Chapter 36
IEEE Arithmetic

Any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete. In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory.
Godel, First incompleteness theorem

Aims
The aims of this chapter are to look in more depth at arithmetic and in particular at the support that Fortran provides for the IEEE 754 and later standards. There is a coverage of:

- hardware support for arithmetic.
- integer formats.
- floating point formats: single and double.
- special values: denormal, infinity and not a number — nan.
- exceptions and flags: divide by zero, inexact, invalid, overflow, underflow.

36.1 Introduction

The literature contains details of the IEEE arithmetic standards. The bibliography contains details of a number of printed and on-line sources.

36.2 History

When we use programming languages to do arithmetic two major concerns are the ability to develop reliable and portable numerical software. Arithmetic is done in hardware and there are a number of things to consider:

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_36

- the range of hardware available both now and in the past.
- the evolution of hardware.

There has been a very considerable change in arithmetic units since the first computers. Table 36.1 is a list of hardware and computing systems that the authors have used or have heard of. It is not exhaustive or definitive, but rather reflects the authors' age and experience.

Table 36.1 Computer hardware and manufacturers

CDC	Cray	IBM	ICL
Fujitsu	DEC	Compaq	Gateway
Sun	Silicon graphics	Hewlett Packard	Data general
Harris	Honeywell	Elliot	Mostek
National semiconductors	Intel	Zilog	Motorola
Signetics	Amdahl	Texas instruments	Cyrix
AMD	NEC		

Table 36.2 lists some of the operating systems.

Table 36.2 Operating systems

NOS	NOS/BE	Kronos	UNIX
VMS	Dos	Windows 3.x	Windows 95
Windows 98	Windows NT	Windows 2000	Windows XP
Windows vista	Windows 7.x	Windows 8.x	MVS
VM	VM/CMS	CP/M	Macintosh
OS/2	Linux (too many)		

Again the list is not exhaustive or definitive. The intention is simply to provide some idea of the wide range of hardware, computer manufacturers and operating systems that have been around in the past 50 years.

To cope with the anarchy in this area Doctor Robert Stewart (acting on behalf of the IEEE) convened a meeting which led to the birth of IEEE 754.

The first draft, which was prepared by William Kahan, Jerome Coonen and Harold Stone, was called the KCS draft and eventually adopted as IEEE 754. A fascinating account of the development of this standard can be found in An Interview with the Old Man of Floating Point, and the bibliography provides a web address for this interview. Kahan went on to get the ACM Turing Award in 1989 for his work in this area.

This has become a de facto standard amongst arithmetic units in modern hardware. Note that it is not possible to describe precisely the answers a program will give, and the authors of the standard knew this. This goal is virtually impossible to achieve when one considers floating point arithmetic. Reasons for this include:

- the conversions of numbers between decimal and binary formats.
- the use of elementary library functions.
- results of calculations may be in hardware inaccessible to the programmer.
- intermediate results in subexpressions or arguments to procedures.

The bibliography contains details of a paper that addresses this issue in much greater depth — Differences Among IEEE 754 Implementations.

Fortran is one of a small number of languages that provides access to IEEE arithmetic, and it achieves this via TR1880 which is an integral part of Fortran 2003. The C standard (C9X) addresses this issue and Java offers limited IEEE arithmetic support. More information can be found in the references at the end of the chapter.

36.3 IEEE Specifications

There have been several IEEE arithmetic standards. The following information is taken from the ISO site.

The url is

```
https://www.iso.org/standard/57469.html
```

ISO/IEC/IEEE 60559:2011(E) specifies formats and methods for floating-point arithmetic in computer systems - standard and extended functions with single, double, extended, and extendable precision and recommends formats for data interchange. Exception conditions are defined and standard handling of these conditions is specified. It provides a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. Errors, and error conditions, in the mathematical processing will be reported in a consistent manner regardless of implementation. This first edition, published as ISO/IEC/IEEE 60559, replaces the second edition of IEC 60559.

