Coverage Analysis Report of main parts from Dinkumware Library and AEABI

Version:	1.0
Date:	2019-06-29
Status:	Draft **/** Reviewed / **Final**
Author:	Dr. Oscar Slotosch, Thomas Escherle
File:	Manual_analysis_of_code_coverage.docx
Size:	78 Pages

VALIDAS

1 History

Version	Date	Status	Autor	Change
0.1	2019/05/28	Draft	Escherle	Created intial version
0.2	2019/05/28	Draft	Escherle	Added analysis for function _Xp_getw
0.3	2019/05/29	Draft	Escherle	Added function imaxdiv
0.4	2019/05/29	Draft	Escherle	Added explanation for function imaxdiv
0.5	2019/06/02	Draft	Slotosch	recomputed with updated analysis
0.6	2019/06/03	Draft	Escherle	imaxdiv analysed
0.7	2019/06/03	Draft	Escherle	Added analysis of function __aeabi_d2uiz
0.8	2019/06/05	Draft	Escherle	Analysis of function __aeabi_dadd
0.9	2019/06/05	Draft	Escherle	Added analysis of several functions
0.10	2019/06/06	Draft	Escherle	Added analysis of several functions
0.11	2019/06/07	Draft	Escherle	Analysis of __udivmoddi4
0.12	2019/06/11	Draft	Escherle	Added analysis of function wideRightShiftWithSticky
0.13	2019/06/12	Draft	Escherle	Analysis of code coverage of __divdi3
0.14	2019/06/14	Draft	Slotosch	updated coverage for sinf
0.15	2019/06/14	Draft	Slotosch	Zwischenstand atanhf
0.16	2019/06/14	Draft	Escherle	Restructured chapter 19 / 20
0.17	2019/06/15	Draft	Slotosch	Analyzed atanh/f
0.18	2019/06/16	Draft	Slotosch	updated pow analysis
0.19	2019/06/16	Draft	Slotosch	finished coverage analysis of pow
0.20	2019/06/16	Draft	Slotosch	added powf
0.21	2019/06/17	Draft	Slotosch	updated quad (still not complete)
0.22	2019/06/17	Draft	Escherle	Removed chapters about __udivmoddi4, since coverage gaps were closed
0.23	2019/06/18	Draft	Escherle	Removed remarks about added test cases
0.24	2019/06/18	Draft	Slotosch	redone Quad Analysis
0.25	2019/06/19	Draft	Escherle	Chapter "List of functions to be covered with computed coverage" added
0.26	2019/06/19	Draft	Escherle	Added sub-section for manually analysed sub-functions

0.27	2019/06/25	Draft	Escherle	Added analysis of function _Getmem
0.28	2019/06/29	Draft	Slotosch	Added last functions _Xp_addh and _FExp
0.9	2019/06/29	Reviewed	Slotosch	Reviewed and completed
1.0	2019/06/29	Final	Slotosch	Finalized document

Contents

2 Introduction

This document identifies which functions need to be covered including their dependent functions. It complements the MC/DC code coverage reports generated by CTC. Functions which show an MC/DC coverage value less than 100 % in the CTC coverage report are analyzed within this document and safety analysis is provided for the untested parts.

This document is structured as follows:

- In chapter 3 all functions - as well as the functions they depend on - are identified which need to be qualified and for which MC/DC coverage needs to be measured
- Chapter 4 contains the coverage results measured by CTC
- Based on the coverage results, the functions which have code coverage gaps (i.e. coverage value < 100 %) and therefore need further analysis, are identified in chapter 5.
- Chapter 6 provides the analysis of the coverage gaps and explanations for them
- For some functions a more detailed analysis of the coverage gaps is required. These can be found in chapter 7.

Most math functions have two variants (single/double variant) and are derived from the same source, hence it suffices to analyze the source once.

3 List of all functions with their dependencies

The following lists identify all 183 functions which need to be qualified for ASIL D as well as the functions they depend on. For all these functions the MC/DC coverage is measured. Refer to chapter 4 for the measured MC/DC values.

Main functions for which the code coverage needs to be measured:

- __aeabi_cdcmpeq
- __aeabi_cdcmple
- __aeabi_cdrcmple
- __aeabi_cfcmpeq
- __aeabi_cfcmple
- __aeabi_cfrcmple
- __aeabi_d2f
- __aeabi_d2h
- __aeabi_d2iz
- __aeabi_d2lz
- __aeabi_d2uiz
- __aeabi_d2ulz
- __aeabi_dadd
- __aeabi_dcmpeq
- __aeabi_dcmpge
- __aeabi_dcmpgt
- __aeabi_dcmple
- __aeabi_dcmplt
- __aeabi_dcmpun
- __aeabi_ddiv
- __aeabi_dmul
- __aeabi_drsub
- __aeabi_dsub
- __aeabi_f2d
- __aeabi_f2h
- __aeabi_f2iz
- __aeabi_f2lz
- __aeabi_f2uiz
- __aeabi_f2ulz
- __aeabi_fadd
- __aeabi_fcmpeq
- __aeabi_fcmpge
- __aeabi_fcmpgt
- __aeabi_fcmple
- __aeabi_fcmplt
- __aeabi_fcmpun
- __aeabi_fdiv
- __aeabi_fmul
- __aeabi_frsub

- __aeabi_fsub
- __aeabi_h2f
- __aeabi_i2d
- __aeabi_i2f
- __aeabi_idivmod
- __aeabi_idiv
- __aeabi_idiv0
- __aeabi_l2d
- __aeabi_l2f
- __aeabi_lasr
- __aeabi_lcmp
- __aeabi_ldivmod
- __aeabi_ldiv0
- __aeabi_llsl
- __aeabi_llsr
- __aeabi_lmul
- __aeabi_memclr
- __aeabi_memclr4
- __aeabi_memclr8
- __aeabi_memcpy
- __aeabi_memcpy4
- __aeabi_memcpy8
- __aeabi_memmove
- __aeabi_memmove4
- __aeabi_memmove8
- __aeabi_memset
- __aeabi_memset4
- __aeabi_memset8
- __aeabi_ui2d
- __aeabi_ui2f
- __aeabi_uidiv
- __aeabi_uidivmod
- __aeabi_ul2d
- __aeabi_ul2f
- __aeabi_ulcmp
- __aeabi_uldivmod
- abs
- acos
- acosf
- acosh
- acoshf
- add
- addf
- asin
- asinf
- asinh
- asinhf
- atan

- atan2
- atan2f
- atanf
- atanh
- atanhf
- cbrt
- cbrtf
- ceil
- ceilf
- cos
- cosf
- cosh
- coshf
- divide
- dividef
- exp
- expm1f
- exp2
- exp2f
- expm1
- expf
- fabs
- fabsf
- fdim
- fdimf
- floor
- floorf
- fma
- fmaf
- fmax
- fmaxf
- fmin
- fminf
- fmod
- fmodf
- HUGE_VAL
- hypot
- hypotf
- ilogb
- ilogbf
- imaxabs
- imaxdiv
- ldexp
- ldexpf
- ldiv
- llabs
- lldiv
- log

- log10
- log10f
- log1p
- log1pf
- log2
- log2f
- logb
- logbf
- logf
- lrint
- lrintf
- lround
- lroundf
- modf
- modff
- multiply
- multiplyf
- nextafter
- nextafterf
- NULL
- pow
- powf
- remainder
- remainderf
- rint
- rintf
- round
- roundf
- scalbln
- scalblnf
- scalbn
- scalbnf
- sin
- sinf
- sinh
- sinhf
- size_t
- sqrt
- sqrtf
- srand
- subtract
- subtractf
- tan
- tanf
- tanh
- tanhf
- trunc
- truncf

Sub-functions for which the code coverage needs to be measured:

- __adddf3
- __addsf3
- __addXf3__
- __cmpdi2
- __divdf3
- __divsf3
- __extendhfsf2
- __extendsfdf2
- __extendXfYf2__
- __fixdfdi
- __fixdfsi
- __fixint
- __fixsfdi
- __fixsfsi
- __fixuint
- __fixunsdfdi
- __fixunsdfsi
- __fixunssfdi
- __fixunssfsi
- __floatdidf
- __floatdisf
- __floatsidf
- __floatsisf
- __floatundidf
- __floatundisf
- __floatunsidf
- __floatunsisf
- __muldf3
- __mulsf3
- __mulXf3__
- __Remquo_subtract
- __subdf3
- __subsf3
- __truncdfhf2
- __truncdfsf2
- __truncsfhf2
- __truncXfYf2__
- __ucmpdi2
- __unorddf2
- _Atan
- _Atan2_divide
- _Cosh
- _Dint
- _Dnorm
- _Dscale
- _Dscalex

- _Dtest
- _Dunscale
- _Exp
- _Expm1_approx
- _F_Remquo_subtract
- _FAtan
- _FAtan2_divide
- _FCosh
- _FDint
- _FDnorm
- _FDscale
- _FDscalex
- _FDtest
- _FDunscale
- _Feraise
- _FExp
- _FExpm1_approx
- _FHypot
- _FLog
- _FLogpoly
- _Force_raise
- _FPmsw
- _FPow
- _FQuad
- _FQuad_multiply
- _FRint
- _FSinh
- _FSinh_small
- _FSinx
- _FTan
- _FTan_approx
- _FXp_addh
- _FXp_addx
- _FXp_getw
- _FXp_mulh
- _FXp_setw
- _Hypot
- _LDtest
- _Log
- _Logpoly
- _Pmsw
- _Pow
- _Quad
- _Quad_multiply
- _Rint
- _Sinh
- _Sinh_small
- _Sinx

- _Tan
- _Tan_approx
- _Tanh_approx
- _Xp_addh
- _Xp_addx
- _Xp_getw
- _Xp_mulh
- _Xp_setw
- copysign
- copysignf
- dstFromRep
- fegetenv
- feraiseexcept
- fesetenv
- fromRep
- memcpy
- nearbyint
- nearbyintf
- nexttoward
- nexttowardf
- normalize
- remquo
- remquof
- rep_clz
- srcToRep
- toRep
- wideLeftShift
- wideMultiply
- wideRightShiftWithSticky

4 List of functions to be covered with computed coverage

The following list contains the functions which need to be covered and the MC/DC coverage values measured by CTC.

- ***TER 100 % (2/ 2) of FUNCTION __adddf3()
- ***TER 100 % (2/ 2) of FUNCTION __addsf3()
- ***TER 98 % (57/ 58) of FUNCTION __addXf3__()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2h()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2iz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2lz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2uiz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_d2ulz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_dadd()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_dcmpun()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ddiv()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_dmul()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_drsub()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_dsub()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2d()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2h()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2iz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2lz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2uiz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_f2ulz()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_fadd()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_fdiv()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_fmul()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_frsub()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_fsub()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_h2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_i2d()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_i2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_idiv0()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_l2d()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_l2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_lcmp()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ldiv0()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ui2d()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ui2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ul2d()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ul2f()
- ***TER 100 % (2/ 2) of FUNCTION __aeabi_ulcmp()
- ***TER 100 % (14/ 14) of FUNCTION __cmpdi2()
- ***TER 100 % (42/ 42) of FUNCTION __divdf3()
- ***TER 100 % (42/ 42) of FUNCTION __divsf3()
- ***TER 100 % (2/ 2) of FUNCTION __extendhfsf2()

- ***TER 100 % (2/ 2) of FUNCTION __extendsfdf2()
- ***TER 100 % (8/ 8) of FUNCTION __extendXfYf2__()
- ***TER 100 % (5/ 5) of FUNCTION __fixdfdi()
- ***TER 100 % (2/ 2) of FUNCTION __fixdfsi()
- ***TER 100 % (15/ 15) of FUNCTION __fixint()
- ***TER 100 % (5/ 5) of FUNCTION __fixsfdi()
- ***TER 100 % (2/ 2) of FUNCTION __fixsfsi()
- ***TER 100 % (15/ 15) of FUNCTION __fixuint()
- ***TER 100 % (5/ 5) of FUNCTION __fixunsdfdi()
- ***TER 100 % (2/ 2) of FUNCTION __fixunsdfsi()
- ***TER 100 % (5/ 5) of FUNCTION __fixunssfdi()
- ***TER 100 % (2/ 2) of FUNCTION __fixunssfsi()
- ***TER 100 % (2/ 2) of FUNCTION __floatdidf()
- ***TER 100 % (14/ 14) of FUNCTION __floatdisf()
- ***TER 100 % (7/ 7) of FUNCTION __floatsidf()
- ***TER 100 % (13/ 13) of FUNCTION __floatsisf()
- ***TER 100 % (2/ 2) of FUNCTION __floatundidf()
- ***TER 100 % (14/ 14) of FUNCTION __floatundisf()
- ***TER 100 % (5/ 5) of FUNCTION __floatunsidf()
- ***TER 100 % (11/ 11) of FUNCTION __floatunsisf()
- ***TER 100 % (2/ 2) of FUNCTION __muldf3()
- ***TER 100 % (2/ 2) of FUNCTION __mulsf3()
- ***TER 100 % (48/ 48) of FUNCTION __mulXf3__()
- ***TER 100 % (6/ 6) of FUNCTION __Remquo_subtract()
- ***TER 100 % (2/ 2) of FUNCTION __subdf3()
- ***TER 100 % (2/ 2) of FUNCTION __subsf3()
- ***TER 100 % (2/ 2) of FUNCTION __truncdfhf2()
- ***TER 100 % (2/ 2) of FUNCTION __truncdfsf2()
- ***TER 100 % (2/ 2) of FUNCTION __truncsfhf2()
- ***TER 100 % (18/ 18) of FUNCTION __truncXfYf2__()
- ***TER 100 % (14/ 14) of FUNCTION __ucmpdi2()
- ***TER 100 % (2/ 2) of FUNCTION __unorddf2()
- ***TER 100 % (16/ 16) of FUNCTION _Atan()
- ***TER 100 % (4/ 4) of FUNCTION _Atan2_divide()
- ***TER 75 % (18/ 24) of FUNCTION _Cosh()
- ***TER 100 % (25/ 25) of FUNCTION _Dint()
- ***TER 100 % (14/ 14) of FUNCTION _Dnorm()
- ***TER 100 % (2/ 2) of FUNCTION _Dscale()
- ***TER 61 % (42/ 69) of FUNCTION _Dscalex()
- ***TER 100 % (16/ 16) of FUNCTION _Dtest()
- ***TER 100 % (12/ 12) of FUNCTION _Dunscale()
- ***TER 53 % (26/ 49) of FUNCTION _Exp()
- ***TER 100 % (2/ 2) of FUNCTION _Expm1_approx()
- ***TER 100 % (6/ 6) of FUNCTION _F_Remquo_subtract()
- ***TER 100 % (18/ 18) of FUNCTION _FAtan()
- ***TER 100 % (4/ 4) of FUNCTION _FAtan2_divide()
- ***TER 75 % (18/ 24) of FUNCTION _FCosh()
- ***TER 100 % (22/ 22) of FUNCTION _FDint()

- ***TER 100 % (12/ 12) of FUNCTION _FDnorm()
- ***TER 100 % (2/ 2) of FUNCTION _FDscale()
- ***TER 63 % (41/ 65) of FUNCTION _FDscalex()
- ***TER 100 % (14/ 14) of FUNCTION _FDtest()
- ***TER 100 % (12/ 12) of FUNCTION _FDunscale()
- ***TER 83 % (10/ 12) of FUNCTION _Feraise()
- ***TER 59 % (29/ 49) of FUNCTION _FExp()
- ***TER 100 % (2/ 2) of FUNCTION _FExpm1_approx()
- ***TER 100 % (30/ 30) of FUNCTION _FHypot()
- ***TER 100 % (20/ 20) of FUNCTION _FLog()
- ***TER 100 % (2/ 2) of FUNCTION _FLogpoly()
- ***TER   0 % (0/ 6) of FUNCTION _Force_raise()
- ***TER 100 % (2/ 2) of FUNCTION _FPmsw()
- ***TER 59 % (61/104) of FUNCTION _FPow()
- ***TER 66 % (29/ 44) of FUNCTION _FQuad()
- ***TER   0 % (0/ 4) of FUNCTION _FQuad_multiply()
- ***TER 50 % (15/ 30) of FUNCTION _FRint()
- ***TER 74 % (29/ 39) of FUNCTION _FSinh()
- ***TER 100 % (2/ 2) of FUNCTION _FSinh_small()
- ***TER 96 % (22/ 23) of FUNCTION _FSinx()
- ***TER 96 % (22/ 23) of FUNCTION _FTan()
- ***TER 100 % (5/ 5) of FUNCTION _FTan_approx()
- ***TER 79 % (55/ 70) of FUNCTION _FXp_addh()
- ***TER 100 % (6/ 6) of FUNCTION _FXp_addx()
- ***TER 76 % (16/ 21) of FUNCTION _FXp_getw()
- ***TER 70 % (21/ 30) of FUNCTION _FXp_mulh()
- ***TER 48 % (10/ 21) of FUNCTION _FXp_setw()
- ***TER 100 % (30/ 30) of FUNCTION _Hypot()
- ***TER 100 % (2/ 2) of FUNCTION _LDtest()
- ***TER 100 % (20/ 20) of FUNCTION _Log()
- ***TER 100 % (2/ 2) of FUNCTION _Logpoly()
- ***TER 100 % (2/ 2) of FUNCTION _Pmsw()
- ***TER 89 % (93/104) of FUNCTION _Pow()
- ***TER 66 % (29/ 44) of FUNCTION _Quad()
- ***TER   0 % (0/ 4) of FUNCTION _Quad_multiply()
- ***TER 50 % (15/ 30) of FUNCTION _Rint()
- ***TER 74 % (29/ 39) of FUNCTION _Sinh()
- ***TER 100 % (2/ 2) of FUNCTION _Sinh_small()
- ***TER 96 % (22/ 23) of FUNCTION _Sinx()
- ***TER 90 % (19/ 21) of FUNCTION _Tan()
- ***TER 100 % (5/ 5) of FUNCTION _Tan_approx()
- ***TER 100 % (2/ 2) of FUNCTION _Tanh_approx()
- ***TER 77 % (54/ 70) of FUNCTION _Xp_addh()
- ***TER 100 % (6/ 6) of FUNCTION _Xp_addx()
- ***TER 76 % (16/ 21) of FUNCTION _Xp_getw()
- ***TER 70 % (21/ 30) of FUNCTION _Xp_mulh()
- ***TER 76 % (16/ 21) of FUNCTION _Xp_setw()
- ***TER 100 % (4/ 4) of FUNCTION abs()

- ***TER 100 % (13/ 13) of FUNCTION acos()
- ***TER 100 % (16/ 16) of FUNCTION acosf()
- ***TER 100 % (19/ 19) of FUNCTION acosh()
- ***TER 100 % (22/ 22) of FUNCTION acoshf()
- ***TER 100 % (19/ 19) of FUNCTION asin()
- ***TER 100 % (24/ 24) of FUNCTION asinf()
- ***TER 100 % (15/ 15) of FUNCTION asinh()
- ***TER 100 % (15/ 15) of FUNCTION asinhf()
- ***TER 100 % (14/ 14) of FUNCTION atan()
- ***TER 100 % (28/ 28) of FUNCTION atan2()
- ***TER 100 % (28/ 28) of FUNCTION atan2f()
- ***TER 100 % (14/ 14) of FUNCTION atanf()
- ***TER 96 % (24/ 25) of FUNCTION atanh()
- ***TER 96 % (24/ 25) of FUNCTION atanhf()
- ***TER 100 % (15/ 15) of FUNCTION cbrt()
- ***TER 100 % (15/ 15) of FUNCTION cbrtf()
- ***TER 100 % (4/ 4) of FUNCTION ceil()
- ***TER 100 % (4/ 4) of FUNCTION ceilf()
- ***TER 75 % (3/ 4) of FUNCTION copysign()
- ***TER 75 % (3/ 4) of FUNCTION copysignf()
- ***TER 100 % (2/ 2) of FUNCTION cos()
- ***TER 100 % (2/ 2) of FUNCTION cosf()
- ***TER 100 % (2/ 2) of FUNCTION cosh()
- ***TER 100 % (2/ 2) of FUNCTION coshf()
- ***TER 100 % (2/ 2) of FUNCTION dstFromRep()
- ***TER 100 % (11/ 11) of FUNCTION exp()
- ***TER 100 % (21/ 21) of FUNCTION exp2()
- ***TER 100 % (21/ 21) of FUNCTION exp2f()
- ***TER 100 % (11/ 11) of FUNCTION expf()
- ***TER 100 % (28/ 28) of FUNCTION expm1()
- ***TER 100 % (28/ 28) of FUNCTION expm1f()
- ***TER 100 % (5/ 5) of FUNCTION fabs()
- ***TER 100 % (5/ 5) of FUNCTION fabsf()
- ***TER 100 % (11/ 11) of FUNCTION fdim()
- ***TER 100 % (11/ 11) of FUNCTION fdimf()
- ***TER 100 % (2/ 2) of FUNCTION fegetenv()
- ***TER 67 % (4/ 6) of FUNCTION feraiseexcept()
- ***TER 100 % (2/ 2) of FUNCTION fesetenv()
- ***TER 100 % (4/ 4) of FUNCTION floor()
- ***TER 100 % (4/ 4) of FUNCTION floorf()
- ***TER 100 % (45/ 45) of FUNCTION fma()
- ***TER 80 % (36/ 45) of FUNCTION fmaf()
- ***TER 100 % (14/ 14) of FUNCTION fmax()
- ***TER 100 % (14/ 14) of FUNCTION fmaxf()
- ***TER 100 % (14/ 14) of FUNCTION fmin()
- ***TER 100 % (14/ 14) of FUNCTION fminf()
- ***TER 97 % (34/ 35) of FUNCTION fmod()
- ***TER 97 % (34/ 35) of FUNCTION fmodf()

- ***TER 100 % (2/ 2) of FUNCTION fromRep()
- ***TER 57 % (4/ 7) of FUNCTION hypot()
- ***TER 57 % (4/ 7) of FUNCTION hypotf()
- ***TER 100 % (9/ 9) of FUNCTION ilogb()
- ***TER 100 % (9/ 9) of FUNCTION ilogbf()
- ***TER 100 % (4/ 4) of FUNCTION imaxabs()
- ***TER 43 % (3/ 7) of FUNCTION imaxdiv()
- ***TER 100 % (9/ 9) of FUNCTION ldexp()
- ***TER 100 % (9/ 9) of FUNCTION ldexpf()
- ***TER 43 % (3/ 7) of FUNCTION ldiv()
- ***TER 100 % (4/ 4) of FUNCTION llabs()
- ***TER 43 % (3/ 7) of FUNCTION lldiv()
- ***TER 100 % (2/ 2) of FUNCTION log()
- ***TER 100 % (2/ 2) of FUNCTION log10()
- ***TER 100 % (2/ 2) of FUNCTION log10f()
- ***TER 100 % (21/ 21) of FUNCTION log1p()
- ***TER 100 % (21/ 21) of FUNCTION log1pf()
- ***TER 100 % (2/ 2) of FUNCTION log2()
- ***TER 100 % (2/ 2) of FUNCTION log2f()
- ***TER 100 % (9/ 9) of FUNCTION logb()
- ***TER 100 % (9/ 9) of FUNCTION logbf()
- ***TER 100 % (2/ 2) of FUNCTION logf()
- ***TER 100 % (13/ 13) of FUNCTION lrint()
- ***TER 100 % (13/ 13) of FUNCTION lrintf()
- ***TER 100 % (11/ 11) of FUNCTION lround()
- ***TER 100 % (11/ 11) of FUNCTION lroundf()
- ***TER 100 % (4/ 4) of FUNCTION memcpy()
- ***TER 100 % (13/ 13) of FUNCTION modf()
- ***TER 100 % (13/ 13) of FUNCTION modff()
- ***TER 43 % (3/ 7) of FUNCTION nearbyint()
- ***TER 43 % (3/ 7) of FUNCTION nearbyintf()
- ***TER 100 % (2/ 2) of FUNCTION nextafter()
- ***TER 100 % (2/ 2) of FUNCTION nextafterf()
- ***TER 97 % (32/ 33) of FUNCTION nexttoward()
- ***TER 96 % (27/ 28) of FUNCTION nexttowardf()
- ***TER 100 % (2/ 2) of FUNCTION normalize()
- ***TER   0 % (0/ 2) of FUNCTION normalize()
- ***TER 100 % (2/ 2) of FUNCTION pow()
- ***TER 100 % (2/ 2) of FUNCTION powf()
- ***TER 100 % (2/ 2) of FUNCTION remainder()
- ***TER 100 % (2/ 2) of FUNCTION remainderf()
- ***TER 92 % (47/ 51) of FUNCTION remquo()
- ***TER 92 % (47/ 51) of FUNCTION remquof()
- ***TER 100 % (5/ 5) of FUNCTION rep_clz()
- ***TER 100 % (9/ 9) of FUNCTION rint()
- ***TER 100 % (9/ 9) of FUNCTION rintf()
- ***TER 100 % (11/ 11) of FUNCTION round()
- ***TER 100 % (11/ 11) of FUNCTION roundf()

- ***TER 100 % (9/ 9) of FUNCTION scalbln()
- ***TER 100 % (9/ 9) of FUNCTION scalblnf()
- ***TER 100 % (9/ 9) of FUNCTION scalbn()
- ***TER 100 % (9/ 9) of FUNCTION scalbnf()
- ***TER 100 % (2/ 2) of FUNCTION sin()
- ***TER 100 % (2/ 2) of FUNCTION sinf()
- ***TER 100 % (2/ 2) of FUNCTION sinh()
- ***TER 100 % (2/ 2) of FUNCTION sinhf()
- ***TER 100 % (15/ 15) of FUNCTION sqrt()
- ***TER 100 % (15/ 15) of FUNCTION sqrtf()
- ***TER 100 % (2/ 2) of FUNCTION srand()
- ***TER 100 % (2/ 2) of FUNCTION srcToRep()
- ***TER 100 % (2/ 2) of FUNCTION tan()
- ***TER 100 % (2/ 2) of FUNCTION tanf()
- ***TER 100 % (23/ 23) of FUNCTION tanh()
- ***TER 100 % (21/ 21) of FUNCTION tanhf()
- ***TER 100 % (2/ 2) of FUNCTION toRep())
- ***TER 100 % (2/ 2) of FUNCTION trunc()
- ***TER 100 % (2/ 2) of FUNCTION truncf()
- ***TER 100 % (2/ 2) of FUNCTION wideLeftShift()
- ***TER 100 % (2/ 2) of FUNCTION wideMultiply()
- ***TER 50 % (3/ 6) of FUNCTION wideRightShiftWithSticky()

5 List of functions to be analyzed

The following list contains the functions which have only a MC/DC coverage value less than 100 % according to the CTC report. For each function a reference to the manual analysis in chapter 6 is provided. If no such reference is given, the analysis is still pending.

MC/DC coverage / Function name	Item no. in chapter 6
***TER 98 % (57/ 58) of FUNCTION __addXf3__()	1
***TER 75 % (18/ 24) of FUNCTION _Cosh()	88
***TER 61 % (42/ 69) of FUNCTION _Dscalex()	5
***TER 53 % (26/ 49) of FUNCTION _Exp()	???
***TER 75 % (18/ 24) of FUNCTION _FCosh()	89
***TER 63 % (41/ 65) of FUNCTION _FDscalex()	6
***TER 83 % (10/ 12) of FUNCTION _Feraise()	3
***TER 59 % (29/ 49) of FUNCTION _FExp()	???
***TER 0 % (0/ 6) of FUNCTION _Force_raise()	2
***TER 59 % (61/104) of FUNCTION _FPow()	91
***TER 66 % (29/ 44) of FUNCTION _FQuad()	108
***TER 0 % (0/ 4) of FUNCTION _FQuad_multiply()	26
***TER 50 % (15/ 30) of FUNCTION _FRint()	39
***TER 74 % (29/ 39) of FUNCTION _FSinh()	107
***TER 96 % (22/ 23) of FUNCTION _FSinx()	33
***TER 96 % (22/ 23) of FUNCTION _FTan()	110
***TER 79 % (55/ 70) of FUNCTION _FXp_addh()	68
***TER 76 % (16/ 21) of FUNCTION _FXp_getw()	31
***TER 70 % (21/ 30) of FUNCTION _FXp_mulh()	69

***TER 48 % (10/ 21) of FUNCTION _FXp_setw()	30
***TER 89 % (93/104) of FUNCTION _Pow()	90
***TER 66 % (29/ 44) of FUNCTION _Quad()	103
***TER 0 % (0/ 4) of FUNCTION _Quad_multiply()	26
***TER 50 % (15/ 30) of FUNCTION _Rint()	37
***TER 74 % (29/ 39) of FUNCTION _Sinh()	106
***TER 96 % (22/ 23) of FUNCTION _Sinx()	32
***TER 90 % (19/ 21) of FUNCTION _Tan()	109
***TER 77 % (54/ 70) of FUNCTION _Xp_addh()	???
***TER 76 % (16/ 21) of FUNCTION _Xp_getw()	29
***TER 70 % (21/ 30) of FUNCTION _Xp_mulh()	67
***TER 76 % (16/ 21) of FUNCTION _Xp_setw()	28
***TER 96 % (24/ 25) of FUNCTION atanh()	46
***TER 96 % (24/ 25) of FUNCTION atanhf()	47
***TER 75 % (3/ 4) of FUNCTION copysign()	35
***TER 75 % (3/ 4) of FUNCTION copysignf()	36
***TER 67 % (4/ 6) of FUNCTION feraiseexcept()	4
***TER 80 % (36/ 45) of FUNCTION fmaf()	44
***TER 97 % (34/ 35) of FUNCTION fmod()	7
***TER 97 % (34/ 35) of FUNCTION fmodf()	8
***TER 57 % (4/ 7) of FUNCTION hypot()	74
***TER 57 % (4/ 7) of FUNCTION hypotf()	75
***TER 43 % (3/ 7) of FUNCTION imaxdiv()	45
***TER 43 % (3/ 7) of FUNCTION ldiv()	41

***TER 43 % (3/ 7) of FUNCTION lldiv()	42
***TER 43 % (3/ 7) of FUNCTION nearbyint()	38
***TER 43 % (3/ 7) of FUNCTION nearbyintf()	40
***TER 97 % (32/ 33) of FUNCTION nexttoward()	81
***TER 96 % (27/ 28) of FUNCTION nexttowardf()	82
***TER 92 % (47/ 51) of FUNCTION remquo()	83
***TER 92 % (47/ 51) of FUNCTION remquof()	84
***TER 50 % (3/ 6) of FUNCTION wideRightShiftWithSticky()	80

The following list contains assembly functions. Since no coverage reports could be generated by CTC, these functions were analyzed manually. Refer to the mentioned items in chapter 6.

Assembly function name	Item in chapter 6
__aeabi_cdcmpeq	86
__aeabi_cdcmple	85
__aeabi_cdrcmple	87
__aeabi_cfcmpeq	100
__aeabi_cfcmple	98
__aeabi_cfrcmple	99
__aeabi_dcmpeq	60
__aeabi_dcmpge	63
__aeabi_dcmpgt	64
__aeabi_dcmple	62
__aeabi_dcmplt	61
__aeabi_dcmpun	65
__aeabi_fcmpeq	95
__aeabi_fcmpge	101
__aeabi_fcmpgt	102
__aeabi_fcmple	96
__aeabi_fcmplt	97
__aeabi_fcmpun	94
__aeabi_idiv	72
__aeabi_idivmod	70
__aeabi_lasr	76
__aeabi_ldivmod	92
__aeabi_llsl	77
__aeabi_llsr	78

__aeabi_lmul	79
__aeabi_memclr	48
__aeabi_memclr4	49
__aeabi_memclr8	50
__aeabi_memcpy	57
__aeabi_memcpy4	58
__aeabi_memcpy8	59
__aeabi_memmove	54
__aeabi_memmove4	55
__aeabi_memmove8	56
__aeabi_memset	51
__aeabi_memset4	52
__aeabi_memset8	53
__aeabi_uidiv	73
__aeabi_uidivmod	71
__aeabi_uldivmod	93

6 List of manually analyzed functions

The following list contains functions where not 100 % MC/DC was reached, but they were manually analyzed and an explanation is given why these coverage gaps remain.

1. Manually Analyzed __addXf3__: The code coverage of the function is complete. The branches remaining cannot be reached. See chapters 7.10, 7.11, 7.12, 7.14, 7.15 and 7.17
2. Manually Analyzed _Force_raise(): This is not called, since for FPP_ARM there are only 5 Exceptions (<0x10) which does not exceed the number _FE_EXMASK_OFF=8)
3. Manually Analyzed _Feraise(): has empty-exception checks that are not used, hence OK
4. Manually Analyzed feraiseexcept(): This is not executed, since for FPP_ARM there are only 5 Exceptions (<0x10) which does not exceed the number _FE_EXMASK_OFF=8)
5. Manually Analyzed _Dscalex(): This is used only four rounding mode 4, not called with overflows and used the ? operator. MCDC terms have dependent vaiables
6. Manually Analyzed _FDscalex(): This is used only four rounding mode 4, not called with overflows and used the ? operator. MCDC terms have dependent vaiables
7. Manually Analyzed fmod(): There is only one MCDC case true in which the second condition impacts the third and hence cannot be satisfied ("FNAME(Dunscale)(&xchar, &t) == 0" implies that xchar=0 and since ychar>0 "xchar - ychar" is always <0)
8. Manually Analyzed fmodf(): There is only one MCDC case true in which the seond condition impacts the third and hence cannot be satisfied ("FNAME(Dunscale)(&xchar, &t) == 0" implies that xchar=0 and since ychar>0 "xchar - ychar" is always <0)
9. Manually Analyzed _Sbrk(): Konstantin Schwarz: This is a macro, and is defined to sbrk in our build. In general, this function has to be provided by the user. We implemented a dummy version in libruntime.a, but it was not instrumented with CTC. It can be ignored.
10. Manually Analyzed add(): operator, no coverage required
11. Manually Analyzed addf(): operator, no coverage required
12. Manually Analyzed multiply(): operator, no coverage required
13. Manually Analyzed multiplyf(): operator, no coverage required
14. Manually Analyzed divide(): operator, no coverage required
15. Manually Analyzed dividef(): operator, no coverage required
16. Manually Analyzed subtract(): operator, no coverage required
17. Manually Analyzed subtractf(): operator, no coverage required
18. Manually Analyzed _Tls_setup__Randinit(): Konstantin Schwarz: Those are function pointers, which are statically initialized

to nullptr in our build. Thus, they will never be called and can be ignored.

19. Manually Analyzed _Tls_setup__Randseed(): Konstantin Schwarz: Those are function pointers, which are statically initialized to nullptr in our build. Thus, they will never be called and can be ignored.
20. Manually Analyzed _Tls_setup_idx(): Konstantin Schwarz: Those are function pointers, which are statically initialized to nullptr in our build. Thus, they will never be called and can be ignored.
21. Manually Analyzed _Tls_setup_rv(): Konstantin Schwarz: Those are function pointers, which are statically initialized to nullptr in our build. Thus, they will never be called and can be ignored.
22. Manually Analyzed _Tls_setup_ssave(): Konstantin Schwarz: Those are function pointers, which are statically initialized to nullptr in our build. Thus, they will never be called and can be ignored.
23. Manually Analyzed NULL(): MACRO-Konstant, no coverage required
24. Manually Analyzed HUGE_VAL(): MACRO-Konstant, no coverage required
25. Manually Analyzed size_t(): typedef, no coverage required
26. Manually Analyzed _Quad_multiply(): See chapter 7.3
27. Manually Analyzed _FQuad_multiply(): same as Quad_multiply
28. Manually Analyzed _Xp_setw(): All cases analyzed. See chapter 7.7
29. Manually Analyzed _Xp_getw(): All cases analyzed. See chapter 7.4
30. Manually Analyzed _FXp_setw(): same as _Xp_setw
31. Manually Analyzed _FXp_getw(): same as _Xp_getw
32. Manually Analyzed _Sinx(): See chapter 7.1
33. Manually Analyzed _FSinx(): Same as _Sinx.
34. Manually Analyzed __aeabi_d2f(): Contains only simple cases: normal,NaN,Inf,round-up/down that are all covered by tests in TP_d2f
35. Manually Analyzed fabs(): called copysign is only with 0, hence a branch is not used here for sure fabs is OK. Function contains call of copysign() function with 2rd parameter always zero(copysign(x,0.0)). Uncovered code in copysign() will be executed only when 2rd parameter is negative (sign bit is 1). Thus, copysign() contains code(branch) that will not be executed when called from fabs(). For more detailes see #123 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/123)
36. Manually Analyzed fabsf(): called copysignf is only with 0, hence a branch is not used here for sure fabs is OK. Function contains call of copysignf() function with 2rd parameter always zero(copysignf(x,0.0)). Uncovered code in copysignf() will be executed only when 2rd parameter is negative (sign bit is 1). Thus, copysignf() contains code(branch) that will not be executed when

called from fabsf(). For more detailes see #123 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/123)

37. Manually Analyzed _Rint(): only one rounding mode used, hence rest is dead code
38. Manually Analyzed nearbyint(): only called from rint that catches the uncovered cases 0,NAN,INF that are dead code therefore in this setting
39. Manually Analyzed _FRint(): only one rounding mode used, hence rest is dead code
40. Manually Analyzed nearbyintf(): only called from rintf that catches the uncovered cases 0,NAN,INF that are dead code therefore in this setting
41. Manually Analyzed ldiv(): The function presents dead code. The missing MC/DC branches to cover cannot be traversed. The details of this analysis can be found on the trac ticket #86 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/86).
42. Manually Analyzed lldiv(): The function presents dead code. The missing MC/DC branches to cover cannot be traversed. The details of this analysis can be found on the trac ticket #86 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/86).
43. Manually Analyzed fma(): 100% locally to be confirmed and removed afterwards
44. Manually Analyzed fmaf(): 100% locally to be confirmed and removed afterwards
45. Manually Analyzed imaxdiv(): See chapter 7.8
46. Manually Analyzed atanh(): Same as atanhf
47. Manually Analyzed atanhf(): condition always true, see chapter 7.18
48. Manually Analyzed __aeabi_memclr(): just a single flow (no branches, except the branch to the called C function memset). The validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
49. Manually Analyzed __aeabi_memclr4(): just a single flow (no branches, except the branch to the called C function memset). The validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
50. Manually Analyzed __aeabi_memclr8(): just a single flow (no branches, except the branch to the called C function memset). The validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
51. Manually Analyzed __aeabi_memset(): just a single flow (no branches, except the branch to the called C function memset). The

validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).

