

## Intel Corporation

### **DIVIDE, SQUARE ROOT AND REMAINDER ALGORITHMS FOR THE IA-64 ARCHITECTURE**

**Marius Cornea** ([Marius.Cornea@intel.com](mailto:Marius.Cornea@intel.com))

**John Harrison** ([johnh@ichips.intel.com](mailto:johnh@ichips.intel.com))

**Cristina Iordache** ([Cristina.S.Iordache@intel.com](mailto:Cristina.S.Iordache@intel.com))

**Bob Norin** ([Bob.Norin@intel.com](mailto:Bob.Norin@intel.com))

**Shane Story** ([Shane.Story@intel.com](mailto:Shane.Story@intel.com))

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. Product Name may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's website at <http://developer.intel.com/design/itcentr>.

Copyright © Intel Corporation, 2000

\*Third-party brands and names are the property of their respective owners.

**TABLE OF CONTENTS**

<b>1. Introduction</b>	<b>4</b>
<b>2. Floating-Point Divide Algorithms for the IA-64 Architecture</b>	<b>8</b>
<b>3. Floating-Point Square Root Algorithms for the IA-64 Architecture</b>	<b>38</b>
<b>4. Integer Divide and Remainder Algorithms for the IA-64 Architecture</b>	<b>73</b>
<b>5. References</b>	<b>112</b>

## 1. Introduction

### 1.1. Background

The IA-64 architecture specifies two approximation instructions **frcpa** and **frsqrrta** that are designed to support efficient and IEEE-correct software implementations of division, square root and remainder.

Deferring most of the division and square root computations to software offers several advantages. Most importantly, since each operation is broken down into several simpler instructions, these individual instructions can be scheduled more flexibly in conjunction with the rest of the code, increasing the potential for parallelism. In particular:

- Since the underlying operations are fully pipelined, the division and square root operations inherit the pipelining, allowing high throughput.
- If a perfectly rounded IEEE-correct result is not required (e.g. in graphics applications), faster algorithms can be substituted.

Intel provides a number of recommended division and square root algorithms, in the form of short sequences of straight-line code written in assembly language for the IA-64 architecture. The intention is that these can be inlined by compilers, used as the core of mathematical libraries, or called on as macros by assembly language programmers. It is expected that these algorithms will serve all the needs of typical users, who when using a high-level language may be unaware of how division and square root are actually implemented.

All the Intel-provided algorithms have been carefully designed to provide IEEE-correct results and trigger IEEE flags and exceptions appropriately. Subject to this correctness constraint, they have been written to maximize performance on the Itanium™ processor, the first silicon implementation of the IA-64 architecture. However, they are also likely to be the most appropriate algorithms for future IA-64 processors, even those with significantly different hardware characteristics.

### 1.2. Versions of the algorithms

The IA-64 architecture provides full support for the following three floating-point formats specified in the IEEE 754 floating-point Standard [1].

Format	Effective precision	Total bits in encoding
Single	24	32
Double	53	64
Double-extended	64	80

In addition, the IA-64 architecture specifies a packed type of two parallel single precision floating-point numbers, intended to support SIMD (single instruction, multiple data stream) computation where the same operation is applied to more than one data set in parallel.

Accordingly, different division and square root algorithms are provided for single, double, double-extended and SIMD formats. Generally, the algorithms for single precision are the fastest, those for double-extended precision the slowest and those for double precision in between. SIMD algorithms are typically slightly slower than those for single precision, since intermediate computation steps cannot take advantage of higher intermediate precision.

As well as the multiplicity of formats, most algorithms have two separate variants, one of which is designed to minimize latency (i.e. the number of clock cycles between starting the operation and having the result available), and the other to maximize throughput (the number of cycles used to execute an instruction, averaged over a large number of independent instances). Which variant is best to use depends on the kind of program within which it is being invoked. For example, when taking the square root of each element of an array and placing the results in another array, one would wish to maximize throughput: because the operations are independent they can fully exploit parallelism, and the latency of an individual operation is hardly significant. If on the other hand the operation is part of a chain of serially dependent computations where later computations cannot proceed till the result is available, one would wish to minimize latency. Generally speaking, an intelligent choice should be made by the compiler, but when hand-optimizing code the user may be able to make a better decision.

### 1.3. Performance

The following tables show the cycle times for the functions of various precisions to execute on an Itanium™ processor. Note that special versions of the SIMD algorithms are provided that rescale to set the denormal flag; these have slightly higher figures.

DIVISION	Single	Double	Extended	SIMD (not scaled)	SIMD (scaled)
Optimized latency	30	35	40	31	35
Optimized throughput	3.5	5.0	7.0	5.5	7.0

SQUARE ROOT	Single	Double	Extended	SIMD (not scaled)	SIMD (scaled)
Optimized latency	35	45	50	35	40
Optimized throughput	5.0	7.0	7.5	6.5	-

The square root algorithms rely on loading constants, and the time taken to load these constants is not included in the overall latencies. If the function is inlined by an optimizing compiler, these loads should be issued early as part of normal operation reordering.

### 1.4. Algorithm details

Section 2 of the current document details the Intel-provided algorithms for floating-point division, and section 3 the square root algorithms. Section 4 covers the algorithms for integer division and remainder, which are based on floating-point cores (see below). This document is addressed to writers of compilers, mathematical libraries, floating-point emulation libraries, floating-point exception handlers, binary translators, and test and validation suites for the IA-64 architecture.

All the algorithms are implemented as segments of straight-line code, which perform an initial approximation step (**frepa** or **frsqrrta**) and then refine the resulting approximation to give a correctly rounded result, using power series or iterative methods such as the Newton-Raphson or Goldschmidt iteration.

All of these algorithms have been mathematically analyzed and proved IEEE-correct, both by hand [3], [4] and machine-checked proofs, for the core case where SWA faults do not occur (see below). Correctness means providing the correctly rounded result whatever the ambient IEEE rounding mode, and setting all the IEEE flags or triggering exceptions appropriately. The Intel-specific denormal flag or exception is also triggered correctly whenever either input argument is unnormal. However, in some of the SIMD algorithms, it may also be triggered spuriously when the input argument (first argument in the case of division) is very small. The precise conditions are stated for each algorithm.

The document also contains optimized assembly language implementations of the divide and square root algorithms, together with simple test drivers. The optimization is aimed at minimizing the number of clock cycles necessary to execute each algorithm on the Itanium™ processor, and also at minimizing register usage (only scratch registers were used). The code is presented as short routines in assembly language for the IA-64 architecture, but using them as inlined sequences is straightforward. As can be seen from the assembly language implementations, all the computation steps of the 14 different algorithms map directly to IA-64 architecture instructions. For all the algorithms, the first and the last instruction share the status field in the Floating-Point Status Register [2] that contains the user settings (usually status field 0; wre set to 0 is assumed, otherwise underflow or overflow conditions might not be reported correctly), while all the other instructions use status field 1 (whose settings include 64-bit precision, rounding to nearest, and widest range exponent, wre, set to 1). The FPSR default value is 0x0009804c02700b23 (see [2], [5] for more details).

### 1.5. Software assistance

For certain inputs, typically when the inputs are extremely large or small, the main path of the algorithms may not compute the result correctly, because of overflow or underflow in intermediate steps. However, in these situations the initial approximation instruction, **frcpa** or **frsqrrta**, will either return the correct result, or generate an IA-64 Architecturally Mandated Software Assistance (SWA) fault, which should cause the floating-point SWA handler to compute the correct result via the IA-64 Architecture Floating-Point Emulation Library. The SWA conditions are stated precisely for each algorithm. In such cases where the body of the algorithm should not be executed, the initial approximation instructions will always clear their output predicate register. Therefore, all the other instructions in the body of the algorithm, which are predicated on this register, will not be executed in the exceptional cases. For finite nonzero arguments in the scalar formats, no exceptional cases occur. They can however occur for certain register format inputs beyond the double-extended range.

For correct operation of the software, it is imperative to use the same destination register for the first instruction of the computation (the reciprocal approximation instruction), and for the last one (an **fma** instruction that generates the final result). In this way, whether an IA-64 architecturally mandated SWA fault occurs or not, the result of the scalar computation will be provided correctly in the destination register of the last instruction.

For the parallel algorithms, the computation beyond the first instruction is disabled too by the cleared output predicate from the initial approximation instruction **fprcpa** or **fprsqrrta** but the parallel approximation does not raise a SWA fault, and does not provide the IEEE correct result. Instead, it provides an approximation (see [2]) that can be possibly used in a "dirty" (not IEEE correct) operation. If an IEEE correct result is desired when the output predicate of the reciprocal approximation instruction is cleared, then the input operands have to be unpacked, one of the scalar algorithms for single precision computations has to be applied to the low and then to the high half, and the results have to be packed into a SIMD result, in the same destination register as that of the last instruction of the parallel algorithm. This method was used in the assembly language implementations of the algorithms described below.