Here is the standard history.

- ISO/IEC/IEEE 60559:2011(E)
- IEC 559:1989
- IEC 559:1982

The standard provides coverage of the following areas, which is taken from the table of contents.

- Floating-point formats
  - Overview
  - Specification levels

- Sets of floating-point data
- Binary interchange format encodings
- Decimal interchange format encodings
- Interchange format parameters
- Extended and extendable precisions

• Attributes and rounding

- Attribute specification
- Dynamic modes for attributes
- Rounding-direction attributes

• Operations

- Overview
- Decimal exponent calculation
- Homogeneous general-computational operations
- Format of general-computational operations
- Quiet-computational operations
- Signaling-computational operations
- Non-computational operations
- Details of conversions from floating-point to integer formats
- Details of operations to round a floating-point datum to integral value
- Details of totalorder predicate
- Details of comparison predicates
- Details of conversion between floating-point data and external character sequences

• Infinity, NaNs, and sign bit

- Infinity arithmetic
- Operations with NaNs
- The sign bit

• Default exception handling

- Overview: exceptions and flags
- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

• Alternate exception handling attributes

- Overview
- Resuming alternate exception handling attributes
- Immediate and delayed alternate exception handling attributes

- Recommended operations
  - Conforming language- and implementation-defined functions
  - Recommended correctly rounded functions
  - Operations on dynamic modes for attributes
  - Reduction operations
- Expression evaluation
  - Expression evaluation rules
  - Assignments, parameters, and function values
  - preferred width attributes for expression evaluation
  - Literal meaning and value-changing optimizations
- Reproducible floating-point results

36.4 Floating Point Formats

Table 36.3 summarises the formats specified in the IEEE 754-2008 standard.

Table 36.3 IEEE formats

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[1]	E min E min	
Binary16	Half precision	2	11	3.31	5	4.51	2**4−1 = 15	−14 +15	[2]
Binary32	Single precision	2	24	7.22	8	38.23	2**7−1 = 127	−126 +127	
Binary64	Double precision	2	53	15.95	11	307.95	2**10−1 = 1023	−1022 +1023	
Binary128	Quadruple precision	2	113	34.02	15	4931.77	2**14−1 = 16383	−16382 +16383	
Binary256	Octuple precision	2	237	71.34	19	78913.2	2**18−1 = 262143	−262142 +262143	[2]
Decimal32		10	7	7	7.58	96	101	−95 +96	[2]
Decimal64		10	16	16	9.58	384	398	−383 +384	
Decimal128		10	34	34	13.58	6144	6176	−6143 +6144	

36.5 Procedure Summary

Tables 36.4 and 36.5 summarise the procedures.