52. Manually Analyzed __aeabi_memset4(): just a single flow (no branches, except the branch to the called C function memset). The validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
53. Manually Analyzed __aeabi_memset8(): just a single flow (no branches, except the branch to the called C function memset). The validation and complete code coverage of memset is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
54. Manually Analyzed __aeabi_memmove(): just a single flow (no branches, except the branch to the called C function memmove). The validation and complete code coverage of memmove is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
55. Manually Analyzed __aeabi_memmove4(): just a single flow (no branches, except the branch to the called C function memmove). The validation and complete code coverage of memmove is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
56. Manually Analyzed __aeabi_memmove8(): just a single flow (no branches, except the branch to the called C function memmove). The validation and complete code coverage of memmove is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
57. Manually Analyzed __aeabi_memcpy(): just a single flow (no branches, except the branch to the called C function memcpy). The validation and complete code coverage of memcpy is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
58. Manually Analyzed __aeabi_memcpy4(): just a single flow (no branches, except the branch to the called C function memcpy). The validation and complete code coverage of memcpy is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).
59. Manually Analyzed __aeabi_memcpy8(): just a single flow (no branches, except the branch to the called C function memcpy). The

validation and complete code coverage of memcpy is detailed in ticket #115 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/115#comment:1).

60. Manually Analyzed __aeabi_dcmpeq(): just two branches (OK/NOK that are covered by testing). Subroutine __eqdf2 is in C and has 100% in CTC
61. Manually Analyzed __aeabi_dcmplt(): just two branches (OK/NOK that are covered by testing). Subroutine __ltdf2 is in C and has 100% in CTC
62. Manually Analyzed __aeabi_dcmple(): just two branches (OK/NOK that are covered by testing). Subroutine __ledf2 is in C and has 100% in CTC
63. Manually Analyzed __aeabi_dcmpge(): just two branches (OK/NOK that are covered by testing). Subroutine __gedf2 is in C and has 100% in CTC
64. Manually Analyzed __aeabi_dcmpgt(): just two branches (OK/NOK that are covered by testing). Subroutine __gtdf2 is in C and has 100% in CTC
65. Manually Analyzed __aeabi_dcmpun(): just two branches (OK/NOK that are covered by testing). Subroutine __unorddf is in C and has 100% in CTC
66. Manually Analyzed _Xp_addh(): See chapter 7.5
67. Manually Analyzed _Xp_mulh(): See chapter 7.6
68. Manually Analyzed _FXp_addh(): Same as _Xp_addh()
69. Manually Analyzed _FXp_mulh(): Same as _Xp_mulh()
70. Manually Analyzed __aeabi_idivmod(): Assembly code only shows one branch, separating cases where the denominator is equal or different to zero. Both types of inputs are present in test, therefore, it has 100% code coverage. This can be verified on the test by using the following regex expression on the corresponding test.c file "denom\[1\] = {[0]\D".
71. Manually Analyzed __aeabi_uidivmod(): Assembly code only shows one branch, separating cases where the denominator is equal or different to zero. Both types of inputs are present in test, therefore, it has 100% code coverage. This can be verified on the test by using the following regex expression on the corresponding test.c file "denom\[1\] = {[0]\D".
72. Manually Analyzed __aeabi_idiv(): Assembly code only shows one branch, separating cases where the denominator is equal or different to zero. Both types of inputs are present in test, therefore, it has 100% code coverage. Tests containing both cases can be found in the file __aeabi_idiv_uv_uv_extreme.c and __aeabi_idiv_uv_uv_normal.c
73. Manually Analyzed __aeabi_uidiv(): Assembly code only shows one branch, separating cases where the denominator is equal or different to zero. Both types of inputs are present in test, therefore, it has 100% code coverage. Tests containing both cases can be

found in the file __aeabi_uidiv_uv_uv_extreme.c and __aeabi_uidiv_uv_uv_normal.c

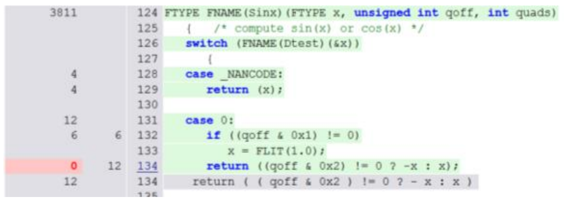
74. Manually Analyzed hypot(): The switch statment is not reachable. if result of _Hypot is NaN, Inf or 0, then reference zexp is set to 0. So for NaN, Inf or 0 "if" statement will always be false. Switch checks if returned value is Inf or Zero. Thus switch will never been invoked. See #116 https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/116 for more details.
75. Manually Analyzed hypotf(): The switch statment is not reachable. if result of _Hypot is NaN, Inf or 0, then reference zexp is set to 0. So for NaN, Inf or 0 "if" statement will always be false. Switch checks if returned value is Inf or Zero. Thus switch will never been invoked. See #116 https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/116 for more details.
76. Manually Analyzed __aeabi_lasr(): The function is just a wrapper for the function `__ashrdi3` which is implemented in C and CTC shows full code coverage.
77. Manually Analyzed __aeabi_llsl(): The function is just a wrapper for the function `__ashldi3` which is implemented in C and CTC shows full code coverage.
78. Manually Analyzed __aeabi_llsr(): The function is just a wrapper for the function `__lshrdi3` which is implemented in C and CTC shows full code coverage.
79. Manually Analyzed __aeabi_lmul(): The function is just a wrapper for the function `__muldi3` which is implemented in C and CTC shows full code coverage.
80. Manually Analyzed wideRightShiftWithSticky(): The function contains some branches which cannot be traverse. For the details please refer to chapter 7.16.1 within this document.
81. Manually Analyzed nextafter(): else if contains mutually exclusive conditions, that can't be "true" or "false" simultaneously. See comment#6 to ticket #121 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/121#comment:6)
82. Manually Analyzed nextafterf(): else if contains mutually exclusive conditions, that can't be "true" or "false" simultaneously. See comment#6 to ticket #121 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/121#comment:6)
83. Manually Analyzed remainder(): Function contains call of remquo function with 3rd parameter (..., int *pquo) always zero. Uncovered code in remquo will be executed only when 3rd parameter is not zero. Thus, remquo contains dead code. For more details see #122 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/122)

84. Manually Analyzed remainderf(): Function contains call of remquof function with 3rd parameter (..., int *pquo) always zero. Uncovered code in remquof will be executed only when 3rd parameter is not zero. Thus, remquof contains dead code. For more details see #122 (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/122)
85. Manually Analyzed __aeabi_cdcmple(): The code coverage of the function is complete. See ticket #124 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/124)
86. Manually Analyzed __aeabi_cdcmpeq(): The code coverage of the function is complete. See ticket #125 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/125)
87. Manually Analyzed __aeabi_cdrcmple(): The code coverage of the function is complete. See ticket #126 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/126)
88. Manually Analyzed cosh(): For the details please refer to chapter 7.19
89. Manually Analyzed coshf(): Same as cosh().
90. Manually Analyzed pow():For the details please refer to chapter 7.20
91. Manually Analyzed powf(): Same as pow()
92. Manually Analyzed __aeabi_ldivmod(): Assembly code is only an assembly wrapper without branches to call functions in C. The functions which the SUT calls are: __aeabi_ldivmod, __divmoddi4, __divdi3, udivmoddi4. Only udivmoddi4 has MC-DC branches, and it has 100% code coverage as it can be observed in the report.txt.
93. Manually Analyzed __aeabi_uldivmod(): Assembly code is only an assembly wrapper without branches to call udivmoddi4. The function udivmoddi4 has 100% code coverage as it can be observed in the report.txt.
94. Manually Analyzed __aeabi_fcmpun(): The code coverage of the function is complete. See ticket #131 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/131)
95. Manually Analyzed __aeabi_fcmpeq(): The function is an alias for `__eqsf2` which code coverage is complete. See ticket #132 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/132)
96. Manually Analyzed __aeabi_fcmple(): The function is an alias for `__lesf2`. The semantics of the function are identical to __eqsf2 (__aeabi_fcmpeq), so it uses the same `__eqsf2` implementation for which code coverage is complete. See ticket #132 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/132)
97. Manually Analyzed __aeabi_fcmplt(): The function is an alias for `__ltsf2`.The semantics of the function are identical to __eqsf2 (__aeabi_fcmpeq), so it uses the same `__eqsf2` implementation for which code coverage is complete. See ticket #132 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/132)

98. Manually Analyzed __aeabi_cfcmple(): The only branches may occur at the calls to __aeabi_fcmplt and __aeabi_fcmpeq to be complete. These functions have been already analyzed above. . See ticket #126
99. Manually Analyzed __aeabi_cfrcmple(): The only branches may occur at __aeabi_cfcmple to be complete. This functions have been already analyzed above. See ticket #126
100. Manually Analyzed __aeabi_cfcmpeq(): The only branches may occur at __aeabi_cfcmple to be complete. This functions have been already analyzed above. See ticket #125
101. Manually Analyzed __aeabi_fcmpge(): The function is an alias for `__gesf2` which code coverage is complete. See ticket #132 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/132)
102. Manually Analyzed __aeabi_fcmpgt(): The function is an alias for `__gtsf2`. The semantics of the function are identical to __gesf2 (__aeabi_fcmpge), so it uses the same `__gesf2` implementation for which code coverage is complete. See ticket #132 for details (https://opentrac.teststatt.de/tracs/qkithightecarm/ticket/132)
103. Manually Analyzed _Quad: See chapter 7.2 in this document.
104. Manually Analyzed __aeabi_fadd: See chapter 7.17 in this document
105. Manually Analyzed _Getmem: See chapter 7.21 in this document.
106. Manually Analyzed sinh: See chapter 7.23
107. Manually Analyzed sinhf: Same as for sinh
108. Manually Analyzed _FQuad: Same as for _Quad
109. Manually Analyzed _Tan: _Feraise, _Pmsw, _Dscale, Dunscale, DTest call this function which have already been analyzed or have 100 % coverage
110. Manually Analyzed _FTan: Same as for _Tan

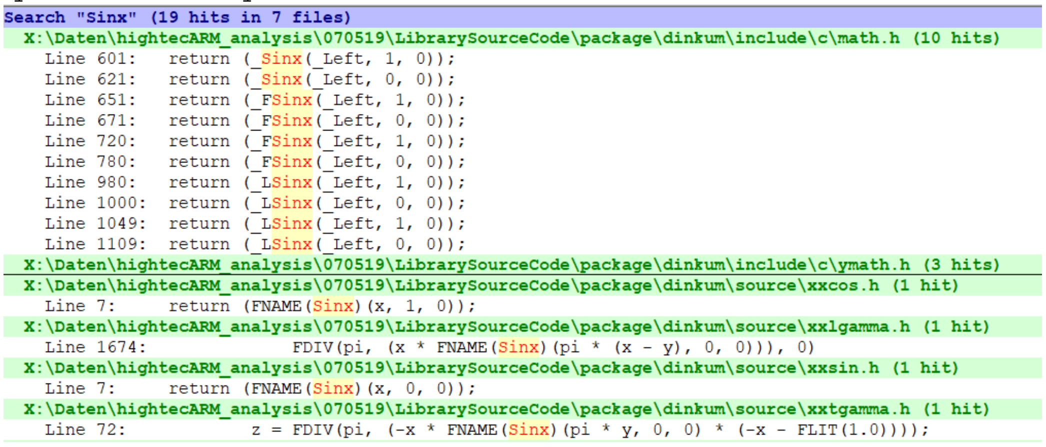
7 Coverage Analysis Details

7.1 Analysis of code coverage of function _Sinx: OK



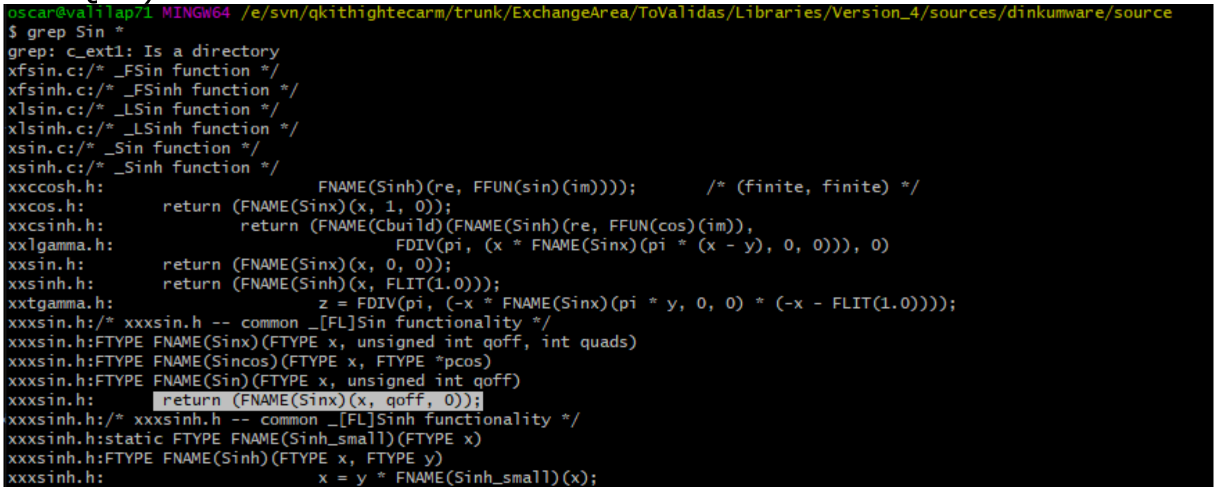
```
  3811       124 FTYPE FNAME(Sinx)(FTYPE x, unsigned int qoff, int quads)
             125     {   /* compute sin(x) or cos(x) */
             126     switch (FNAME(Dtest)(&x))
             127         {
     4       128     case _NANCODE:
     4       129         return (x);
             130
    12       131     case 0:
     6    6  132         if ((qoff & 0x1) != 0)
             133             x = FLIT(1.0);
     0   12  134         return ((qoff & 0x2) != 0 ? -x : x);
    12       134      return ( ( qoff & 0x2 ) != 0 ? - x : x )
```

In order to cover line 134, for function parameter `qoff` of `_Sinx` it must be fulfilled: `quoff & 0x2 != 0`. However function `_Sinx` is only called with `qoff == 0` or `qoff == 1`:



```
Search "Sinx" (19 hits in 7 files)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\include\c\math.h (10 hits)
	Line 601: 	return (_Sinx(_Left, 1, 0));
	Line 621: 	return (_Sinx(_Left, 0, 0));
	Line 651: 	return (_FSinx(_Left, 1, 0));
	Line 671: 	return (_FSinx(_Left, 0, 0));
	Line 720: 	return (_FSinx(_Left, 1, 0));
	Line 780: 	return (_FSinx(_Left, 0, 0));
	Line 980: 	return (_LSinx(_Left, 1, 0));
	Line 1000: 	return (_LSinx(_Left, 0, 0));
	Line 1049: 	return (_LSinx(_Left, 1, 0));
	Line 1109: 	return (_LSinx(_Left, 0, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\include\c\ymath.h (3 hits)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxcos.h (1 hit)
	Line 7: 	return (FNAME(Sinx)(x, 1, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxlgamma.h (1 hit)
	Line 1674: 			FDIV(pi, (x * FNAME(Sinx)(pi * (x - y), 0, 0))), 0)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxsin.h (1 hit)
	Line 7: 	return (FNAME(Sinx)(x, 0, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxtgamma.h (1 hit)
	Line 72: 		z = FDIV(pi, (-x * FNAME(Sinx)(pi * y, 0, 0) * (-x - FLIT(1.0))));
```

Looking into the sources shows that Sinx is also called from Sin which is called from ccos (i.e. the complex cos function, which is out of scope of this QKit)



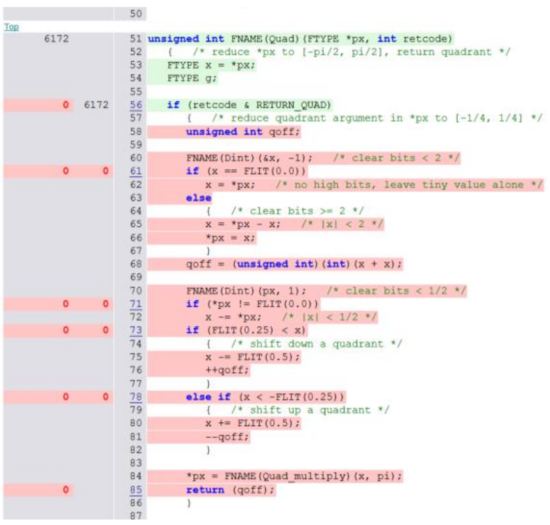
```
oscar@valilap71 MINGW64 /e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/Libraries/Version_4/sources/dinkumware/source
$ grep Sin *
grep: c_ext1: Is a directory
xfsin.c:/* _FSin function */
xfsinh.c:/* _FSinh function */
xlsin.c:/* _LSin function */
xlsinh.c:/* _LSinh function */
xsin.c:/* _Sin function */
xsinh.c:/* _Sinh function */
xxccosh.h:                              FNAME(Sinh)(re, FFUN(sin)(im))));       /* (finite, finite) */
xxcos.h:        return (FNAME(Sinx)(x, 1, 0));
xxcsinh.h:              return (FNAME(Cbuild)(FNAME(Sinh)(re, FFUN(cos)(im)),
xxlgamma.h:                             FDIV(pi, (x * FNAME(Sinx)(pi * (x - y), 0, 0))), 0)
xxsin.h:        return (FNAME(Sinx)(x, 0, 0));
xxsinh.h:       return (FNAME(Sinh)(x, FLIT(1.0)));
xxtgamma.h:                     z = FDIV(pi, (-x * FNAME(Sinx)(pi * y, 0, 0) * (-x - FLIT(1.0))));
xxxsin.h:/* xxxsin.h -- common _[FL]Sin functionality */
xxxsin.h:FTYPE FNAME(Sinx)(FTYPE x, unsigned int qoff, int quads)
xxxsin.h:FTYPE FNAME(Sincos)(FTYPE x, FTYPE *pcos)
xxxsin.h:FTYPE FNAME(Sin)(FTYPE x, unsigned int qoff)
xxxsin.h:       return (FNAME(Sinx)(x, qoff, 0));
xxxsinh.h:/* xxxsinh.h -- common _[FL]Sinh functionality */
xxxsinh.h:static FTYPE FNAME(Sinh_small)(FTYPE x)
xxxsinh.h:FTYPE FNAME(Sinh)(FTYPE x, FTYPE y)
xxxsinh.h:                      x = y * FNAME(Sinh_small)(x);
```

➔Line 134 cannot be covered by adding additional test cases.

7.2 Analysis of code coverage of function _Quad: OK

7.2.1 Retcode & RETURN_QUAD: OK

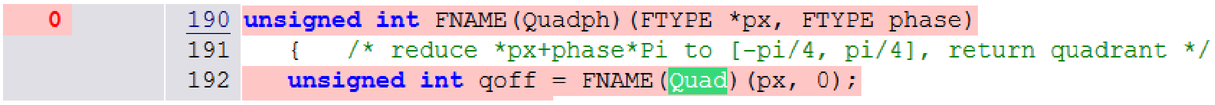
Cannot be covered, since retcode is always zero



```
Top
6172         unsigned int FNAME(Quad)(FTYPE *px, int retcode)
                {   /* reduce *px to [-pi/2, pi/2], return quadrant */
                FTYPE x = *px;
                FTYPE g;

   0   6172     if (retcode & RETURN_QUAD)
                    {   /* reduce quadrant argument in *px to [-1/4, 1/4] */
                    unsigned int qoff;

                    FNAME(Dint)(&x, -1);    /* clear bits < 2 */
   0      0         if (x == FLIT(0.0))
                        x = *px;    /* no high bits, leave tiny value alone */
                    else
                        {   /* clear bits >= 2 */
                        x = *px - x;    /* |x| < 2 */
                        *px = x;
                        }
                    qoff = (unsigned int)(int)(x + x);

                    FNAME(Dint)(px, 1);    /* clear bits < 1/2 */
   0      0         if (*px != FLIT(0.0))
                        x -= *px;    /* |x| < 1/2 */
   0      0         if (FLIT(0.25) < x)
                        {   /* shift down a quadrant */
                        x -= FLIT(0.5);
                        ++qoff;
                        }
   0      0         else if (x < -FLIT(0.25))
                        {   /* shift up a quadrant */
                        x += FLIT(0.5);
                        --qoff;
                        }

                    *px = FNAME(Quad_multiply)(x, pi);
   0                return (qoff);
                    }
```

In order to cover the "true"-branch of if condition (retcode & RETURN_QUAD), it must be parameter retcode & 1 != 0.
Function _Quad is called directly by the following functions:

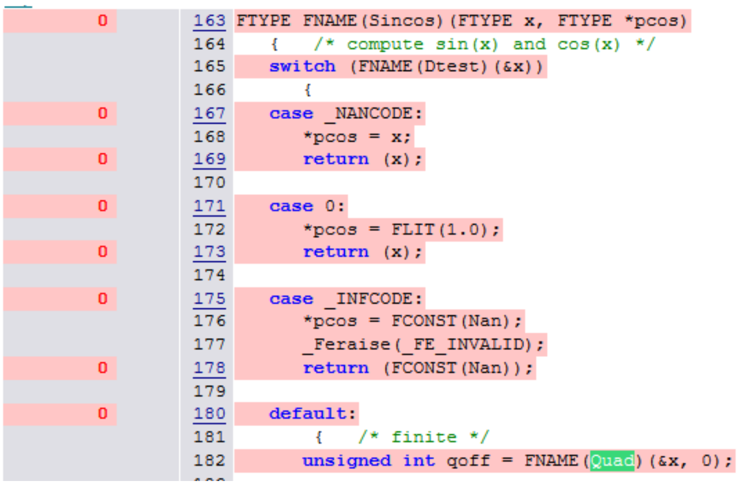
- _Quadph (OS: Quadph is dead)



```
   0         unsigned int FNAME(Quadph)(FTYPE *px, FTYPE phase)
                {   /* reduce *px+phase*Pi to [-pi/4, pi/4], return quadrant */
                unsigned int qoff = FNAME(Quad)(px, 0);
```

➔Parameter retcode is set to 0 by _Quadph

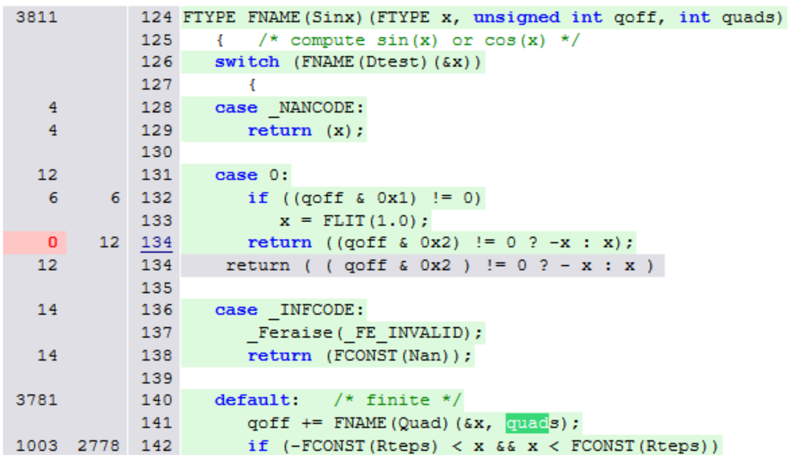
- _Sincos (OS: Sincos is dead)



```
   0         FTYPE FNAME(Sincos)(FTYPE x, FTYPE *pcos)
                {   /* compute sin(x) and cos(x) */
                switch (FNAME(Dtest)(&x))
                    {
   0            case _NANCODE:
                    *pcos = x;
   0                return (x);

   0            case 0:
                    *pcos = FLIT(1.0);
   0                return (x);

   0            case _INFCODE:
                    *pcos = FCONST(Nan);
                    _Feraise(_FE_INVALID);
   0                return (FCONST(Nan));

   0            default:
                    {   /* finite */
                    unsigned int qoff = FNAME(Quad)(&x, 0);
```

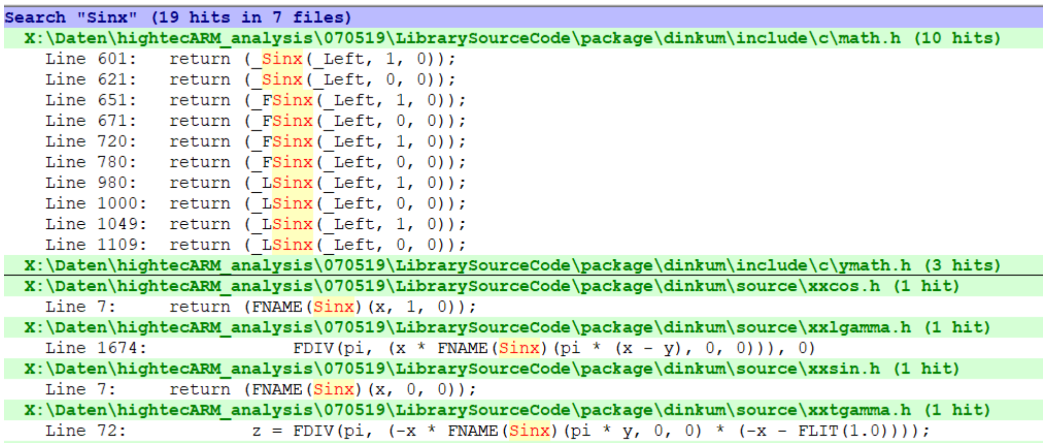
➔Parameter retcode is set to 0 by _Sincos

- _Sinx



```
3811         FTYPE FNAME(Sinx)(FTYPE x, unsigned int qoff, int quads)
                {   /* compute sin(x) or cos(x) */
                switch (FNAME(Dtest)(&x))
                    {
   4            case _NANCODE:
   4                return (x);

  12            case 0:
   6      6         if ((qoff & 0x1) != 0)
                        x = FLIT(1.0);
   0     12         return ((qoff & 0x2) != 0 ? -x : x);
  12           return ( ( qoff & 0x2 ) != 0 ? - x : x )

  14            case _INFCODE:
                    _Feraise(_FE_INVALID);
  14                return (FCONST(Nan));

3781            default:    /* finite */
                    qoff += FNAME(Quad)(&x, quads);
1003   2778         if (-FCONST(Rteps) < x && x < FCONST(Rteps))
```

➔Parameter retcode of _Quad is set to value of parameter quads of _Sinx

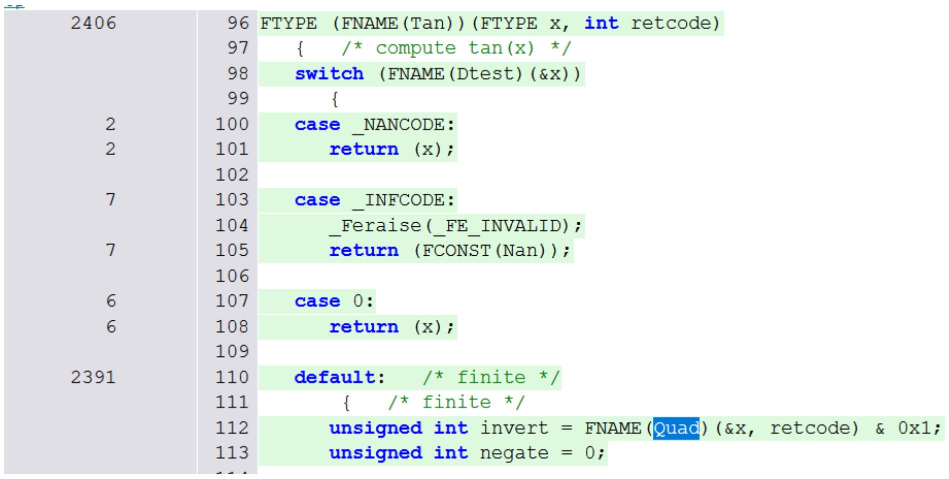


```
Search "Sinx" (19 hits in 7 files)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\include\c\math.h (10 hits)
    Line 601:   return (_Sinx(_Left, 1, 0));
    Line 621:   return (_Sinx(_Left, 0, 0));
    Line 651:   return (_FSinx(_Left, 1, 0));
    Line 671:   return (_FSinx(_Left, 0, 0));
    Line 720:   return (_FSinx(_Left, 1, 0));
    Line 780:   return (_FSinx(_Left, 0, 0));
    Line 980:   return (_LSinx(_Left, 1, 0));
    Line 1000:  return (_LSinx(_Left, 0, 0));
    Line 1049:  return (_LSinx(_Left, 1, 0));
    Line 1109:  return (_LSinx(_Left, 0, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\include\c\ymath.h (3 hits)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxcos.h (1 hit)
    Line 7:     return (FNAME(Sinx)(x, 1, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxlgamma.h (1 hit)
    Line 1674:              FDIV(pi, (x * FNAME(Sinx)(pi * (x - y), 0, 0))), 0)
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxsin.h (1 hit)
    Line 7:     return (FNAME(Sinx)(x, 0, 0));
  X:\Daten\hightecARM_analysis\070519\LibrarySourceCode\package\dinkum\source\xxtgamma.h (1 hit)
    Line 72:            z = FDIV(pi, (-x * FNAME(Sinx)(pi * y, 0, 0) * (-x - FLIT(1.0))));
```

➔Parameter quads of _Sinx is in all cases set to 0.

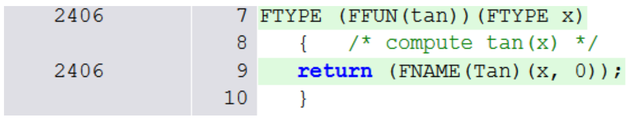
➔ Parameter retcode of _Quad is in all cases set to 0.

- _Tan



```
2406    FTYPE (FNAME(Tan))(FTYPE x, int retcode)
            {   /* compute tan(x) */
            switch (FNAME(Dtest)(&x))
                {
   2        case _NANCODE:
   2            return (x);

   7        case _INFCODE:
                _Feraise(_FE_INVALID);
   7            return (FCONST(Nan));

   6        case 0:
   6            return (x);

2391        default:    /* finite */
                {   /* finite */
                unsigned int invert = FNAME(Quad)(&x, retcode) & 0x1;
                unsigned int negate = 0;
```

➔ Parameter retcode of _Quad is set to value of parameter retcode of _Tan



```
2406    FTYPE (FFUN(tan))(FTYPE x)
            {   /* compute tan(x) */
2406        return (FNAME(Tan)(x, 0));
            }
```

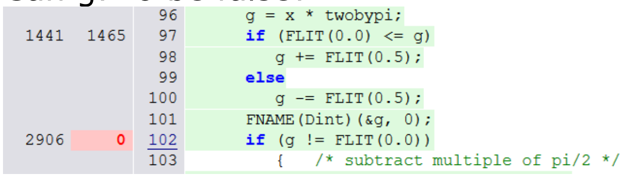
➔ Parameter retcode of _Tan is set to 0

➔Parameter retcode of _Quad is set to 0

Since retcode = 0, condition (retcode & RETURN_QUAD) can never be fulfilled.