### 1.6. Integer division

The IA-64 architecture does not provide any integer divide or remainder operations, and these too are meant to be implemented by software. In section 4, Intel provides some recommended algorithms that have been mathematically proved correct, and are designed to be efficient. Integer division works by transferring the operands to floating-point registers, performing an approximate floating-point division, and truncating the answer before returning it to an integer register. The remainder operation is based on integer division: one more multiply-subtract is performed at the end.

The integer divide and remainder computations are not affected by the rounding mode set by the user in the FPSR main status field (sf0); all floating-point operations use the reserved status field sf1, and thus are performed in register file size format (17-bit exponent, 64-bit mantissa) and round to nearest mode.

More efficient algorithms are provided for short integers. The fastest algorithms are for 8-bit operands, and the slowest but most general for 64-bit operands; 16-bit and 32-bit variants of intermediate speed are also provided.

The latencies and throughput of the integer divide and remainder operations are as follows:

Operation (signed/unsigned)	Latency	Throughput
8-bit integer divide	36	3
8-bit integer remainder	43	3.5
16-bit integer divide	41	3.5
16-bit integer remainder	48	4
32-bit integer divide	46	4.5
32-bit integer remainder	53	5
64-bit integer divide	56	6.5
64-bit integer remainder	63	7

The integer inputs must be transferred to floating-point registers and converted to floating-point format. After the final step of the quotient computation, the quotient must be truncated to integer. Therefore the latencies of these conversions, as well as the latencies of transfers between the general and floating-point register files, must be added to the FP computation latencies. For 8, 16, and 32 bit integer division and remainder, the arguments may need to be sign extended to 64 bits (if they are not already passed in 64-bit format); the latency of the sign extension operation (sxt, zxt) was not added to the table above.

The throughput was estimated as the number of floating point instructions divided by the number of floating-point units available.

The integer operations will not cause software assistance requests. All input arguments are representable as 64-bit integers, which are representable as double extended precision numbers (15-bit exponent, 64-bit mantissa). Software assistance conditions for scalar floating-point divide occur only for 17-bit exponent arguments.

The only special case when the quotient is provided by the `frcpa` instruction, and not by the Newton-Raphson or similar sequence, is division by 0. To get the correct result in all cases, it is important that the destination register be the same for the first instruction in the quotient computation sequence (`frcpa`) and for the last `fma` before quotient truncation.

## 2. Floating-Point Divide Algorithms for the IA-64 Architecture

Six different algorithms are provided for scalar floating point divide operations: one latency optimized and one throughput optimized version for each IEEE format (single, double and double-extended precision). The algorithms were proven to be IEEE compliant (see [3],[4]) and in addition, the architecture specific Denormal flag is always correctly set to indicate a denormal input. The double extended precision algorithms will also yield IEEE compliant results and correctly set IEEE flags for 82-bit floating-point register format, when the floating-point status register (FPSR) is set for rounding to a 17-bit exponent.

For parallel (SIMD) floating-point divide, one IEEE compliant algorithm that also sets the Denormal flag correctly is provided. A slightly faster version, that preserves IEEE compliance but does not provide a correct Denormal flag is also available.

### 2.1. Single Precision Floating-Point Divide, Latency Optimized

The following algorithm calculates  $q'_3 = a/b$  in single precision, where  $a$  and  $b$  are single precision numbers.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $q_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(3) $e_0 = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(4) $q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(5) $e_1 = (e_0 \cdot e_0)_{rn}$	82-bit register format precision
(6) $q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(7) $e_2 = (e_1 \cdot e_1)_{rn}$	82-bit register format precision
(8) $q_3 = (q_2 + e_2 \cdot q_2)_{rn}$	17-bit exponent, 53-bit mantissa
(9) $q'_3 = (q_3)_{rnd}$	single precision

The assembly language implementation:

```
.file "sgl_div_min_lat.s"
.section .text
.proc sgl_div_min_lat#
.align 32
.global sgl_div_min_lat#
.align 32

sgl_div_min_lat:
{ .mmi
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34

  // &a is in r32
  // &b is in r33
  // &div is in r34 (the address of the divide result)

  // general registers used: r31, r32, r33, r34
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8

  nop.m 0
  nop.i 0;;
} { .mmi
  // load a, the first argument, in f6
  ldfs f6 = [r32]
```



```

// load b, the second argument, in f7
ldfs f7 = [r33]
nop.i 0;;
} { .mfi

// BEGIN SINGLE PRECISION MINIMUM LATENCY DIVIDE ALGORITHM

nop.m 0
// Step (1)
// y0 = 1 / b in f8
frcpa.s0 f8,p6=f6,f7
nop.i 0;;
} { .mfi
nop.m 0
// Step (2)
// q0 = a * y0 in f6
(p6) fma.sl f6=f6,f8,f0
nop.i 0
} { .mfi
nop.m 0
// Step (3)
// e0 = 1 - b * y0 in f7
(p6) fnma.sl f7=f7,f8,f1
nop.i 0;;
} { .mfi
nop.m 0
// Step (4)
// q1 = q0 + e0 * q0 in f6
(p6) fma.sl f6=f7,f6,f6
nop.i 0
} { .mfi
nop.m 0
// Step (5)
// e1 = e0 * e0 in f7
(p6) fma.sl f7=f7,f7,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (6)
// q2 = q1 + e1 * q1 in f6
(p6) fma.sl f6=f7,f6,f6
nop.i 0
} { .mfi
nop.m 0
// Step (7)
// e2 = e1 * e1 in f7
(p6) fma.sl f7=f7,f7,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// q3 = q2 + e2 * q2 in f6
(p6) fma.d.sl f6=f7,f6,f6
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// q3' = q3 in f8
(p6) fma.s.s0 f8=f6,f1,f0
nop.i 0;;

// END SINGLE PRECISION MINIMUM LATENCY DIVIDE ALGORITHM

} { .mmb
nop.m 0
// store result
stfs [r34]=f8
// return
br.ret.sptk b0;;
}

```

```
.endp sgl_div_min_lat
```

Sample test driver:

```
#include<stdio.h>

void sgl_div_min_lat(float*,float*,float*);

void run_test(unsigned long ia,unsigned long ib,unsigned long iq)
{
    float a,b,q;

    *(unsigned long*)(&a)=ia;
    *(unsigned long*)(&b)=ib;

    sgl_div_min_lat(&a,&b,&q);

    printf("\nNumerator: %lx\nDenominator: %lx\nQuotient: %lx\n",ia,ib,iq);
    if(iq==(unsigned long*)(&q)) printf("Passed\n");
    else printf("Failed\n");
}

void main()
{
    /* 1/1=1 */
    run_test(0x3f800000,0x3f800000,0x3f800000);

    /* 1/0=Infinity */
    run_test(0x3f800000,0x00000000,0x7f800000);

    /* -1/Infinity=-Zero */
    run_test(0xbf800000,0x7f800000,0x80000000);
}
```

## 2.2. Single Precision Floating-Point Divide, Throughput Optimized

This throughput optimized algorithm calculates  $q = a/b$  in single precision, where  $a$  and  $b$  are single precision numbers.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |                                  |
|--|----------------------------------|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup                     |
| (2) $d = (1 - b \cdot y_0)_{rn}$                                       | 82-bit register format precision |
| (3) $e = (d + d \cdot d)_{rn}$   | 82-bit register format precision |
| (4) $y_1 = (y_0 + e \cdot y_0)_{rn}$                                   | 82-bit register format precision |
| (5) $q_1 = (a \cdot y_1)_{rn}$   | 17-bit exponent, 24-bit mantissa |
| (6) $r = (a - b \cdot q_1)_{rn}$                                       | 82-bit register format precision |
| (7) $q = (q_1 + r \cdot y_1)_{rnd}$                                    | single precision                 |