Table 36.4 IEEE Arithmetic module procedure summary

Procedure arguments	Class	Description
IEEE_CLASS(X)	E	Classify number
IEEE_COPY_SIGN(X,Y)	E	Copy sign
IEEE_FMA(A,B,C)	E	Fused multiply-add operation
IEEE_GET_ROUNDING_MODE	S	Get rounding mode
(ROUND_VALUE[,RADIX])	S	Get rounding mode
IEEE_GET_UNDERFLOW_MODE	S	Get underflow mode
(GRADUAL)	S	Get underflow mode
IEEE_INT(A,ROUND[, KIND])	E	Conversion to integer type
IEEE_IS_FINITE(X)	E	Whether a value is finite
IEEE_IS_NAN(X)	E	Whether a value is an IEEE NaN
IEEE_IS_NEGATIVE(X)	E	Whether a value is negative
IEEE_IS_NORMAL(X)	E	Whether a value is a normal number
IEEE_LOGB(X)	E	Exponent
IEEE_MAX_NUM(X,Y)	E	Maximum numeric value
IEEE_MAX_NUM_MAG(X,Y)	E	Maximum magnitude numeric value
IEEE_MIN_NUM(X,Y)	E	Minimum numeric value
IEEE_MIN_NUM_MAG(X,Y)	E	Minimum magnitude numeric value
IEEE_NEXT_AFTER(X,Y)	E	Adjacent machine number
IEEE_NEXT_DOWN(X)	E	Adjacent lower machine number
IEEE_NEXT_UP(X)	E	Adjacent higher machine number
IEEE_QUIET_EQ(A,B)	E	Quiet compares equal
IEEE_QUIET_GE(A,B)	E	Quiet compares greater than or equal
IEEE_QUIET_GT(A,B)	E	Quiet compares greater than
IEEE_QUIET_LE(A,B)	E	Quiet compares less than or equal
IEEE_QUIET_LT(A,B)	E	Quiet compares less than
IEEE_QUIET_NE(A,B)	E	Quiet compares not equal
IEEE_REAL(A[,KIND])	E	Conversion to real type
IEEE_REM(X,Y)	E	Exact remainder
IEEE_RINT(X)	E	Round to integer
IEEE_SCALB(X,I)	E	X 2I
IEEE_SELECTED_REAL_KIND	T	IEEE kind type parameter value
([P,R,RADIX])	S	IEEE kind type parameter value
IEEE_SET_ROUNDING_MODE	S	Set
(ROUND_VALUE[,RADIX])	S	Set
IEEE_SET_UNDERFLOW_MODE	S	Set underflow mode
(GRADUAL)	S	Set underflow mode
IEEE_SIGNALING_EQ(A,B)	E	Signaling compares equal
IEEE_SIGNALING_GE(A,B)	E	Signaling compares greater than or equal
IEEE_SIGNALING_GT(A,B)	E	Signaling compares greater than
IEEE_SIGNALING_LE(A,B)	E	Signaling compares less than or equal
IEEE_SIGNALING_LT(A,B)	E	Signaling compares less than
IEEE_SIGNALING_NE(A,B)	E	Signaling compares not equal
IEEE_SIGNBIT(X)	E	Test sign bit
IEEE_SUPPORT_DATATYPE([X])	I	Query IEEE arithmetic support
IEEE_SUPPORT_DENORMAL([X])	I	Query subnormal number support
IEEE_SUPPORT_DIVIDE([X])	I	Query IEEE division support
IEEE_SUPPORT_INF([X])	I	Query IEEE infinity support
IEEE_SUPPORT_IO([X])	I	Query IEEE formatting support
IEEE_SUPPORT_NAN([X])	I	Query IEEE NaN support
IEEE_SUPPORT_ROUNDING	T	Query IEEE rounding support
(ROUND_VALUE[,X])	T	Query IEEE rounding support

Table 36.4 (continued)

Procedure Arguments	Class	Description
IEEE_SUPPORT_SQRT([X])	I	Query IEEE square root support
IEEE_SUPPORT_SUBNORMAL([X])	I	Query subnormal number support
IEEE_SUPPORT_STANDARD([X])	I	Query IEEE standard support
IEEE_SUPPORT_UNDERFLOW	I	Query underflow control support
_CONTROL([X])	I	Query underflow control support
IEEE_UNORDERED(X,Y)	E	Whether two values are unordered
IEEE_VALUE(X,CLASS)	E	Return number in a class

Table 36.5 IEEE Exceptions module procedure summary

Procedure	Arguments	Class	Description
IEEE_GET_FLAG	(FLAG,FLAG_VALUE)	ES	Get an exception flag
IEEE_GET_HALTING_MODE	(FLAG,HALTING)	ES	Get a halting mode
IEEE_GET_MODES	(MODES)	S	Get floating-point modes
IEEE_GET_STATUS	(STATUS_VALUE)	S	Get floating-point status
IEEE_SET_FLAG	(FLAG,FLAG_VALUE)	PS	Set an exception flag
IEEE_SET_HALTING_MODE	(FLAG,HALTING)	PS	Set a halting mode
IEEE_SET_MODES	(MODES)	S	Set floating-point modes
IEEE_SET_STATUS	(STATUS_VALUE)	S	Restore floating-point status
IEEE_SUPPORT_FLAG	(FLAG [,X])	T	Query exception support
IEEE_SUPPORT_HALTING	(FLAG)	T	Query halting mode support