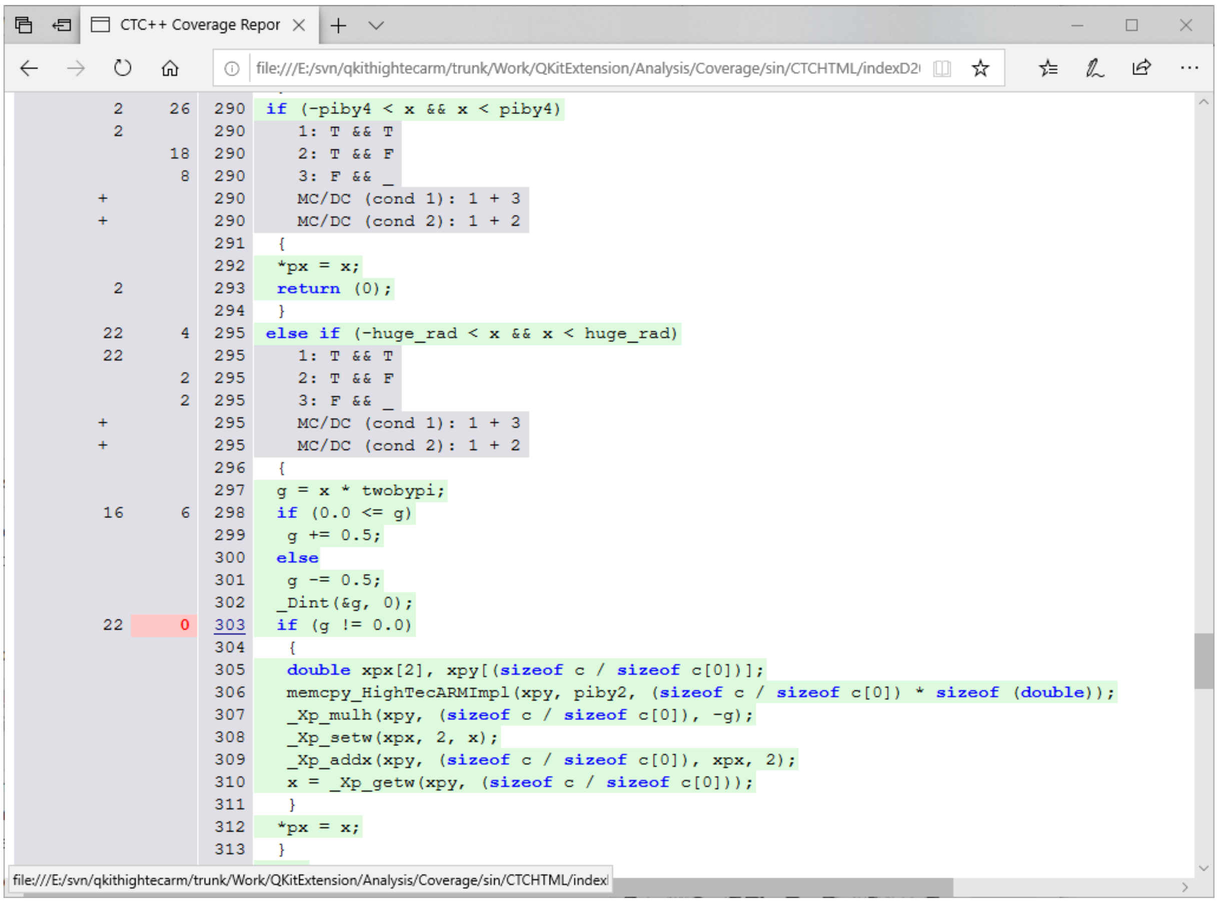
7.2.2 g==0: OK

Can g!=0 be false?



```
                    g = x * twobypi;
1441   1465         if (FLIT(0.0) <= g)
                        g += FLIT(0.5);
                    else
                        g -= FLIT(0.5);
                    FNAME(Dint)(&g, 0);
2906      0         if (g != FLIT(0.0))
                        {   /* subtract multiple of pi/2 */
```

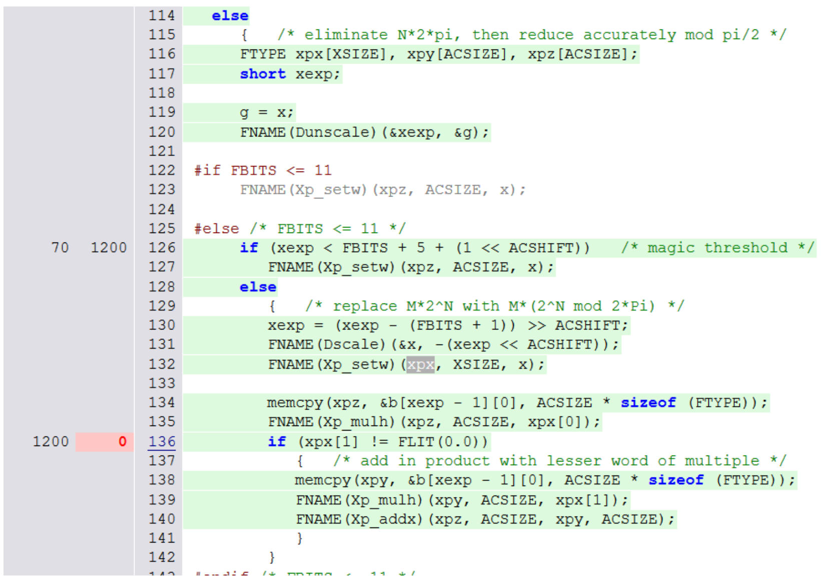
Ore more detailed (precompiled code):


CTC++ Coverage Repor
file:///E:/svn/qkithightecarm/trunk/Work/QKitExtension/Analysis/Coverage/sin/CTCHTML/indexD2
file:///E:/svn/qkithightecarm/trunk/Work/QKitExtension/Analysis/Coverage/sin/CTCHTML/index

```
   2    26  290  if (-piby4 < x && x < piby4)
   2        290     1: T && T
        18  290     2: T && F
         8  290     3: F && _
  +         290     MC/DC (cond 1): 1 + 3
  +         290     MC/DC (cond 2): 1 + 2
            291    {
            292    *px = x;
   2        293    return (0);
            294    }
  22     4  295  else if (-huge_rad < x && x < huge_rad)
  22        295     1: T && T
         2  295     2: T && F
         2  295     3: F && _
  +         295     MC/DC (cond 1): 1 + 3
  +         295     MC/DC (cond 2): 1 + 2
            296    {
            297    g = x * twobypi;
  16     6  298    if (0.0 <= g)
            299     g += 0.5;
            300    else
            301     g -= 0.5;
            302    _Dint(&g, 0);
  22     0  303    if (g != 0.0)
            304     {
            305     double xpx[2], xpy[(sizeof c / sizeof c[0])];
            306     memcpy_HighTecARMImpl(xpy, piby2, (sizeof c / sizeof c[0]) * sizeof (double));
            307     _Xp_mulh(xpy, (sizeof c / sizeof c[0]), -g);
            308     _Xp_setw(xpx, 2, x);
            309     _Xp_addx(xpy, (sizeof c / sizeof c[0]), xpx, 2);
            310     x = _Xp_getw(xpy, (sizeof c / sizeof c[0]));
            311     }
            312    *px = x;
            313    }
```

Can g be 0 (after rounding via "if (0.0<=g) g+=0.5; else g-=0.5; _Dint(&g,0);")
Only if g would be between -0.5 and 0.5 before rounding.
g is the value of x*twobypi, i.e. g=2x/Pi;
The case that x is between –Pi/4 and Pi/4 is handeled before (line 290)
The corner cases are x=Pi/4 and x=-Pi/4 for those values g=2*x is -0.5 and 0.5 which are rounded to
-1 and 1. Therefore the case g==0 cannot occur if the floating point routines computing g=x*twobypi and g-=0.5 and g+=0.5 work exactly.
Only in case of a rounding error occurs for x=Pi/4;
The values –Pi/4 and Pi/4 have been added to the test to test exactly tis corner case of the algorithm.

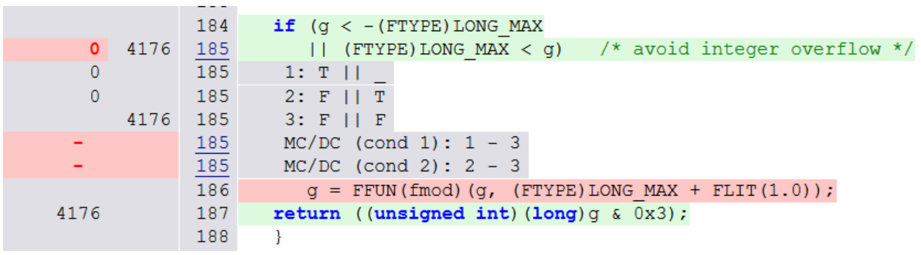
7.2.3 Xpz[1]==0 OK



```
                else
                    {   /* eliminate N*2*pi, then reduce accurately mod pi/2 */
                    FTYPE xpx[XSIZE], xpy[ACSIZE], xpz[ACSIZE];
                    short xexp;

                    g = x;
                    FNAME(Dunscale)(&xexp, &g);

              #if FBITS <= 11
                    FNAME(Xp_setw)(xpz, ACSIZE, x);

              #else /* FBITS <= 11 */
  70    1200        if (xexp < FBITS + 5 + (1 << ACSHIFT))    /* magic threshold */
                        FNAME(Xp_setw)(xpz, ACSIZE, x);
                    else
                        {   /* replace M*2^N with M*(2^N mod 2*Pi) */
                        xexp = (xexp - (FBITS + 1)) >> ACSHIFT;
                        FNAME(Dscale)(&x, -(xexp << ACSHIFT));
                        FNAME(Xp_setw)(xpx, XSIZE, x);

                        memcpy(xpz, &b[xexp - 1][0], ACSIZE * sizeof (FTYPE));
                        FNAME(Xp_mulh)(xpz, ACSIZE, xpx[0]);
1200       0            if (xpx[1] != FLIT(0.0))
                            {   /* add in product with lesser word of multiple */
                            memcpy(xpy, &b[xexp - 1][0], ACSIZE * sizeof (FTYPE));
                            FNAME(Xp_mulh)(xpy, ACSIZE, xpx[1]);
                            FNAME(Xp_addx)(xpz, ACSIZE, xpy, ACSIZE);
                            }
                        }
```

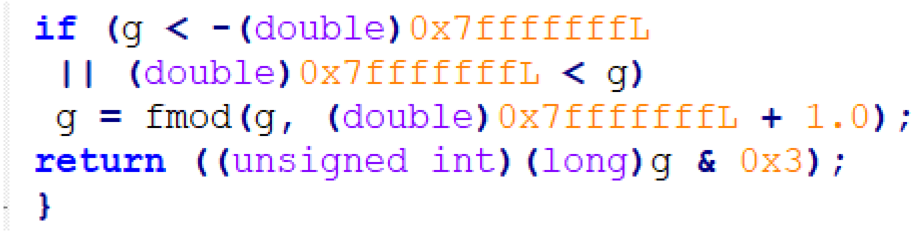
This corner case can only be valid, if the floating point value of x (after descaling) is loaded into the two floats xpx[0] and xpx[1] happens to have only 0x0000000000 in the second part. This is very hard to trigger and not necessary, since the effect (empty else branch of that if) would be just a multiplication of by 0.0 and adding 0 to z. It is safe to omit these operations as a matter of speed optimization and it is therefore not required to cover this case.

7.2.4 G>-LONG_MAX && g<-LONG_MAX OK



```
                  if (g < -(FTYPE)LONG_MAX
        0   4176      || (FTYPE)LONG_MAX < g)    /* avoid integer overflow */
        0          1: T || _
        0          2: F || T
            4176   3: F || F
        -          MC/DC (cond 1): 1 - 3
        -          MC/DC (cond 2): 2 - 3
                      g = FFUN(fmod)(g, (FTYPE)LONG_MAX + FLIT(1.0));
    4176          return ((unsigned int)(long)g & 0x3);
                  }
```

For the given architecture (sizeof(long)=sizeof(double)=8) this is an impossible case as can be seen by looking to the precompiled code. Hence the code is dead and cannot be covered.



```
 if (g < -(double)0x7fffffffL
  || (double)0x7fffffffL < g)
  g = fmod(g, (double)0x7fffffffL + 1.0);
 return ((unsigned int)(long)g & 0x3);
 }
```

7.3 Analysis of code coverage of function _Quad_multiply: OK

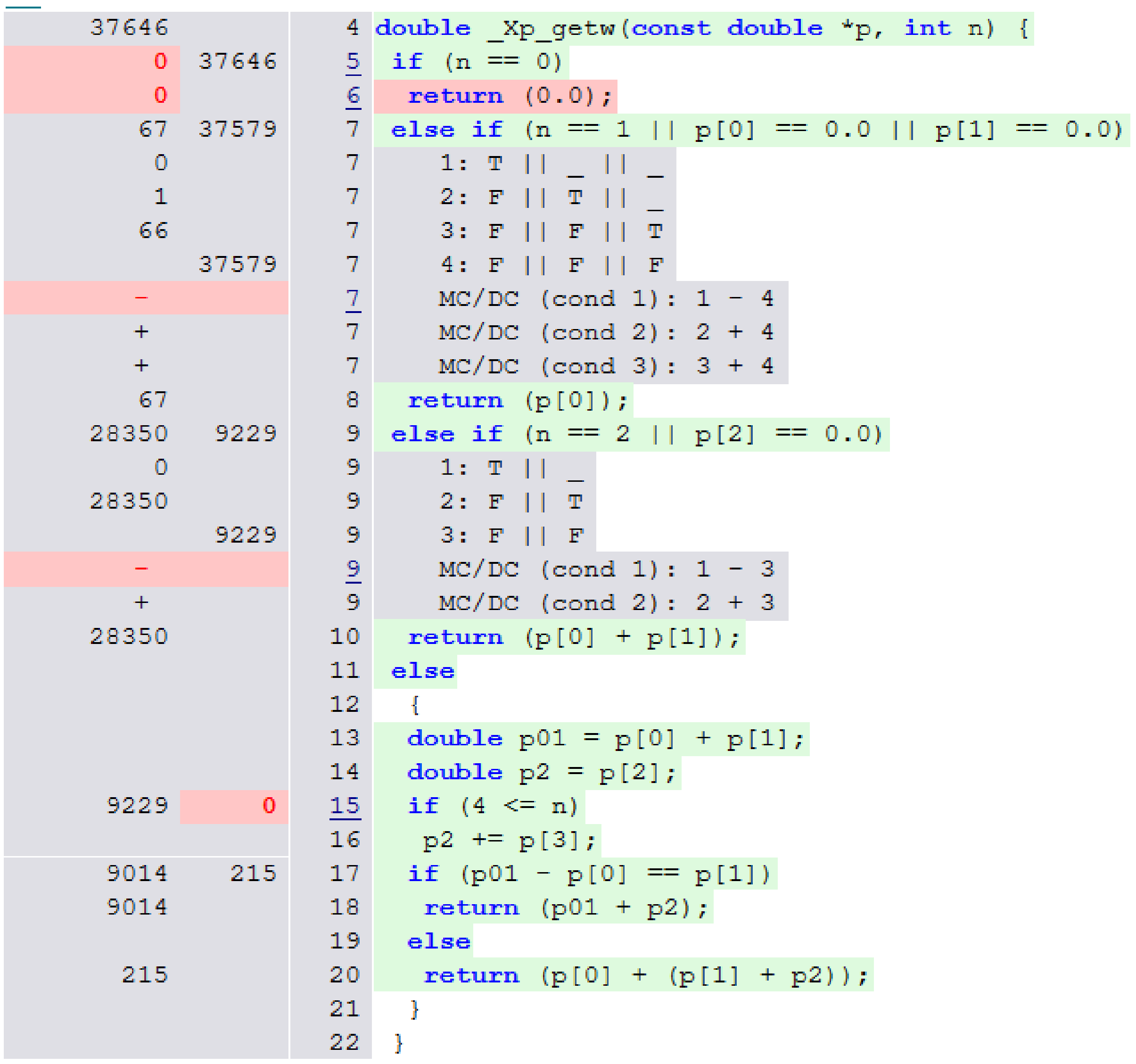
Quad_multiply is never called, since condition in section 7.2.1 is never fulfilled.

7.4 Analysis of code coverage of function _Xp_getw: OK

Analyzing the precompiled code shows that _Xp_getw is only called with n==4 or 6 (sizeof c / sizeof c[0]), sizeof c / sizeof c[0]) = 6 (see definition of c in xxxquad.hx):

```
oscar@valilap71 MINGW64 /e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/preprocessed/preprocessed
$ grep Xp_getw *| grep -v double| grep -v FXp_getw
fma.i.c:  ans = _Xp_getw(xpx, 4);
fmal.i.c:  ans = _LXp_getw(xpx, 4);
pow.i.c:  z = _Xp_getw(xpz, 4);
powl.i.c:  z = _LXp_getw(xpz, 4);
xdtento.i.c:  return (_Xp_getw(xpx, 4));
xdtento.i.c: x = _Xp_getw(xpx, 4);
xldtento.i.c:  return (_LXp_getw(xpx, 4));
xldtento.i.c: x = _LXp_getw(xpx, 4);
xlquad.i.c:   x = _LXp_getw(xpy, (sizeof c / sizeof c[0]));
xlquad.i.c:  *px = _LXp_getw(xpz, (sizeof c / sizeof c[0]));
xquad.i.c:   x = _Xp_getw(xpy, (sizeof c / sizeof c[0]));
xquad.i.c:  *px = _Xp_getw(xpz, (sizeof c / sizeof c[0]));
```

This simplifies further analysis of uncovered parts



```
   37646          4  double _Xp_getw(const double *p, int n) {
       0   37646  5   if (n == 0)
       0          6    return (0.0);
      67   37579  7   else if (n == 1 || p[0] == 0.0 || p[1] == 0.0)
       0          7      1: T || _ || _
       1          7      2: F || T || _
      66          7      3: F || F || T
           37579  7      4: F || F || F
       -          7      MC/DC (cond 1): 1 - 4
       +          7      MC/DC (cond 2): 2 + 4
       +          7      MC/DC (cond 3): 3 + 4
      67          8    return (p[0]);
   28350    9229  9   else if (n == 2 || p[2] == 0.0)
       0          9      1: T || _
   28350          9      2: F || T
            9229  9      3: F || F
       -          9      MC/DC (cond 1): 1 - 3
       +          9      MC/DC (cond 2): 2 + 3
   28350         10    return (p[0] + p[1]);
                 11   else
                 12    {
                 13    double p01 = p[0] + p[1];
                 14    double p2 = p[2];
    9229       0 15    if (4 <= n)
                 16     p2 += p[3];
    9014     215 17    if (p01 - p[0] == p[1])
    9014         18     return (p01 + p2);
                 19    else
     215         20     return (p[0] + (p[1] + p2));
                 21    }
                 22  }
```

Line 5/6 are unreachable since n>=4
Line 9, cond 1: is also not reachable sine n>=4
Line 15 is never false, since n>=4
Therefore _Xp_getw is completely covered.

7.5 Analysis of code coverage of function _Xp_addh: OK

The analysis of _XP_addh starts from the part in the coverage report.txt (see CTC-User Guide for explanations):

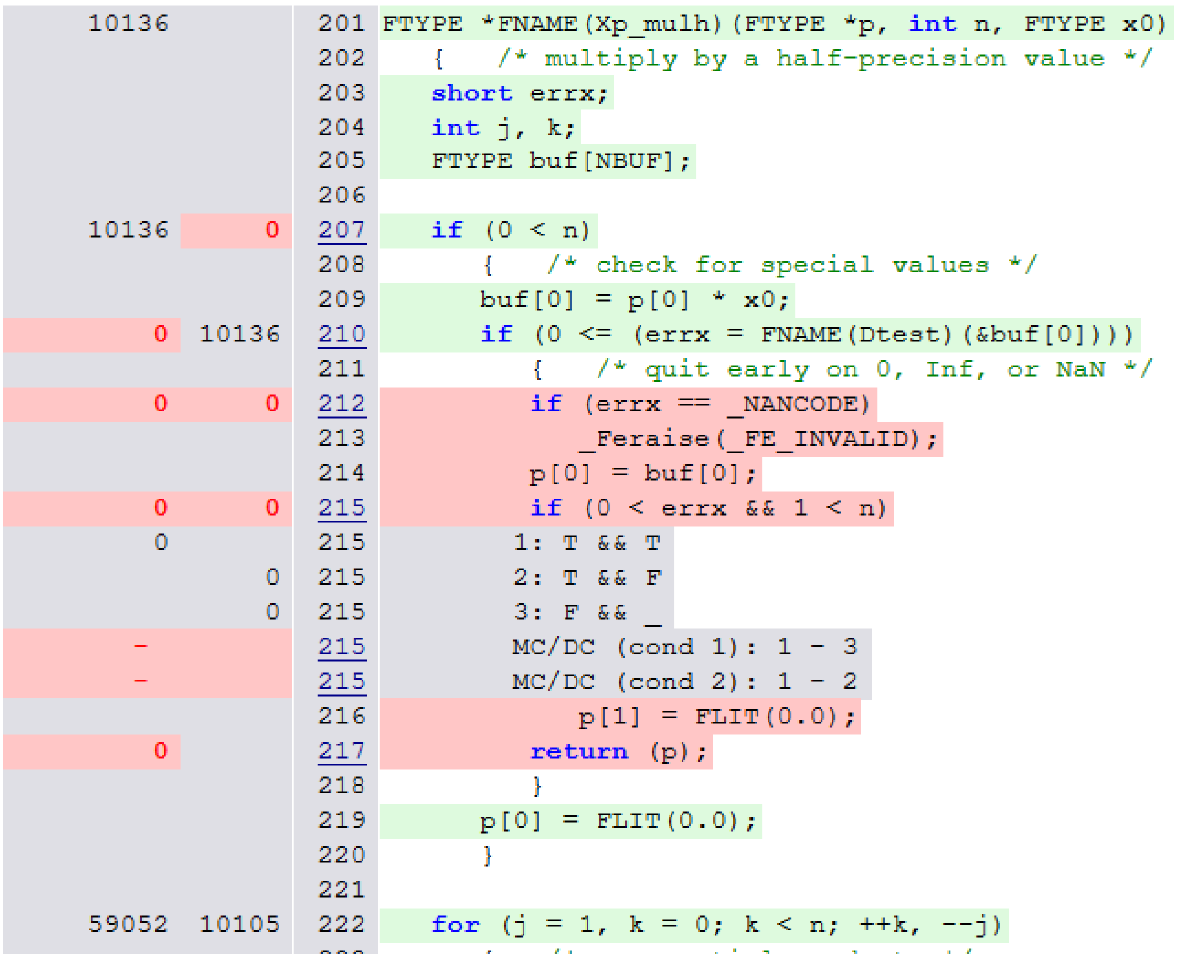
```
   116287                 92 FUNCTION _FXp_addh()
        0     116287 -     97   if (n == 0)
                           98   }+
        0     116287 -     99   else if (0 < ( errx = _FDunscale ( & xexp , & xscaled ) ))
        0          0 -    100     if (errx == 2 || ( errx = _FDtest ( & p [ 0 ] ) ) <= 0)
        0                 100       1: T || _
        0                 100       2: F || T
                   0      100       3: F || F
                     -    100       MC/DC (cond 1): 1 - 3
                     -    100       MC/DC (cond 2): 2 - 3
                          101     }-
        0          0 -    102     else if (errx == 2 || ( ( * _FPmsw ( & ( x0 ) ) ) & ( ( unsigned short ) 0x8000 ) ) == ( (
* _FPmsw ( & ( p [ 0 ] ) ) ) & ( ( unsigned short ) 0x8000 ) ))
        0                 102       1: T || _
        0                 102       2: F || T
                   0      102       3: F || F
                     -    102       MC/DC (cond 1): 1 - 3
                     -    102       MC/DC (cond 2): 2 - 3
                          103     }-
                          104     else
        0          0 -    108       if (1 < n)
                          109       }-
                          110     }-
                          110   }+
    97488      18799      111   else if (errx < 0)
```

```
 549635       1546    116      for (;k < n;)
      0     549635 -  123        if (0 < ( errx = _FDunscale ( & yexp , & yscaled ) ))
      0            -  124          break
                      124        }+
  78174     471461    125        else if (errx == 0)
  60294      17880    128          if (k + 1 < n)
                      129          }+
  78174               130          break
                      131        }+
   2307     469154    133        else if (( diff = ( long ) yexp - xexp ) <= - mybits && x0 != 0.0F)
   2307               133          1: T && T
             78181    133          2: T && F
            390973    133          3: F && _
                      133          MC/DC (cond 1): 1 + 3
                      133          MC/DC (cond 2): 1 + 2
   3582       2307    137          for (;++ j < n && p [ j ] != 0.0F;)
   3582               137            1: T && T
               994    137            2: T && F
              1313    137            3: F && _
                      137            MC/DC (cond 1): 1 + 3
                      137            MC/DC (cond 2): 1 + 2
                      138          }+
    859       1448    139          if (j < n - 1)
                      140          }+
   1313        135    141          else if (j == n)
                      142          }+
   5435       2307    143          for (;k < j;)
                      144          }+
                      147        }+
 249119     220035    148        else if (mybits <= diff && x0 != 0.0F)
 249119               148          1: T && T
                 0    148          2: T && F
            220035    148          3: F && _
                      148          MC/DC (cond 1): 1 + 3
                   -  148          MC/DC (cond 2): 1 - 2
                      152        }+
                      153        else
    909     219126    155          if (( p [ k ] += x0 ) == 0.0F)
   4516        909    157            for (;++ m < n && ( p [ m - 1 ] = p [ m ] ) != 0.0F;)
   4516               157              1: T && T
               323    157              2: T && F
               586    157              3: F && _
                      157              MC/DC (cond 1): 1 + 3
                      157              MC/DC (cond 2): 1 + 2
                      157            }+
     15        894    158            if (p [ k ] == 0.0F)
     15               159              break
                      159            }+
                      160          }+
  31892     188128    163          if (prevexp - mybits < xexp)
   1464      30428    167            if (( p [ k ] -= x0 ) == 0.0F)
   2521       1464    169              for (;++ m < n && ( p [ m - 1 ] = p [ m ] ) != 0.0F;)
   2521               169                1: T && T
               561    169                2: T && F
               903    169                3: F && _
                      169                MC/DC (cond 1): 1 + 3
                      169                MC/DC (cond 2): 1 + 2
                      169              }+
                      170            }+
   7276      24616    171            if (-- k == 0)
                      172            }+
                      173            else
                      178            }+
                      179          }+
  17753     170375    180          else if (k + 1 == n)
  17753               181            break
                      181          }+
                      182          else
  46892     123483    191            ternary-?: x0 != 0.0F
                      194          }+
                      195        }+
                      196      }+
                      197    }+
 116287               198    return ( p )
                      199 }

***TER  79 % ( 55/ 70) of FUNCTION _FXp_addh()
        89 % ( 77/ 87) statement
```

The analysis is done by the lines that are not covered

- 92: n (number of bytes) is either 2 or 4 in our cases
- 100: errx undefined cases for scaled=x0 are handled from the toplevel functions, hence this code is dead for us
- 123: same for y0
- 148: x0 may never be 0.0, since this excluded from errx<0 implying x0!=0.0)

7.6 Analysis of code coverage of function _Xp_mulh: OK



```
     10136          201 FTYPE *FNAME(Xp_mulh)(FTYPE *p, int n, FTYPE x0)
                    202     {   /* multiply by a half-precision value */
                    203     short errx;
                    204     int j, k;
                    205     FTYPE buf[NBUF];
                    206
     10136       0  207     if (0 < n)
                    208         {   /* check for special values */
                    209         buf[0] = p[0] * x0;
         0   10136  210         if (0 <= (errx = FNAME(Dtest)(&buf[0])))
                    211             {   /* quit early on 0, Inf, or NaN */
         0       0  212             if (errx == _NANCODE)
                    213                 _Feraise(_FE_INVALID);
                    214             p[0] = buf[0];
         0       0  215             if (0 < errx && 1 < n)
         0          215            1: T && T
                 0  215            2: T && F
                 0  215            3: F && _
         -          215            MC/DC (cond 1): 1 - 3
         -          215            MC/DC (cond 2): 1 - 2
                    216                 p[1] = FLIT(0.0);
         0          217             return (p);
                    218             }
                    219         p[0] = FLIT(0.0);
                    220         }
                    221
     59052   10105  222     for (j = 1, k = 0; k < n; ++k, --j)
```

_Xp_mulh is not called with special values because

1) N is always >0
2) There are no out of range values INF/NAN in buf[0]

7.6.1 N is always >0 OK

```
$ grep Xp_mulh `find .`
./sources/dinkumware/source/xldtob.c:                     FNAME(Xp_mulh)(xpx, ACSIZE, SCALE_NDIG);
./sources/dinkumware/source/xxfma.h:                FNAME(Xp_mulh)(xpx, ACSIZE, xpy[0]);
./sources/dinkumware/source/xxfma.h:                    FNAME(Xp_mulh)(xpw, ACSIZE, xpy[1]);
./sources/dinkumware/source/xxpow.h:                FNAME(Xp_mulh)(xpy, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxpow.h:                    FNAME(Xp_mulh)(xpw, ACSIZE, xpx[i]);
./sources/dinkumware/source/xxpow.h:            FNAME(Xp_mulh)(xpz, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxpow.h:                FNAME(Xp_mulh)(xpw, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxprec.h:FTYPE *FNAME(Xp_mulh)(FTYPE *p, int n, FTYPE x0)
./sources/dinkumware/source/xxxprec.h:  FNAME(Xp_mulh)(p, n, (FTYPE)10000);
./sources/dinkumware/source/xxxprec.h:          FNAME(Xp_mulh)(p, n, q[0]);
./sources/dinkumware/source/xxxprec.h:          FNAME(Xp_mulh)(p, n, q[0]);    /* form first partial product in place */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulh)(pac, n, q[j]);
./sources/dinkumware/source/xxxprec.h:          FNAME(Xp_mulh)(p, n, q[0]);
./sources/dinkumware/source/xxxprec.h:          FNAME(Xp_mulh)(p, n, q[0]);    /* form first partial product in place */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulh)(pac, n, q[j]);
```

```
./sources/dinkumware/source/xxxprec.h:          FNAME(Xp_mulh)(py, n, -FLIT(1.0));      /* py = -x */
./sources/dinkumware/source/xxxprec.h:                FNAME(Xp_mulh)(pac, n, -FLIT(0.5));
./sources/dinkumware/source/xxxquad.h:                FNAME(Xp_mulh)(xpy, ACSIZE, -g);
./sources/dinkumware/source/xxxquad.h:                FNAME(Xp_mulh)(xpz, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxxquad.h:                      FNAME(Xp_mulh)(xpy, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxquad.h:                FNAME(Xp_mulh)(xpy, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxxquad.h:                      FNAME(Xp_mulh)(xpw, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxquad.h:                FNAME(Xp_mulh)(xpw, ACSIZE, -g * FLIT(0.25) *
inv_fracbits);
```

Out of xxxprec it is always called with ACSIZE as argument which is defined

```
$ grep ACSIZE `find .`|grep define

./sources/dinkumware/source/xgetint.c:#define ACSIZE    32      /* holds only prefix, m.s. digits */
./sources/dinkumware/source/xldtob.c:#define ACSIZE     3       /* size of extended-precision accumulators */
./sources/dinkumware/source/xwgetint.c:#define ACSIZE   32      /* holds only prefix, m.s. digits */
./sources/dinkumware/source/xxfma.h:#define ACSIZE      4
./sources/dinkumware/source/xxpow.h:#define ACSIZE      4       /* size of extended-precision accumulators */
./sources/dinkumware/source/xxstod.h:#define ACSIZE     4       /* size of extended-precision accumulators */
./sources/dinkumware/source/xxxdtent.h:#define ACSIZE   4       /* size of extended-precision accumulators */
./sources/dinkumware/source/xxxdtent.h:#define BIAS     (ACSIZE * (FBITS / 2))  /* avoid denorms for finite
values */
./sources/dinkumware/source/xxxquad.h:#define ACSIZE    (sizeof c / sizeof c[0])
```

Since sizeof c is 6 times sizeof c[0] ACSIZE>0
Within xxxprec.h n is passed from the functions (by passing n without changing it)

- _Xp_mulx(double *p, int n, const double *q, int m, double *ptemp2)
- _Xp_invx(double *p, int n, double *ptemp4)
- _Xp_sqrtx(double *p, int n, double *ptemp4)

_Xp_invx and _Xp_sqrtx are not called (verified by not finding a call in the sources, not a computed call tree in the analysis) at all in the library and should be removed.
_Xp_mulx is not covered at all (hence very likely not used in our functions), and the analysis of the code confirms that: It used in the following files:

```
oscar@valilap71 MINGW64 /e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/Libraries/Version_3
$ grep Xp_mulx `find .`

./sources/dinkumware/source/xldtob.c:                  FNAME(Xp_mulx)(xpx, ACSIZE, xpf, ACSIZE, xpt);
./sources/dinkumware/source/xldtob.c:             FNAME(Xp_mulx)(xpf, ACSIZE, xpw, ACSIZE, xpt);  /*
square 10^n */
./sources/dinkumware/source/xxstod.h:                  FNAME(Xp_mulx)(xpx, ACSIZE, xpf, ACSIZE, xpt);
./sources/dinkumware/source/xxstod.h:                  FNAME(Xp_mulx)(xpx, ACSIZE, xpf, ACSIZE, xpt);
./sources/dinkumware/source/xxxdtent.h:             FNAME(Xp_mulx)(xpx, ACSIZE, xpf, ACSIZE, xpt);
./sources/dinkumware/source/xxxdtent.h:        FNAME(Xp_mulx)(xpf, ACSIZE, xpw, ACSIZE, xpt);  /* square
10^n */
./sources/dinkumware/source/xxxprec.h:FTYPE *FNAME(Xp_mulx)(FTYPE *p, int n,
./sources/dinkumware/source/xxxprec.h:FTYPE *FNAME(__qcom_Xp_mulx)(FTYPE *p, int n,
./sources/dinkumware/source/xxxprec.h:                FNAME(Xp_mulx)(pac, n, py, n, ptemp2);
```

```
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulx)(pac, n, p, n, ptemp2);   /* y*(1-x*y) */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulx)(pac, n, p, n, ptemp2);
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulx)(pac, n, py, n, ptemp2);
./sources/dinkumware/source/xxxprec.h:
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulx)(p, n, py, n, ptemp2);    /* x*sqrt(1/x) */
```

- xldtop.c: contains long double functions that are not in scope/not used
- xxstod.h: strong to double (also not used/in scope)
- xxxdtent.h: contains the function Dtento which his only used in xxstod (see above)

Therefore _Xp_mult is not used and N is always >0

7.6.2 p[0]*x0 is always valid OK

Valid means that the result errx = dTest(&buf[0]) is always <0 (line 210) dTest returns 1 for INF, 2 for NAN and 0 for 0 -1 for normal numbers and -2 for denormalized numbers
Since _Xp_mulh is only called with valid value (INF/NAN checks are done in the main functions) the only ways to cover invalidate buf[0]=p[0]*x0 is

- P[0] or x0 are zero
- P[0]*x0 flows over

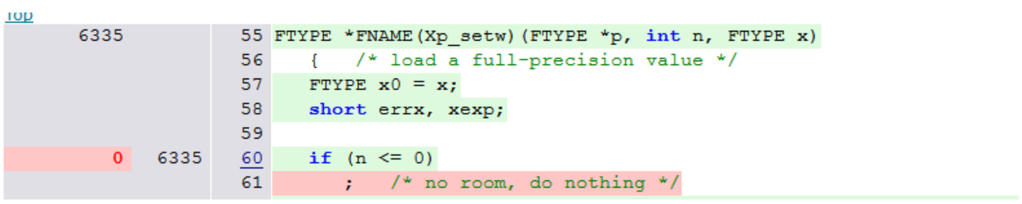
Xp_mulh is called in the following places:

```
oscar@valilap71 MINGW64 /e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/Libraries/Version_3
$ grep Xp_mulh `find .`
./sources/dinkumware/source/xldtob.c:                   FNAME(Xp_mulh)(xpx, ACSIZE, SCALE_NDIG);
./sources/dinkumware/source/xxfma.h:                FNAME(Xp_mulh)(xpx, ACSIZE, xpy[0]);
./sources/dinkumware/source/xxfma.h:                    FNAME(Xp_mulh)(xpw, ACSIZE, xpy[1]);
./sources/dinkumware/source/xxpow.h:                FNAME(Xp_mulh)(xpy, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxpow.h:                    FNAME(Xp_mulh)(xpw, ACSIZE, xpx[i]);
./sources/dinkumware/source/xxpow.h:            FNAME(Xp_mulh)(xpz, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxpow.h:                FNAME(Xp_mulh)(xpw, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxprec.h:FTYPE *FNAME(Xp_mulh)(FTYPE *p, int n, FTYPE x0)
./sources/dinkumware/source/xxxprec.h:  FNAME(Xp_mulh)(p, n, (FTYPE)10000);
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulh)(p, n, q[0]);
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulh)(p, n, q[0]);     /* form first partial product in
place */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulh)(pac, n, q[j]);
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulh)(p, n, q[0]);
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulh)(p, n, q[0]);     /* form first partial product in
place */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulh)(pac, n, q[j]);
./sources/dinkumware/source/xxxprec.h:        FNAME(Xp_mulh)(py, n, -FLIT(1.0));      /* py = -x */
./sources/dinkumware/source/xxxprec.h:              FNAME(Xp_mulh)(pac, n, -FLIT(0.5));
./sources/dinkumware/source/xxxquad.h:              FNAME(Xp_mulh)(xpy, ACSIZE, -g);
./sources/dinkumware/source/xxxquad.h:              FNAME(Xp_mulh)(xpz, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxxquad.h:                  FNAME(Xp_mulh)(xpy, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxquad.h:              FNAME(Xp_mulh)(xpy, ACSIZE, xpx[0]);
./sources/dinkumware/source/xxxquad.h:                  FNAME(Xp_mulh)(xpw, ACSIZE, xpx[1]);
./sources/dinkumware/source/xxxquad.h:              FNAME(Xp_mulh)(xpw, ACSIZE, -g * FLIT(0.25) *
inv_fracbits);
```

7.7 Analysis of code coverage of function _Xp_setw: OK

_Xp_setw is completely covered, since all 5 places are successfully analyzed to be not reachable.

