nearest floating-point number

[7] With the Simulink MIL mode we mean the regular normal mode simulation of a Simulink model.

[8] This can also be achieved from within Matlab running `maxNumCompThreads(1)`.

[9] The flag `-msse2` for GCC and Clang and `-march=sse2` for the Intel C Compiler may also be used for the same purpose on older x86 processors that do not have AVX2

DOI: 10.13009/EUCASS2019-790

- `-prec-sqrt=true`: compute square root to the nearest floating-point number
- `-fmad=false`: do not merge multiply and add operations
- **Note**: If subnormal support is not desired, as when using a GRFPU, set `-ftz=true` to emulate this behavior on the GPU.

5.2 Additional Recommendations Regarding Numerical Accuracy and Error Condition Handling

1. Check for subnormal floating-point numbers being generated by the numerical computation (e.g., by running the software on a GRFPU configured to trap on subnormal floating-point numbers, see FSR configuration in Figure 6). This will point to computations where precision is being lost.
2. Set floating-point variables to sNaN on creation. This will raise an *Invalid Operation* exception when uninitialized data is used.
3. Check for NaN results after numerical computation blocks. Be aware that using NaNs in relational operators can produce perfectly valid results and not necessarily signal an Invalid Operation exception depending on the compiler and its version (e.g., in the case of GCC this was solved in version 8.1).

6. Appendices

For the numerical experiments carried out the following computing platforms have been used:

- Host1:
  - Processor: Intel Core i5-3360M
  - Operating system: Debian GNU/Linux running within Oracle VirtualBox
  - Compiler: GCC 6.3.0
  - *Note*: This system has no hardware FMA support thus, the GCC software FMA implementation has been used for test purposes in those cases where it is indicated so. For the rest of the tests, this platform shall e regarded as having no FMA support.
- Host2:
  - Processor: Intel Core i5-6400
  - Operating system: Debian GNU/Linux running within WSL (Windows Subsystem for Linux)
  - Compiler: GCC 6.3.0
  - *Note*: This system does have hardware FMA support.
- Target
  - Processor: Atmel AT697E LEON2 with MEIKO FPU
  - Operating system: Edisoft RTEMS 4.8
  - Compiler: GCC 4.2.1

Note that we only identify the relevant attributes with a version number (e.g., it does not really matter what Linux kernel version we are using).

The altered logistic map Simulink model described in Section 3.1 and using the standard Simulink mathematical functions has been executed in the following modes, running in each case 482 070 iterations:

(a) Simulink normal mode

(b) Simulink accelerator mode

(c) Simulink rapid accelerator mode

(d) Simulink SIL mode

DOI: 10.13009/EUCASS2019-790

The altered logistic map Simulink model described in Section 3.1 and using the `MLFS` mathematical functions has been executed in the following modes, running in each case 482 070 iterations:

(I) Simulink normal mode

(II) Simulink accelerator mode

(III) Simulink rapid accelerator mode

(IV) Simulink SIL mode

(V) PIL mode (on the target LEON2)

The auto-code generation has been configured to (the rest is configured in default modus for Intel x86-64 targets on Linux):

- maximum parentheses level (specify precedence with parentheses) and
- preserve operand order in expression.

All compilations follow the guidelines described in Section 5 except for the cases where FMA support has been included. In addition the `-O2` optimization flag has been used in all cases.

In the text, we will reference these experiments as follows:

- **Host1(c)** means a rapid accelerator simulation in Simulink on Host1 (without FMA support) using standard Simulink mathematical functions.
- **Host1(IV)** means a SIL simulation on Host1 using the `MLFS` mathematical functions.
- **Target(V)** means a PIL simulation on Target using the `MLFS` mathematical functions.
- **Host2(d)** means a SIL simulation on Host2 using the standard Simulink mathematical functions.

References

[1] ISO/IEC 9899 - Programming Languages - C, 2005.

[2] IEEE P1003.1 - Portable Operating System Interface (POSIX), 2007.

[3] IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754-2008, Aug 2008.

[4] Cobham-Gaisler. Handling Denormalized Numbers with the GRFPU, 2015.

[5] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic, 1991.

[6] S. Graillat, Ph. Langlois, and N. Louvet. Improving the Compensated Horner Scheme with a Fused Multiply and Add, 2005.

[7] W. G. Horner. A New Method of Solving Numerical Equations of All Orders, by Continuous Approximation. *Philosophical Transactions of the Royal Society of London*, 109:308–335, January 1819.

[8] William Kahan. Matlab's Loss is Nobody's Gain, 1998.

[9] Eugene Loh and G Walster. Rump's Example Revisited. In *Reliable Computing*, volume 8, pages 245–248. 01 2002.

[10] Siegfried M. Rump. Reliability in Computing: The Role of Interval Methods in Scientific Computing. chapter Algorithms for Verified Inclusions: Theory and Practice, pages 109–126. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[11] James Hardy Wilkinson. *Rounding Errors in Algebraic Processes*. H.M.S.O London, 1963.

[12] Georg Zitzlsberger. FP Accuracy & Reproducibility; Intel C++/Fortran Compiler, Intel Math Kernel Library, and Intel Threading Building Blocks, September 2014.