36.6 General Comments About the Standard

The special bit patterns provide the following:

- $+0$
- -0
- subnormal numbers in the range 1.17549421E-38 to 1.40129846E-45
- $+\infty$
- $-\infty$
- quiet NaN (Not a Number)
- signalling NaN

One of the first systems that the authors worked with that had special bit patterns set aside was the CDC 6000 range of computers that had negative indefinite and infinity. Thus the ideas are not new, as this was in the late 1970s.

The support of positive and negative zero means that certain problems can be handled correctly including:

- The evaluation of the log function which has a discontinuity at zero.
- The equation $\sqrt{1/z} = 1/z$ can be solved when $z = -1$

See also the Kahan paper *Branch Cuts for complex Elementary functions, or Much Ado About Nothing's Sign Bit* for more details.

Subnormals, which permit gradual underflow, fill the gap between 0 and the smallest normal number.

Simply stated underflow occurs when the result of an arithmetic operation is so small that it is subject to a larger than normal rounding error when stored. The existence of subnormals means that greater precision is available with these small numbers than with normal numbers. The key features of gradual underflow are:

- When underflow does occur there should never be a loss of accuracy any greater than that from ordinary roundoff.
- The operations of addition, subtraction, comparison and remainder are always exact.
- Algorithms written to take advantage of subnormal numbers have smaller error bounds than other systems.
- if x and y are within a factor of 2 then x-y is error free, which is used in a number of algorithms that increase the precision at critical regions.

The combination of positive and negative zero and subnormal numbers means that when x and y are small and x-y has been flushed to zero the evaluation of $1/(x - y)$ can be flagged and located.

Certain arithmetic operations cause problems including:

- $0 * \infty$
- $0/0$
- $\sqrt{x}$ when $x < 0$

and the support for NaN handles these cases.

The support for positive and negative infinity allows the handling of $x/0$ when x is nonzero and of either sign, and the outcome of this means that we write our programs to take the appropriate action. In some cases this would mean recalculating using another approach.

For more information see the references in the bibliography.

36.7 Resume

The above has provided a quick tour of the IEEE standard. We'll now look at what Fortran has to offer to support it.

36.8 Fortran Support for IEEE Arithmetic

Fortran first introduced support for IEEE arithmetic in ISO TR 15580. The Fortran 2003 standard integrated support into the main standard. Fortran 2018 offers more support, and for more details one should consult Chap. 17 of that document.

The intrinsic modules

- `ieee_features`
- `ieee_exceptions`
- `ieee_arithmetic`

provide support for exceptions and IEEE arithmetic. Whether the modules are provided is processor dependent. If the module `ieee_features` is provided, which of the named constants defined in this standard are included is processor dependent. The module `ieee_arithmetic` behaves as if it contained a `use` statement for `ieee_exceptions`; everything that is public in `ieee_exceptions` is public in`ieee_arithmetic`.

The first thing to consider is the degree of conformance to the IEEE standard. It is possible that not all of the features are supported. Thus the first thing to do is to run one or more test programs to determine the degree of support for a particular system.

36.9 Derived Types and Constants Defined in the Modules

The modules

- `ieee_exceptions`
- `ieee_arithmetic`
- `ieee_features`

define five derived types, whose components are all private.

36.9.1 `ieee_exceptions`

This module defines `ieee_flag_type`, for identifying a particular exception flag.

Possible values are

```
ieee_invalid
ieee_overflow
ieee_divide_by_zero
ieee_underflow
ieee_inexact
```

The module also defines the array named constants

```
ieee_usual = (/ ieee_overflow,
  ieee_divide_by_zero, ieee_invalid /)
```

```
ieee_all = (/ ieee_usual, ieee_underflow,
              ieee_inexact /)
```

```
ieee_status_type
```

The last is for saving the current floating point status.