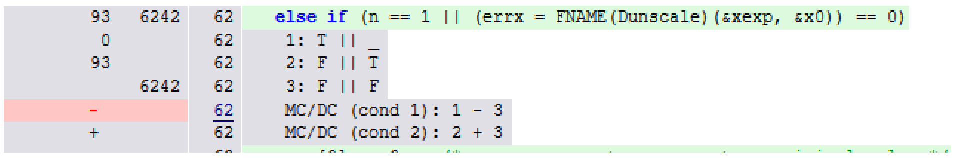
7.7.1 N>0



```
    6335           55 FTYPE *FNAME(Xp_setw)(FTYPE *p, int n, FTYPE x)
                   56    {   /* load a full-precision value */
                   57    FTYPE x0 = x;
                   58    short errx, xexp;
                   59
       0    6335   60    if (n <= 0)
                   61        ;   /* no room, do nothing */
```

N always greater, see Xp_mulh

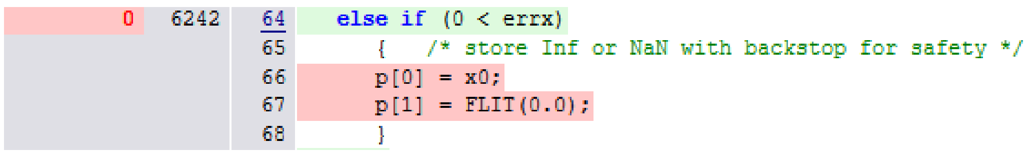
7.7.2 N>1



```
   93   6242   62     else if (n == 1 || (errx = FNAME(Dunscale)(&xexp, &x0)) == 0)
    0          62       1: T || _
   93          62       2: F || T
        6242   62       3: F || F
    -          62       MC/DC (cond 1): 1 - 3
    +          62       MC/DC (cond 2): 2 + 3
```

Only n==1 not coverable, see Xp_mulh, which holds also for >1, since 2 is the minimal value of N

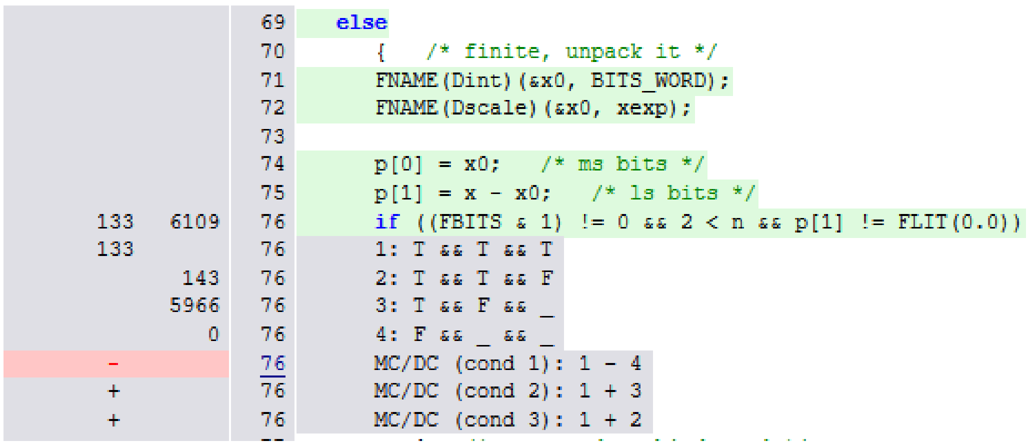
7.7.3 No NAN,INF



```
       0    6242   64     else if (0 < errx)
                   65         {   /* store Inf or NaN with backstop for safety */
                   66         p[0] = x0;
                   67         p[1] = FLIT(0.0);
                   68         }
```

Inf & NAN are always handled from the main library functions, such that this cannot occur on inputs and since all functions have INF/NAN test that are handled obviously not in _Xp_setw. So this is never reached here

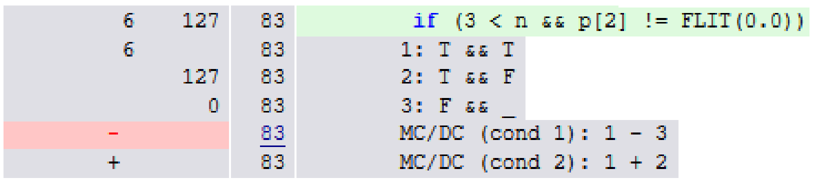
7.7.4 FBITS&1



```
                   69     else
                   70         {   /* finite, unpack it */
                   71         FNAME(Dint)(&x0, BITS_WORD);
                   72         FNAME(Dscale)(&x0, xexp);
                   73
                   74         p[0] = x0;    /* ms bits */
                   75         p[1] = x - x0;    /* ls bits */
     133    6109   76         if ((FBITS & 1) != 0 && 2 < n && p[1] != FLIT(0.0))
     133           76         1: T && T && T
             143   76         2: T && T && F
            5966   76         3: T && F && _
               0   76         4: F && _ && _
       -           76         MC/DC (cond 1): 1 - 4
       +           76         MC/DC (cond 2): 1 + 3
       +           76         MC/DC (cond 3): 1 + 2
```

FBITS is 53 (instead of 52) for double, see #107 and 24 (instead of 23) for float. Nevertheless it is constant (see #108) and here just use to re-use code. Therefore it is constant and not modifiable / coverable

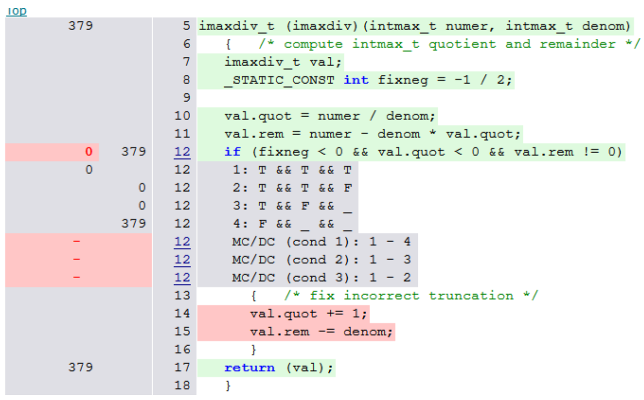
7.7.5 N!=3



```
     6   127    83       if (3 < n && p[2] != FLIT(0.0))
     6          83      1: T && T
         127    83      2: T && F
           0    83      3: F && _
     -          83      MC/DC (cond 1): 1 - 3
     +          83      MC/DC (cond 2): 1 + 2
```

N is either 2 or 4 but never three (except in long double case in ACSIZE in xldtob.c, see also Analysis of _Xp_mulh). Since N>2 is checked in line 62 (see previous section) N is always >3 and this cannot be reached here. Note the case for single float will be the opposite argumentation with FBITS&1 and N!=3, i.e. always false, always true,..

7.8 Analysis of code coverage of function imaxdiv: OK



```
Top
  379          5 imaxdiv_t (imaxdiv)(intmax_t numer, intmax_t denom)
               6    {   /* compute intmax_t quotient and remainder */
               7    imaxdiv_t val;
               8    _STATIC_CONST int fixneg = -1 / 2;
               9
              10    val.quot = numer / denom;
              11    val.rem = numer - denom * val.quot;
    0    379  12    if (fixneg < 0 && val.quot < 0 && val.rem != 0)
    0         12     1: T && T && T
           0  12     2: T && T && F
           0  12     3: T && F && _
         379  12     4: F && _ && _
    -         12     MC/DC (cond 1): 1 - 4
    -         12     MC/DC (cond 2): 1 - 3
    -         12     MC/DC (cond 3): 1 - 2
              13       {   /* fix incorrect truncation */
              14       val.quot += 1;
              15       val.rem -= denom;
              16       }
  379         17    return (val);
              18    }
```

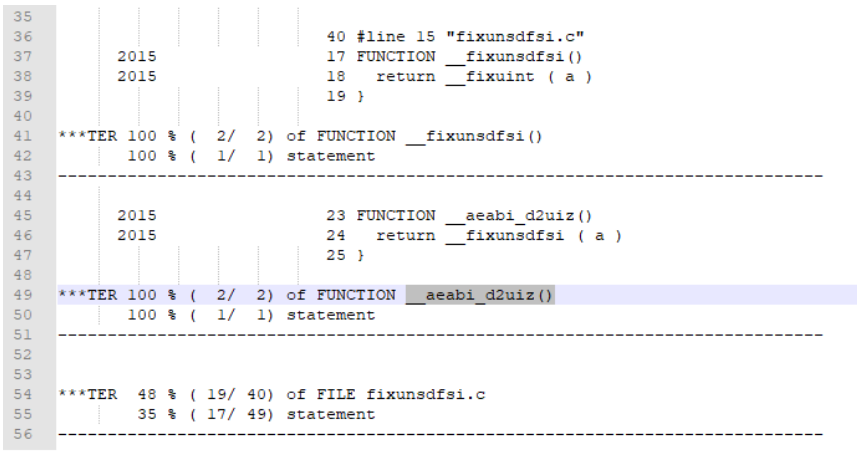
To clarify: Can condition fixneg < 0 be covered?

➔If a C Standard newer or equal to C99 is used the “if” condition cannot be covered. If C89 / C90 is used, fixneg can be also negative

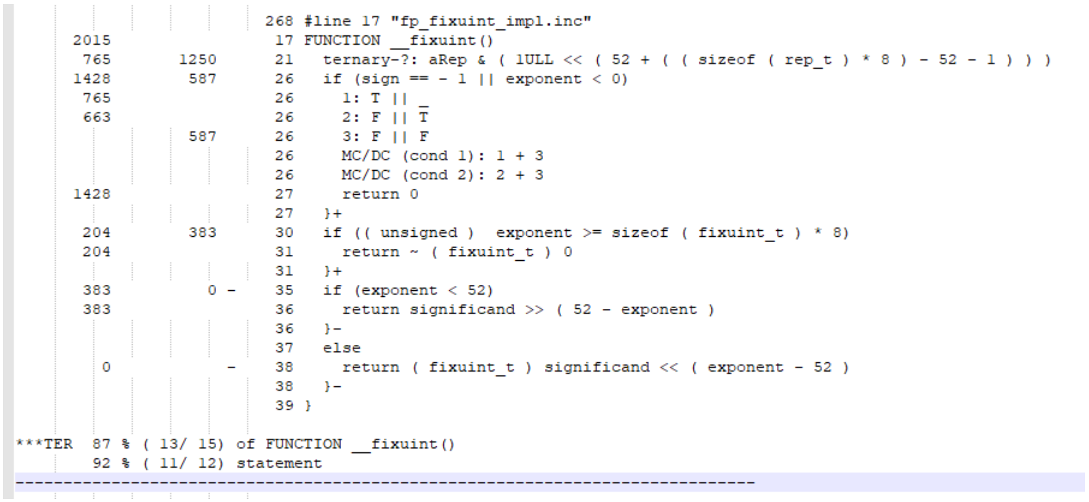
➔Statement from Hightec: C99 Standard is used

➔Condition fixneg < 0 cannot be covered.

7.9 Analysis of code coverage of function __aeabi_d2uiz: OK



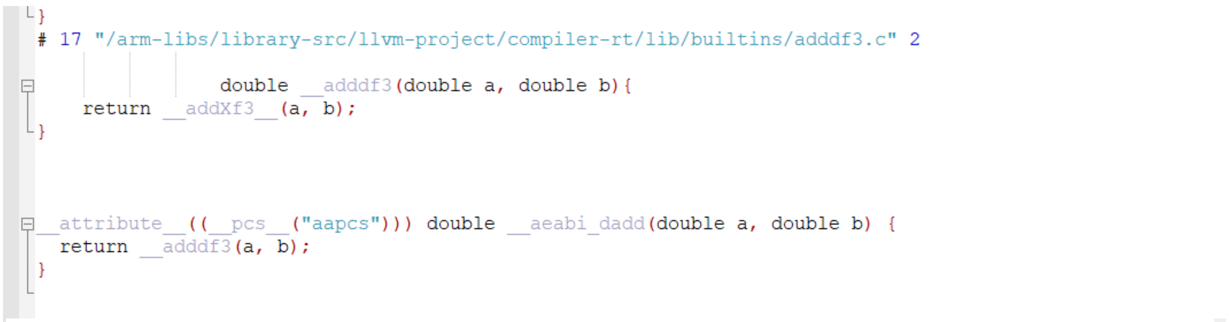
```

                                40 #line 15 "fixunsdfsi.c"
       2015                     17 FUNCTION __fixunsdfsi()
       2015                     18   return __fixuint ( a )
                                19 }

***TER 100 % (  2/  2) of FUNCTION __fixunsdfsi()
        100 % (  1/  1) statement
----------------------------------------------------------------------------

       2015                     23 FUNCTION __aeabi_d2uiz()
       2015                     24   return __fixunsdfsi ( a )
                                25 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_d2uiz()
        100 % (  1/  1) statement
----------------------------------------------------------------------------


***TER  48 % ( 19/ 40) of FILE fixunsdfsi.c
         35 % ( 17/ 49) statement
----------------------------------------------------------------------------
```



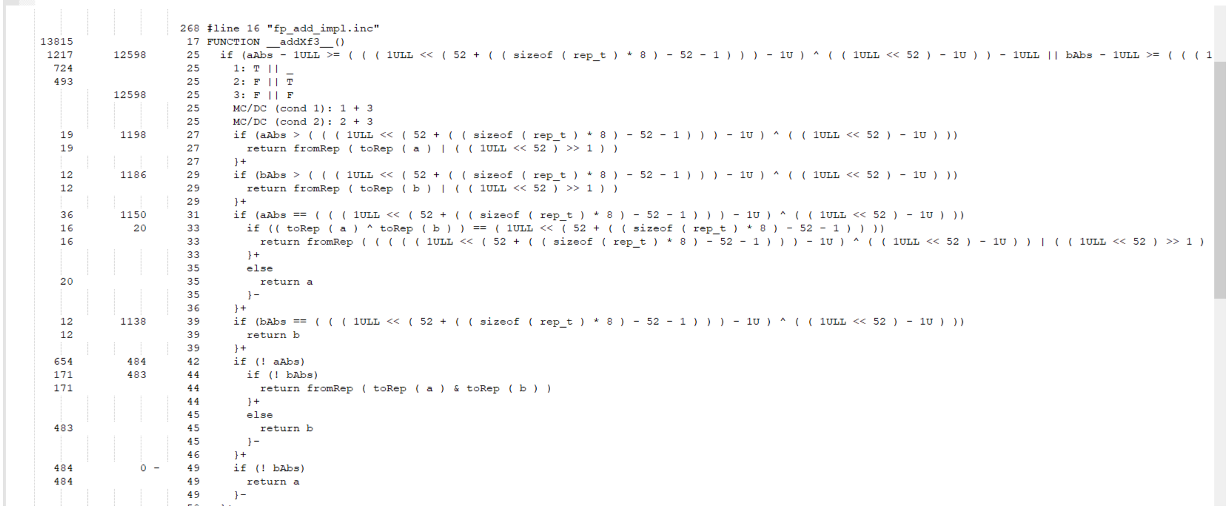
```
                        268 #line 17 "fp_fixuint_impl.inc"
  2015                   17 FUNCTION __fixuint()
   765      1250         21   ternary-?: aRep & ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) )
  1428       587         26   if (sign == - 1 || exponent < 0)
   765                   26     1: T || _
   663                   26     2: F || T
             587         26     3: F || F
                         26     MC/DC (cond 1): 1 + 3
                         26     MC/DC (cond 2): 2 + 3
  1428                   27     return 0
                         27   }+
   204       383         30   if (( unsigned )  exponent >= sizeof ( fixuint_t ) * 8)
   204                   31     return ~ ( fixuint_t ) 0
                         31   }+
   383         0 -       35   if (exponent < 52)
   383                   36     return significand >> ( 52 - exponent )
                         36   }-
                         37   else
     0             -     38     return ( fixuint_t ) significand << ( exponent - 52 )
                         38   }-
                         39 }

***TER  87 % ( 13/ 15) of FUNCTION __fixuint()
        92 % ( 11/ 12) statement
----------------------------------------------------------------------------
```

The “if” condition in line 30 checks whether exponent >= 32. In this case the function returns.
Thus, if exponent >= 52, it never reaches the else branch in lines 37 to 38.

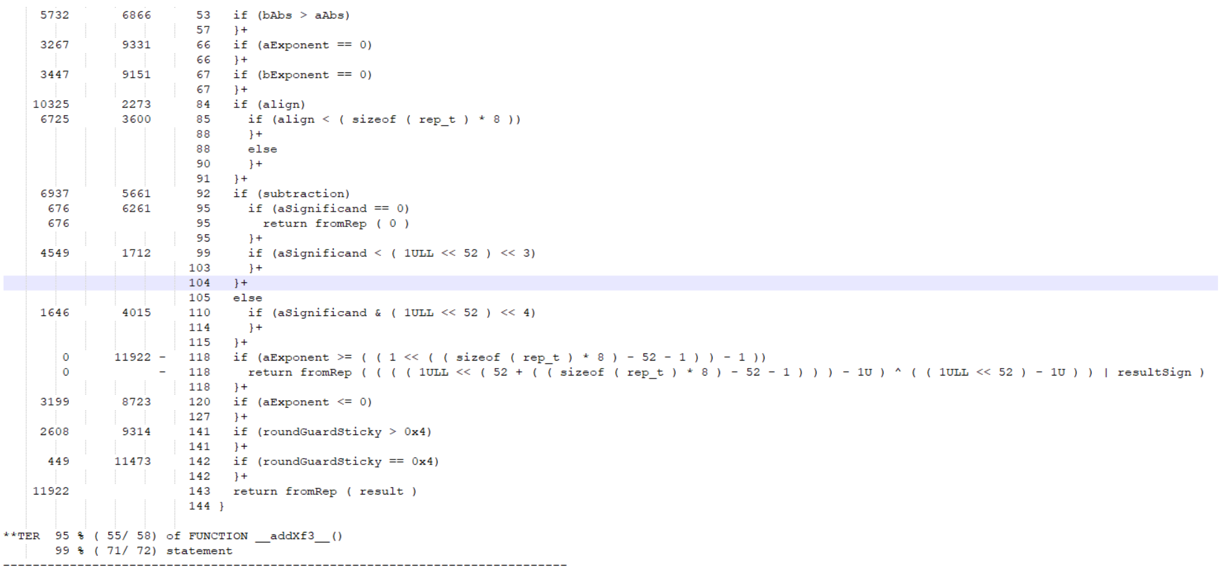
7.10 Analysis of code coverage of function __aeabi_dadd / __addXf3__: OK



```
}
# 17 "/arm-libs/library-src/llvm-project/compiler-rt/lib/builtins/adddf3.c" 2

                double __adddf3(double a, double b){
    return __addXf3__(a, b);
}



__attribute__((__pcs__("aapcs"))) double __aeabi_dadd(double a, double b) {
  return __adddf3(a, b);
}
```

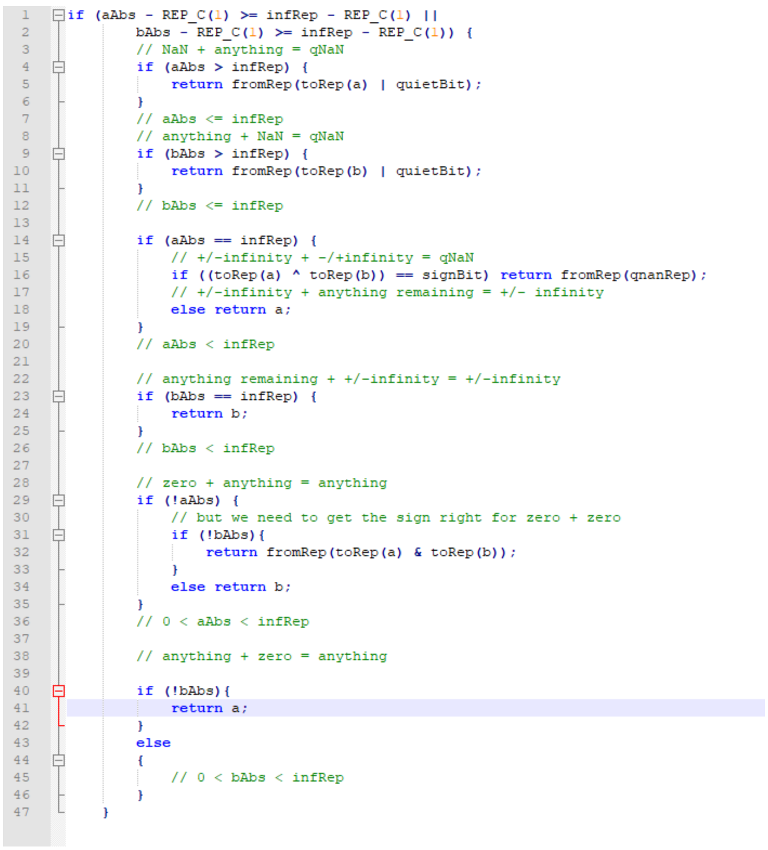


```
                      268 #line 16 "fp_add_impl.inc"
13815                  17 FUNCTION __addXf3__()
 1217      12598       25   if (aAbs - 1ULL >= ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) - 1ULL || bAbs - 1ULL >= ( ( ( 1
  724                  25     1: T || _
  493                  25     2: F || T
           12598       25     3: F || F
                       25     MC/DC (cond 1): 1 + 3
                       25     MC/DC (cond 2): 2 + 3
   19       1198       27     if (aAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   19                  27       return fromRep ( toRep ( a ) | ( ( 1ULL << 52 ) >> 1 ) )
                       27     }+
   12       1186       29     if (bAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                  29       return fromRep ( toRep ( b ) | ( ( 1ULL << 52 ) >> 1 ) )
                       29     }+
   36       1150       31     if (aAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   16         20       33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ))
   16                  33         return fromRep ( ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | ( ( 1ULL << 52 ) >> 1 )
                       33       }+
                       35       else
   20                  35         return a
                       35       }-
                       36     }+
   12       1138       39     if (bAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                  39       return b
                       39     }+
  654        484       42     if (! aAbs)
  171        483       44       if (! bAbs)
  171                  44         return fromRep ( toRep ( a ) & toRep ( b ) )
                       44       }+
                       45       else
  483                  45         return b
                       45       }-
                       46     }+
  484          0 -     49     if (! bAbs)
  484                  49       return a
                       49     }-
```



```
  5732      6866      53  if (bAbs > aAbs)
                      57  }+
  3267      9331      66  if (aExponent == 0)
                      66  }+
  3447      9151      67  if (bExponent == 0)
                      67  }+
 10325      2273      84  if (align)
  6725      3600      85    if (align < ( sizeof ( rep_t ) * 8 ))
                      88    }+
                      88    else
                      90    }+
                      91  }+
  6937      5661      92  if (subtraction)
   676      6261      95    if (aSignificand == 0)
   676                95      return fromRep ( 0 )
                      95    }+
  4549      1712      99    if (aSignificand < ( 1ULL << 52 ) << 3)
                     103    }+
                     104  }+
                     105  else
  1646      4015     110    if (aSignificand & ( 1ULL << 52 ) << 4)
                     114    }+
                     115  }+
     0     11922 -   118  if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) - 1 ))
     0           -   118    return fromRep ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | resultSign )
                     118  }+
  3199      8723     120  if (aExponent <= 0)
                     127  }+
  2608      9314     141  if (roundGuardSticky > 0x4)
                     141  }+
   449     11473     142  if (roundGuardSticky == 0x4)
                     142  }+
 11922               143  return fromRep ( result )
                     144 }

**TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
       99 % ( 71/ 72) statement
-----------------------------------------------------------------------------
```

7.10.1 “Else” branch of “If” condition if (! bAbs)



```
1   if (aAbs - REP_C(1) >= infRep - REP_C(1) ||
2           bAbs - REP_C(1) >= infRep - REP_C(1)) {
3           // NaN + anything = qNaN
4           if (aAbs > infRep) {
5               return fromRep(toRep(a) | quietBit);
6           }
7           // aAbs <= infRep
8           // anything + NaN = qNaN
9           if (bAbs > infRep) {
10              return fromRep(toRep(b) | quietBit);
11          }
12          // bAbs <= infRep
13
14          if (aAbs == infRep) {
15              // +/-infinity + -/+infinity = qNaN
16              if ((toRep(a) ^ toRep(b)) == signBit) return fromRep(qnanRep);
17              // +/-infinity + anything remaining = +/- infinity
18              else return a;
19          }
20          // aAbs < infRep
21
22          // anything remaining + +/-infinity = +/-infinity
23          if (bAbs == infRep) {
24              return b;
25          }
26          // bAbs < infRep
27
28          // zero + anything = anything
29          if (!aAbs) {
30              // but we need to get the sign right for zero + zero
31              if (!bAbs){
32                  return fromRep(toRep(a) & toRep(b));
33              }
34              else return b;
35          }
36          // 0 < aAbs < infRep
37
38          // anything + zero = anything
39
40          if (!bAbs){
41              return a;
42          }
43          else
44          {
45              // 0 < bAbs < infRep
46          }
47      }
```

The “Else” branch of “If” condition if(!bAbs) in line 49 cannot be covered, because

a) On the one hand, the “If” condition if(aAbs – REP_C(1) >= infRep – REP_C(1) || bAbs – REP_C(1) >= infRep – REP_C(1)) needs to be fulfilled

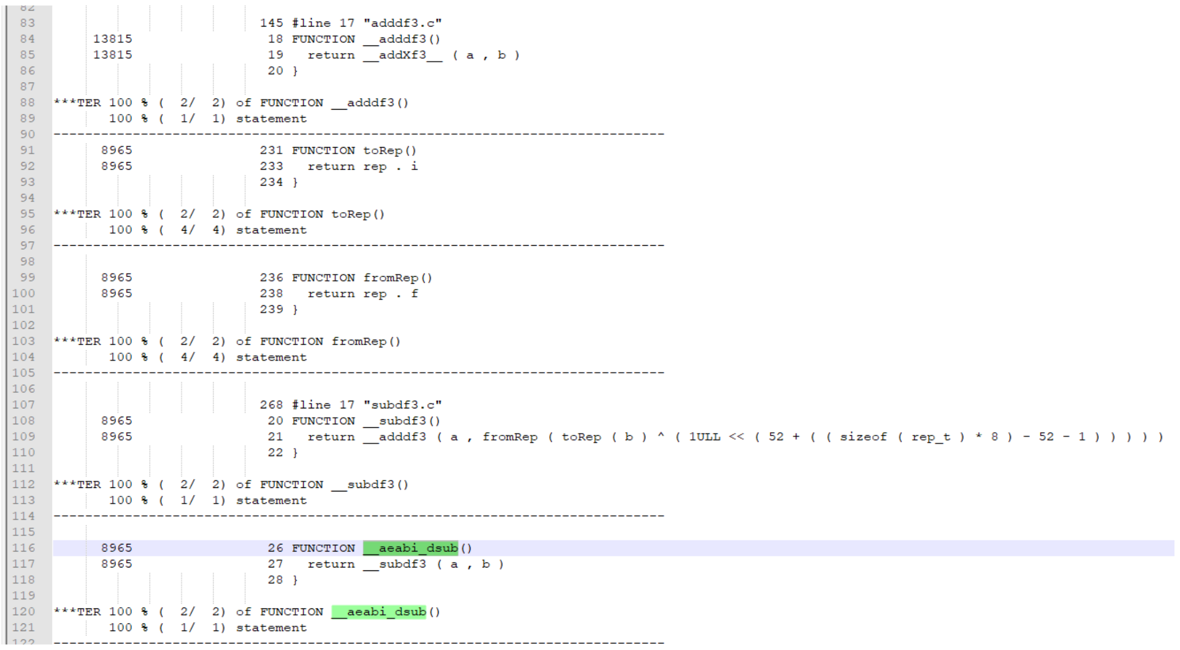
b) On the other hand in order to reach the “Else” branch in line 43

   - The “if” condition if (aAbs > infRep) in line 4 needs to evaluate to false (needs to be skipped)

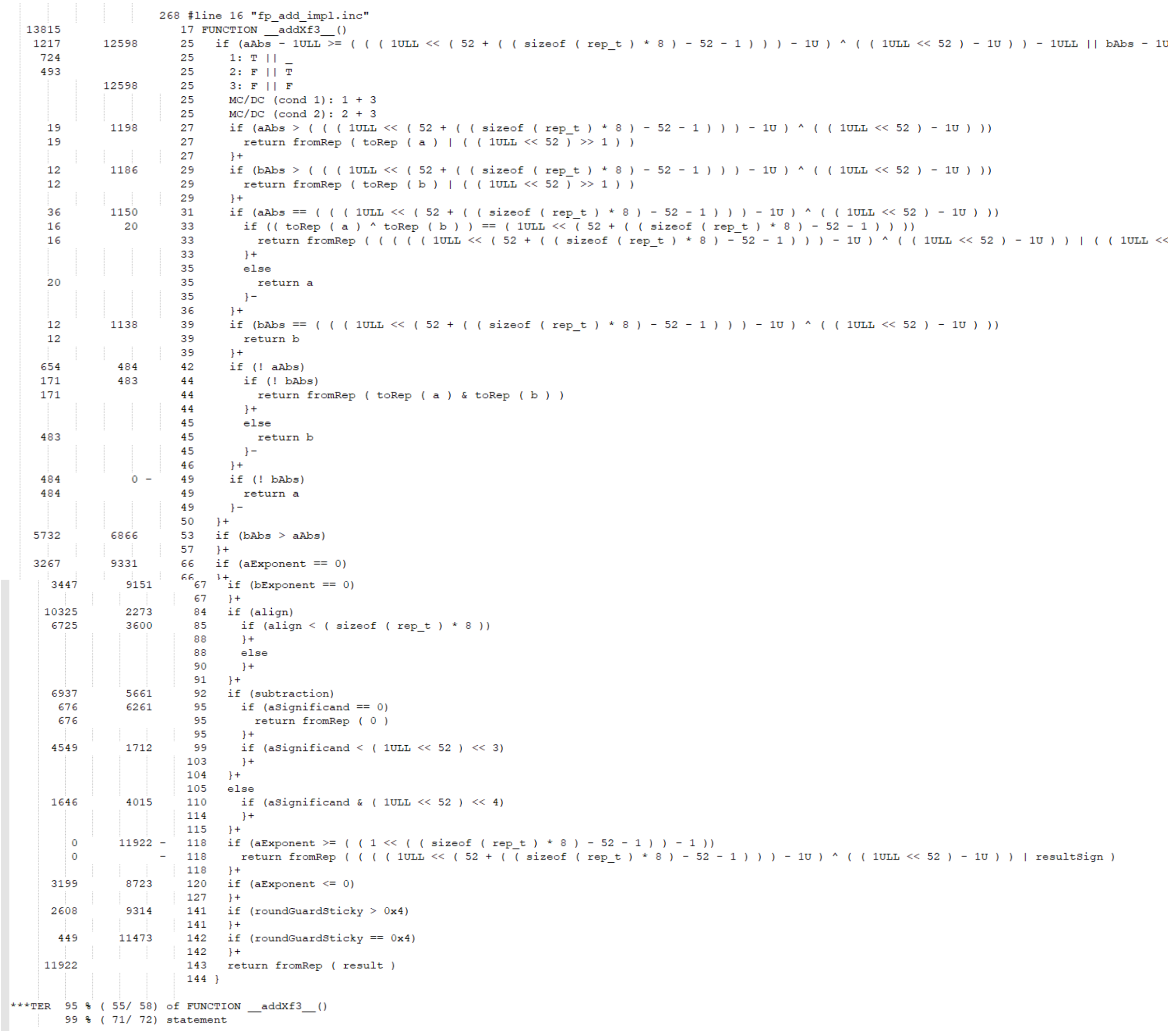
- The “if” condition if (bAbs > infRep) in line 9 needs to evaluate to false (needs to be skipped)
- The “if” condition if (aAbs == infRep) in line 14 needs to evaluate to false (needs to be skipped)
- The “if” condition if (bAbs == infRep) in line 23 needs to evaluate to false (needs to be skipped)
- The “if” condition if (!aAbs) in line 29 needs to evaluate to false (needs to be skipped)
- The “if” condition if (!bAbs) in line 40 needs to evaluate to false (needs to be skipped)

➔ So in the end the following should apply when reaching the “Else” branch in line 43:
0 < aAbs < infRep AND 0 < bAbs < infRep. But this is not possible because also the condition in a) needs to be fulfilled.

7.11 Analysis of code coverage of function __aeabi_dsub / __addXf3__: OK



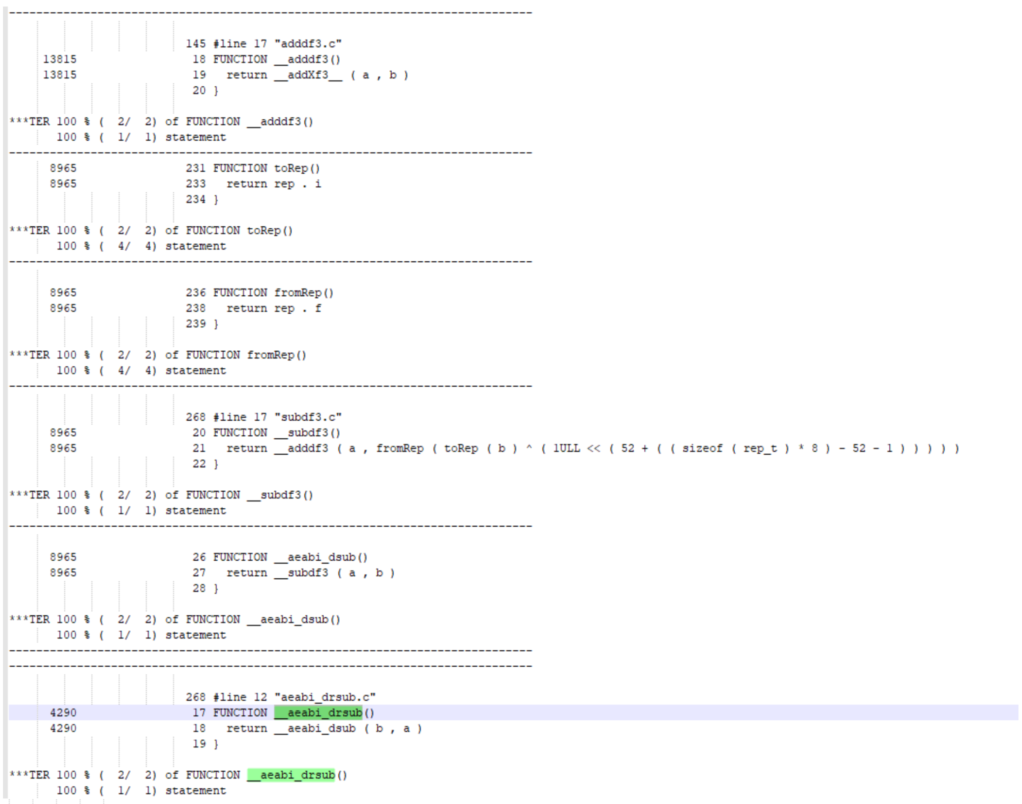
```
                           145 #line 17 "adddf3.c"
     13815                  18 FUNCTION __adddf3()
     13815                  19   return __addXf3__ ( a , b )
                            20 }

***TER 100 % (  2/  2) of FUNCTION __adddf3()
       100 % (  1/  1) statement
------------------------------------------------------------------------------
      8965                 231 FUNCTION toRep()
      8965                 233   return rep . i
                           234 }

***TER 100 % (  2/  2) of FUNCTION toRep()
       100 % (  4/  4) statement
------------------------------------------------------------------------------

      8965                 236 FUNCTION fromRep()
      8965                 238   return rep . f
                           239 }

***TER 100 % (  2/  2) of FUNCTION fromRep()
       100 % (  4/  4) statement
------------------------------------------------------------------------------

                           268 #line 17 "subdf3.c"
      8965                  20 FUNCTION __subdf3()
      8965                  21   return __adddf3 ( a , fromRep ( toRep ( b ) ^ ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) ) )
                            22 }

***TER 100 % (  2/  2) of FUNCTION __subdf3()
       100 % (  1/  1) statement
------------------------------------------------------------------------------

      8965                  26 FUNCTION __aeabi_dsub()
      8965                  27   return __subdf3 ( a , b )
                            28 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_dsub()
       100 % (  1/  1) statement
------------------------------------------------------------------------------
```



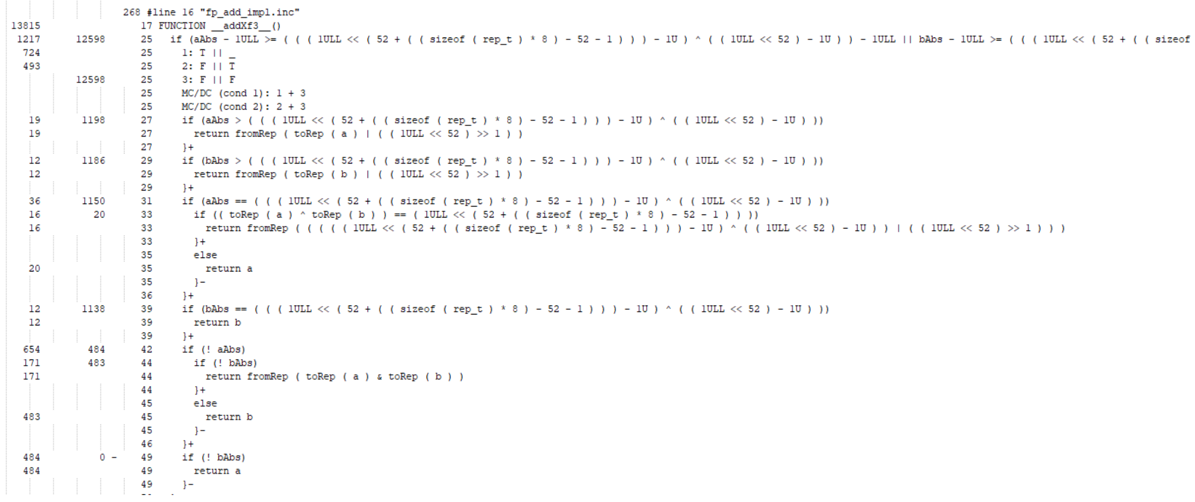
```
                    268 #line 16 "fp_add_impl.inc"
13815                  17 FUNCTION __addXf3__()
 1217     12598        25   if (aAbs - 1ULL >= ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) - 1ULL || bAbs - 1U
  724                  25     1: T || _
  493                  25     2: F || T
          12598        25     3: F || F
                       25     MC/DC (cond 1): 1 + 3
                       25     MC/DC (cond 2): 2 + 3
   19      1198        27     if (aAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   19                  27       return fromRep ( toRep ( a ) | ( ( 1ULL << 52 ) >> 1 ) )
                       27     }+
   12      1186        29     if (bAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                  29       return fromRep ( toRep ( b ) | ( ( 1ULL << 52 ) >> 1 ) )
                       29     }+
   36      1150        31     if (aAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   16        20        33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ))
   16                  33         return fromRep ( ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | ( ( 1ULL <<
                       33       }+
                       35       else
   20                  35         return a
                       35       }-
                       36     }+
   12      1138        39     if (bAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                  39       return b
                       39     }+
  654       484        42     if (! aAbs)
  171       483        44       if (! bAbs)
  171                  44         return fromRep ( toRep ( a ) & toRep ( b ) )
                       44       }+
                       45       else
  483                  45         return b
                       45       }-
                       46     }+
  484         0 -      49     if (! bAbs)
  484                  49       return a
                       49     }-
                       50   }+
 5732      6866        53   if (bAbs > aAbs)
                       57   }+
 3267      9331        66   if (aExponent == 0)
                       66   }+
  3447       9151        67   if (bExponent == 0)
                         67   }+
 10325       2273        84   if (align)
  6725       3600        85     if (align < ( sizeof ( rep_t ) * 8 ))
                         88     }+
                         88     else
                         90     }+
                         91   }+
  6937       5661        92   if (subtraction)
   676       6261        95     if (aSignificand == 0)
   676                   95       return fromRep ( 0 )
                         95     }+
  4549       1712        99     if (aSignificand < ( 1ULL << 52 ) << 3)
                        103     }+
                        104   }+
                        105   else
  1646       4015       110     if (aSignificand & ( 1ULL << 52 ) << 4)
                        114     }+
                        115   }+
     0      11922 -     118   if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) - 1 ))
     0            -     118     return fromRep ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | resultSign )
                        118   }+
  3199       8723       120   if (aExponent <= 0)
                        127   }+
  2608       9314       141   if (roundGuardSticky > 0x4)
                        141   }+
   449      11473       142   if (roundGuardSticky == 0x4)
                        142   }+
 11922                  143   return fromRep ( result )
                        144 }

***TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
        99 % ( 71/ 72) statement
```

Since __aeabi_dsub calls indirectly __addXf3__, the same argumentation regarding code coverage as in chapter 7.10 applies.