The assembly language implementation:

```

.file "sgl_div_max_thr.s"
.section .text
.proc sgl_div_max_thr#
.align 32
.global sgl_div_max_thr#
.align 32

sgl_div_max_thr:
{ .mmi
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34

  // &a is in r32
  // &b is in r33
  // &div is in r34 (the address of the divide result)

  // general registers used: r31, r32, r33, r34
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9

  nop.m 0
  nop.i 0;;
} { .mmi
  // load a, the first argument, in f6
  ldfs f6 = [r32]
  // load b, the second argument, in f7
  ldfs f7 = [r33]
  nop.i 0;;
} { .mfi

  // BEGIN SINGLE PRECISION MAXIMUM THROUGHPUT DIVIDE ALGORITHM

  nop.m 0
  // Step (1)
  // y0 = 1 / b in f8
  frcpa.s0 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  // d = 1 - b * y0 in f9
  (p6) fnma.s1 f9=f7,f8,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3)
  // e = d + d * d in f9
  (p6) fma.s1 f9=f9,f9,f9
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  // y1 = y0 + e * y0 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  // q1 = a * y1 in f9
  (p6) fma.s.s1 f9=f6,f8,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6)
  // r = a - b * q1 in f6
  (p6) fnma.s1 f6=f7,f9,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  // q = q1 + r * y1 in f8

```

```

(p6) fma.s.s0 f8=f6,f8,f9
nop.i 0;;

// END SINGLE PRECISION MAXIMUM THROUGHPUT DIVIDE ALGORITHM
} { .mmb
  nop.m 0
  // store result
  stfs [r34]=f8
  // return
  br.ret.sptk b0;;
}

.endp sgl_div_max_thr

```

### Sample test driver:

```

#include<stdio.h>

void sgl_div_max_thr (float*,float*,float*);

void run_test(unsigned long ia,unsigned long ib,unsigned long iq)
{
  float a,b,q;

  *(unsigned long*)&a=ia;
  *(unsigned long*)&b=ib;

  sgl_div_max_thr(&a,&b,&q);

  printf("\nNumerator: %lx\nDenominator: %lx\nQuotient: %lx\n",ia,ib,iq);
  if(iq==*(unsigned long*)&q) printf("Passed\n");
  else printf("Failed\n");
}

void main()
{
  /* 1/1=1 */
  run_test(0x3f800000,0x3f800000,0x3f800000);

  /* 1/0=Infinity */
  run_test(0x3f800000,0x00000000,0x7f800000);

  /* -1/Infinity=-Zero */
  run_test(0xbf800000,0x7f800000,0x80000000);
}

```

## 2.3. Double Precision Floating-Point Divide, Latency Optimized

The quotient  $q_4 = a/b$  is calculated in double precision, where  $a$  and  $b$  are double precision numbers.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |                                  |
|--|----------------------------------|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup                     |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision |
| (3) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision |

(4) $q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(5) $e_1 = (e_0 \cdot e_0)_{rn}$	82-bit register format precision
(6) $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit register format precision
(7) $q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(8) $e_2 = (e_1 \cdot e_1)_{rn}$	82-bit register format precision
(9) $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	82-bit register format precision
(10) $q_3 = (q_2 + e_2 \cdot q_2)_{rn}$	17-bit exponent, 53-bit mantissa
(11) $y_3 = (y_2 + e_2 \cdot y_2)_{rn}$	82-bit register format precision
(12) $r_0 = (a - b \cdot q_3)_{rn}$	17-bit exponent, 53-bit mantissa
(13) $q_4 = (q_3 + r_0 \cdot y_3)_{rnd}$	double precision

The assembly language implementation:

```
.file "dbl_div_min_lat.s"
.section .text
.proc   dbl_div_min_lat#
.align 32
.global dbl_div_min_lat#
.align 32

dbl_div_min_lat:
{ .mmi
    alloc   r31=ar.pfs,3,0,0,0 // r32, r33, r34

    // &a is in r32
    // &b is in r33
    // &div is in r34 (the address of the divide result)

    // general registers used: r32, r33, r34
    // predicate registers used: p6
    // floating-point registers used: f6, f7, f8, f9, f10, f11

    // load a, the first argument, in f6
    ldld f6 = [r32]
    nop.i 0
} { .mmi
    // load b, the second argument, in f7
    ldld f7 = [r33]
    nop.m 0
    nop.i 0;;
} { .mfi

    // BEGIN DOUBLE PRECISION MINIMUM LATENCY DIVIDE ALGORITHM

    nop.m 0
    // Step (1)
    // y0 = 1 / b in f8
    frcpa.s0 f8,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // q0 = a * y0 in f9
    (p6) fma.sl f9=f6,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // e0 = 1 - b * y0 in f10
    (p6) fnma.sl f10=f7,f8,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
```

```

    // q1 = q0 + e0 * q0 in f9
    (p6) fma.s1 f9=f10,f9,f9
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // e1 = e0 * e0 in f11
    (p6) fma.s1 f11=f10,f10,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (6)
    // y1 = y0 + e0 * y0 in f8
    (p6) fma.s1 f8=f10,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // q2 = q1 + e1 * q1 in f9
    (p6) fma.s1 f9=f11,f9,f9
    nop.i 0
} { .mfi
    nop.m 0
    // Step (8)
    // e2 = e1 * e1 in f10
    (p6) fma.s1 f10=f11,f11,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (9)
    // y2 = y1 + e1 * y1 in f8
    (p6) fma.s1 f8=f11,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // q3 = q2 + e2 * q2 in f9
    (p6) fma.d.s1 f9=f10,f9,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // y3 = y2 + e2 * y2 in f8
    (p6) fma.s1 f8=f10,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (12)
    // r0 = a - b * q3 in f6
    (p6) fnma.d.s1 f6=f7,f9,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (13)
    // q4 = q3 + r0 * y3 in f8
    (p6) fma.d.s0 f8=f6,f8,f9
    nop.i 0;;

    // END DOUBLE PRECISION MINIMUM LATENCY DIVIDE ALGORITHM
} { .mmi
    nop.m 0;;
    // store result
    stfd [r34]=f8
    nop.i 0
} { .mib
    nop.m 0
    nop.i 0
    // return
    br.ret.sptk b0;;
}

```

```
.endp dbl_div_min_lat
```

Sample test driver:

```
#include<stdio.h>

typedef struct
{
    unsigned long W[2];
} _FP64;

void dbl_div_min_lat(_FP64*,_FP64*,_FP64*);

void run_test(unsigned long ial,unsigned long ia0,unsigned long ibl,unsigned long
ib0,unsigned long iq1,unsigned long iq0)
{
    _FP64 a,b,q;

    a.W[0]=ia0; a.W[1]=ial;
    b.W[0]=ib0; b.W[1]=ibl;

    dbl_div_min_lat(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx\nDenominator: %08lx%08lx\nQuotient:
%08lx%08lx\n",ial,ia0,ibl,ib0,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* 1/1=1 */
    run_test(0x3ff00000,0x00000000,0x3ff00000,0x00000000,0x3ff00000,0x00000000);

    /* 1/0=Infinity */
    run_test(0x3ff00000,0x00000000,0x00000000,0x00000000,0x7ff00000,0x00000000);

    /* -1/Infinity=-Zero */
    run_test(0xbff00000,0x00000000,0x7ff00000,0x00000000,0x80000000,0x00000000);
}
```

## 2.4. Double Precision Floating-Point Divide, Throughput Optimized

The quotient  $q_1 = a/b$  is calculated in double precision, where  $a$  and  $b$  are double precision numbers.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |                                  |
|--|----------------------------------|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup                     |
| (2) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision |
| (3) $y_1 = (y_0 + e_0 \cdot y_0)_m$                                    | 82-bit register format precision |
| (4) $e_1 = (e_0 \cdot e_0)_m$  | 82-bit register format precision |
| (5) $y_2 = (y_1 + e_1 \cdot y_1)_m$                                    | 82-bit register format precision |
| (6) $e_2 = (e_1 \cdot e_1)_m$  | 82-bit register format precision |
| (7) $y_3 = (y_2 + e_2 \cdot y_2)_m$                                    | 82-bit register format precision |
| (8) $q_0 = (a \cdot y_3)_m$  | 17-bit exponent, 53-bit mantissa |

$$(9) \ r_0 = (a - b \cdot q_0)_m$$

$$(10) \ q_1 = (q_0 + r_0 \cdot y_3)_{rd}$$

17-bit exponent, 53-bit mantissa  
double precision

The assembly language implementation:

```
.file "dbl_div_max_thr.s"
.section .text
.proc   dbl_div_max_thr#
.align 32
.global dbl_div_max_thr#
.align 32

dbl_div_max_thr:
{ .mii
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34

  // &a is in r32
  // &b is in r33
  // &div is in r34 (the address of the divide result)

  // general registers used: r31, r32, r33, r34
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9

  nop.i 0
  nop.i 0;;
} { .mmi
  // load a, the first argument, in f6
  ldld f6 = [r32]
  // load b, the second argument, in f7
  ldld f7 = [r33]
  nop.i 0;;
} { .mfi

  // BEGIN DOUBLE PRECISION MAXIMUM THROUGHPUT DIVIDE ALGORITHM

  nop.m 0
  // Step (1)
  // y0 = 1 / b in f8
  frcpa.s0 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  // e0 = 1 - b * y0 in f9
  (p6) fnma.s1 f9=f7,f8,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (3)
  // y1 = y0 + e0 * y0 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (4)
  // e1 = e0 * e0 in f9
  (p6) fma.s1 f9=f9,f9,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  // y2 = y1 + e1 * y1 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
```



```

    // e2 = e1 * e1 in f9
    (p6) fma.s1 f9=f9,f9,f0
    nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  // y3 = y2 + e2 * y2 in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // q0 = a * y3 in f9
  (p6) fma.d.s1 f9=f6,f8,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // r0 = a - b * q0 in f6
  (p6) fnma.d.s1 f6=f7,f9,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // q1 = q0 + r0 * y3 in f8
  (p6) fma.d.s0 f8=f6,f8,f9
  nop.i 0;;

  // END DOUBLE PRECISION MAXIMUM THROUGHPUT DIVIDE ALGORITHM
} { .mmb
  // store result
  stfd [r34]=f8
  nop.m 0
  // return
  br.ret.sptk b0;;
}

.endp dbl_div_max_thr