36.9.2 `ieee_arithmetic`

This module defines `ieee_class_type`, for identifying a class of floating-point values.

Possible values are:

```
ieee_signalling_nan
ieee_quiet_nan
ieee_negative_inf
ieee_negative_normal
ieee_negative_denormal
ieee_negative_zero
ieee_positive_zero
ieee_positive_denormal
ieee_positive_normal
ieee_positive_inf
ieee_other_value
```

The module defines `ieee_round_type`, for identifying a particular rounding mode. Its only possible values are those of named constants defined in the module: `ieee_nearest`, `ieee_to_zero`, `ieee_up`, and `ieee_down` for the `ieee_modes`; and `ieee_other` for any other mode.

The elemental operator == for two values of one of these types to return true if the values are the same and false otherwise.

The elemental operator /= for two values of one of these types to return true if the values differ and false otherwise.

36.9.3 `ieee_features`

This module defines `ieee_features_type`, for expressing the need for particular `ieee_features`. Its only possible values are those of named constants defined in the module:

- `ieee_datatype`
- `ieee_denormal`
- `ieee_divide`
- `ieee_halting`
- `ieee_inexact_flag`
- `ieee_inf`
- `ieee_invalid_flag`
- `ieee_nan`
- `ieee_rounding`
- `ieee_sqrt`
- `ieee_underflow_flag`

36.9.4 Further Information

There are a number of additional sources of information.

- the Fortran standard.
- documentation that comes with your compiler.

The latter has the benefit of describing what is supported in that compiler.

36.10 Example 1: Testing IEEE Support

The first examples test basic IEEE arithmetic support.

Here is a program to illustrate the above.

```
include 'precision_module.f90'

program ch3601

  use precision_module
  use ieee_arithmetic

  implicit none

  real (sp) :: x = 1.0
```

```
  real (dp) :: y = 1.0_dp
  real (qp) :: z = 1.0_qp

  if (ieee_support_datatype(x)) then
    print *, '  32 bit IEEE support'
  end if
  if (ieee_support_datatype(y)) then
    print *, '  64 bit IEEE support'
  end if
  if (ieee_support_datatype(z)) then
    print *, ' 128 bit IEEE support'
  end if

end program ch3601
```

Table 36.6 summarises the support for a number of compilers.

Table 36.6 Compiler IEEE support for various precisions

Precision	gfortran	intel	nag	sun
32 bit IEEE support	Yes	Yes	Yes	Yes
64 bit IEEE support	Yes	Yes	Yes	Yes
128 bit IEEE support	No	Yes	No	Yes

36.11 Example 2: Testing What Flags Are Supported

Here is a program to illustrate the above.

```
include 'precision_module.f90'

program ch3602

  use precision_module
  use ieee_arithmetic

  implicit none

  real (sp) :: x = 1.0
```

```
  real (dp) :: y = 1.0_dp
  real (qp) :: z = 1.0_qp

  integer :: i

  character *20, dimension (5) :: flags = (/ &
    'IEEE_DIVIDE_BY_ZERO ', &
    'IEEE_INEXACT        ', &
    'IEEE_INVALID        ', &
    'IEEE_OVERFLOW       ', &
    'IEEE_UNDERFLOW      ' /)

  do i = 1, 5
    if (ieee_support_flag(ieee_all(i),x)) then
      write (unit=*, fmt=100) flags(i)
100   format (a20, ' 32 bit support')
    end if
    if (ieee_support_flag(ieee_all(i),y)) then
      write (unit=*, fmt=110) flags(i)
110   format (a20, ' 64 bit support')

    end if
    if (ieee_support_flag(ieee_all(i),z)) then
      write (unit=*, fmt=120) flags(i)
120   format (a20, '128 bit support')
    end if
  end do

end program ch3602
```