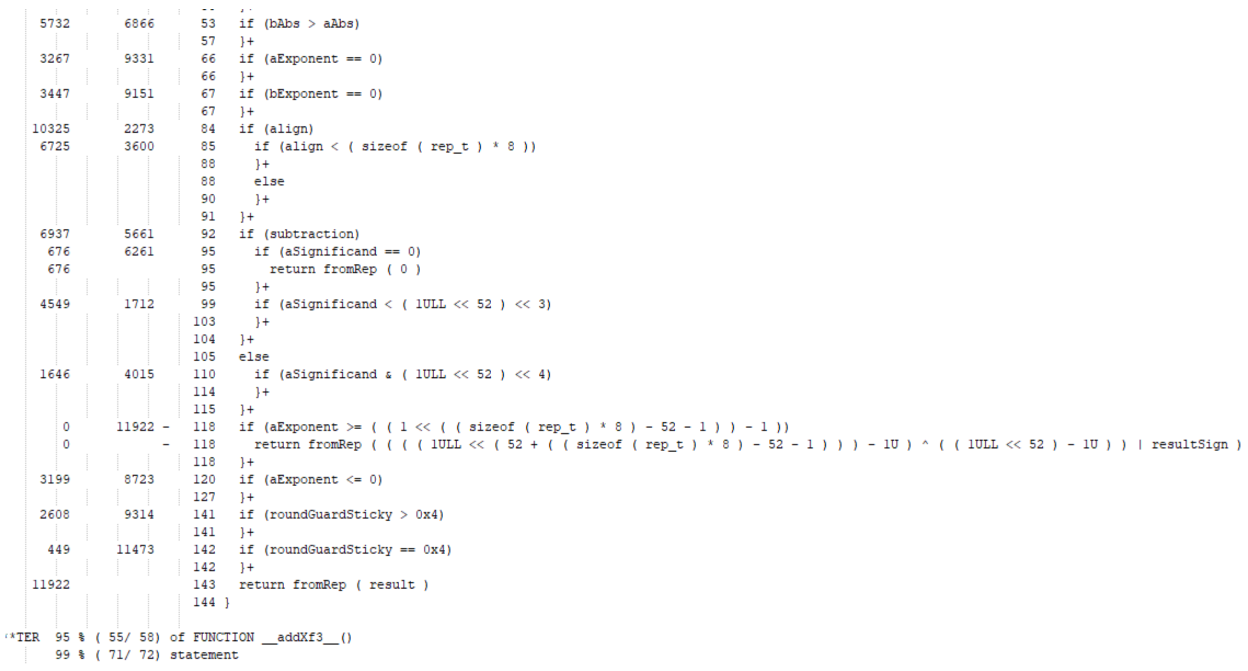
7.12 Analysis of code coverage of function __aeabi_drsub / __addXf3__: OK



```
-----------------------------------------------------------------------------

                      145 #line 17 "adddf3.c"
     13815             18 FUNCTION __adddf3()
     13815             19   return __addXf3__ ( a , b )
                       20 }

***TER 100 % (  2/  2) of FUNCTION __adddf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------
      8965            231 FUNCTION toRep()
      8965            233   return rep . i
                      234 }

***TER 100 % (  2/  2) of FUNCTION toRep()
       100 % (  4/  4) statement
-----------------------------------------------------------------------------

      8965            236 FUNCTION fromRep()
      8965            238   return rep . f
                      239 }

***TER 100 % (  2/  2) of FUNCTION fromRep()
       100 % (  4/  4) statement
-----------------------------------------------------------------------------

                      268 #line 17 "subdf3.c"
      8965             20 FUNCTION __subdf3()
      8965             21   return __adddf3 ( a , fromRep ( toRep ( b ) ^ ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) ) )
                       22 }

***TER 100 % (  2/  2) of FUNCTION __subdf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------

      8965             26 FUNCTION __aeabi_dsub()
      8965             27   return __subdf3 ( a , b )
                       28 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_dsub()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

                      268 #line 12 "aeabi_drsub.c"
      4290             17 FUNCTION __aeabi_drsub()
      4290             18   return __aeabi_dsub ( b , a )
                       19 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_drsub()
       100 % (  1/  1) statement
```



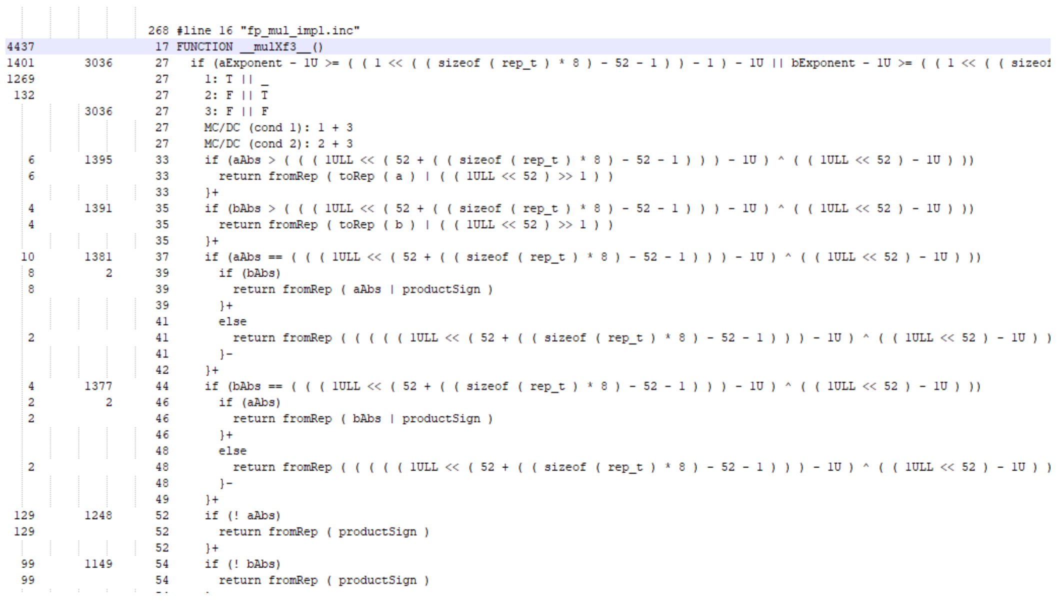
```
                 268 #line 16 "fp_add_impl.inc"
13815                17 FUNCTION __addXf3__()
 1217     12598      25   if (aAbs - 1ULL >= ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) - 1ULL || bAbs - 1ULL >= ( ( ( 1ULL << ( 52 + ( ( sizeof
  724                25     1: T || _
  493                25     2: F || T
          12598      25     3: F || F
                     25     MC/DC (cond 1): 1 + 3
                     25     MC/DC (cond 2): 2 + 3
   19      1198      27     if (aAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   19                27       return fromRep ( toRep ( a ) | ( ( 1ULL << 52 ) >> 1 ) )
                     27     }+
   12      1186      29     if (bAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                29       return fromRep ( toRep ( b ) | ( ( 1ULL << 52 ) >> 1 ) )
                     29     }+
   36      1150      31     if (aAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   16        20      33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ))
   16                33         return fromRep ( ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | ( ( 1ULL << 52 ) >> 1 ) ) )
                     33       }+
                     35       else
   20                35         return a
                     35       }-
                     36     }+
   12      1138      39     if (bAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   12                39       return b
                     39     }+
  654       484      42     if (! aAbs)
  171       483      44       if (! bAbs)
  171                44         return fromRep ( toRep ( a ) & toRep ( b ) )
                     44       }+
                     45       else
  483                45         return b
                     45       }-
                     46     }+
  484         0 -    49     if (! bAbs)
  484                49       return a
                     49     }-
```



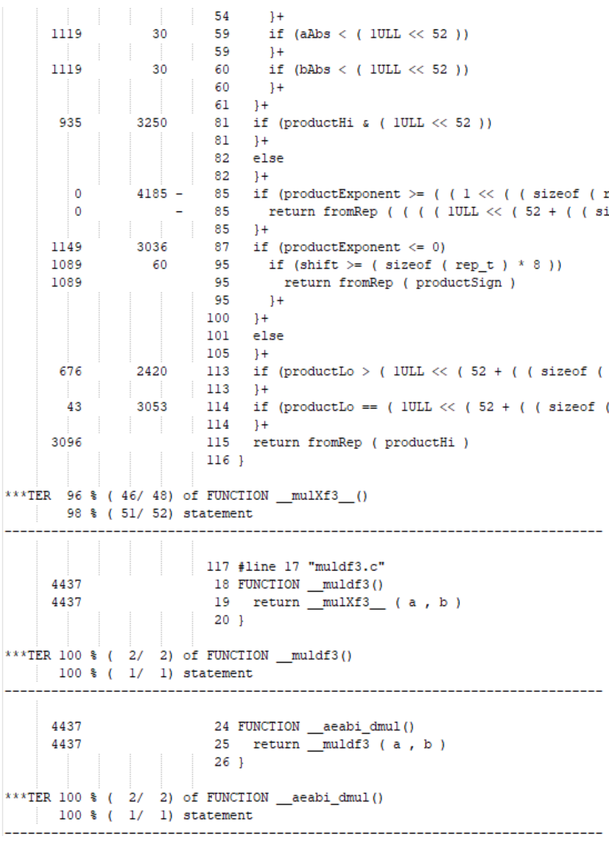
```
 5732      6866      53   if (bAbs > aAbs)
                     57   }+
 3267      9331      66   if (aExponent == 0)
                     66   }+
 3447      9151      67   if (bExponent == 0)
                     67   }+
10325      2273      84   if (align)
 6725      3600      85     if (align < ( sizeof ( rep_t ) * 8 ))
                     88     }+
                     88     else
                     90     }+
                     91   }+
 6937      5661      92   if (subtraction)
  676      6261      95     if (aSignificand == 0)
  676                95       return fromRep ( 0 )
                     95     }+
 4549      1712      99     if (aSignificand < ( 1ULL << 52 ) << 3)
                    103     }+
                    104   }+
                    105   else
 1646      4015     110     if (aSignificand & ( 1ULL << 52 ) << 4)
                    114     }+
                    115   }+
    0     11922 -   118   if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) - 1 ))
    0           -   118     return fromRep ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ) | resultSign )
                    118   }+
 3199      8723     120   if (aExponent <= 0)
                    127   }+
 2608      9314     141   if (roundGuardSticky > 0x4)
                    141   }+
  449     11473     142   if (roundGuardSticky == 0x4)
                    142   }+
11922               143   return fromRep ( result )
                    144 }

'*TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
       99 % ( 71/ 72) statement
```

Since __aeabi_drsub calls indirectly __addXf3__, the same argumentation regarding code coverage as in chapter 7.10 applies.

7.13 Analysis of code coverage of function __aeabi_dmul: OK

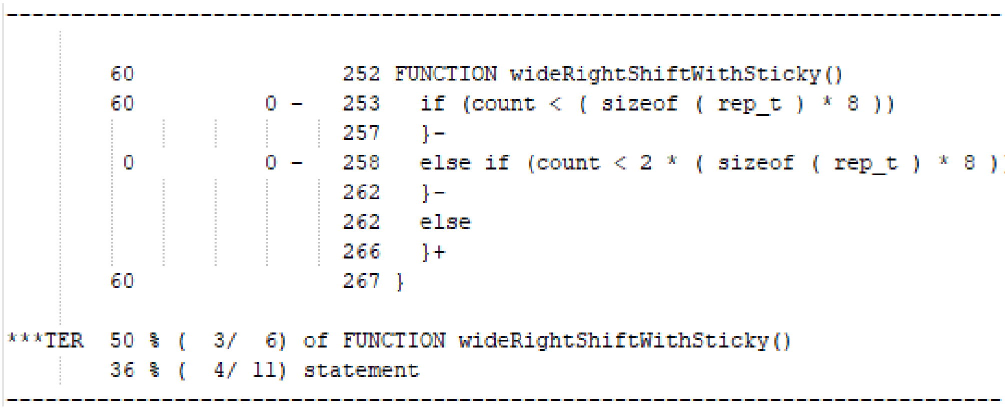


```
                   268 #line 16 "fp_mul_impl.inc"
4437                17 FUNCTION __mulXf3__()
1401      3036      27   if (aExponent - 1U >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) - 1 ) - 1U || bExponent - 1U >= ( ( 1 << ( ( sizeof
1269                27     1: T || _
 132                27     2: F || T
          3036      27     3: F || F
                    27     MC/DC (cond 1): 1 + 3
                    27     MC/DC (cond 2): 2 + 3
   6      1395      33     if (aAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   6                33       return fromRep ( toRep ( a ) | ( ( 1ULL << 52 ) >> 1 ) )
                    33     }+
   4      1391      35     if (bAbs > ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   4                35       return fromRep ( toRep ( b ) | ( ( 1ULL << 52 ) >> 1 ) )
                    35     }+
  10      1381      37     if (aAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   8         2      39       if (bAbs)
   8                39         return fromRep ( aAbs | productSign )
                    39       }+
                    41       else
   2                41         return fromRep ( ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) )
                    41       }-
                    42     }+
   4      1377      44     if (bAbs == ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) ))
   2         2      46       if (aAbs)
   2                46         return fromRep ( bAbs | productSign )
                    46       }+
                    48       else
   2                48         return fromRep ( ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U ) )
                    48       }-
                    49     }+
 129      1248      52     if (! aAbs)
 129                52       return fromRep ( productSign )
                    52     }+
  99      1149      54     if (! bAbs)
  99                54       return fromRep ( productSign )
```



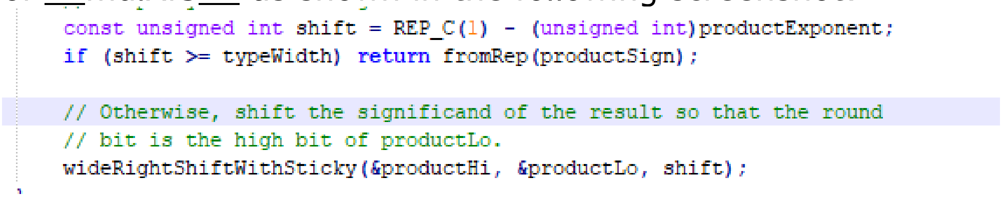
```
                        54     }+
   1119        30       59     if (aAbs < ( 1ULL << 52 ))
                        59     }+
   1119        30       60     if (bAbs < ( 1ULL << 52 ))
                        60     }+
                        61   }+
    935      3250       81   if (productHi & ( 1ULL << 52 ))
                        81   }+
                        82   else
                        82   }+
      0      4185 -     85   if (productExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) - 1 ))
      0           -     85     return fromRep ( ( ( ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ) - 1U ) ^ ( ( 1ULL << 52 ) - 1U )
                        85   }+
   1149      3036       87   if (productExponent <= 0)
   1089        60       95     if (shift >= ( sizeof ( rep_t ) * 8 ))
   1089                 95       return fromRep ( productSign )
                        95     }+
                       100   }+
                       101   else
                       105   }+
    676      2420      113   if (productLo > ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ))
                       113   }+
     43      3053      114   if (productLo == ( 1ULL << ( 52 + ( ( sizeof ( rep_t ) * 8 ) - 52 - 1 ) ) ))
                       114   }+
   3096                115   return fromRep ( productHi )
                       116 }

***TER  96 % ( 46/ 48) of FUNCTION __mulXf3__()
        98 % ( 51/ 52) statement
----------------------------------------------------------------------------

                       117 #line 17 "muldf3.c"
   4437                 18 FUNCTION __muldf3()
   4437                 19   return __mulXf3__ ( a , b )
                        20 }

***TER 100 % (  2/  2) of FUNCTION __muldf3()
       100 % (  1/  1) statement
----------------------------------------------------------------------------

   4437                 24 FUNCTION __aeabi_dmul()
   4437                 25   return __muldf3 ( a , b )
                        26 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_dmul()
       100 % (  1/  1) statement
----------------------------------------------------------------------------
```

7.13.1 Sub-function wideRightShiftWithSticky: OK



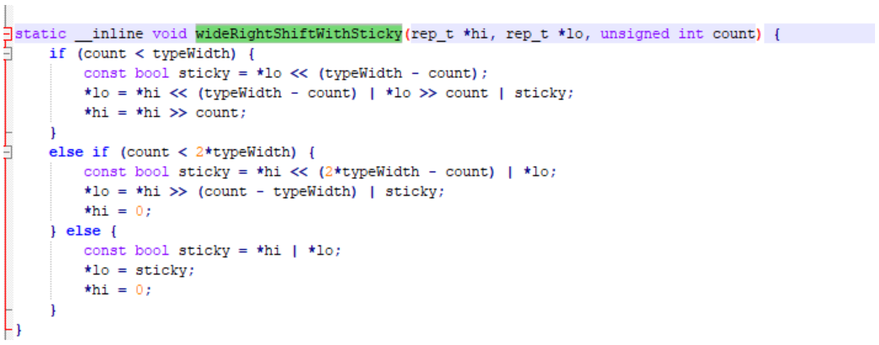
```
----------------------------------------------------------------------------

     60                   252 FUNCTION wideRightShiftWithSticky()
     60            0 -    253   if (count < ( sizeof ( rep_t ) * 8 ))
                          257   }-
      0            0 -    258   else if (count < 2 * ( sizeof ( rep_t ) * 8 ))
                          262   }-
                          262   else
                          266   }+
     60                   267 }

***TER  50 % (  3/  6) of FUNCTION wideRightShiftWithSticky()
        36 % (  4/ 11) statement
----------------------------------------------------------------------------
```

The sub-function wideRightShiftWithSticky is only called within the context of __mulXf3__ as shown in the following screenshot:



```
    const unsigned int shift = REP_C(1) - (unsigned int)productExponent;
    if (shift >= typeWidth) return fromRep(productSign);

    // Otherwise, shift the significand of the result so that the round
    // bit is the high bit of productLo.
    wideRightShiftWithSticky(&productHi, &productLo, shift);
```

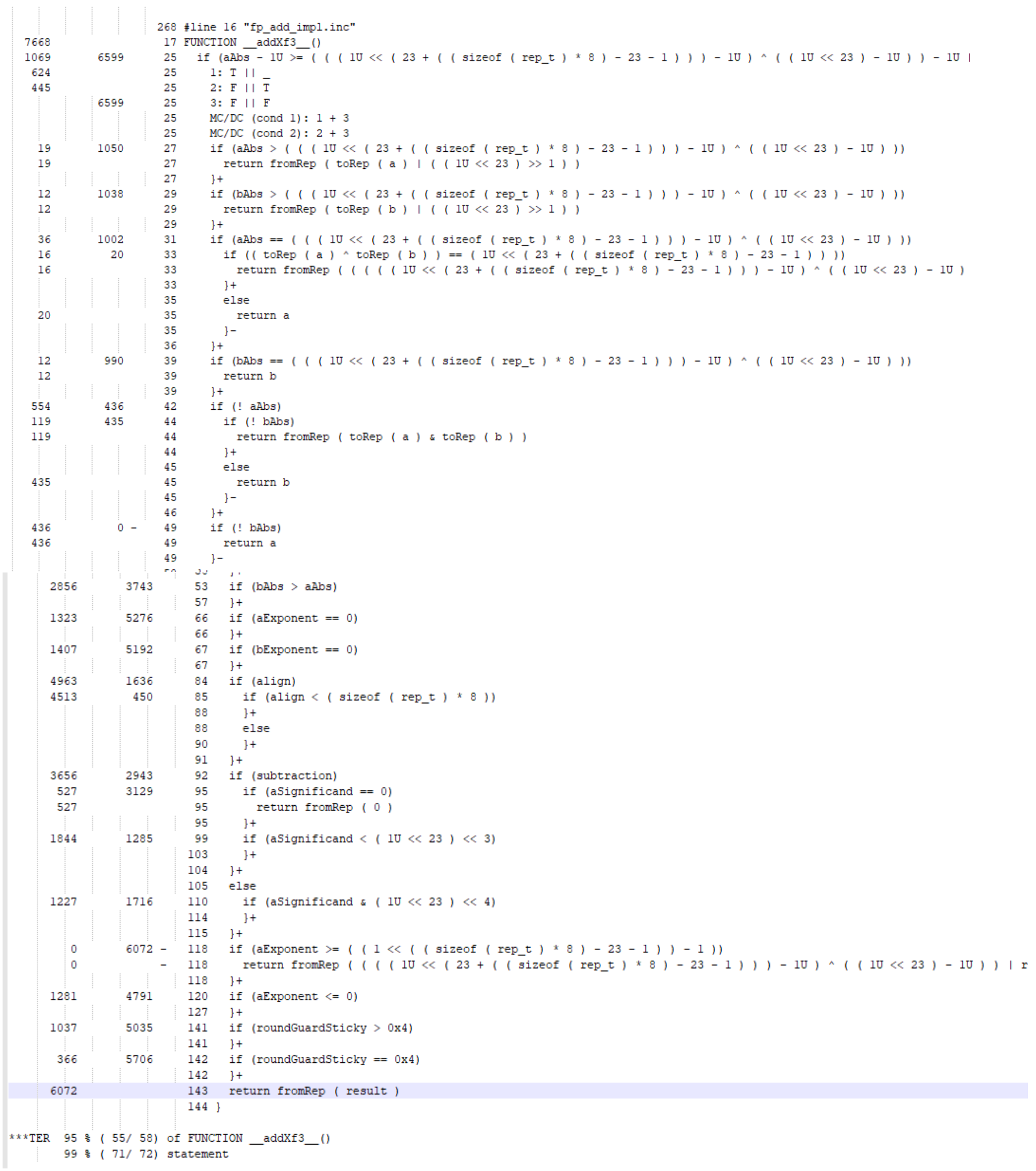
Due to the “If” statement if(shift >= typeWidth) return fromRep (productSign) it is ensured that wideRightShiftWithSticky is only called with shift < typeWidth.



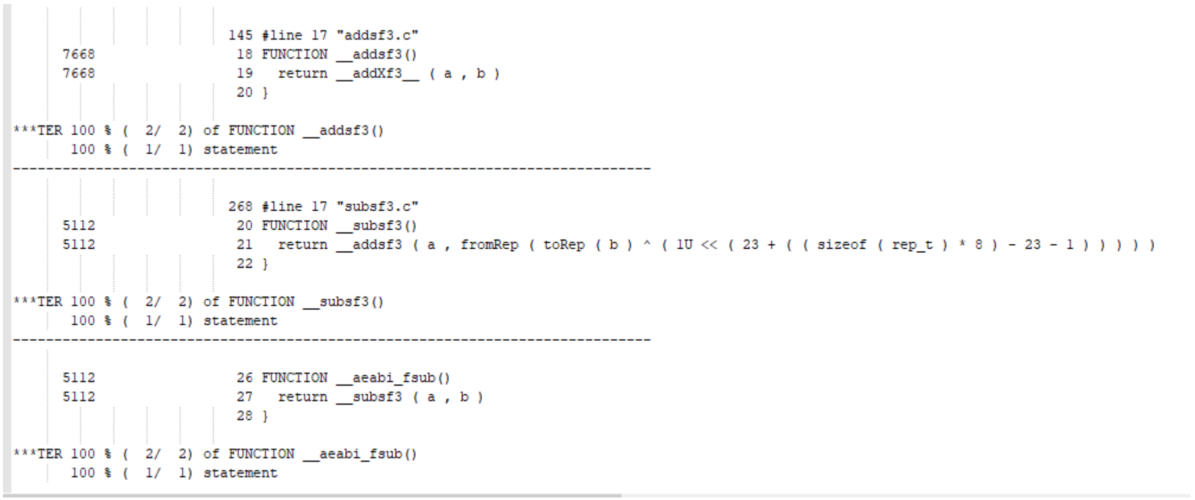
```
static __inline void wideRightShiftWithSticky(rep_t *hi, rep_t *lo, unsigned int count) {
    if (count < typeWidth) {
        const bool sticky = *lo << (typeWidth - count);
        *lo = *hi << (typeWidth - count) | *lo >> count | sticky;
        *hi = *hi >> count;
    }
    else if (count < 2*typeWidth) {
        const bool sticky = *hi << (2*typeWidth - count) | *lo;
        *lo = *hi >> (count - typeWidth) | sticky;
        *hi = 0;
    } else {
        const bool sticky = *hi | *lo;
        *lo = sticky;
        *hi = 0;
    }
}
```

Thus, within wideRightShiftWithSticky the “If” condition if(count < typeWidth) is always fulfilled and the else branch cannot be covered.

7.14 Analysis of code coverage of function __aeabi_fsub / __addXf3__: OK

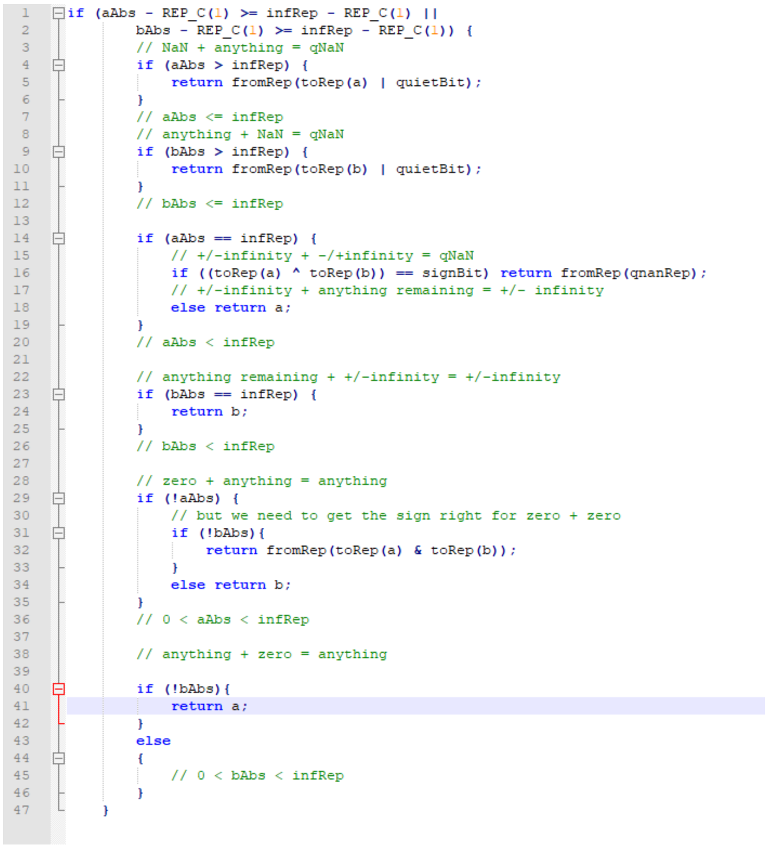


```
                      268 #line 16 "fp_add_impl.inc"
 7668                  17 FUNCTION __addXf3__()
 1069       6599       25   if (aAbs - 1U >= ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) - 1U |
  624                  25     1: T || _
  445                  25     2: F || T
            6599       25     3: F || F
                       25     MC/DC (cond 1): 1 + 3
                       25     MC/DC (cond 2): 2 + 3
   19       1050       27     if (aAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
   19                  27       return fromRep ( toRep ( a ) | ( ( 1U << 23 ) >> 1 ) )
                       27     }+
   12       1038       29     if (bAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
   12                  29       return fromRep ( toRep ( b ) | ( ( 1U << 23 ) >> 1 ) )
                       29     }+
   36       1002       31     if (aAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
   16         20       33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ))
   16                  33         return fromRep ( ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U )
                       33       }+
                       35       else
   20                  35         return a
                       35       }-
                       36     }+
   12        990       39     if (bAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
   12                  39       return b
                       39     }+
  554        436       42     if (! aAbs)
  119        435       44       if (! bAbs)
  119                  44         return fromRep ( toRep ( a ) & toRep ( b ) )
                       44       }+
                       45       else
  435                  45         return b
                       45       }-
                       46     }+
  436          0 -     49     if (! bAbs)
  436                  49       return a
                       49     }-
   2856       3743       53   if (bAbs > aAbs)
                         57   }+
   1323       5276       66   if (aExponent == 0)
                         66   }+
   1407       5192       67   if (bExponent == 0)
                         67   }+
   4963       1636       84   if (align)
   4513        450       85     if (align < ( sizeof ( rep_t ) * 8 ))
                         88     }+
                         88     else
                         90     }+
                         91   }+
   3656       2943       92   if (subtraction)
    527       3129       95     if (aSignificand == 0)
    527                  95       return fromRep ( 0 )
                         95     }+
   1844       1285       99     if (aSignificand < ( 1U << 23 ) << 3)
                        103     }+
                        104   }+
                        105   else
   1227       1716      110     if (aSignificand & ( 1U << 23 ) << 4)
                        114     }+
                        115   }+
      0       6072 -    118   if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) - 1 ))
      0            -    118     return fromRep ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) | r
                        118   }+
   1281       4791      120   if (aExponent <= 0)
                        127   }+
   1037       5035      141   if (roundGuardSticky > 0x4)
                        141   }+
    366       5706      142   if (roundGuardSticky == 0x4)
                        142   }+
   6072                 143   return fromRep ( result )
                        144 }

***TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
        99 % ( 71/ 72) statement
```



```
                         145 #line 17 "addsf3.c"
     7668                 18 FUNCTION __addsf3()
     7668                 19   return __addXf3__ ( a , b )
                          20 }

***TER 100 % (  2/  2) of FUNCTION __addsf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------

                         268 #line 17 "subsf3.c"
     5112                 20 FUNCTION __subsf3()
     5112                 21   return __addsf3 ( a , fromRep ( toRep ( b ) ^ ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) ) )
                          22 }

***TER 100 % (  2/  2) of FUNCTION __subsf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------

     5112                 26 FUNCTION __aeabi_fsub()
     5112                 27   return __subsf3 ( a , b )
                          28 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_fsub()
       100 % (  1/  1) statement
```

7.14.1 “Else” branch of “If” condition if (! bAbs)



```
1   if (aAbs - REP_C(1) >= infRep - REP_C(1) ||
2           bAbs - REP_C(1) >= infRep - REP_C(1)) {
3           // NaN + anything = qNaN
4           if (aAbs > infRep) {
5               return fromRep(toRep(a) | quietBit);
6           }
7           // aAbs <= infRep
8           // anything + NaN = qNaN
9           if (bAbs > infRep) {
10              return fromRep(toRep(b) | quietBit);
11          }
12          // bAbs <= infRep
13
14          if (aAbs == infRep) {
15              // +/-infinity + -/+infinity = qNaN
16              if ((toRep(a) ^ toRep(b)) == signBit) return fromRep(qnanRep);
17              // +/-infinity + anything remaining = +/- infinity
18              else return a;
19          }
20          // aAbs < infRep
21
22          // anything remaining + +/-infinity = +/-infinity
23          if (bAbs == infRep) {
24              return b;
25          }
26          // bAbs < infRep
27
28          // zero + anything = anything
29          if (!aAbs) {
30              // but we need to get the sign right for zero + zero
31              if (!bAbs){
32                  return fromRep(toRep(a) & toRep(b));
33              }
34              else return b;
35          }
36          // 0 < aAbs < infRep
37
38          // anything + zero = anything
39
40          if (!bAbs){
41              return a;
42          }
43          else
44          {
45              // 0 < bAbs < infRep
46          }
47      }
```

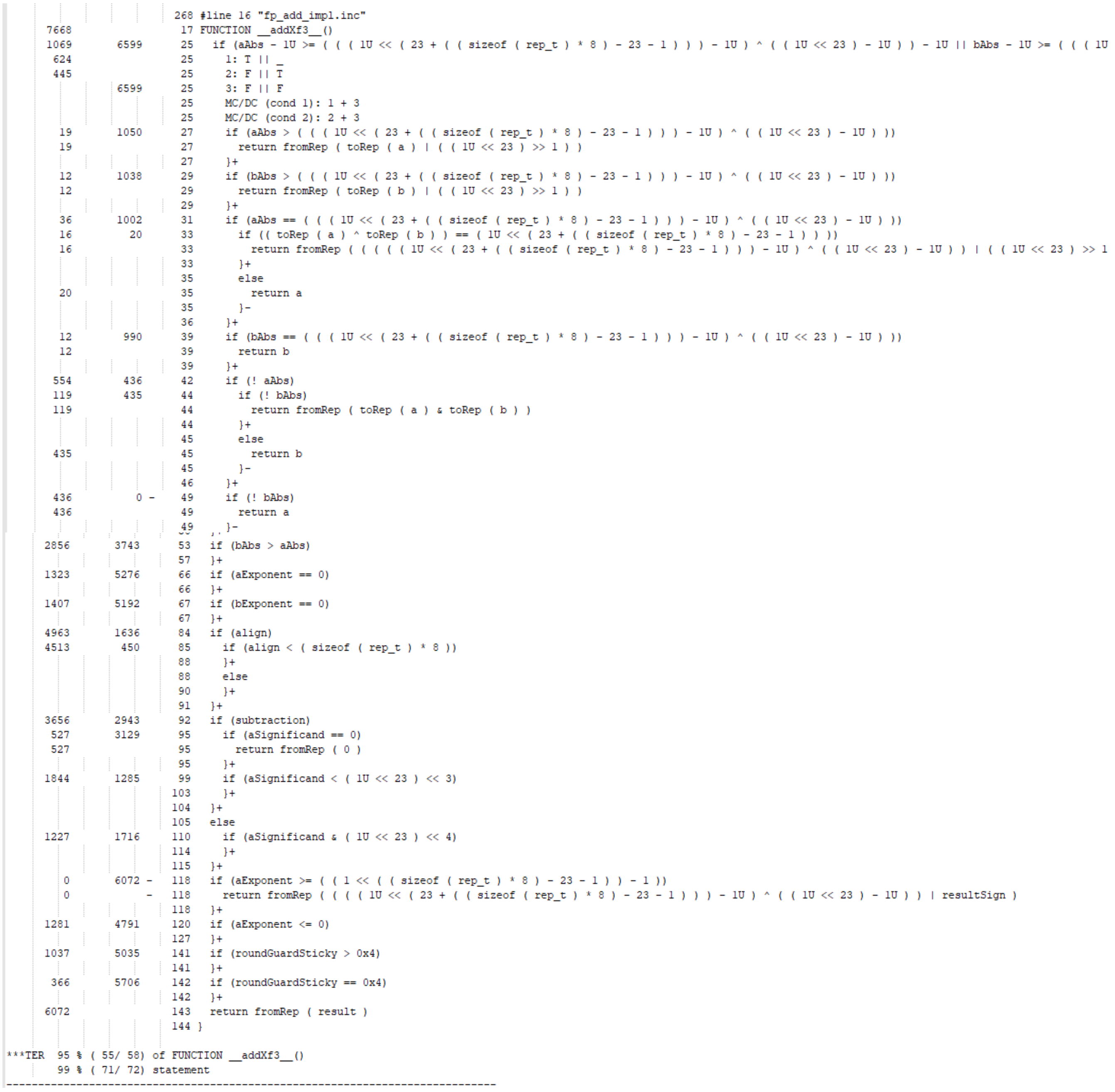
The “Else” branch of “If” condition if(!bAbs) in line 49 cannot be covered, because

c) On the one hand, the “If” condition if(aAbs – REP_C(1) >= infRep – REP_C(1) || bAbs – REP_C(1) >= infRep – REP_C(1)) needs to be fulfilled
d) On the the other hand in order to reach the “Else” branch in line 43
    - The “if” condition if (aAbs > infRep) in line 4 needs to evaluate to false (needs to be skipped)
    - The “if” condition if (bAbs > infRep) in line 9 needs to evaluate to false (needs to be skipped)

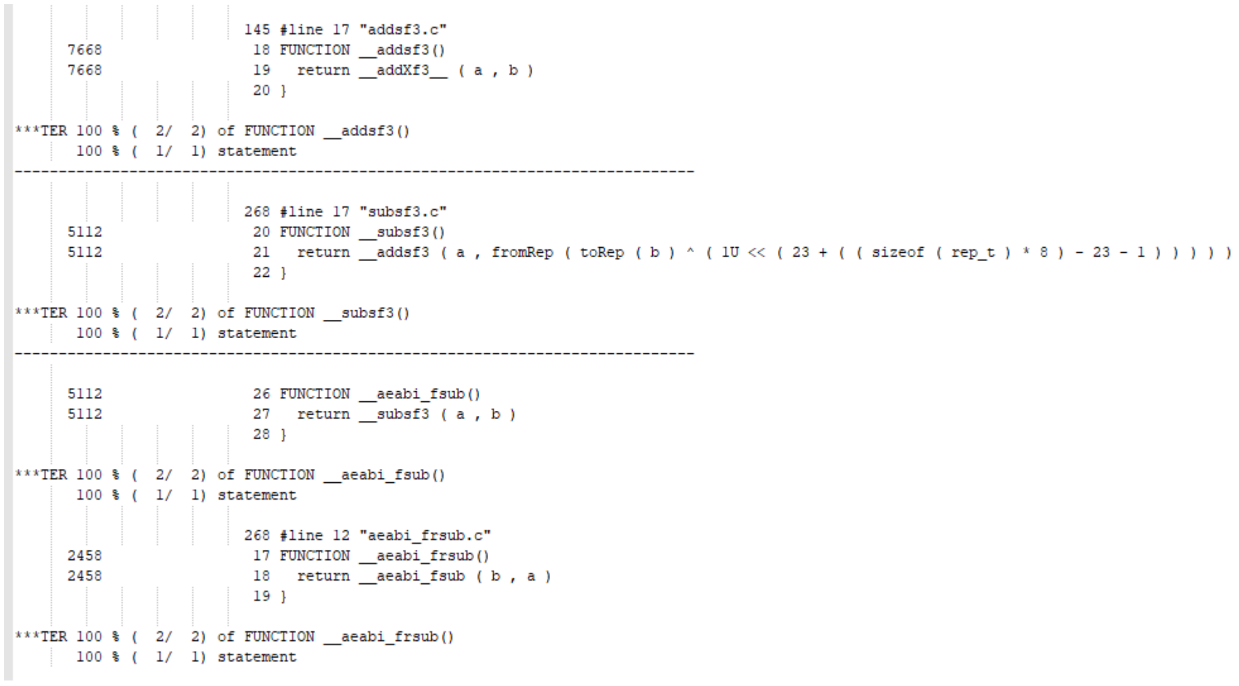
- The “if” condition if (aAbs == infRep) in line 14 needs to evaluate to false (needs to be skipped)
- The “if” condition if (bAbs == infRep) in line 23 needs to evaluate to false (needs to be skipped)
- The “if” condition if (!aAbs) in line 29 needs to evaluate to false (needs to be skipped)
- The “if” condition if (!bAbs) in line 40 needs to evaluate to false (needs to be skipped)

➔ So in the end the following should apply when reaching the “Else” branch in line 43:
0 < aAbs < infRep AND 0 < bAbs < infRep. But this is not possible because also the condition in a) needs to be fulfilled.