```

### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[2];
} _FP64;

void dbl_div_max_thr(_FP64*,_FP64*,_FP64*);

void run_test(unsigned long ial,unsigned long ia0,unsigned long ib1,
              unsigned long  ib0,unsigned long iql,unsigned long iq0)
{
    _FP64 a,b,q;

    a.W[0]=ia0; a.W[1]=ial;
    b.W[0]=ib0; b.W[1]=ib1;

    dbl_div_max_thr(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx\nDenominator: %08lx%08lx\nQuotient: %08lx%08lx\n",ial,ia0,ib1,ib0,iql,iq0);
    if(iq0==q.W[0] && iql==q.W[1]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

```

```

void main()
{
    /* 1/1=1 */
    run_test(0x3fff00000,0x00000000,0x3fff00000,0x00000000,0x3fff00000,0x00000000);

    /* 1/0=Infinity */
    run_test(0x3fff00000,0x00000000,0x00000000,0x00000000,0x7fff00000,0x00000000);

    /* -1/Infinity=-Zero */
    run_test(0xbfff00000,0x00000000,0x7fff00000,0x00000000,0x80000000,0x00000000);
}

```

## 2.5. Double Extended Precision Floating-Point Divide, Latency Optimized

The quotient  $q = a/b$  is calculated in double extended precision, where  $a$  and  $b$  are double extended precision numbers.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $d = (1 - b \cdot y_0)_m$	82-bit register format precision
(3) $q_0 = (a \cdot y_0)_m$	82-bit register format precision
(4) $d_2 = (d \cdot d)_m$	82-bit register format precision
(5) $d_3 = (d \cdot d + d)_m$	82-bit register format precision
(6) $d_5 = (d_2 \cdot d_2 + d)_m$	82-bit register format precision
(7) $y_1 = (y_0 + d_3 \cdot y_0)_m$	82-bit register format precision
(8) $y_2 = (y_0 + d_5 \cdot y_1)_m$	82-bit register format precision
(9) $r_0 = (a - b \cdot q_0)_m$	82-bit register format precision
(10) $q_1 = (q_0 + r_0 \cdot y_2)_m$	82-bit register format precision
(11) $e = (1 - b \cdot y_2)_m$	82-bit register format precision
(12) $y_3 = (y_2 + e \cdot y_2)_m$	82-bit register format precision
(13) $r = (a - b \cdot q_1)_m$	82-bit register format precision
(14) $q = (q_1 + r \cdot y_3)_{rnd}$	double extended precision

The assembly language implementation:

```

.file "dbl_ext_div_min_lat.s"
.section .text
.proc dbf_ext_div_min_lat#
.align 32
.global dbf_ext_div_min_lat#
.align 32

dbf_ext_div_min_lat:
{ .mmi
    alloc    r31=ar.pfs,3,0,0,0 // r32, r33, r34

    // &a is in r32
    // &b is in r33
    // &div is in r34 (the address of the divide result)

    // general registers used: r32, r33, r34
    // predicate registers used: p6
    // floating-point registers used: f6, f7, f8, f9, f10, f11, f12

    nop.m 0

```

```

    nop.i 0;;
} { .mmi
    // load a, the first argument, in f6
    ldfe f6 = [r32]
    // load b, the second argument, in f7
    ldfe f7 = [r33]
    nop.i 0;;
} { .mfi

    // BEGIN DOUBLE-EXTENDED PRECISION MINIMUM LATENCY DIVIDE ALGORITHM

    nop.m 0
    // Step (1)
    // y0 = 1 / b in f8
    frcpa.s0 f8,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // d = 1 - b * y0 in f9
    (p6) fnma.s1 f9=f7,f8,f1
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // q0 = a * y0 in f10
    (p6) fma.s1 f10=f6,f8,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // d2 = d * d in f11
    (p6) fma.s1 f11=f9,f9,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // d3 = d * d + d in f12
    (p6) fma.s1 f12=f9,f9,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (6)
    // d5 = d2 * d2 + d in f9
    (p6) fma.s1 f9=f11,f11,f9
    nop.i 0
} { .mfi
    nop.m 0
    // Step (7)
    // y1 = y0 + d3 * y0 in f11
    (p6) fma.s1 f11=f12,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // y2 = y0 + d5 * y1 in f8
    (p6) fma.s1 f8=f11,f9,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (9)
    // r0 = a - b * q0 in f9
    (p6) fnma.s1 f9=f7,f10,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // q1 = q0 + r0 * y2 in f9
    (p6) fma.s1 f9=f9,f8,f10
    nop.i 0
} { .mfi

```

```

    nop.m 0
    // Step (11)
    // e = 1 - b * y2 in f10
    (p6) fnma.s1 f10=f7,f8,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (12)
    // y3 = y2 + e * y2 in f8
    (p6) fma.s1 f8=f10,f8,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (13)
    // r = a - b * q1 in f10
    (p6) fnma.s1 f10=f7,f9,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (14)
    // q = q1 + r * y3 in f8
    (p6) fma.s0 f8=f10,f8,f9
    nop.i 0;;

    // END DOUBLE-EXTENDED PRECISION MINIMUM LATENCY DIVIDE ALGORITHM

} { .mmb
    // store result
    stfe [r34]=f8
    nop.m 0
    // return
    br.ret.sptk b0;;
}

.endp dbl_ext_div_min_lat

```

### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void dbl_ext_div_min_lat(_FP128*,_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long ib3,unsigned long ib2,unsigned long ib1,unsigned long ib0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    b.W[0]=ib0; b.W[1]=ib1; b.W[2]=ib2; b.W[3]=ib3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    dbl_ext_div_min_lat(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx%08lx\nDenominator: %08lx%08lx%08lx\nQuotient: %08lx%08lx%08lx\n",ia2,ia1,ia0,ib2,ib1,ib0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()

```

```

{
    /* 1/1=1 */

    run_test(0,0x3fff,0x80000000,0x00000000,0,0x3fff,0x80000000,0x00000000,0,0x3fff,0x8000
    0000,0x00000000);

    /* 1/0=Infinity */

    run_test(0,0x3fff,0x80000000,0x00000000,0,0,0x00000000,0x00000000,0,0x7fff,0x80000000,
    0x00000000);

    /* -1/Infinity=-Zero */

    run_test(0,0xbfff,0x80000000,0x00000000,0,0x7fff,0x80000000,0x00000000,0,0x8000,0x0000
    0000,0x00000000);
}

```

## 2.6. Double Extended Precision Floating-Point Divide, Throughput Optimized

The quotient  $q_2 = a/b$  is calculated in double extended precision, where  $a$  and  $b$  are double extended precision numbers.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $e_0 = (1 - b \cdot y_0)_m$	82-bit register format precision
(3) $y_1 = (y_0 + e_0 \cdot y_0)_m$	82-bit register format precision
(4) $e_1 = (e_0 \cdot e_0)_m$	82-bit register format precision
(5) $y_2 = (y_1 + e_1 \cdot y_1)_m$	82-bit register format precision
(6) $q_0 = (a \cdot y_0)_m$	82-bit register format precision
(7) $e_2 = (1 - b \cdot y_2)_m$	82-bit register format precision
(8) $y_3 = (y_2 + e_2 \cdot y_2)_m$	82-bit register format precision
(9) $r_0 = (a - b \cdot q_0)_m$	82-bit register format precision
(10) $q_1 = (q_0 + r_0 \cdot y_3)_m$	82-bit register format precision
(11) $e_3 = (1 - b \cdot y_3)_m$	82-bit register format precision
(12) $y_4 = (y_3 + e_3 \cdot y_3)_m$	82-bit register format precision
(13) $r_1 = (a - b \cdot q_1)_m$	82-bit register format precision
(14) $q_2 = (q_1 + r_1 \cdot y_4)$	double extended precision