Here is the output from the Intel compiler.

```
IEEE_DIVIDE_BY_ZERO  32 bit support
IEEE_DIVIDE_BY_ZERO  64 bit support
IEEE_DIVIDE_BY_ZERO 128 bit support
IEEE_INEXACT         32 bit support
IEEE_INEXACT         64 bit support
IEEE_INEXACT        128 bit support
IEEE_INVALID         32 bit support
IEEE_INVALID         64 bit support
IEEE_INVALID        128 bit support
IEEE_OVERFLOW        32 bit support
IEEE_OVERFLOW        64 bit support
IEEE_OVERFLOW       128 bit support
IEEE_UNDERFLOW       32 bit support
```

```
IEEE_UNDERFLOW        64 bit support
IEEE_UNDERFLOW       128 bit support
```

36.12 Example 3: Overflow

Here is a program to illustrate the above.

```
program ch3603

  use ieee_arithmetic

  implicit none
  integer :: i
  real :: x = 1.0
  logical :: overflow_happened = .false.

  if (ieee_support_datatype(x)) then
    print *, &
      ' IEEE support for default precision'
  end if

  do i = 1, 50
    if (overflow_happened) then
      print *, ' overflow occurred '
      print *, ' program terminates'
      stop 20
    else
      print 100, i, x
100   format (' ', i3, ' ', e12.4)
    end if
    x = x*10.0
    call ieee_get_flag(ieee_overflow, &
      overflow_happened)
  end do
end program ch3603
```

36.13 Example 4: Underflow

Here is a program to illustrate the above.

```
program ch3604

  use ieee_arithmetic

  implicit none
  integer :: i
  real :: x = 1.0
  logical :: underflow_happened = .false.

  if (ieee_support_datatype(x)) then
    print *, ' IEEE arithmetic '
    print *, &
      ' is supported for default precision'
  end if

  do i = 1, 50
    if (underflow_happened) then
      print *, ' underflow occurred '
      print *, ' program terminates'
      stop 20
    else
      print 100, i, x
100   format (' ', i3, ' ', e12.4)
    end if
    x = x/10.0
    call ieee_get_flag(ieee_underflow, &
      underflow_happened)
  end do
end program ch3604
```

36.14 Example 5: Inexact Summation

Here is a program to illustrate the above.

```
program ch3605

  use ieee_arithmetic
  implicit none

  integer :: i
  real :: computed_sum
  real :: real_sum
```

```
  integer :: array_size

  logical :: inexact_happened = .false.
  integer :: allocate_status

  character *13, dimension (3) :: heading = (/ &
    '   10,000,000', '  100,000,000', &
    '1,000,000,000' /)

  real, allocatable, dimension (:) :: x

  if (ieee_support_datatype(x)) then
    print *, &
      ' IEEE support for default precision'
  end if

!            10,000,000

  array_size = 10000000

  do i = 1, 3
    write (unit=*, fmt=100) array_size, &
      heading(i)
100 format (' Array size = ', i15, 2x, a13)
    allocate (x(1:array_size), stat= &
      allocate_status)
    if (allocate_status/=0) then
      print *, ' Allocate fails, program ends'
      stop
    end if
    x = 1.0
    computed_sum = sum(x)
    call ieee_get_flag(ieee_inexact, &
      inexact_happened)
    real_sum = array_size*1.0
    write (unit=*, fmt=110) computed_sum
110 format (' Computed sum = ', e12.4)
    write (unit=*, fmt=120) real_sum
120 format (' Real sum     = ', e12.4)
    if (inexact_happened) then
      print *, ' inexact arithmetic'
      print *, ' in the summation'
      print *, ' program terminates'
      stop 20
    end if
```

```
    deallocate (x)
    array_size = array_size*10
  end do

end program ch3605
```