7.15 Analysis of code coverage of function __aeabi_frsub / __addXf3__: OK



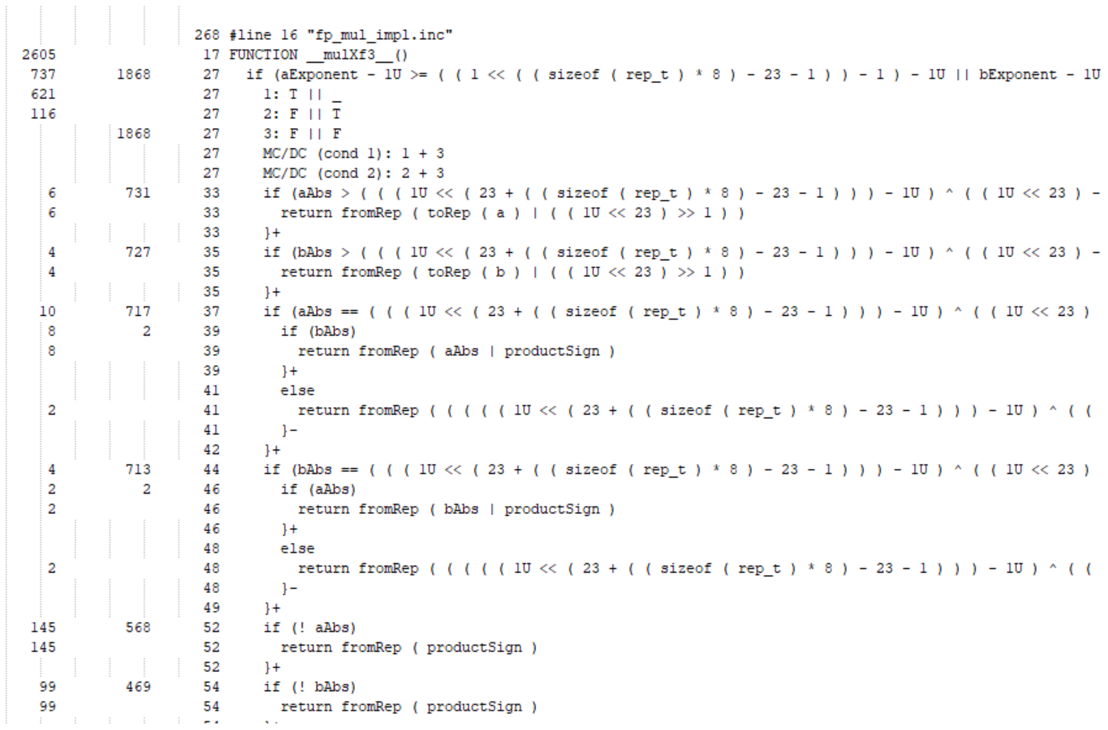
```
                       268 #line 16 "fp_add_impl.inc"
   7668                 17 FUNCTION __addXf3__()
   1069       6599      25   if (aAbs - 1U >= ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) - 1U || bAbs - 1U >= ( ( ( 1U
    624                 25     1: T || _
    445                 25     2: F || T
              6599      25     3: F || F
                        25     MC/DC (cond 1): 1 + 3
                        25     MC/DC (cond 2): 2 + 3
     19       1050      27     if (aAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
     19                 27       return fromRep ( toRep ( a ) | ( ( 1U << 23 ) >> 1 ) )
                        27     }+
     12       1038      29     if (bAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
     12                 29       return fromRep ( toRep ( b ) | ( ( 1U << 23 ) >> 1 ) )
                        29     }+
     36       1002      31     if (aAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
     16         20      33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ))
     16                 33         return fromRep ( ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) | ( ( 1U << 23 ) >> 1
                        33       }+
                        35       else
     20                 35         return a
                        35       }-
                        36     }+
     12        990      39     if (bAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ))
     12                 39       return b
                        39     }+
    554        436      42     if (! aAbs)
    119        435      44       if (! bAbs)
    119                 44         return fromRep ( toRep ( a ) & toRep ( b ) )
                        44       }+
                        45       else
    435                 45         return b
                        45       }-
                        46     }+
    436          0 -    49     if (! bAbs)
    436                 49       return a
                        49     }-
   2856       3743      53   if (bAbs > aAbs)
                        57   }+
   1323       5276      66   if (aExponent == 0)
                        66   }+
   1407       5192      67   if (bExponent == 0)
                        67   }+
   4963       1636      84   if (align)
   4513        450      85     if (align < ( sizeof ( rep_t ) * 8 ))
                        88     }+
                        88     else
                        90     }+
                        91   }+
   3656       2943      92   if (subtraction)
    527       3129      95     if (aSignificand == 0)
    527                 95       return fromRep ( 0 )
                        95     }+
   1844       1285      99     if (aSignificand < ( 1U << 23 ) << 3)
                       103     }+
                       104   }+
                       105   else
   1227       1716     110     if (aSignificand & ( 1U << 23 ) << 4)
                       114     }+
                       115   }+
      0       6072 -   118   if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) - 1 ))
      0            -   118     return fromRep ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) | resultSign )
                       118   }+
   1281       4791     120   if (aExponent <= 0)
                       127   }+
   1037       5035     141   if (roundGuardSticky > 0x4)
                       141   }+
    366       5706     142   if (roundGuardSticky == 0x4)
                       142   }+
   6072                143   return fromRep ( result )
                       144 }

***TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
        99 % ( 71/ 72) statement
-----------------------------------------------------------------------------
```



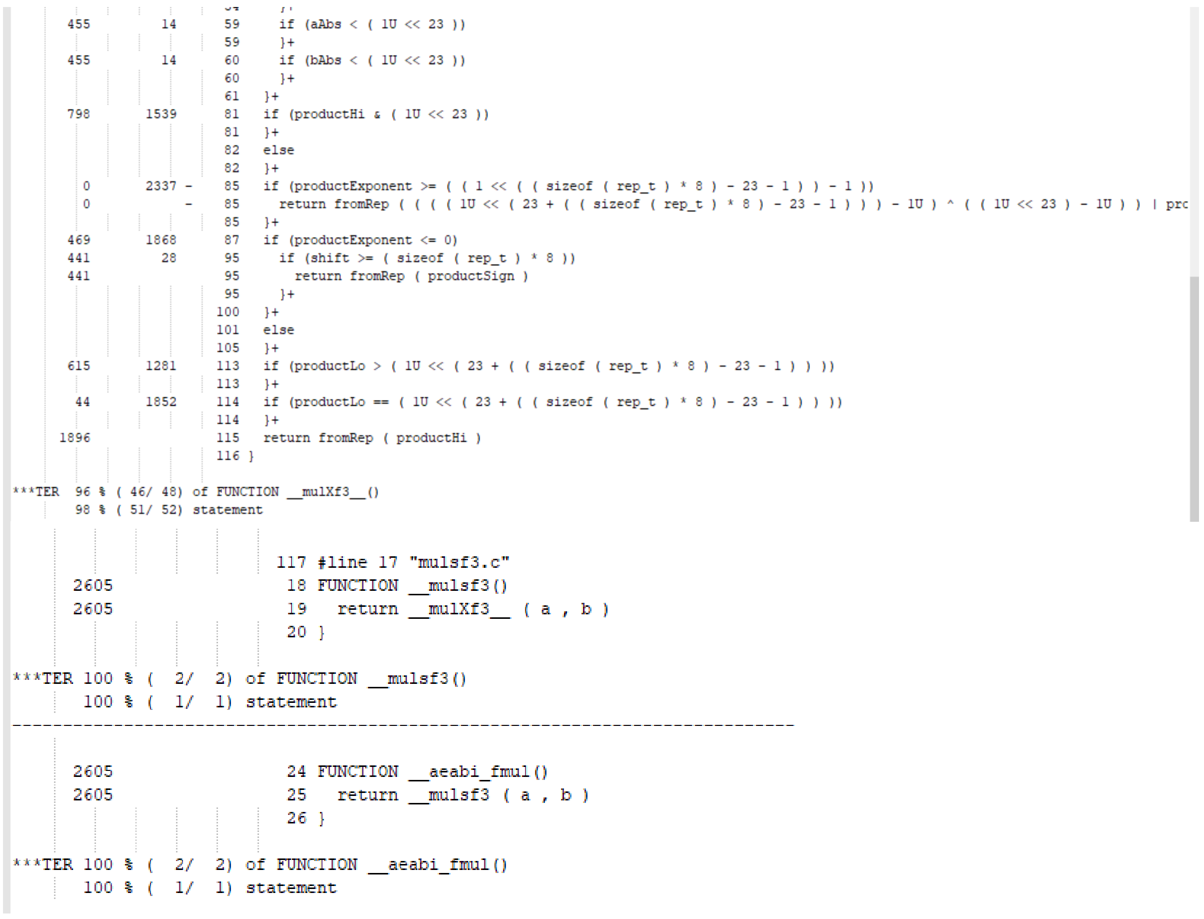
```
                          145 #line 17 "addsf3.c"
     7668                  18 FUNCTION __addsf3()
     7668                  19   return __addXf3__ ( a , b )
                           20 }

***TER 100 % (  2/  2) of FUNCTION __addsf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------
                          268 #line 17 "subsf3.c"
     5112                  20 FUNCTION __subsf3()
     5112                  21   return __addsf3 ( a , fromRep ( toRep ( b ) ^ ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) ) )
                           22 }

***TER 100 % (  2/  2) of FUNCTION __subsf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------

     5112                  26 FUNCTION __aeabi_fsub()
     5112                  27   return __subsf3 ( a , b )
                           28 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_fsub()
       100 % (  1/  1) statement

                          268 #line 12 "aeabi_frsub.c"
     2458                  17 FUNCTION __aeabi_frsub()
     2458                  18   return __aeabi_fsub ( b , a )
                           19 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_frsub()
       100 % (  1/  1) statement
```

Since __aeabi_frsub calls indirectly __addXf3__, the same argumentation regarding code coverage as in chapter 7.14 applies.

7.16 Analysis of code coverage of function __aeabi_fmul: OK

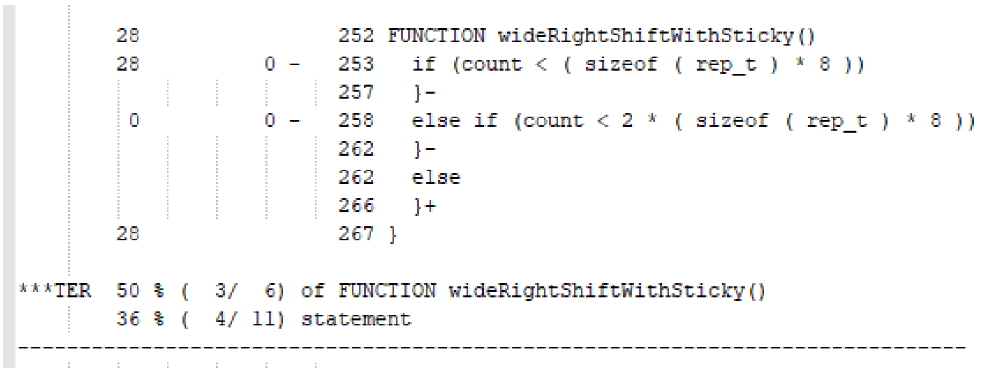


```
                        268 #line 16 "fp_mul_impl.inc"
 2605                    17 FUNCTION __mulXf3__()
  737        1868        27   if (aExponent - 1U >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) - 1 ) - 1U || bExponent - 1U
  621                    27     1: T || _
  116                    27     2: F || T
             1868        27     3: F || F
                         27     MC/DC (cond 1): 1 + 3
                         27     MC/DC (cond 2): 2 + 3
    6         731        33     if (aAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
    6                    33       return fromRep ( toRep ( a ) | ( ( 1U << 23 ) >> 1 ) )
                         33     }+
    4         727        35     if (bAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
    4                    35       return fromRep ( toRep ( b ) | ( ( 1U << 23 ) >> 1 ) )
                         35     }+
   10         717        37     if (aAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 )
    8           2        39       if (bAbs)
    8                    39         return fromRep ( aAbs | productSign )
                         39       }+
                         41       else
    2                    41         return fromRep ( ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( (
                         41       }-
                         42     }+
    4         713        44     if (bAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 )
    2           2        46       if (aAbs)
    2                    46         return fromRep ( bAbs | productSign )
                         46       }+
                         48       else
    2                    48         return fromRep ( ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( (
                         48       }-
                         49     }+
  145         568        52     if (! aAbs)
  145                    52       return fromRep ( productSign )
                         52     }+
   99         469        54     if (! bAbs)
   99                    54       return fromRep ( productSign )
```



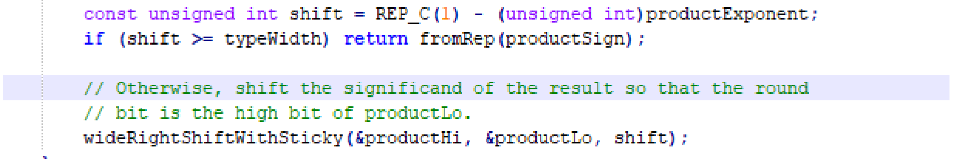
```
   455        14      59     if (aAbs < ( 1U << 23 ))
                      59     }+
   455        14      60     if (bAbs < ( 1U << 23 ))
                      60     }+
                      61   }+
   798      1539      81   if (productHi & ( 1U << 23 ))
                      81   }+
                      82   else
                      82   }+
     0      2337 -    85   if (productExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) - 1 ))
     0           -    85     return fromRep ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) - 1U ) ) | prc
                      85   }+
   469      1868      87   if (productExponent <= 0)
   441        28      95     if (shift >= ( sizeof ( rep_t ) * 8 ))
   441                95       return fromRep ( productSign )
                      95     }+
                     100   }+
                     101   else
                     105   }+
   615      1281     113   if (productLo > ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ))
                     113   }+
    44      1852     114   if (productLo == ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ))
                     114   }+
  1896               115   return fromRep ( productHi )
                     116 }

***TER  96 % ( 46/ 48) of FUNCTION __mulXf3__()
        98 % ( 51/ 52) statement

                           117 #line 17 "mulsf3.c"
   2605                     18 FUNCTION __mulsf3()
   2605                     19   return __mulXf3__ ( a , b )
                            20 }

***TER 100 % (  2/   2) of FUNCTION __mulsf3()
       100 % (  1/   1) statement
-------------------------------------------------------------------------------

   2605                     24 FUNCTION __aeabi_fmul()
   2605                     25   return __mulsf3 ( a , b )
                            26 }

***TER 100 % (  2/   2) of FUNCTION __aeabi_fmul()
       100 % (  1/   1) statement
```

7.16.1 Sub-function wideRightShiftWithSticky



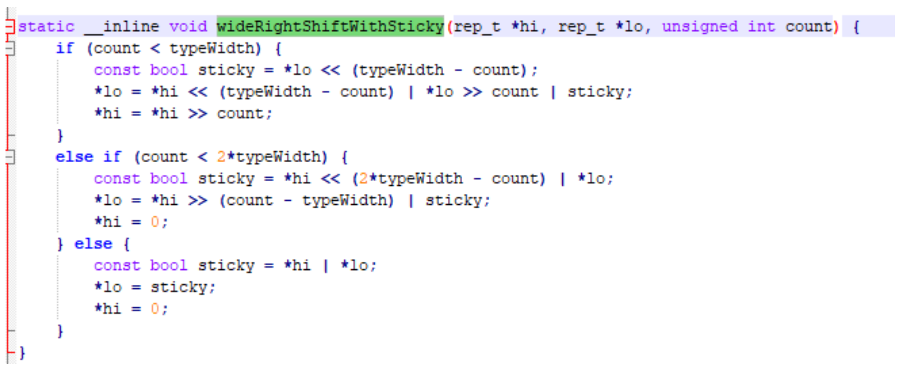
```
     28                 252 FUNCTION wideRightShiftWithSticky()
     28             0 - 253   if (count < ( sizeof ( rep_t ) * 8 ))
                        257   }-
      0             0 - 258   else if (count < 2 * ( sizeof ( rep_t ) * 8 ))
                        262   }-
                        262   else
                        266   }+
     28                 267 }

***TER  50 % (  3/   6) of FUNCTION wideRightShiftWithSticky()
        36 % (  4/ 11) statement
-------------------------------------------------------------------------------
```

The sub-function wideRightShiftWithSticky is only called within the context of __mulXf3__ as shown in the following screenshot:



```c
    const unsigned int shift = REP_C(1) - (unsigned int)productExponent;
    if (shift >= typeWidth) return fromRep(productSign);

    // Otherwise, shift the significand of the result so that the round
    // bit is the high bit of productLo.
    wideRightShiftWithSticky(&productHi, &productLo, shift);
```

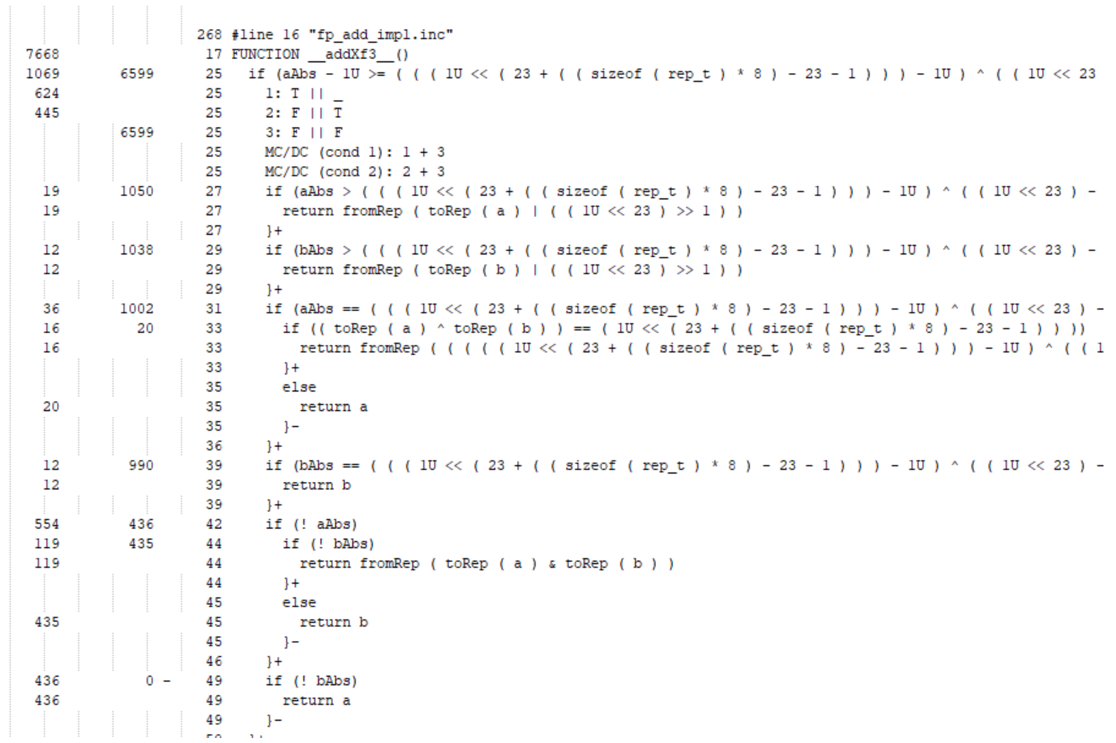
Due to the "If" statement if(shift >= typeWidth) return fromRep (productSign) it is ensured that wideRightShiftWithSticky is only called with shift < typeWidth.



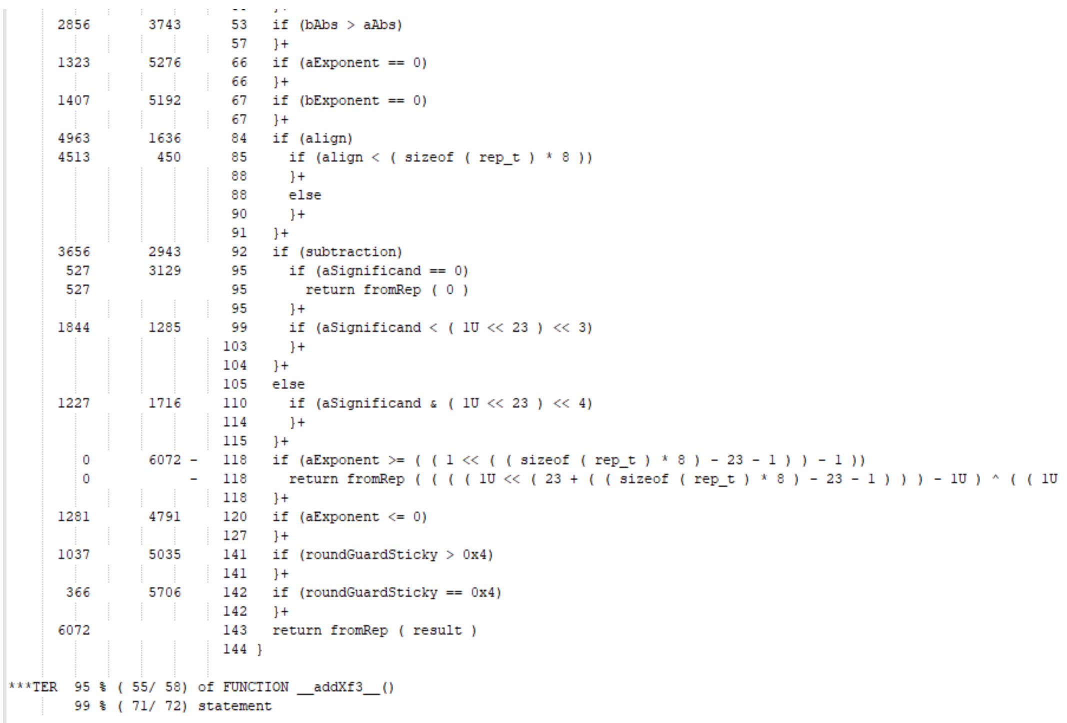
```c
static __inline void wideRightShiftWithSticky(rep_t *hi, rep_t *lo, unsigned int count) {
    if (count < typeWidth) {
        const bool sticky = *lo << (typeWidth - count);
        *lo = *hi << (typeWidth - count) | *lo >> count | sticky;
        *hi = *hi >> count;
    }
    else if (count < 2*typeWidth) {
        const bool sticky = *hi << (2*typeWidth - count) | *lo;
        *lo = *hi >> (count - typeWidth) | sticky;
        *hi = 0;
    } else {
        const bool sticky = *hi | *lo;
        *lo = sticky;
        *hi = 0;
    }
}
```

Thus, within wideRightShiftWithSticky the “If” condition if(count < typeWidth) is always fulfilled and the else branch cannot be covered.

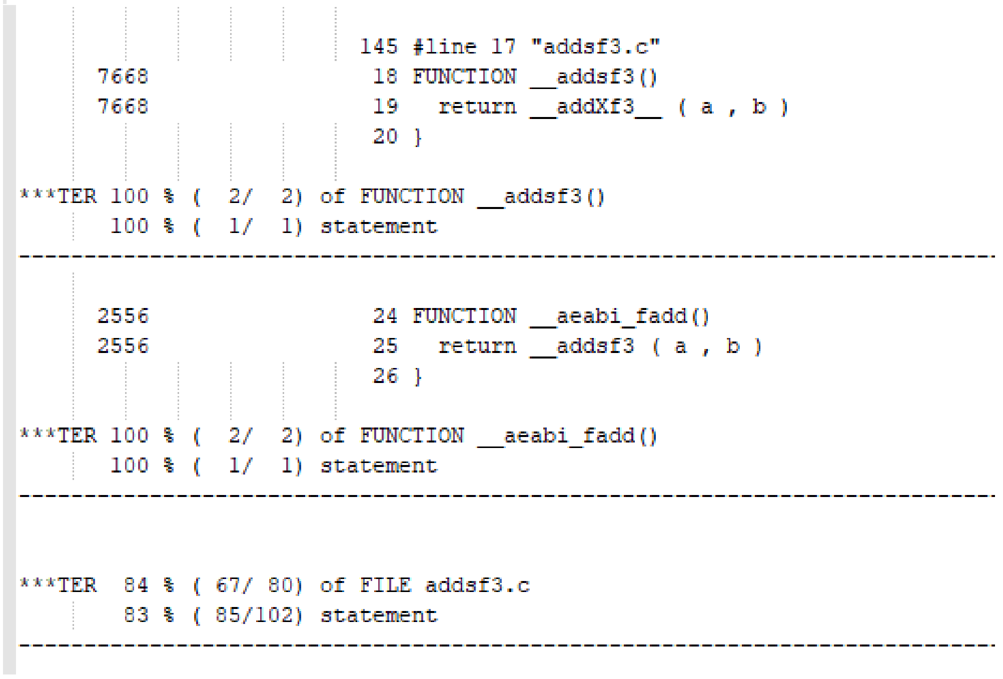
7.17 Analysis of code coverage of function __aeabi_fadd / __addXf3__: OK



```
                          268 #line 16 "fp_add_impl.inc"
 7668                      17 FUNCTION __addXf3__()
 1069          6599        25   if (aAbs - 1U >= ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23
  624                      25     1: T || _
  445                      25     2: F || T
               6599        25     3: F || F
                           25     MC/DC (cond 1): 1 + 3
                           25     MC/DC (cond 2): 2 + 3
   19          1050        27     if (aAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
   19                      27       return fromRep ( toRep ( a ) | ( ( 1U << 23 ) >> 1 ) )
                           27     }+
   12          1038        29     if (bAbs > ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
   12                      29       return fromRep ( toRep ( b ) | ( ( 1U << 23 ) >> 1 ) )
                           29     }+
   36          1002        31     if (aAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
   16            20        33       if (( toRep ( a ) ^ toRep ( b ) ) == ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ))
   16                      33         return fromRep ( ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1
                           33       }+
                           35       else
   20                      35         return a
                           35       }-
                           36     }+
   12           990        39     if (bAbs == ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U << 23 ) -
   12                      39       return b
                           39     }+
  554           436        42     if (! aAbs)
  119           435        44       if (! bAbs)
  119                      44         return fromRep ( toRep ( a ) & toRep ( b ) )
                           44       }+
                           45       else
  435                      45         return b
                           45       }-
                           46     }+
  436             0 -      49     if (! bAbs)
  436                      49       return a
                           49     }-
```



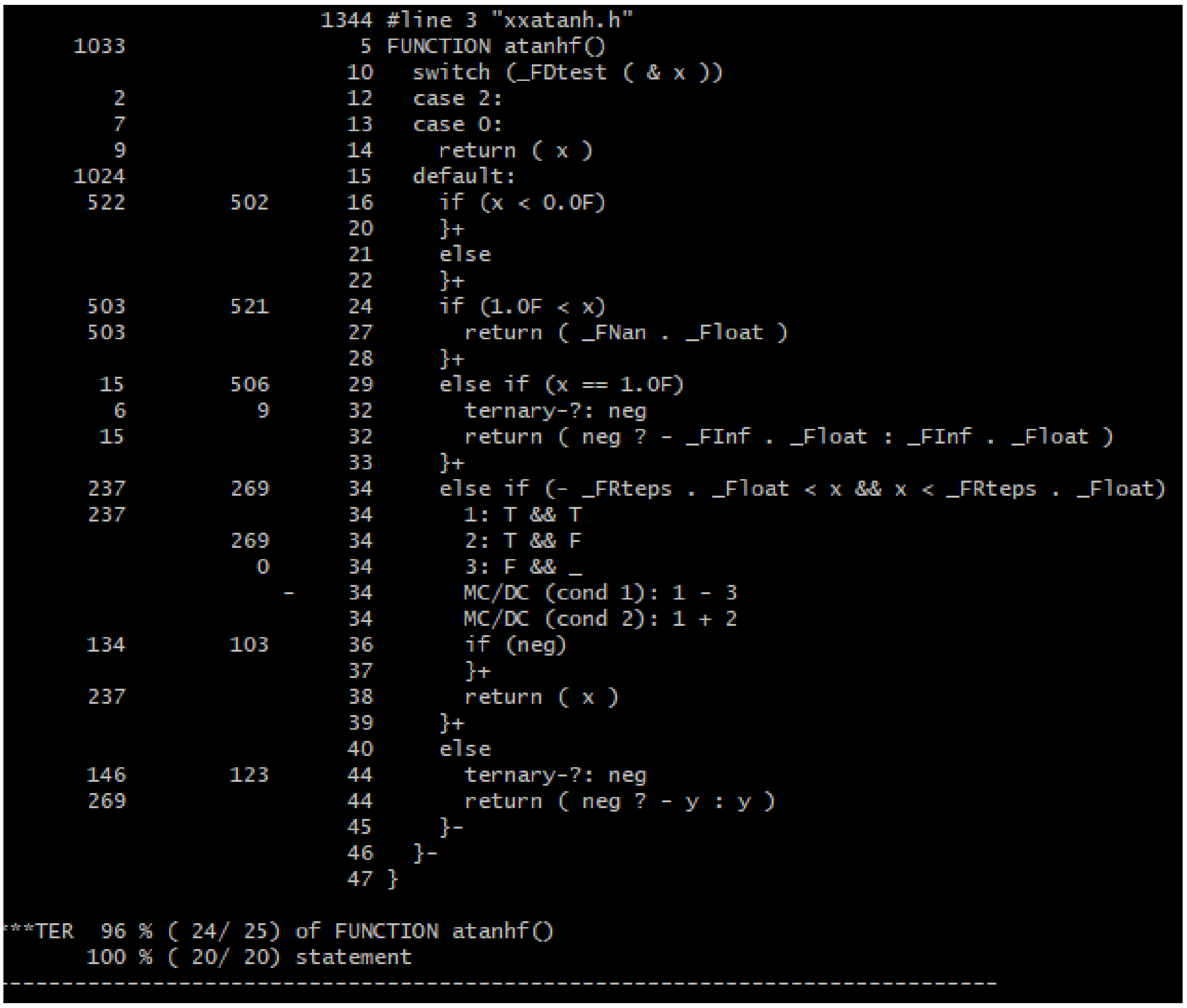
```
      2856       3743      53   if (bAbs > aAbs)
                           57   }+
      1323       5276      66   if (aExponent == 0)
                           66   }+
      1407       5192      67   if (bExponent == 0)
                           67   }+
      4963       1636      84   if (align)
      4513        450      85     if (align < ( sizeof ( rep_t ) * 8 ))
                           88     }+
                           88     else
                           90     }+
                           91   }+
      3656       2943      92   if (subtraction)
       527       3129      95     if (aSignificand == 0)
       527                 95       return fromRep ( 0 )
                           95     }+
      1844       1285      99     if (aSignificand < ( 1U << 23 ) << 3)
                          103     }+
                          104   }+
                          105   else
      1227       1716     110     if (aSignificand & ( 1U << 23 ) << 4)
                          114     }+
                          115   }+
         0       6072 -   118   if (aExponent >= ( ( 1 << ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) - 1 ))
         0            -   118     return fromRep ( ( ( ( 1U << ( 23 + ( ( sizeof ( rep_t ) * 8 ) - 23 - 1 ) ) ) - 1U ) ^ ( ( 1U
                          118   }+
      1281       4791     120   if (aExponent <= 0)
                          127   }+
      1037       5035     141   if (roundGuardSticky > 0x4)
                          141   }+
       366       5706     142   if (roundGuardSticky == 0x4)
                          142   }+
      6072                143   return fromRep ( result )
                          144 }

***TER  95 % ( 55/ 58) of FUNCTION __addXf3__()
        99 % ( 71/ 72) statement
```



```
                                 145 #line 17 "addsf3.c"
      7668                        18 FUNCTION __addsf3()
      7668                        19   return __addXf3__ ( a , b )
                                  20 }

***TER 100 % (  2/  2) of FUNCTION __addsf3()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------

      2556                        24 FUNCTION __aeabi_fadd()
      2556                        25   return __addsf3 ( a , b )
                                  26 }

***TER 100 % (  2/  2) of FUNCTION __aeabi_fadd()
       100 % (  1/  1) statement
-----------------------------------------------------------------------------


***TER  84 % ( 67/ 80) of FILE addsf3.c
        83 % ( 85/102) statement
-----------------------------------------------------------------------------
```

Since __aeabi_fadd calls indirectly __addXf3__, the same argumentation regarding code coverage as in chapter 7.14 applies.