The assembly language implementation:

```

.file "dbl_ext_div_max_thr.s"
.section .text
.proc dbf_ext_div_max_thr#
.align 32
.global dbf_ext_div_max_thr#
.align 32

dbf_ext_div_max_thr:
{ .mmi
    alloc    r31=ar.pfs,3,0,0,0 // r32, r33, r34

    // &a is in r32
    // &b is in r33
    // &div is in r34 (the address of the divide result)

```

```

// general registers used: r32, r33, r34
// predicate registers used: p6
// floating-point registers used: f6, f7, f8, f9, f10

nop.m 0
nop.i 0;;
} { .mmi
// load a, the first argument, in f6
ldfe f6 = [r32]
// load b, the second argument, in f7
ldfe f7 = [r33]
nop.i 0;;
} { .mfi

// BEGIN DOUBLE EXTENDED PRECISION MAX. THROUGHPUT DIVIDE ALGORITHM

nop.m 0
// Step (1)
//  $y_0 = 1 / b$  in f8
frcpa.s0 f8,p6=f6,f7
nop.i 0;;
} { .mfi
nop.m 0
// Step (2)
//  $e_0 = 1 - b * y_0$  in f9
(p6) fnma.s1 f9=f7,f8,f1
nop.i 0;;
} { .mfi
nop.m 0
// Step (3)
//  $y_1 = y_0 + e_0 * y_0$  in f10
(p6) fma.s1 f10=f9,f8,f8
nop.i 0
} { .mfi
nop.m 0
// Step (4)
//  $e_1 = e_0 * e_0$  in f9
(p6) fma.s1 f9=f9,f9,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (5)
//  $y_2 = y_1 + e_1 * y_1$  in f9
(p6) fma.s1 f9=f9,f10,f10
nop.i 0;;
} { .mfi
nop.m 0
// Step (6)
//  $q_0 = a * y_0$  in f10
(p6) fma.s1 f10=f6,f8,f0
nop.i 0
} { .mfi
nop.m 0
// Step (7)
//  $e_2 = 1 - b * y_2$  in f8
(p6) fnma.s1 f8=f7,f9,f1
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
//  $y_3 = y_2 + e_2 * y_2$  in f8
(p6) fma.s1 f8=f8,f9,f9
nop.i 0
} { .mfi
nop.m 0
// Step (9)
//  $r_0 = a - b * q_0$  in f9
(p6) fnma.s1 f9=f7,f10,f6
nop.i 0;;
} { .mfi

```

```

    nop.m 0
    // Step (10)
    // q1 = q0 + r0 * y3 in f9
    (p6) fma.s1 f9=f9,f8,f10
    nop.i 0
} { .mfi
    nop.m 0
    // Step (11)
    // e3 = 1 - b * y3 in f10
    (p6) fnma.s1 f10=f7,f8,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (12)
    // y4 = y3 + e3 * y3 in f8
    (p6) fma.s1 f8=f10,f8,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (13)
    // r1 = a - b * q1 in f10
    (p6) fnma.s1 f10=f7,f9,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (14)
    // q2 = q1 + r1 * y4 in f8
    (p6) fma.s0 f8=f10,f8,f9
    nop.i 0;;

    // END DOUBLE-EXTENDED PRECISION MAX. THROUGHPUT DIVIDE ALGORITHM
} { .mmb
    // store result
    stfe [r34]=f8
    nop.m 0
    // return
    br.ret.sptk b0;;
}

.endp dbl_ext_div_max_thr

```

### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void dbl_ext_div_max_thr(_FP128*,_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long ib3,unsigned long ib2,unsigned long ib1,unsigned long ib0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    b.W[0]=ib0; b.W[1]=ib1; b.W[2]=ib2; b.W[3]=ib3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    dbl_ext_div_max_thr(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx%08lx\nDenominator: %08lx%08lx%08lx\nQuotient: %08lx%08lx%08lx\n", ia2,ia1,ia0,ib2,ib1,ib0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
}

```

```

    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* 1/1=1 */

    run_test(0,0x3fff,0x80000000,0x00000000,0,0x3fff,0x80000000,0x00000000,0,0x3fff,0x8000
0000,0x00000000);

    /* 1/0=Infinity */

    run_test(0,0x3fff,0x80000000,0x00000000,0,0,0x00000000,0x00000000,0,0x7fff,0x80000000,
0x00000000);

    /* -1/Infinity=-Zero */

    run_test(0,0xbfff,0x80000000,0x00000000,0,0x7fff,0x80000000,0x00000000,0,0x8000,0x0000
0000,0x00000000);
}

```

## 2.7. Parallel Single Precision (SIMD) Floating-Point Divide, Version 1

The single precision quotients of two packed pairs of single precision numbers,  $a$  (numerators) and  $b$  (denominators) are computed in parallel, when both pairs satisfy the conditions that ensure the correctness of the iterations used. Otherwise, the arguments are unpacked and the two quotients are computed separately, then returned packed together. The parallel single precision divide algorithm only is described below. The assembly code provided also unpacks the inputs, computes the quotients separately and packs them together, when necessary.

$rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All of the symbols used are packed single precision numbers. Each parallel step is performed in single precision.

To ensure that the Denormal flag is correctly set, scaling coefficients ( $pscale$ ,  $nscale$ ) are computed in parallel with the steps shown, and applied before the last multiply-add, which sets the flags in the FPSR status field.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $d = (1 - b \cdot y_0)_{rn}$                                       | single precision  |
| (3) $q_0 = (a \cdot y_0)_{rn}$   | single precision  |
| (4) $y_1 = (y_0 + d \cdot y_0)_{rn}$                                   | single precision  |
| (5) $r_0 = (a - b \cdot q_0)_{rn}$                                     | single precision  |
| (6) $e = (1 - b \cdot y_1)_{rn}$                                       | single precision  |
| (7) $y_2 = (y_0 + d \cdot y_1)_{rn}$                                   | single precision  |
| (8) $q_1 = (q_0 + r_0 \cdot y_1)_{rn}$                                 | single precision  |
| (9) $y_3 = (y_1 + e \cdot y_2)_{rn}$                                   | single precision  |
| (10) $a_{ps} = (a \cdot pscale)_{rn}$                                  | single precision (optional step)  |
| (11) $b_{ps} = (b \cdot pscale)_{rn}$                                  | single precision (optional step)  |
| (12) $y_{3ns} = (y_3 \cdot nscale)_{rn}$                               | single precision (optional step)  |
| (13) $r_{1ps} = (a_{ps} - b_{ps} \cdot q_1)_{rn}$                      | single precision ( $a$ , $b$ , $y_3$ are optionally scaled to ensure a correct Denormal flag setting) |
| (14) $q = (q_1 + r_{1ps} \cdot y_{3ns})_{rnd}$                         | single precision  |



The assembly language implementation:

```
.file "simd_div_sc.s"
.section .text
.proc  simd_div_sc #
.align 32
.global simd_div_sc #
.align 32

simd_div_sc:

    // &a is in r32
    // &b is in r33
    // &div is in r34 (the address of the divide result)

    // general registers used: r2, r3,r8,r9,r10,r11,r14,r15,r16, r31, r32, r33, r34
    // predicate registers used: p6, p7, p8
    // floating-point registers used: f6 to f33

    // expects the user status field sf0 in the FPSR to have wre = 0

{ .mmi
  alloc  r31=ar.pfs,3,0,0,0  // r32, r33, r34
  nop.m 0
  nop.i 0;;
} { .mmb
  // load a, the first argument, in f6
  ldf.fill f6 = [r32]
  // load b, the second argument, in f7
  ldf.fill f7 = [r33]
  nop.b 0
}
// BEGIN SCALED SIMD DIVIDE ALGORITHM
{ .mlx
  nop.m 0
  // +1.0, +1.0 in r2
  movl r2 = 0x3f8000003f800000;;
} { .mlx
  nop.m 0
  // get negative scale factor (1,2^{-24})
  movl r14=0x3f80000033800000;;
} { .mlx
  // set 8-bit exponent mask
  mov r16=0xff
  // get positive scale factor (1,2^{24})
  movl r15=0x3f8000004b800000;;
} { .mii
  // +1.0, +1.0 in f9
  setf.sig f9=r2
  // assume negative scale factor (1,1)
  mov r10=r2
  nop.i 0;;
} { .mmi
  nop.m 0;;
  // extract in advance a_high, a_low from f6 into r2
  getf.sig r2 = f6
  nop.i 0
} { .mfi
  // extract in advance b_high, b_low from f7 into r3
  getf.sig r3 = f7
  // Step (1)
  // y0 = 1 / b in f8
  fprcpa.s1 f8,p6=f6,f7
  nop.i 0;;
} { .mmi
  // unpack in advance a_low in f14
  setf.s f14 = r2
  nop.m 0
```