Here is the output from several compilers.

```
gfortran

  IEEE support for default precision
 Array size =        10000000     10,000,000
 Computed sum =   0.1000E+08
 Real sum     =   0.1000E+08
 Array size =       100000000    100,000,000
 Computed sum =   0.1000E+09
 Real sum     =   0.1000E+09
  inexact arithmetic
  in the summation
  program terminates


Intel

  IEEE support for default precision
 Array size =        10000000     10,000,000
 Computed sum =   0.1000E+08
 Real sum     =   0.1000E+08
 Array size =       100000000    100,000,000
 Computed sum =   0.1000E+09
 Real sum     =   0.1000E+09
  inexact arithmetic
  in the summation
  program terminates


nag

  IEEE support for default precision
 Array size =        10000000     10,000,000
 Computed sum =   0.1000E+08
 Real sum     =   0.1000E+08
 Array size =       100000000    100,000,000
 Computed sum =   0.1678E+08
 Real sum     =   0.1000E+09
  inexact arithmetic
```

```
  in the summation
  program terminates

sun/oracle

  IEEE support for default precision
 Array size =        10000000     10,000,000
 Computed sum =   0.1000E+08
 Real sum     =   0.1000E+08
 Array size =       100000000    100,000,000
 Computed sum =   0.1678E+08
 Real sum     =   0.1000E+09
  inexact arithmetic
  in the summation
  program terminates
```

What do you notice about the value of the computed sum?

36.15 Example 6: NAN and Other Specials

Here is a program to illustrate some additional IEEE functionality.

```
program ch3606

  use precision_module
  use ieee_arithmetic
  implicit none

  real (sp) :: x0 = 0.0
  real (dp) :: y0 = 0.0_dp
  real (qp) :: z0 = 0.0_qp

  real (sp) :: x1 = 1.0
  real (dp) :: y1 = 1.0_dp
  real (qp) :: z1 = 1.0_qp

  real (sp) :: xnan = 1.0
  real (dp) :: ynan = 1.0_dp
  real (qp) :: znan = 1.0_qp

  real (sp) :: xinfinite = 1.0
  real (dp) :: yinfinite = 1.0_dp
```

```
    real (qp) :: zinfinite = 1.0_qp

    xinfinite = x1/x0
    yinfinite = y1/y0
    zinfinite = z1/z0
    xnan = x0/x0
    ynan = y0/y0
    znan = z0/z0

    if (ieee_support_datatype(x1)) then
      print *, '  32 bit IEEE support'
      print *, '     inf ', ieee_support_inf(x1)
      print *, '     nan ', ieee_support_nan(x1)
      print *, ' 1/0 finite', ieee_is_finite( &
        xinfinite)
      print *, ' 0/0 nan', ieee_is_nan(xnan)
    end if

    if (ieee_support_datatype(y1)) then
      print *, '  64 bit IEEE support'
      print *, '     inf ', ieee_support_inf(y1)
      print *, '     nan ', ieee_support_nan(y1)
      print *, ' 1/0 finite', ieee_is_finite( &
        yinfinite)
      print *, ' 0/0 nan', ieee_is_nan(ynan)
    end if
    if (ieee_support_datatype(z1)) then
      print *, ' 128 bit IEEE support'
      print *, '     inf ', ieee_support_inf(z1)
      print *, '     nan ', ieee_support_nan(z1)
      print *, ' 1/0 finite', ieee_is_finite( &
        zinfinite)
      print *, ' 0/0 nan', ieee_is_nan(znan)
    end if

  end program ch3606
```

36.16 Summary

Compiler support in this area is now quite widespread as the above examples have shown.

36.17 Bibliography

Hauser J.R., Handling Floating Point Exceptions in Numeric programs, ACM Transaction on programming Languages and Systems, Vol. 18, No. 2, March 1996, pp. 139–174.

- The paper looks at a number of techniques for handling floating point exceptions in numeric code. One of the conclusions is for better structured support for floating point exception handling in new programming languages, or of course better standards for existing languages.

IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-2008, Institute of Electrical and Electronic Engineers Inc.