7.18 Analysis of code coverage of function atanhf: OK

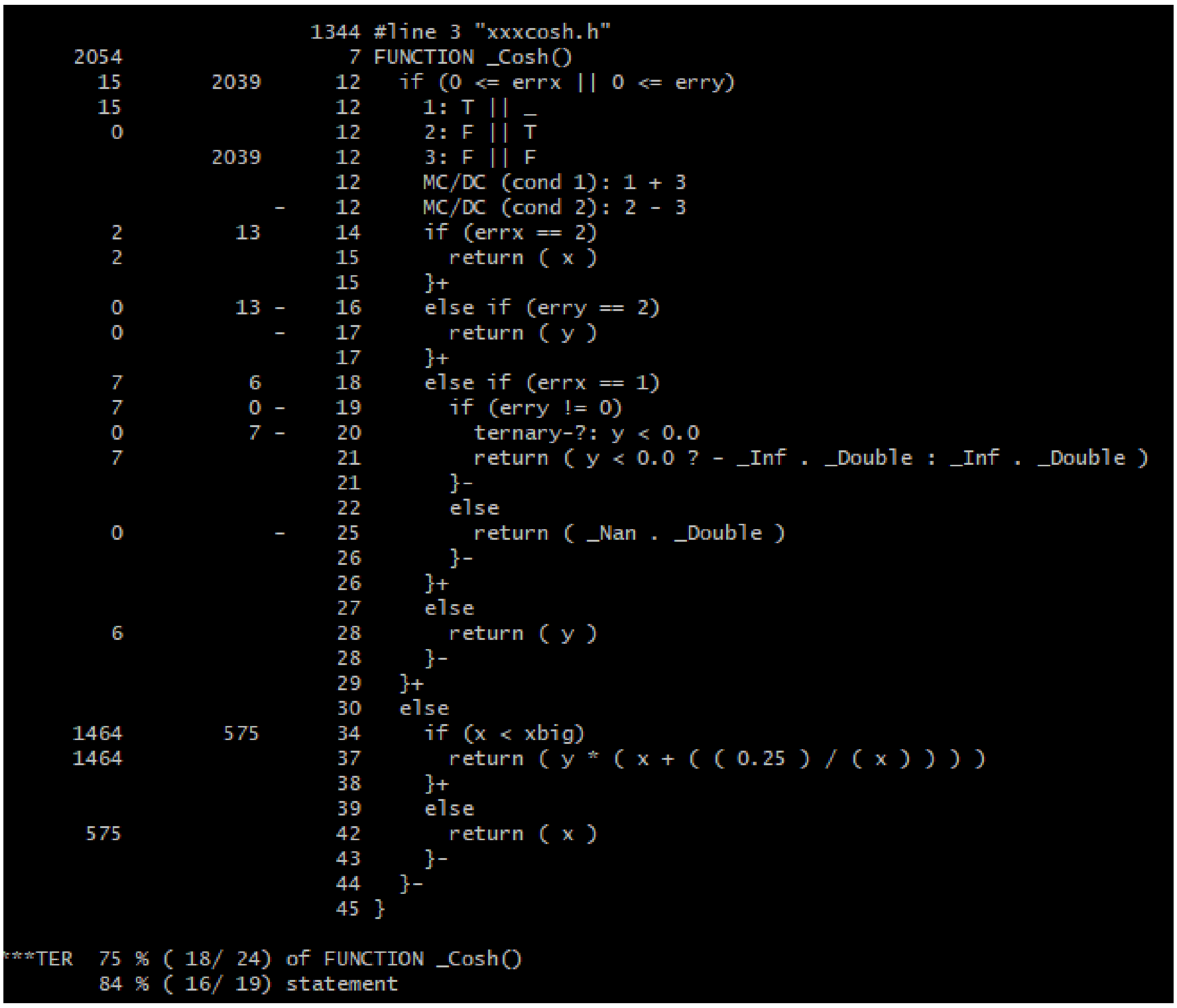


```
                       1344 #line 3 "xxatanh.h"
    1033                  5 FUNCTION atanhf()
                         10   switch (_FDtest ( & x ))
       2                 12   case 2:
       7                 13   case 0:
       9                 14     return ( x )
    1024                 15   default:
     522       502       16     if (x < 0.0F)
                         20     }+
                         21     else
                         22     }+
     503       521       24     if (1.0F < x)
     503                 27       return ( _FNan . _Float )
                         28     }+
      15       506       29     else if (x == 1.0F)
       6         9       32       ternary-?: neg
      15                 32       return ( neg ? - _FInf . _Float : _FInf . _Float )
                         33     }+
     237       269       34     else if (- _FRteps . _Float < x && x < _FRteps . _Float)
     237                 34       1: T && T
               269       34       2: T && F
                 0       34       3: F && _
                   -     34       MC/DC (cond 1): 1 - 3
                         34       MC/DC (cond 2): 1 + 2
     134       103       36       if (neg)
                         37       }+
     237                 38       return ( x )
                         39     }+
                         40     else
     146       123       44       ternary-?: neg
     269                 44       return ( neg ? - y : y )
                         45     }-
                         46   }-
                         47 }

***TER  96 % ( 24/ 25) of FUNCTION atanhf()
       100 % ( 20/ 20) statement
-----------------------------------------------------------------------------
```

The only uncovered case in this function is better seen in the source co-de

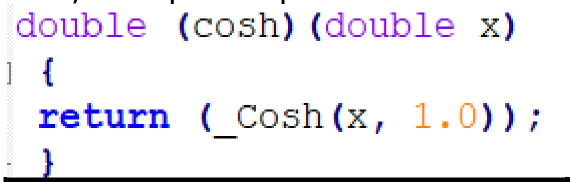
7.19 Analysis of code coverage of function cosh: OK

Used the function _Cosh that has insufficient code coverage

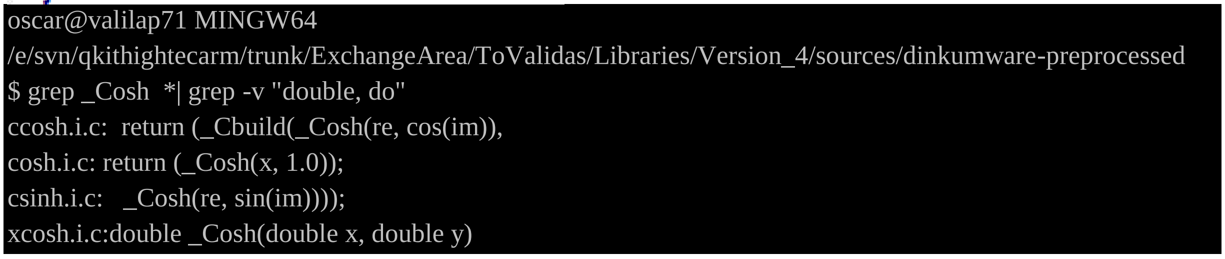


```
                         1344 #line 3 "xxxcosh.h"
     2054                   7 FUNCTION _Cosh()
       15        2039      12   if (0 <= errx || 0 <= erry)
       15                  12     1: T || _
        0                  12     2: F || T
                 2039      12     3: F || F
                           12     MC/DC (cond 1): 1 + 3
                      -    12     MC/DC (cond 2): 2 - 3
        2          13      14     if (errx == 2)
        2                  15       return ( x )
                           15     }+
        0          13 -    16     else if (erry == 2)
        0             -    17       return ( y )
                           17     }+
        7           6      18     else if (errx == 1)
        7           0 -    19       if (erry != 0)
        0           7 -    20         ternary-?: y < 0.0
        7                  21         return ( y < 0.0 ? - _Inf . _Double : _Inf . _Double )
                           21       }-
                           22       else
        0             -    25         return ( _Nan . _Double )
                           26       }-
                           26     }+
                           27     else
        6                  28       return ( y )
                           28     }-
                           29   }+
                           30   else
     1464         575      34     if (x < xbig)
     1464                  37       return ( y * ( x + ( ( 0.25 ) / ( x ) ) ) )
                           38     }+
                           39     else
      575                  42       return ( x )
                           43     }-
                           44   }-
                           45 }

***TER  75 % ( 18/ 24) of FUNCTION _Cosh()
        84 % ( 16/ 19) statement
```

Is called by cosh with the second argument fixed to one (and nowhere else, except complex functions that are not qualified)



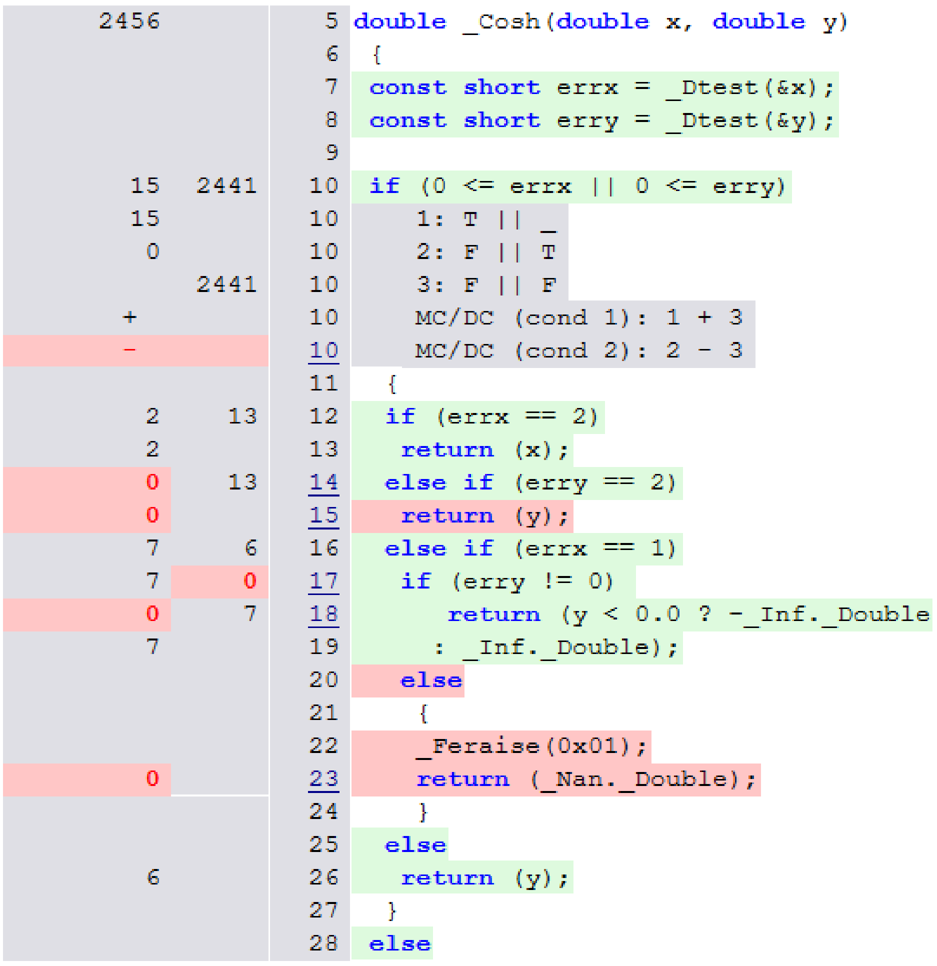
```
double (cosh)(double x)
 {
 return (_Cosh(x, 1.0));
 }
```



```
oscar@valilap71 MINGW64
/e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/Libraries/Version_4/sources/dinkumware-preprocessed
$ grep _Cosh  *| grep -v "double, do"
ccosh.i.c:  return (_Cbuild(_Cosh(re, cos(im)),
cosh.i.c: return (_Cosh(x, 1.0));
csinh.i.c:   _Cosh(re, sin(im))));
xcosh.i.c:double _Cosh(double x, double y)
```

y=1.0 leads to erry=-1. Therefore the uncovered parts cannot be reached

- 0<=yerr: is always false
- elseif (yerr==2) is always false
- if (erry!=0) is always true

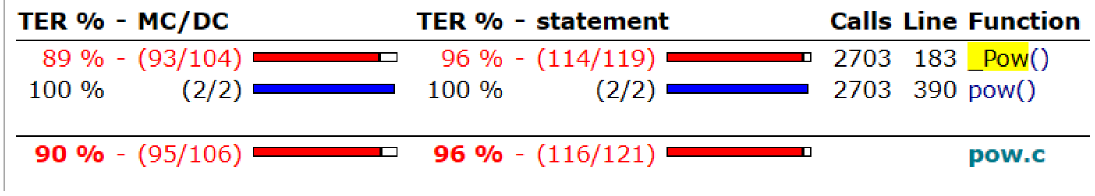


```
2456         5 double _Cosh(double x, double y)
             6  {
             7  const short errx = _Dtest(&x);
             8  const short erry = _Dtest(&y);
             9
  15  2441  10  if (0 <= errx || 0 <= erry)
  15        10      1: T || _
   0        10      2: F || T
      2441  10      3: F || F
   +        10      MC/DC (cond 1): 1 + 3
   -        10      MC/DC (cond 2): 2 - 3
            11   {
   2    13  12   if (errx == 2)
   2        13    return (x);
   0    13  14   else if (erry == 2)
   0        15    return (y);
   7     6  16   else if (errx == 1)
   7     0  17    if (erry != 0)
   0     7  18       return (y < 0.0 ? -_Inf._Double
   7        19      : _Inf._Double);
            20    else
            21     {
            22     _Feraise(0x01);
   0        23     return (_Nan._Double);
            24     }
            25   else
   6        26    return (y);
            27   }
            28  else
```

Therefore coverage in cosh is maximal. Same argumentation holds for coshf.

7.20 Analysis of code coverage of function pow: OK

Uses the function _Pow that has insufficient code coverage, see https://opensvn.teststatt.de/repos/qkithightecarm/trunk/Work/QKitExtension/Analysis/Coverage/pow/CTCHTML/index.html for the full report
The report shows a coverage of 93 / 104

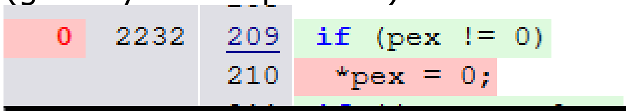


TER % - MC/DC	TER % - statement	Calls	Line	Function
89 % - (93/104)	96 % - (114/119)	2703	183	_Pow()
100 % (2/2)	100 % (2/2)	2703	390	pow()
90 % - (95/106)	**96 %** - (116/121)			**pow.c**

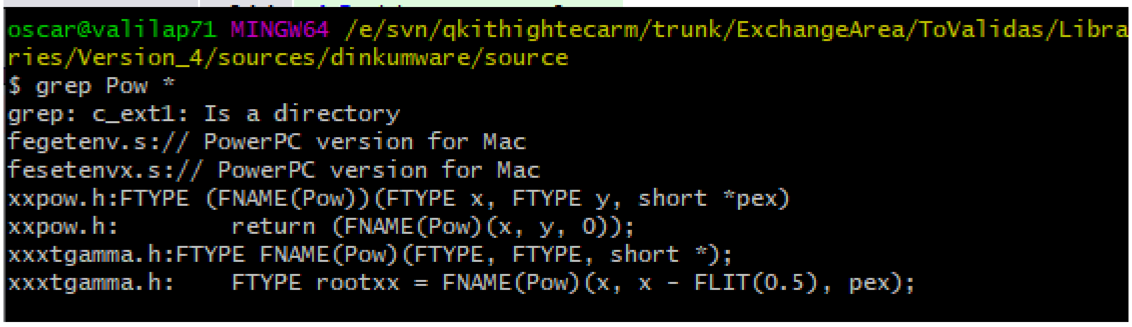
Hence 11 places have been analyzed. This was done successfully in the following subsections.
Note that pow/powf use the same source code (xxpow.h) and hence the results carry over also to powf.

7.20.1 _Pow: `if (pex != 0)`

Trivial (since pex==0) in the call of Pow, this code is dead for Pow (gammy is not qualified)

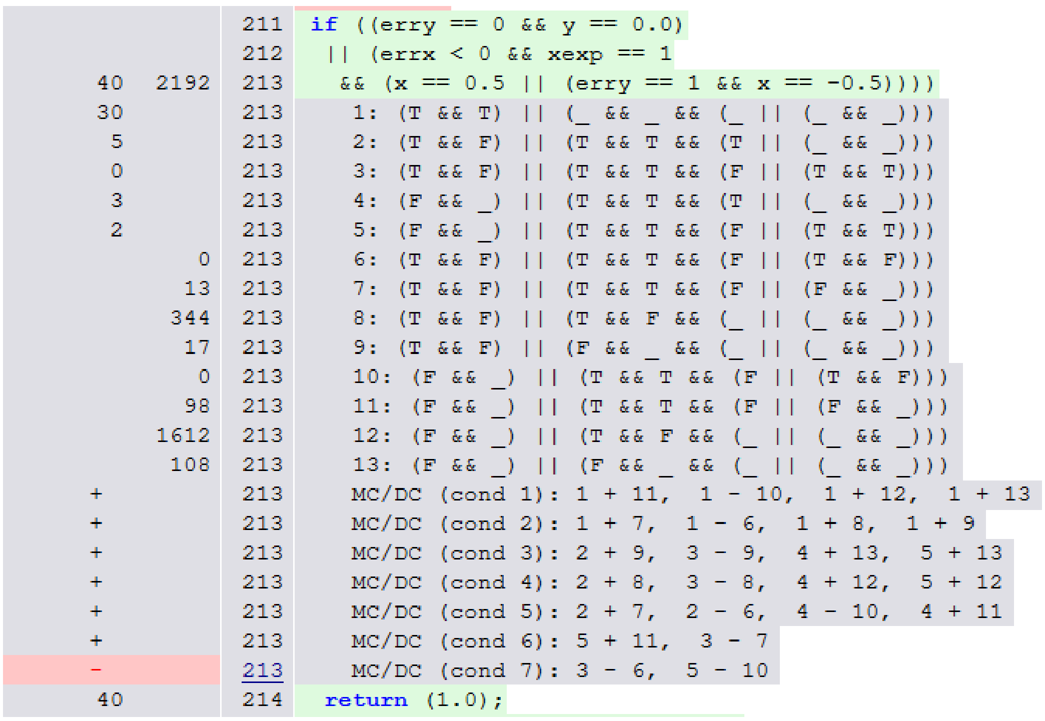


```
    0   2232    209   if (pex != 0)
                210     *pex = 0;
```



```
oscar@valilap71 MINGW64 /e/svn/qkithightecarm/trunk/ExchangeArea/ToValidas/Libra
ries/Version_4/sources/dinkumware/source
$ grep Pow *
grep: c_ext1: Is a directory
fegetenv.s:// PowerPC version for Mac
fesetenvx.s:// PowerPC version for Mac
xxpow.h:FTYPE (FNAME(Pow))(FTYPE x, FTYPE y, short *pex)
xxpow.h:        return (FNAME(Pow)(x, y, 0));
xxxtgamma.h:FTYPE FNAME(Pow)(FTYPE, FTYPE, short *);
xxxtgamma.h:    FTYPE rootxx = FNAME(Pow)(x, x - FLIT(0.5), pex);
```

7.20.2 _Pow: `erry == 0 && y == 0.0`

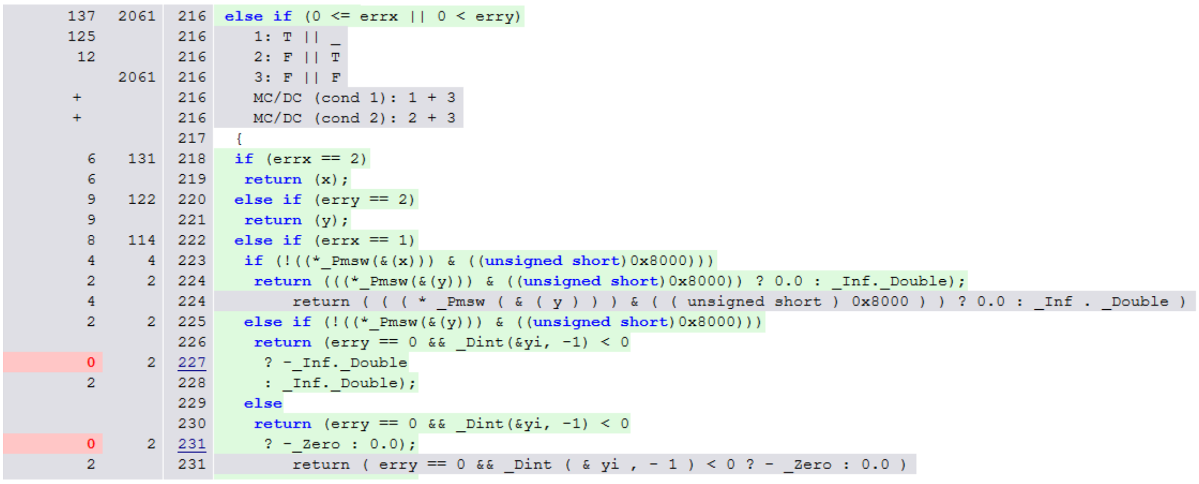


```
                 211   if ((erry == 0 && y == 0.0)
                 212     || (errx < 0 && xexp == 1
      40  2192   213      && (x == 0.5 || (erry == 1 && x == -0.5))))
      30         213       1: (T && T) || (_ && _ && (_ || (_ && _)))
       5         213       2: (T && F) || (T && T && (T || (_ && _)))
       0         213       3: (T && F) || (T && T && (F || (T && T)))
       3         213       4: (F && _) || (T && T && (T || (_ && _)))
       2         213       5: (F && _) || (T && T && (F || (T && T)))
             0   213       6: (T && F) || (T && T && (F || (T && F)))
            13   213       7: (T && F) || (T && T && (F || (F && _)))
           344   213       8: (T && F) || (T && F && (_ || (_ && _)))
            17   213       9: (T && F) || (F && _ && (_ || (_ && _)))
             0   213       10: (F && _) || (T && T && (F || (T && F)))
            98   213       11: (F && _) || (T && T && (F || (F && _)))
          1612   213       12: (F && _) || (T && F && (_ || (_ && _)))
           108   213       13: (F && _) || (F && _ && (_ || (_ && _)))
      +          213       MC/DC (cond 1): 1 + 11,  1 - 10,  1 + 12,  1 + 13
      +          213       MC/DC (cond 2): 1 + 7,  1 - 6,  1 + 8,  1 + 9
      +          213       MC/DC (cond 3): 2 + 9,  3 - 9,  4 + 13,  5 + 13
      +          213       MC/DC (cond 4): 2 + 8,  3 - 8,  4 + 12,  5 + 12
      +          213       MC/DC (cond 5): 2 + 7,  2 - 6,  4 - 10,  4 + 11
      +          213       MC/DC (cond 6): 5 + 11,  3 - 7
      -          213       MC/DC (cond 7): 3 - 6,  5 - 10
      40         214     return (1.0);
```

There are two excluding cases in this complex condition

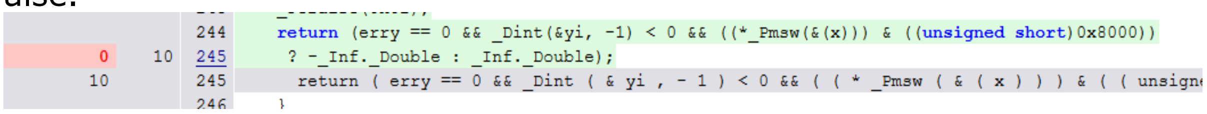
1) Erry==0 && y==0 have always the same results (even if y==-0.0). Hence MCDC cann0t be complete (condition 3-6)
2) xexp==1 is true for x in [1,2[it is impossible that x==0.5 and x==-0-5 are true at the same time (Condition 5-10)

7.20.3 _Pow: erry==0



```
 137  2061  216  else if (0 <= errx || 0 < erry)
 125        216     1: T || _
  12        216     2: F || T
      2061  216     3: F || F
   +        216     MC/DC (cond 1): 1 + 3
   +        216     MC/DC (cond 2): 2 + 3
            217   {
   6   131  218   if (errx == 2)
   6        219    return (x);
   9   122  220   else if (erry == 2)
   9        221    return (y);
   8   114  222   else if (errx == 1)
   4     4  223    if (!((*_Pmsw(&(x))) & ((unsigned short)0x8000)))
   2     2  224     return (((*_Pmsw(&(y))) & ((unsigned short)0x8000)) ? 0.0 : _Inf._Double);
   4        224         return ( ( ( * _Pmsw ( & ( y ) ) ) & ( ( unsigned short ) 0x8000 ) ) ? 0.0 : _Inf . _Double )
   2     2  225    else if (!((*_Pmsw(&(y))) & ((unsigned short)0x8000)))
            226     return (erry == 0 && _Dint(&yi, -1) < 0
   0     2  227      ? -_Inf._Double
   2        228      : _Inf._Double);
            229    else
            230     return (erry == 0 && _Dint(&yi, -1) < 0
   0     2  231      ? -_Zero : 0.0);
   2        231         return ( erry == 0 && _Dint ( & yi , - 1 ) < 0 ? - _Zero : 0.0 )
```

Since line 216 checks for 0<erry the case erry==0 in lines 226 and 230 cannot be true (also dead code).
Also erry==0 cannot be true (see above) and this code is also always false.



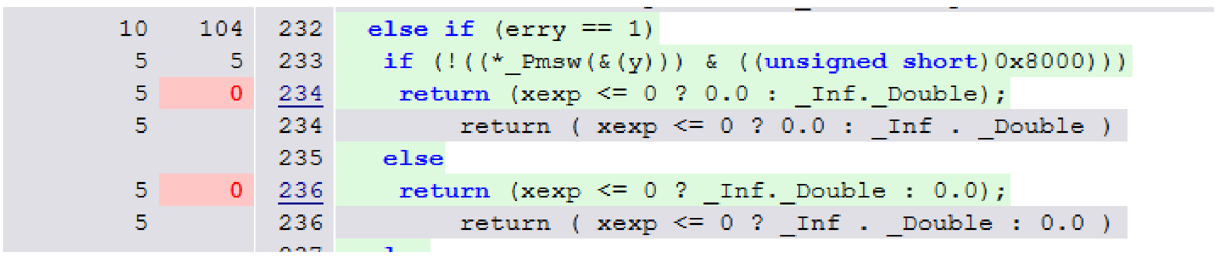
```
            244     return (erry == 0 && _Dint(&yi, -1) < 0 && ((*_Pmsw(&(x))) & ((unsigned short)0x8000))
   0    10  245      ? -_Inf._Double : _Inf._Double);
  10        245       return ( erry == 0 && _Dint ( & yi , - 1 ) < 0 && ( ( * _Pmsw ( & ( x ) ) ) & ( ( unsigne
            246     }
```

7.20.4 _Pow: erry==0

Same as in the previous lines

7.20.5 _Pow:xexp <=0 false

The case xexp==0 is the only case that can occur here, since the main condition is errx>=0 meand that x==0,INF or NAN. INF and NAN are handeled before. Tehrefore x=0 here and the exponen (xexp) is zero in this case. Hence the else branch is dead.



```
 10   104   232  else if (erry == 1)
  5     5   233   if (!((*_Pmsw(&(y))) & ((unsigned short)0x8000)))
  5     0   234    return (xexp <= 0 ? 0.0 : _Inf._Double);
  5         234        return ( xexp <= 0 ? 0.0 : _Inf . _Double )
            235   else
  5     0   236    return (xexp <= 0 ? _Inf._Double : 0.0);
  5         236        return ( xexp <= 0 ? _Inf . _Double : 0.0 )
```

7.20.6 _Pow: xexp<=0 false

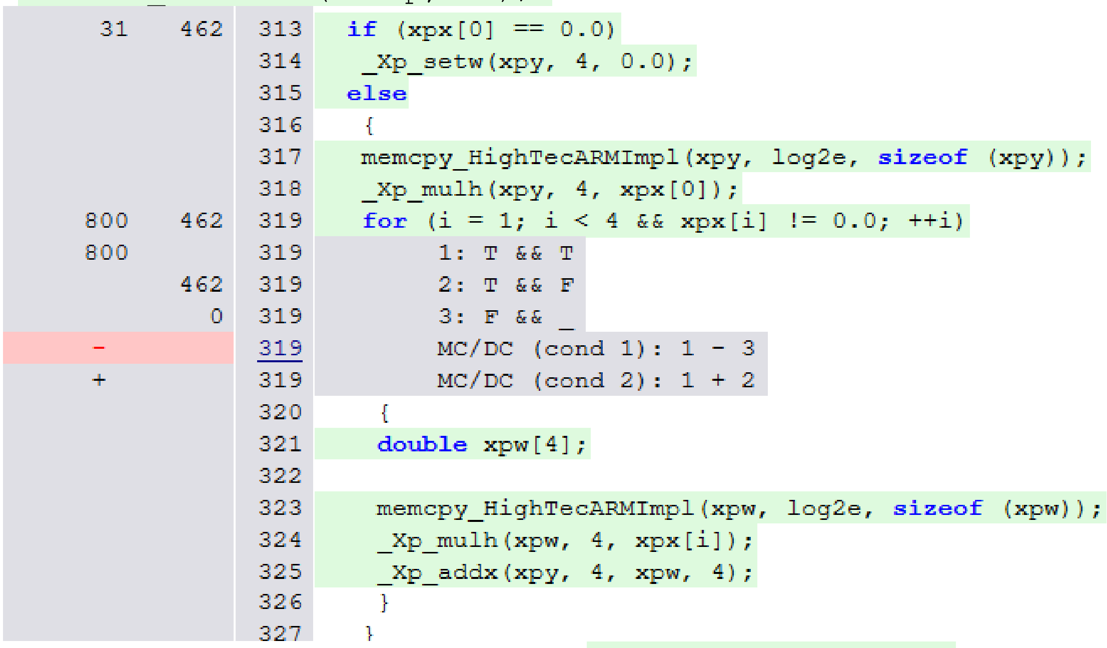
Same as in previous section

7.20.7 _Pow_ erry==0

Cannot occur (same argument as in corresponding previous section).

7.20.8 _Pow: for loop

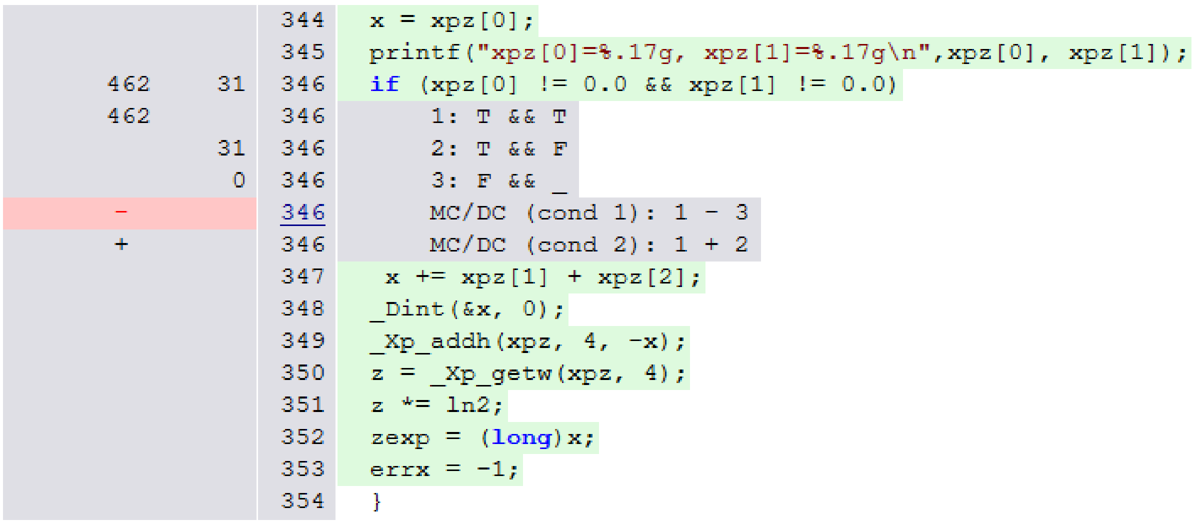
The following loop computes the scaled values until a maximal size of 4, however since the value x is unscaled in the beginning of the function by "`errx = _Dunscale(&xexp, &x);`"



```
 31   462   313  if (xpx[0] == 0.0)
            314   _Xp_setw(xpy, 4, 0.0);
            315  else
            316   {
            317   memcpy_HighTecARMImpl(xpy, log2e, sizeof (xpy));
            318   _Xp_mulh(xpy, 4, xpx[0]);
800   462   319   for (i = 1; i < 4 && xpx[i] != 0.0; ++i)
800         319        1: T && T
      462   319        2: T && F
        0   319        3: F && _
  -         319        MC/DC (cond 1): 1 - 3
  +         319        MC/DC (cond 2): 1 + 2
            320    {
            321    double xpw[4];
            322
            323    memcpy_HighTecARMImpl(xpw, log2e, sizeof (xpw));
            324    _Xp_mulh(xpw, 4, xpx[i]);
            325    _Xp_addx(xpy, 4, xpw, 4);
            326    }
            327   }
```

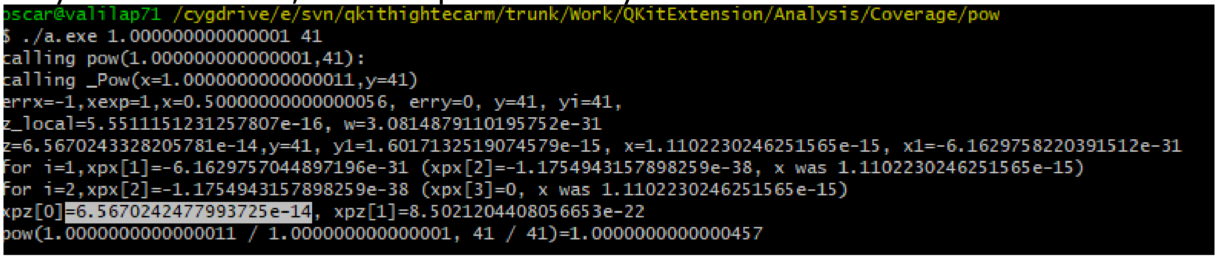
The scaled value x is set to xpx by "`_Xp_setw(xpx, 4, x);`". It occupies only up to two floats, hence the loop always terminates via xpx[3]==0.0 and never by i>=4. Therefore the condition i<4 is always true and could be omitted (even if I also would not like do it;-).

7.20.9 _Pow: xpz[0]!=0



```
                    x = xpz[0];
                    printf("xpz[0]=%.17g, xpz[1]=%.17g\n",xpz[0], xpz[1]);
  462     31        if (xpz[0] != 0.0 && xpz[1] != 0.0)
  462                   1: T && T
          31            2: T && F
           0            3: F && _
   -                    MC/DC (cond 1): 1 - 3
   +                    MC/DC (cond 2): 1 + 2
                     x += xpz[1] + xpz[2];
                    _Dint(&x, 0);
                    _Xp_addh(xpz, 4, -x);
                    z = _Xp_getw(xpz, 4);
                    z *= ln2;
                    zexp = (long)x;
                    errx = -1;
                    }
```

All values have been tested and it has been observed that xpz[0] goes only towards zero, if the input x is very close to 1.