```

    // shift exponent of a_low to rightmost bits
    shr.u r8=r2,23;;
} { .mmi
    nop.m 0
    // mask to get exponent of a_low
    and r8=r8,r16
    // shift exponent of a_high to rightmost bits
    shr.u r9=r2,55;;
} { .mmi
    nop.m 0
    // mask to get exponent of a_high
    and r9=r16,r9
    // set p7 in a_low needs scaling, else set p8
    cmp.gt.unc p7,p8=0x30,r8;;
} { .mfi
    // Set p9 if a_high needs scaling and a_low needs scaling
    // Set p10 if a_high doesn't need scaling and a_low needs scaling
    (p7) cmp.gt.unc p9,p10=0x30,r9
    // Step (2)
    // d = 1 - b * y0 in f10
    (p6) fpmma.s1 f10=f7,f8,f9
    // Set p11 if a_high needs scaling and a_low doesn't need scaling
    // Set p12 if a_high doesn't need scaling and a_low doesn't need scaling
    (p8) cmp.gt.unc p11,p12=0x30,r9
} { .mfi
    // unpack in advance b_low in f15
    setf.s f15 = r3
    // Step (3)
    // q0 = a * y0 in f11
    (p6) fpma.s1 f11=f6,f8,f0
    // assume positive scale factor (1,1)
    mov r11=r10;;
} { .mlx
    nop.m 0
    // get negative scale factor (2^{-24}, 2^{-24})
    (p9) movl r10=0x3380000033800000
} { .mlx
    nop.m 0
    // get positive scale factor (2^{24}, 2^{24})
    (p9) movl r11=0x4b8000004b800000;;
}
.pred.rel "mutex",p10,p11
{ .mlx
    // get negative scale factor (1, 2^{-24})
    (p10) mov r10=r14
    // get negative scale factor (2^{-24}, 1)
    (p11) movl r10=0x338000003f800000
} { .mlx
    // get positive scale factor (1, 2^{24})
    (p10) mov r11=r15
    // get positive scale factor (2^{24}, 1)
    (p11) movl r11=0x4b8000003f800000;;
} { .mfi
    nop.m 0
    // Step (4)
    // y1 = y0 + d * y0 in f12
    (p6) fpma.s1 f12=f10,f8,f8
    // shift right a_high in r2 (in advance)
    shr.u r2 = r2, 0x20
} { .mfi
    nop.m 0
    // Step (5)
    // r0 = a - b * q0 in f13
    (p6) fpmma.s1 f13=f7,f11,f6
    nop.i 0;;
} { .mmi
    // set negative factor, nscale
    setf.sig f32=r10
    // set positive factor, pscale
    setf.sig f33=r11
    // shift right b_high in r3 (in advance)

```

```

    shr.u r3 = r3, 0x20;;
} { .mfi
  nop.m 0
  // Step (6)
  // e = 1 - b * y1 in f9
  (p6) fpmma.s1 f9=f7,f12,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (7)
  // y2 = y0 + d * y1 in f10
  (p6) fpma.s1 f10=f10,f12,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // q1 = q0 + r0 * y1 in f8
  (p6) fpma.s1 f8=f13,f12,f11
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // y3 = y1 + e * y2 in f9
  (p6) fpma.s1 f9=f9,f10,f12
  nop.i 0;;
} { .mfi
  nop.m 0
  // scaling step S2: a_ps=a*pscale
  (p6) fpma.s1 f6=f6,f33,f0
  nop.i 0
} { .mfi
  nop.m 0
  // scaling step S3: b_ps=b*pscale
  (p6) fpma.s1 f7=f7,f33,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // scaling step S1: y3_ns=y3*nscale
  (p6) fpma.s1 f9=f9,f32,f0
  nop.i 0
} { .mfi
  // unpack in advance a_high in f32
  setf.s f32 = r2
  // Step (10)
  // r1_ps = a_ps - b_ps * q1 in f10
  (p6) fpmma.s1 f10=f7,f8,f6
  nop.i 0;;
} { .mfb
  // unpack in advance b_high in f33
  setf.s f33 = r3
  // Step (11)
  // q = q1 + r1_ps * y3_ns in f8
  (p6) fpma.s0 f8=f10,f9,f8
  // jump over the unpacked computation if (p6) was 1
  (p6) br.cond.dptk done
}

// Apply scalar single precision division for the low and high parts

{ .mfi
  nop.m 0
  // Step (1)
  // y0 = 1 / b in f6
  frcpa.s0 f6,p7=f14,f15
  nop.i 0;;
} { .mfi
  nop.m 0
  // normalize a_low in f14
  (p7) fnorm.s1 f14 = f14
  nop.i 0;;
} { .mfi

```

```

    nop.m 0
    // normalize b_low in f15
    (p7) fnorm.s1 f15 = f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1 / b in f7
    frcpa.s0 f7,p8=f32,f33
    nop.i 0;;
} { .mfi
    nop.m 0
    // normalize in advance a_high in f32
    (p8) fnorm.s1 f32 = f32
    nop.i 0
} { .mfi
    nop.m 0
    // normalize in advance b_high in f33
    (p8) fnorm.s1 f33 = f33
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // q0 = a * y0 in f14
    (p7) fma.s1 f14=f14,f6,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // e0 = 1 - b * y0 in f15
    (p7) fnma.s1 f15=f15,f6,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // q0 = a * y0 in f32
    (p8) fma.s1 f32=f32,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // e0 = 1 - b * y0 in f33
    (p8) fnma.s1 f33=f33,f7,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // q1 = q0 + e0 * q0 in f14
    (p7) fma.s1 f14=f15,f14,f14
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // e1 = e0 * e0 in f15
    (p7) fma.s1 f15=f15,f15,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // q1 = q0 + e0 * q0 in f32
    (p8) fma.s1 f32=f33,f32,f32
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // e1 = e0 * e0 in f33
    (p8) fma.s1 f33=f33,f33,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (6)

```

```

    // q2 = q1 + e1 * q1 in f14
    (p7) fma.s1 f14=f15,f14,f14
    nop.i 0
} { .mfi
  nop.m 0
  // Step (7)
  // e2 = e1 * e1 in f15
  (p7) fma.s1 f15=f15,f15,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6)
  // q2 = q1 + e1 * q1 in f32
  (p8) fma.s1 f32=f33,f32,f32
  nop.i 0
} { .mfi
  nop.m 0
  // Step (7)
  // e2 = e1 * e1 in f33
  (p8) fma.s1 f33=f33,f33,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // q3 = q2 + e2 * q2 in f14
  (p7) fma.d.s1 f14=f15,f14,f14
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // q3 = q2 + e2 * q2 in f32
  (p8) fma.d.s1 f32=f33,f32,f32
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // q3' = q3 in f6
  (p7) fma.s.s0 f6=f14,f1,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // q3' = q3 in f7
  (p8) fma.s.s0 f7=f32,f1,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // pack res_low from f6 and res_high from f7 into f8
  fpack f8 = f7, f6
  nop.i 0;;
}

// END SCALED SIMD DIVIDE ALGORITHM

done:

{ .mib
  // store result
  stf.spill [r34]=f8
  nop.i 0
  // return
  br.ret.sptk b0;;
}

.endp simd_div_sc

```

Sample test driver:

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

```

typedef struct
{
    unsigned long W[4];
} _FP128;

void simd_div_sc(_FP128*,_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long ib3,unsigned long ib2,unsigned long ib1,unsigned long ib0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    b.W[0]=ib0; b.W[1]=ib1; b.W[2]=ib2; b.W[3]=ib3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    simd_div_sc(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx%08lx\nDenominator: %08lx%08lx%08lx\nQuotient:
    %08lx%08lx%08lx\n", ia2,ia1,ia0,ib2,ib1,ib0,iq2,iq1,iq0);

    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* 1/1=1, 1.5/1.5=1 */

    run_test(0,0x1003e,0x3f800000,0x3fc00000,0,0x1003e,0x3f800000,0x3fc00000,0,0x1003e,0x3f800000,0x3fc00000);