- The formal definition of IEEE 754. This is available for purchase as both a pdf and printed version - see the address below.

```
http://www.techstreet.com/standards/
IEEE/754_2008?product_id=1745167
```

This standard specifies formats and methods for floating-point arithmetic in computer systems: standard and extended functions with single, double, extended, and extendable precision, and recommends formats for data interchange. Exception conditions are defined and standard handling of these conditions is specified. Keywords: 754-2008, arithmetic, binary, computer, decimal, exponent, floating-point, format, interchange, NaN,number, rounding, significand, subnormal. Product Code(s): STDPD95802,STD95802

Knuth D., Seminumerical Algorithms, Addison-Wesley, 1969.

- There is a coverage of floating point arithmetic, multiple precision arithmetic, radix conversion and rational arithmetic.

Sun, Numerical Computation Guide, SunPro.

- Very good coverage of the numeric formats for IEEE Standard 754 for Binary Floating-Point Arithmetic. All SunPro compiler products support the features of the IEEE 754 standard.

36.17.1 Web-Based Sources

- Differences Among IEEE 754 Implementations. The material in this paper will eventually be included in the Sun Numerical Computation Guide as an addendum to Appendix C, David Goldberg's What Every Computer Scientist Should Know about Floating Point Arithmetic.

http://docs.oracle.com/cd/
E19422-01/819-3693/819-3693.pdf
https://docs.oracle.com/en/

- The Numerical Computation Guide can be browsed on-line or downloaded as a pdf file. The last time we checked it was 294 pages. Good source of information if you have Sun equipment.

http://www-users.math.umn.edu/
~arnold/disasters/ariane.html

- The Explosion of the Ariane 5: A 64-bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,768, the largest integer storeable in a 16-bit signed integer, and thus the conversion failed.

36.17.2 Hardware Sources

Amd - Visit

https://developer.amd.com/resources/

for details of the AMD manuals. The following five manuals are available for download as pdf's from the above site.

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming
- AMD64 Architecture Programmer's Manual Volume 2: System Programming
- AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions
- AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256 bit media instructions
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions

Intel - Visit

https://software.intel.com/en-us/articles/intel-sdm

for a list of manuals. The following three manuals are available for download as pdf's from the above site.

- Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture
- Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 2A and 2B: Instruction Set Reference, A-Z.
- Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2

Osbourne A., Kane G., 4-bit and 8-bit Microprocessor Handbook, Osbourne and McGraw Hill, 1981.

- Good source of information on 4-bit and 8-bit microprocessors.

Osbourne A., Kane G., 16-Bit Microprocessor Handbook, Osbourne and McGraw Hill, 1981.

- Ditto 16-bit microprocessors.

Bhandarkar D.P., Alpha Implementations and Architecture: Complete Reference and Guide, Digital Press, 1996.

- Looks at some of the trade-offs and design philosophy behind the alpha chip. The author worked with VAX, MicroVAX and VAX vectors as well as the Prism. Also looks at the GEM compiler technology that DEC/Compaq use.

Various companies home pages.

```
http://www.ibm.com/
IBM home page.
http://www.sgi.com/
Silicon Graphics home page.
```

36.17.3 Operating Systems

Deitel H.M., An Introduction to Operating Systems, Addison-Wesley, 1990.

- The revised first edition includes case studies of UNIX, VMS, CP/M, MVS and VM. The second edition adds OS/2 and the Macintosh operating systems. There is a coverage of hardware, software, firmware, process management, process concepts, asynchronous concurrent processes, concurrent programming, deadlock and indefinite postponement, storage management, real storage, virtual storage, processor management, distributed computing, disk performance optimisation, file and database systems, performance, coprocessors, risc, data flow, analytic modelling, networks, security and it concludes with case studies of the these operating systems. The book is well written and an easy read.

36.18 Problem

36.1 Compile and run each of the examples in this chapter with your compiler(s). If you have access to more than one compiler do the compilers behave in the same way?