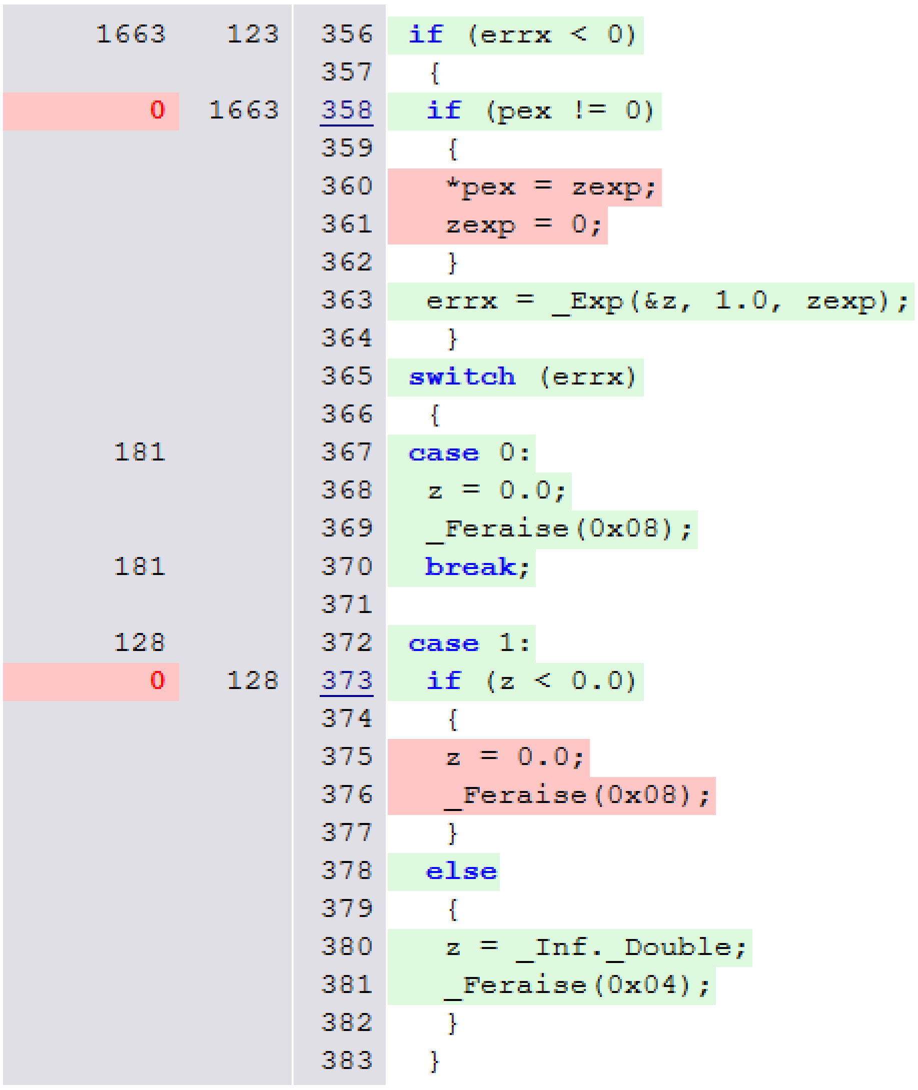
```
oscar@valilap71 /cygdrive/e/svn/qkithightecarm/trunk/Work/QKitExtension/Analysis/Coverage/pow
$ ./a.exe 1.000000000000001 41
calling pow(1.000000000000001,41):
calling _Pow(x=1.0000000000000011,y=41)
errx=-1,xexp=1,x=0.50000000000000056, erry=0, y=41, yi=41,
z_local=5.5511151231257807e-16, w=3.0814879110195752e-31
z=6.5670243328205781e-14,y=41, y1=1.6017132519074579e-15, x=1.1102230246251565e-15, x1=-6.1629758220391512e-31
for i=1,xpx[1]=-6.1629757044897196e-31 (xpx[2]=-1.1754943157898259e-38, x was 1.1102230246251565e-15)
for i=2,xpx[2]=-1.1754943157898259e-38 (xpx[3]=0, x was 1.1102230246251565e-15)
xpz[0]=6.5670242477993725e-14, xpz[1]=8.5021204408056653e-22
pow(1.0000000000000011 / 1.000000000000001, 41 / 41)=1.0000000000000457
```

However the case x=1 is handled differently, such that this case cannot occur and.

7.20.10 _Pow: pex!=0

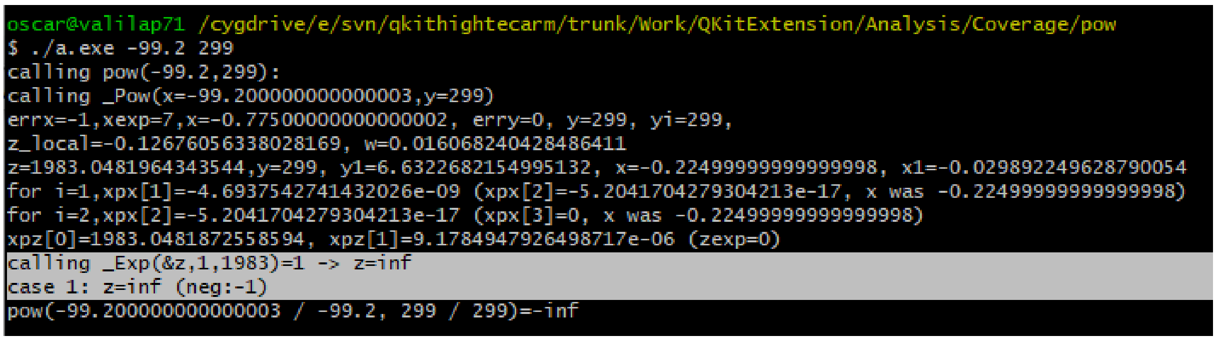
Dead, see identical case in first sub section.

7.20.11 _Pow: case 1 & z<0



```
   1663    123   356  if (errx < 0)
                 357   {
      0   1663   358   if (pex != 0)
                 359    {
                 360    *pex = zexp;
                 361    zexp = 0;
                 362    }
                 363   errx = _Exp(&z, 1.0, zexp);
                 364    }
                 365  switch (errx)
                 366   {
    181          367  case 0:
                 368   z = 0.0;
                 369   _Feraise(0x08);
    181          370   break;
                 371
    128          372  case 1:
      0    128   373   if (z < 0.0)
                 374    {
                 375    z = 0.0;
                 376    _Feraise(0x08);
                 377    }
                 378   else
                 379    {
                 380    z = _Inf._Double;
                 381    _Feraise(0x04);
                 382    }
                 383   }
```

This code is dead, since the negative values are handled before (using variable neg).
Trying to get a negative INF out of _Exp call showed that only neg was used

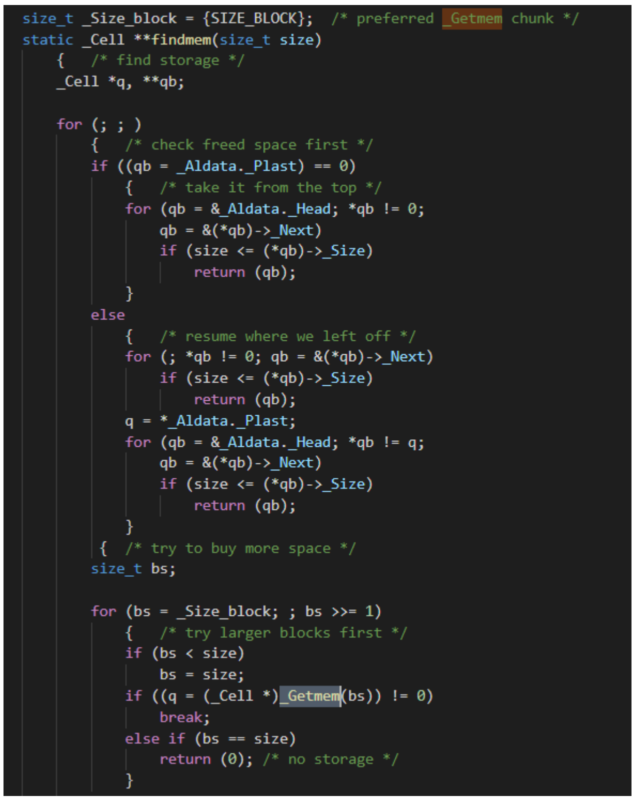


```
oscar@valilap71 /cygdrive/e/svn/qkithightecarm/trunk/Work/QKitExtension/Analysis/Coverage/pow
$ ./a.exe -99.2 299
calling pow(-99.2,299):
calling _Pow(x=-99.200000000000003,y=299)
errx=-1,xexp=7,x=-0.77500000000000002, erry=0, y=299, yi=299,
z_local=-0.12676056338028169, w=0.016068240428486411
z=1983.0481964343544,y=299, y1=6.6322682154995132, x=-0.22499999999999998, x1=-0.029892249628790054
for i=1,xpx[1]=-4.6937542741432026e-09 (xpx[2]=-5.2041704279304213e-17, x was -0.22499999999999998)
for i=2,xpx[2]=-5.2041704279304213e-17 (xpx[3]=0, x was -0.22499999999999998)
xpz[0]=1983.0481872558594, xpz[1]=9.1784947926498717e-06 (zexp=0)
calling _Exp(&z,1,1983)=1 -> z=inf
case 1: z=inf (neg:-1)
pow(-99.200000000000003 / -99.2, 299 / 299)=-inf
```

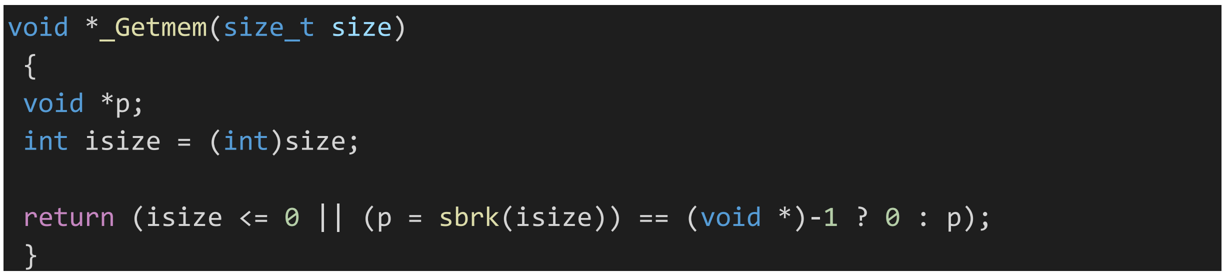
Hence this code is dead.

7.21 Analysis of code coverage of function _Getmem: OK

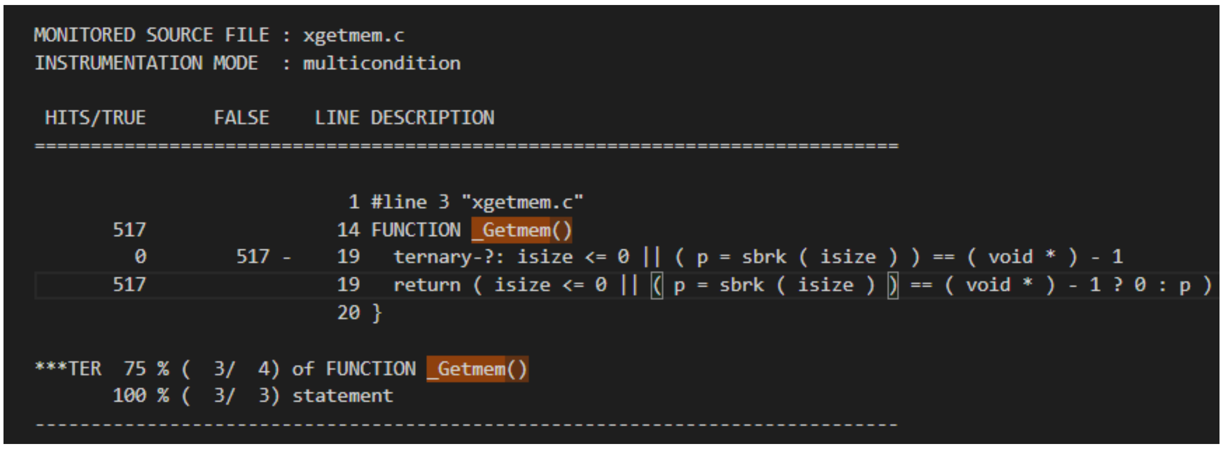
_Getmem is called in the context of function findmem with parameter size = 512.



```
size_t _Size_block = {SIZE_BLOCK};  /* preferred _Getmem chunk */
static _Cell **findmem(size_t size)
    {   /* find storage */
    _Cell *q, **qb;

    for (; ; )
        {   /* check freed space first */
        if ((qb = _Aldata._Plast) == 0)
            {   /* take it from the top */
            for (qb = &_Aldata._Head; *qb != 0;
                qb = &(*qb)->_Next)
                if (size <= (*qb)->_Size)
                    return (qb);
            }
        else
            {   /* resume where we left off */
            for (; *qb != 0; qb = &(*qb)->_Next)
                if (size <= (*qb)->_Size)
                    return (qb);
            q = *_Aldata._Plast;
            for (qb = &_Aldata._Head; *qb != q;
                qb = &(*qb)->_Next)
                if (size <= (*qb)->_Size)
                    return (qb);
            }
         {  /* try to buy more space */
        size_t bs;

        for (bs = _Size_block; ; bs >>= 1)
            {   /* try larger blocks first */
            if (bs < size)
                bs = size;
            if ((q = (_Cell *)_Getmem(bs)) != 0)
                break;
            else if (bs == size)
                return (0); /* no storage */
            }
```



```
void *_Getmem(size_t size)
 {
 void *p;
 int isize = (int)size;

 return (isize <= 0 || (p = sbrk(isize)) == (void *)-1 ? 0 : p);
 }
```



```
 MONITORED SOURCE FILE : xgetmem.c
 INSTRUMENTATION MODE  : multicondition

  HITS/TRUE      FALSE    LINE DESCRIPTION
 ============================================================================

                               1 #line 3 "xgetmem.c"
        517                   14 FUNCTION _Getmem()
          0         517 -     19   ternary-?: isize <= 0 || ( p = sbrk ( isize ) ) == ( void * ) - 1
        517                   19   return ( isize <= 0 || ( p = sbrk ( isize ) ) == ( void * ) - 1 ? 0 : p )
                              20 }

 ***TER  75 % (  3/   4) of FUNCTION _Getmem()
        100 % (  3/   3) statement
 ----------------------------------------------------------------------------
```

- Since isize = size = 512, the condition isize <= 0 is false
- Since sbrk will be in almost all cases successful, it will return the prior value of the program break, i.e. it won't be equal to -1, the condition (p = sbrk(isize)) == (void *)-1 is false

➔The ternary will almost always be false

7.22 Analysis of code coverage of function _FExp[1] OK

Function _FExp is called by

- expf
- exp2f
- _FPow
- _FCosh
- _FSinh

All functions call _FExp with constant or finite (and non-zero for y) (see subsections)

```
    5925              105 FUNCTION _FExp()
      72     5853     110   if (0 <= errx || 0 <= erry)
      72              110     1: T || _
       0              110     2: F || T
             5853     110     3: F || F
                      110     MC/DC (cond 1): 1 + 3
                -     110     MC/DC (cond 2): 2 - 3
       0       72 -   112     if (errx == 2)
       0          -   113       return ( 2 )
                      113     }+
       0       72 -   114     else if (erry == 2)
       0          -   117       return ( 2 )
                      118     }+
       0       72 -   119     else if (erry == 0)
       0        0 -   120       if (* px != _FInf . _Float)
       0          -   123         return ( 0 )
                      124       }-
                      125       else
       0          -   129         return ( 2 )
                      130       }-
                      130     }+
       0       72 -   131     else if (erry == 1)
       0        0 -   132       if (* px != - _FInf . _Float)
       0          -   135         return ( 1 )
                      136       }-
                      137       else
       0          -   141         return ( 2 )
                      142       }-
                      142     }+
      72        0 -   143     else if (errx == 0)
                      146       switch (errx = _FDscale ( px , eoff ))
       4              148       case 0:
       4              150         break
       2              152       case 1:
                      154       }+
      72              155       return ( errx )
                      156     }-
       0        0 -   157     else if (* px == _FInf . _Float)
       0          -   160       return ( 1 )
                      161     }-
                      162     else
       0          -   165       return ( 0 )
                      166     }-
                      167   }+
     264     5589     168   else if (* px < - hugexp)
     264              171     return ( 0 )
                      172   }+
     707     4882     173   else if (hugexp < * px)
     707              177     return ( 1 )
                      178   }+
                      179   else
    1806     3076     182     ternary-?: g < 0.0F
     696     4186     186     if (- _FEps . _Float < g && g < _FEps . _Float)
     696              186       1: T && T
             1752     186       2: T && F
             2434     186       3: F && _
                      186       MC/DC (cond 1): 1 + 3
```

[1] Exp is dervied from the same source code (as most double/single functions are) and hence not analyzed again.

```
                      186        MC/DC (cond 2): 1 + 2
                      187      }+
                      188      else
                      196      }+
                      198      switch (errx = _FDscale ( px , ( long ) xexp + eoff ))
      70              200      case 0:
      70              202        break
      84              204      case 1:
                      206      }+
    4882              207      return ( errx )
                      208    }-
                      209 }

***TER  59 % ( 29/ 49) of FUNCTION _FExp()
        65 % ( 37/ 57) statement
```

The analysis is done based on the line numbers:

- 110: Missing case 0<=erry means y = INF/NAN/0 is never true (see following subsections)
- 112: NAN-case for x is also handled from main functions
- 117: NAN-case for y is dead, see line 110
- 119: y==0-case is excluded when calling the function for efficiency (e.g. see powf)
- 131-142: INF-case for y is dead, see line 110
- 143: only remaining case, since x!=INF, x!=NAN is that errx==0 hence this condition is always true and the false branch is dead.

So it remain to be shown for all callers of the function that

- They do not call _FExp with x=INF/NAN, with y=INF/NAN/0

This is done within the following sections.

7.22.1 Analysis of code coverage of function _FExp in context of expf

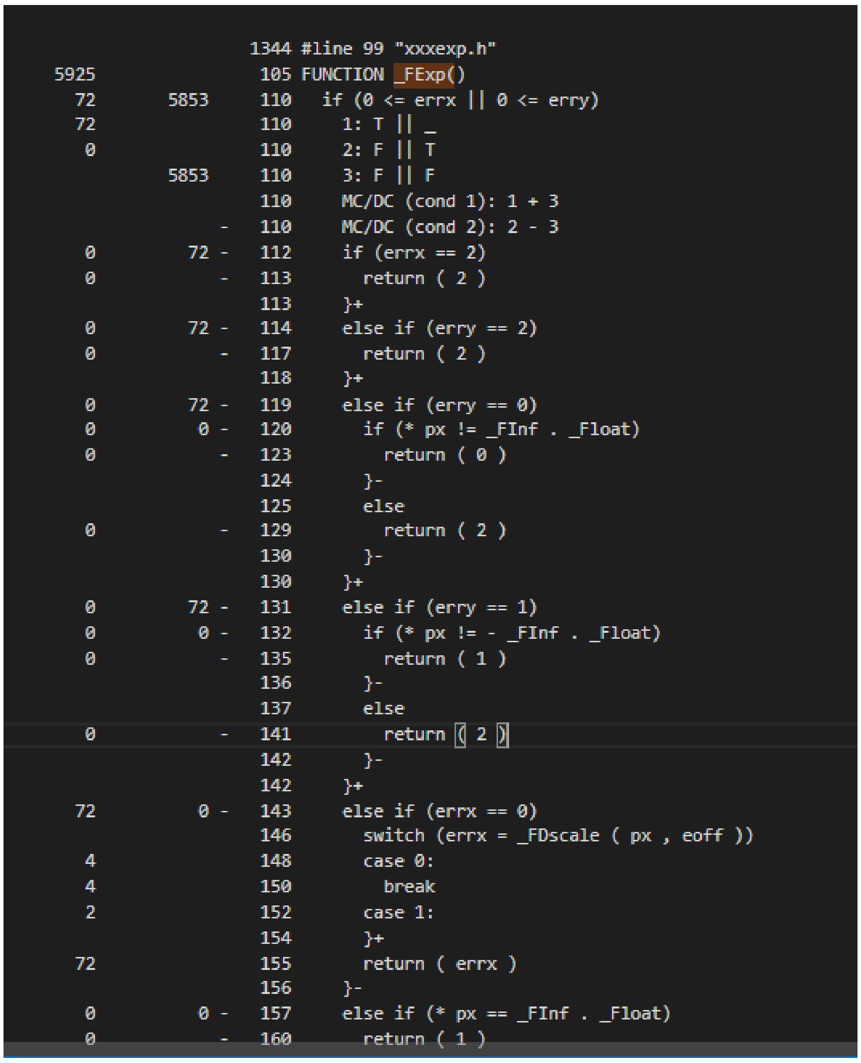
```
float (expf)(float x)
 {
 switch (_FDtest(&x))
  {
 case 2:
  return (x);

 case 1:
  return (((*_FPmsw(&(x))) & ((unsigned short)0x8000)) ? 0.0F : x);

 case 0:
  return (1.0F);

 default:
  _FExp(&x, 1.0F, 0);
  return (x);
  }
 }
```

Parameter x is passed to _FDtest which classifies x into the following categories:

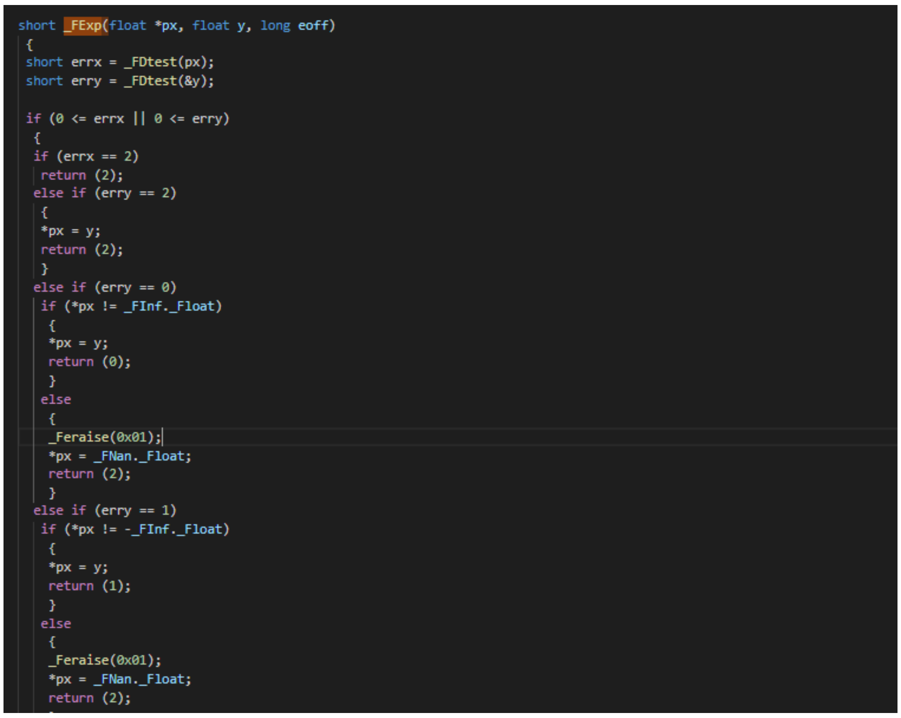
- _DENORM = -2
- _FINITE = -1
- 0 (default value)

- _INFCODE = 1
- _NANCODE = 2

_FExp is only called when either x is classified by _FDtest as _DENORM or _FINITE.
Furthermore _FExp is called with y = 1.0F and eoff = 0.



```
                      1344 #line 99 "xxxexp.h"
  5925                 105 FUNCTION _FExp()
    72       5853      110   if (0 <= errx || 0 <= erry)
    72                 110     1: T || _
     0                 110     2: F || T
             5853      110     3: F || F
                       110     MC/DC (cond 1): 1 + 3
                  -    110     MC/DC (cond 2): 2 - 3
     0         72 -    112     if (errx == 2)
     0            -    113       return ( 2 )
                       113     }+
     0         72 -    114     else if (erry == 2)
     0            -    117       return ( 2 )
                       118     }+
     0         72 -    119     else if (erry == 0)
     0          0 -    120       if (* px != _FInf . _Float)
     0            -    123         return ( 0 )
                       124       }-
                       125       else
     0            -    129         return ( 2 )
                       130       }-
                       130     }+
     0         72 -    131     else if (erry == 1)
     0          0 -    132       if (* px != - _FInf . _Float)
     0            -    135         return ( 1 )
                       136       }-
                       137       else
     0            -    141         return ( 2 )
                       142       }-
                       142     }+
    72          0 -    143     else if (errx == 0)
                       146       switch (errx = _FDscale ( px , eoff ))
     4                 148       case 0:
     4                 150         break
     2                 152       case 1:
                       154       }+
    72                 155       return ( errx )
                       156     }-
     0          0 -    157     else if (* px == _FInf . _Float)
     0            -    160       return ( 1 )
```



```
short _FExp(float *px, float y, long eoff)
 {
 short errx = _FDtest(px);
 short erry = _FDtest(&y);

 if (0 <= errx || 0 <= erry)
  {
  if (errx == 2)
   return (2);
  else if (erry == 2)
   {
   *px = y;
   return (2);
   }
  else if (erry == 0)
   if (*px != _FInf._Float)
    {
    *px = y;
    return (0);
    }
   else
    {
    _Feraise(0x01);
    *px = _FNan._Float;
    return (2);
    }
  else if (erry == 1)
   if (*px != -_FInf._Float)
    {
    *px = y;
    return (1);
    }
   else
    {
    _Feraise(0x01);
    *px = _FNan._Float;
    return (2);
```

7.22.1.1 Condition if (0 <= errx || 0 <= erry)

The Condition if (0 <= errx || 0 <= erry) in line 110 cannot be covered, since errx is either equal to -2 or -1 and erry = _FDtest(1.0) == -1.

7.22.2 Analysis of code coverage of function _FExp in context of exp2f

```
float (exp2f)(float x)
 {
 long xexp;

 switch (_FDtest(&x))
  {
 case 2:
  return (x);

 case 1:
  return (((*_FPmsw(&(x))) & ((unsigned short)0x8000)) ? 0.0F : x);

 case 0:
  return (1.0F);

 default:
  if (x <= (float)(-0x7fffffffL - 1))
   return (0.0F);
  else if ((float)0x7fffffffL <= x)
   return (_FInf._Float);
  else
   {
   xexp = (long)x;
   x -= (float)xexp;
   if (0.5F < x)
    {
    x -= 1.0F;
    ++xexp;
    }
   else if (x < -0.5F)
    {
    x += 1.0F;
    --xexp;
    }
   }
  x *= ln2;
  _FExp(&x, 1.0F, xexp);
  return (x);
  }
 }
```
Parameter x is passed to _FDtest which classifies x into the following categories:

- _DENORM = -2
- _FINITE = -1
- 0 (default value)
- _INFCODE = 1
- _NANCODE = 2

_FExp is only called when either x is classified by _FDtest as _DENORM or _FINITE.
Furthermore _FExp is called with y = 1.0F and eoff = xexp.

7.22.2.1 Condition if (0 <= errx || 0 <= erry)

The Condition if (0 <= errx || 0 <= erry) in line 110 cannot be covered, since errx is either equal to -2 or -1 and erry = _FDtest(1.0) == -1.

7.22.3 Analysis of code coverage of function _FExp in context of _FPow

The only call is

```
errx = _FExp(&z, 1.0F, zexp);
```

so y is constant and z is finite since other cases are checked earlier.

7.22.4 Analysis of code coverage of function _FExp in context of _FCosh

There are two calls in _FCosh:

```
else
    {   /* x and y finite */
    FMAKEPOS(x);

    if (x < xbig)
        {   /* worth adding in exp(-x) */
        FNAME(Exp)(&x, FLIT(1.0), -1);
        return (y * (x + FDIV(FLIT(0.25), x)));
        }
    else
        {   /* x large, compute y*exp(x)/2 */
        FNAME(Exp)(&x, y, -1);
        return (x);
        }
    }
}
```

Both values are finite and even the case y=0 (erry=0) is handeld earlier in cosh

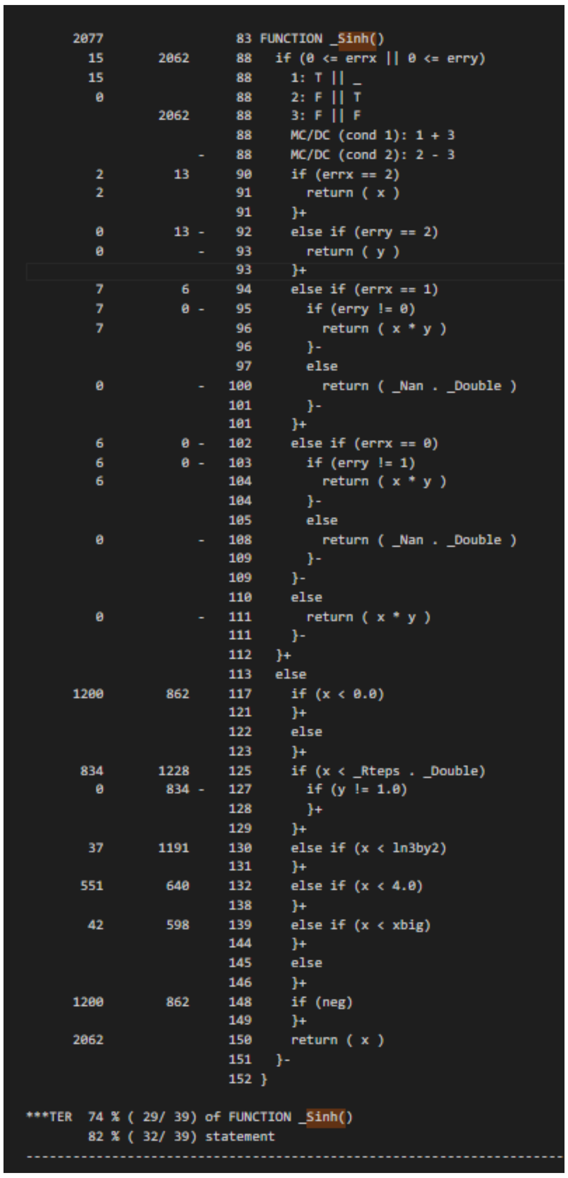
7.22.5 Analysis of code coverage of function _FExp in context of _FSinh

FSinh calls Exp only once and the argument is the same as for FCosh

```
{   /* x and y finite */
short neg;

if (x < FLIT(0.0))
    {   /* make positive and remember sign */
    FNEGATE(x);
    neg = 1;
    }
else
    neg = 0;

if (x < FCONST(Rteps))
    {   /* x tiny, multiply by y */
    if (y != FLIT(1.0)) /* avoid FTZ */
        x *= y;
    }
else if (x < ln3by2)
    x = y * FNAME(Sinh_small)(x);
else if (x < FLIT(4.0))
    {   /* don't factor out expm1(z)  */
    FTYPE z = FFUN(expm1)(x);

    x = z + FLIT(0.5) * z * FFUN(expm1)(-x);
    x *= y;
    }
else if (x < xbig)
    {   /* x small, compute (exp(x)-exp(-x))/2 carefully */
    x = FFUN(expm1)(x)
            * (FLIT(1.0) + FLIT(0.5) * FFUN(expm1)(-x));
    x *= y;
    }
else
    FNAME(Exp)(&x, y, -1);

if (neg)
    FNEGATE(x);
return (x);
}
```

7.23 Analysis of code coverage of function sinh: OK

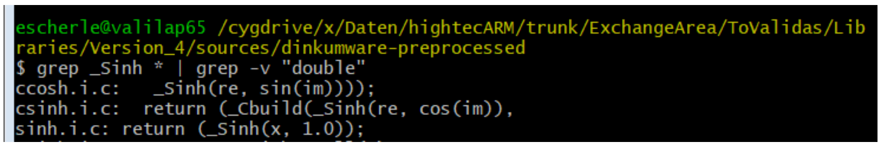
Uses the function _Sinh that has insufficient code coverage



```
   2077                  83 FUNCTION _Sinh()
     15       2062       88   if (0 <= errx || 0 <= erry)
     15                  88     1: T || _
      0                  88     2: F || T
              2062       88     3: F || F
                         88     MC/DC (cond 1): 1 + 3
                     -   88     MC/DC (cond 2): 2 - 3
      2         13       90     if (errx == 2)
      2                  91       return ( x )
                         91     }+
      0         13 -     92     else if (erry == 2)
      0            -     93       return ( y )
                         93     }+
      7          6       94     else if (errx == 1)
      7          0 -     95       if (erry != 0)
      7                  96         return ( x * y )
                         96       }-
                         97       else
      0            -    100         return ( _Nan . _Double )
                        101       }-
                        101     }+
      6          0 -    102     else if (errx == 0)
      6          0 -    103       if (erry != 1)
      6                 104         return ( x * y )
                        104       }-
                        105       else
      0            -    108         return ( _Nan . _Double )
                        109       }-
                        109     }-
                        110     else
      0            -    111       return ( x * y )
                        111     }-
                        112   }+
                        113   else
   1200        862      117     if (x < 0.0)
                        121     }+
                        122     else
                        123     }+
    834       1228      125     if (x < _Rteps . _Double)
      0        834 -    127       if (y != 1.0)
                        128       }+
                        129     }+
     37       1191      130     else if (x < ln3by2)
                        131     }+
    551        640      132     else if (x < 4.0)
                        138     }+
     42        598      139     else if (x < xbig)
                        144     }+
                        145     else
                        146     }+
   1200        862      148     if (neg)
                        149     }+
   2062                 150     return ( x )
                        151   }-
                        152 }

***TER  74 % ( 29/ 39) of FUNCTION _Sinh()
        82 % ( 32/ 39) statement
-----------------------------------------------------------------------------
```

Is called by sinh with the second argument fixed to one (and nowhere else, except complex functions that are not qualified)

```
FTYPE (FFUN(sinh))(FTYPE x)
    {   /* compute sinh(x) */
    return (FNAME(Sinh)(x, FLIT(1.0)));
    }
```



```
escherle@valilap65 /cygdrive/x/Daten/hightecARM/trunk/ExchangeArea/ToValidas/Lib
raries/Version_4/sources/dinkumware-preprocessed
$ grep _Sinh * | grep -v "double"
ccosh.i.c:    _Sinh(re, sin(im))));
csinh.i.c:   return (_Cbuild(_Sinh(re, cos(im)),
sinh.i.c: return (_Sinh(x, 1.0));
```

y=1.0 leads to erry=-1. Therefore the following uncovered parts cannot be reached

- 0<=yerr: is always false
- elseif (yerr==2) is always false
- if (erry!=0) is always true

- if (erry!=1) is always true
- if (y!=1.0) is always false

The condition if (errx == 0) in line 102 is always true (the false branch cannot be covered), since

- the condition if (0 <= errx || 0 <= erry) needs to be true to reach line 102. Since 0 <= erry is false, 0 <= errx needs to be true.
- When line 102 is reached, errx!=2 and errx!=1 and errx >= 0

```
FTYPE FNAME(Sinh)(FTYPE x, FTYPE y)
    {   /* compute sinh(x)*y */
    const short errx = FNAME(Dtest)(&x);
    const short erry = FNAME(Dtest)(&y);

    if (0 <= errx || 0 <= erry)
        {   /* x or y is 0, Inf, or NAN */
        if (errx == _NANCODE)
            return (x); /* sinh(NaN)*y */
        else if (erry == _NANCODE)
            return (y); /* sinh(x)*NaN */
        else if (errx == _INFCODE)
            if (erry != 0)
                return (x * y); /* sinh(Inf)*{finite or Inf} */
            else
                {   /* sinh(Inf)*0, report invalid */
                _Feraise(_FE_INVALID);
                return (FCONST(Nan));
                }
        else if (errx == 0)
            if (erry != _INFCODE)
                return (x * y); /* sinh(0)*{finite or 0} */
            else
                {   /* sinh(0)*Inf, report invalid */
                _Feraise(_FE_INVALID);
                return (FCONST(Nan));
                }
        else
            return (x * y); /* sinh(finite)*{0 or Inf} */
        }
    else
        {   /* x and y finite */
        short neg;

        if (x < FLIT(0.0))
            {   /* make positive and remember sign */
            FNEGATE(x);
            neg = 1;
            }
        else
            neg = 0;

        if (x < FCONST(Rteps))
            {   /* x tiny, multiply by y */
            if (y != FLIT(1.0)) /* avoid FTZ */
                x *= y;
            }
        else if (x < ln3by2)
            x = y * FNAME(Sinh_small)(x);
        else if (x < FLIT(4.0))
            {   /* don't factor out expm1(z) */
            FTYPE z = FFUN(expm1)(x);

            x = z + FLIT(0.5) * z * FFUN(expm1)(-x);
            x *= y;
            }
        else if (x < xbig)
            {   /* x small, compute (exp(x)-exp(-x))/2 carefully */
            x = FFUN(expm1)(x)
                    * (FLIT(1.0) + FLIT(0.5) * FFUN(expm1)(-x));
            x *= y;
            }
        else
            FNAME(Exp)(&x, y, -1);

        if (neg)
            FNEGATE(x);
        return (x);
        }
    }
```

Therefore coverage in sinh is maximal. Same argumentation holds for sinhf.

8 Summary

The code coverage of the selected Dinkumware library and AEABI has been analyzed successfully. Several parts are dead within the analyzed context, e.g. complex value functions have not been considered and other parts are real dead code that could be removed. However this might have been added from the author as kind of “defensive programming”.

9 References

[1] MC/DC Coverage Report generated by CTC

https://opensvn.teststatt.de/repos/qkithightecarm/trunk/Work/QKitExtension/Analysis/report.txt

[2] List of successfully, manually analyzed functions

https://opensvn.teststatt.de/repos/qkithightecarm/trunk/Work/QKitExtension/Analysis/ManualCoverage_OK.txt

[3] List of all functions to be qualified (inclusive sub-functions)

https://opensvn.teststatt.de/repos/qkithightecarm/trunk/Work/QKitExtension/Analysis/Listfun_dependencies_computed.txt