    /* 1/0=Infinity, 0/0=QNaN */

    run_test(0,0x1003e,0x3f800000,0x00000000,0,0x1003e,0x00000000,0x00000000,0,0x1003e,0x7f800000,0xffc00000);

    /* -1/Infinity=-Zero, 3/2=1.5 */

    run_test(0,0x1003e,0xbf800000,0x40400000,0,0x1003e,0x7f800000,0x40000000,0,0x1003e,0x80000000,0x3fc00000);
}

```

## 2.8. Parallel Single Precision (SIMD) Floating-Point Divide, Version 2

A slightly faster version of the SIMD algorithm shown above eliminates the scaling steps, and thus does not guarantee a correct setting of the Denormal flag at the last computation step. The parallel single precision divide algorithm only is described below. The assembly code provided also unpacks the inputs, computes the quotients separately and packs them together, when necessary.

$rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All of the symbols used are packed single precision numbers. Each parallel step is performed in single precision.

- |  |                  |
|--|------------------|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup     |
| (2) $d = (1 - b \cdot y_0)_{rn}$                                       | single precision |
| (3) $q_0 = (a \cdot y_0)_{rn}$   | single precision |

- |  |                  |
|--|------------------|
| (4) $y_1 = (y_0 + d \cdot y_0)_{rn}$   | single precision |
| (5) $r_0 = (a - b \cdot q_0)_{rn}$     | single precision |
| (6) $e = (1 - b \cdot y_1)_{rn}$       | single precision |
| (7) $y_2 = (y_0 + d \cdot y_1)_{rn}$   | single precision |
| (8) $q_1 = (q_0 + r_0 \cdot y_1)_{rn}$ | single precision |
| (9) $y_3 = (y_1 + e \cdot y_2)_{rn}$   | single precision |
| (10) $r_1 = (a - b \cdot q_1)_{rn}$    | single precision |
| (11) $q = (q_1 + r_1 \cdot y_3)_{md}$  | single precision |

The assembly language implementation:

```
.file "simd_div.s"
.section .text
.proc  simd_div #
.align 32
.global simd_div #
.align 32

simd_div:

    // &a is in r32
    // &b is in r33
    // &div is in r34 (the address of the divide result)

    // general registers used: r2, r3, r31, r32, r33, r34
    // predicate registers used: p6, p7, p8
    // floating-point registers used: f6 to f33

    // expects the user status field sf0 in the FPSR to have wre = 0

{ .mmi
  alloc r31=ar.pfs,3,0,0,0 // r32, r33, r34
  nop.m 0
  nop.i 0;;
} { .mmb
  // load a, the first argument, in f6
  ldf.fill f6 = [r32]
  // load b, the second argument, in f7
  ldf.fill f7 = [r33]
  nop.b 0
}
// BEGIN SIMD DIVIDE ALGORITHM
{ .mlx
  nop.m 0
  // +1.0, +1.0 in r2
  movl r2 = 0x3f8000003f800000;;
} { .mfi
  // +1.0, +1.0 in f9
  setf.sig f9=r2
  nop.f 0
  nop.i 0;;
} { .mmi
  nop.m 0;;
  // extract in advance a_high, a_low from f6 into r2
  getf.sig r2 = f6
  nop.i 0
} { .mfi
  // extract in advance b_high, b_low from f7 into r3
  getf.sig r3 = f7
  // Step (1)
  // y0 = 1 / b in f8
  fprcpa.s1 f8,p6=f6,f7
  nop.i 0;;
} { .mmi
```

```

// unpack in advance a_low in f14
setf.s f14 = r2
nop.m 0
nop.i 0;;
} { .mfi
nop.m 0
// Step (2)
// d = 1 - b * y0 in f10
(p6) fpmma.s1 f10=f7,f8,f9
nop.i 0
} { .mfi
// unpack in advance b_low in f15
setf.s f15 = r3
// Step (3)
// q0 = a * y0 in f11
(p6) fpma.s1 f11=f6,f8,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (4)
// y1 = y0 + d * y0 in f12
(p6) fpma.s1 f12=f10,f8,f8
// shift right a_high in r2 (in advance)
shr.u r2 = r2, 0x20
} { .mfi
nop.m 0
// Step (5)
// r0 = a - b * q0 in f13
(p6) fpmma.s1 f13=f7,f11,f6
// shift right b_high in r3 (in advance)
shr.u r3 = r3, 0x20;;
} { .mmi
// unpack in advance a_high in f32
setf.s f32 = r2
nop.m 0
nop.i 0;;
} { .mfi
nop.m 0
// Step (6)
// e = 1 - b * y1 in f9
(p6) fpmma.s1 f9=f7,f12,f9
nop.i 0
} { .mfi
// unpack in advance b_high in f33
setf.s f33 = r3
// Step (7)
// y2 = y0 + d * y1 in f10
(p6) fpma.s1 f10=f10,f12,f8
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// q1 = q0 + r0 * y1 in f8
(p6) fpma.s1 f8=f13,f12,f11
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// y3 = y1 + e * y2 in f9
(p6) fpma.s1 f9=f9,f10,f12
nop.i 0;;
} { .mfi
nop.m 0
// Step (10)
// r1 = a - b * q1 in f10
(p6) fpmma.s1 f10=f7,f8,f6
nop.i 0;;
} { .mfb
nop.m 0
// Step (11)
// q = q1 + r1 * y3 in f8

```



```

(p6) fpma.s0 f8=f10,f9,f8
// jump over the unpacked computation if (p6) was 1
(p6) br.cond.dptk done
}

// Apply scalar single precision division for the low and high parts

{ .mfi
  nop.m 0
  // Step (1)
  // y0 = 1 / b in f6
  frcpa.s0 f6,p7=f14,f15
  nop.i 0;;
} { .mfi
  nop.m 0
  // normalize a_low in f14
  (p7) fnorm.s1 f14 = f14
  nop.i 0;;
} { .mfi
  nop.m 0
  // normalize b_low in f15
  (p7) fnorm.s1 f15 = f15
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (1)
  // y0 = 1 / b in f7
  frcpa.s0 f7,p8=f32,f33
  nop.i 0;;
} { .mfi
  nop.m 0
  // normalize in advance a_high in f32
  (p8) fnorm.s1 f32 = f32
  nop.i 0
} { .mfi
  nop.m 0
  // normalize in advance b_high in f33
  (p8) fnorm.s1 f33 = f33
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  // q0 = a * y0 in f14
  (p7) fma.s1 f14=f14,f6,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  // e0 = 1 - b * y0 in f15
  (p7) fnma.s1 f15=f15,f6,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  // q0 = a * y0 in f32
  (p8) fma.s1 f32=f32,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  // e0 = 1 - b * y0 in f33
  (p8) fnma.s1 f33=f33,f7,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  // q1 = q0 + e0 * q0 in f14
  (p7) fma.s1 f14=f15,f14,f14
  nop.i 0
} { .mfi
  nop.m 0

```

```

// Step (5)
// e1 = e0 * e0 in f15
(p7) fma.s1 f15=f15,f15,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (4)
// q1 = q0 + e0 * q0 in f32
(p8) fma.s1 f32=f33,f32,f32
nop.i 0
} { .mfi
nop.m 0
// Step (5)
// e1 = e0 * e0 in f33
(p8) fma.s1 f33=f33,f33,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (6)
// q2 = q1 + e1 * q1 in f14
(p7) fma.s1 f14=f15,f14,f14
nop.i 0
} { .mfi
nop.m 0
// Step (7)
// e2 = e1 * e1 in f15
(p7) fma.s1 f15=f15,f15,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (6)
// q2 = q1 + e1 * q1 in f32
(p8) fma.s1 f32=f33,f32,f32
nop.i 0
} { .mfi
nop.m 0
// Step (7)
// e2 = e1 * e1 in f33
(p8) fma.s1 f33=f33,f33,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// q3 = q2 + e2 * q2 in f14
(p7) fma.d.s1 f14=f15,f14,f14
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// q3 = q2 + e2 * q2 in f32
(p8) fma.d.s1 f32=f33,f32,f32
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// q3' = q3 in f6
(p7) fma.s.s0 f6=f14,f1,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// q3' = q3 in f7
(p8) fma.s.s0 f7=f32,f1,f0
nop.i 0;;
} { .mfi
nop.m 0
// pack res_low from f6 and res_high from f7 into f8
fpack f8 = f7, f6
nop.i 0;;
}
// END SIMD DIVIDE ALGORITHM

```

```
done:
```

```
{ .mib
  // store result
  stf.spill [r34]=f8
  nop.i 0
  // return
  br.ret.sptk b0;;
}
```

```
.endp simd_div
```

Sample test driver:

```
#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void simd_div(_FP128*,_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long ib3,unsigned long ib2,unsigned long ib1,unsigned long ib0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    b.W[0]=ib0; b.W[1]=ib1; b.W[2]=ib2; b.W[3]=ib3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    simd_div(&a,&b,&q);

    printf("\nNumerator: %08lx%08lx%08lx\nDenominator: %08lx%08lx%08lx\nQuotient:
%08lx%08lx%08lx\n", ia2,ia1,ia0,ib2,ib1,ib0,iq2,iq1,iq0);

    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* 1/1=1, 1.5/1.5=1 */
    run_test(0,0x1003e,0x3f800000,0x3fc00000,0,0x1003e,0x3f800000,0x3fc00000,0,0x1003e,0x3f800000,0x3fc00000);

    /* 1/0=Infinity, 0/0=QNaN */
    run_test(0,0x1003e,0x3f800000,0x00000000,0,0x1003e,0x00000000,0x00000000,0,0x1003e,0x7f800000,0xffc00000);

    /* -1/Infinity=-Zero, 3/2=1.5 */
    run_test(0,0x1003e,0xbf800000,0x40400000,0,0x1003e,0x7f800000,0x40000000,0,0x1003e,0x80000000,0x3fc00000);
}
```

## 2.9. Software Assistance (SWA) Conditions for Floating Point Divide

**Property 1**

Let  $a$  and  $b$  be two floating-point numbers with  $N_{in}$ -bit significands, and  $M_{in}$ -bit exponents (limited exponent range), as described by the IEEE-754 Standard for Binary Floating-Point Arithmetic. Let  $N$  be the size of the significand and  $M$  the size of the exponent, in number of bits, corresponding to a computation step in the algorithms described above. The exact values of  $N_{in}$ ,  $M_{in}$ ,  $N$ , and  $M$  are specified explicitly or implicitly for each algorithm. For all the intermediate steps of scalar floating-point divide (single, double, double extended precision and 82-bit register file format)  $M = 17$ , and for SIMD divide,  $M = 8$ . The value of  $N$  is specified by the precision of the computation step.

Then,  $e_{\min, in} = -2^{M_{in}-1} + 2$ , and  $e_{\max, in} = 2^{M_{in}-1} - 1$  are the minimum and maximum values of the exponents allowed for floating point numbers on input, and  $e_{\min} = -2^{M-1} + 2$ , and  $e_{\max} = 2^{M-1} - 1$  are the minimum and maximum values of the exponents allowed for floating point numbers in a given computation step.

Let  $a = \mathbf{s}_a \cdot s_a \cdot 2^{e_a}$ , with  $\mathbf{s}_a = \pm 1$ ,  $1 \leq s_a < 2$ ,  $s_a$  representable using  $N_{in}$  bits, and  $e_a \in \mathbf{Z}$ ,  $e_{\min, in} - N_{in} + 1 \leq e_a \leq e_{\max, in}$ . In addition, if  $e_a < e_{\min}$  and  $k = e_{\min} - e_a$ , then  $(2^{N_{in}-1} \cdot s_a) \equiv 0 \pmod{2^k}$  (which allows for denormal values of  $a$ ).

Let  $b = \mathbf{s}_b \cdot s_b \cdot 2^{e_b}$ , with  $\mathbf{s}_b = \pm 1$ ,  $1 \leq s_b < 2$ ,  $s_b$  representable using  $N_{in}$  bits, and  $e_b \in \mathbf{Z}$ ,  $e_{\min, in} - N_{in} + 1 \leq e_b \leq e_{\max, in}$ . In addition, if  $e_b < e_{\min}$  and  $k = e_{\min} - e_b$ , then  $(2^{N_{in}-1} \cdot s_b) \equiv 0 \pmod{2^k}$  (which allows for denormal values of  $b$ ).

Assume that the exact value of  $\frac{a}{b}$  will yield through rounding a floating-point number that is neither tiny, nor huge:

$$2^{e_{\min, in}} \leq \left( \frac{a}{b} \right)_{rnd} \leq (2 - 2^{-N+1}) \cdot 2^{e_{\max, in}}$$

where  $rnd$  is one of the IEEE rounding modes.

An *fma* operation for floating-point numbers with  $N$ -bit significands and  $M$ -bit exponents is assumed available, that preserves the  $2 \cdot N$  bits of the product before the summation, and only incurs one rounding error.

Then, the following statements hold.

The algorithms for calculating  $\left( \frac{a}{b} \right)_{rnd}$  in any IEEE rounding mode  $rnd$ , in single or double precision as described above, do not cause in any step overflow, underflow, or loss of precision.

The SIMD algorithms for calculating  $\left( \frac{a}{b} \right)_{rnd}$  in any IEEE rounding mode  $rnd$ , as described above, do not cause in any step overflow, underflow, or loss of precision, if:

$$\left\{ \begin{array}{ll} (a) & e_b \geq e_{\min} - 1 \quad (y_i \text{ will not be huge}) \\ (b) & e_b \leq e_{\max} - 3 \quad (y_i \text{ will not be tiny}) \\ (c) & e_a - e_b \leq e_{\max} - 1 \quad (q_i \text{ will not be huge}) \\ (d) & e_a - e_b \geq e_{\min} + 2 \quad (q_i \text{ will not be tiny}) \\ (e) & e_a \geq e_{\min} + N \quad (r_i \text{ will not lose precision}) \end{array} \right.$$

The algorithm for calculating  $\left( \frac{a}{b} \right)_{rnd}$  in any IEEE rounding mode  $rnd$ , as described in the double extended precision floating point divide algorithm, does not cause in any step overflow, underflow, or loss of precision, if the input values are representable in double-extended precision format  $N_{in} = 64$ , and  $M_{in} = 15$ .

The algorithm for calculating  $\left(\frac{a}{b}\right)_{md}$  in any IEEE rounding mode  $md$ , as described in the double extended precision floating point divide algorithm, with input values in floating-point register format ( $N_{in} = 64$ , and  $M_{in} = 17$ ), does not cause in any step overflow, underflow, or loss of precision, if:

$$\left\{ \begin{array}{ll} (a') & e_b \geq e_{\min} - 1 & (y_i \text{ will not be huge}) \\ (b') & e_b \leq e_{\max} - 3 & (y_i \text{ will not be tiny}) \\ (c') & e_a - e_b \leq e_{\max} - 1 & (q_i \text{ will not be huge}) \\ (d') & e_a - e_b \geq e_{\min} + 2 & (q_i \text{ will not be tiny}) \\ (e') & e_a \geq e_{\min} + N & (r_i \text{ will not lose precision}) \end{array} \right.$$

Note:

The Itanium™ processor will ask for software assistance (SWA), or otherwise indicate by appropriately setting the output predicate, that it cannot provide the correctly rounded result of the divide operation, whenever any of the conditions in Property 1 is not satisfied, i.e. for the SIMD algorithms described, or for the double extended precision floating point divide algorithm, when the input values  $a$  and  $b$  are in floating-point register file format (82-bit floating-point numbers). The SWA conditions are:

$$\left\{ \begin{array}{ll} (a) & e_b \leq e_{\min} - 2 & (y_i \text{ might be huge}) \\ (b) & e_b \geq e_{\max} - 2 & (y_i \text{ might be tiny}) \\ (c) & e_a - e_b \geq e_{\max} & (q_i \text{ might be huge}) \\ (d) & e_a - e_b \leq e_{\min} + 1 & (q_i \text{ might be tiny}) \\ (e) & e_a \leq e_{\min} + N - 1 & (r_i \text{ might lose precision}) \end{array} \right.$$

where  $N_{in} = N = 24$  and  $M_{in} = M = 8$  in the case of SIMD floating-point divide, and  $N_{in} = N = 64$  and  $M_{in} = M = 17$  for the double extended precision divide algorithm with 82-bit inputs.

### 3. Floating-Point Square Root Algorithms for the IA-64 Architecture

Five different algorithms are provided for scalar square root operations: two for single precision (one that optimizes latency, one that optimizes throughput), two for double precision (one that optimizes latency, one that optimizes throughput), and one for double extended precision and 82-bit register file format precision. All of these algorithms are IEEE compliant (see [3],[4]), and the architecture specific Denormal flag is always correctly set for single, double, or double extended precision inputs.

An IEEE compliant parallel (SIMD) single precision square root algorithm that correctly sets the Denormal flag is also presented here.

#### 3.1. Single Precision Floating-Point Square Root, Latency Optimized

The following algorithm calculates  $S = \sqrt{a}$  in single precision, where  $a$  is a single precision number.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |   |                                  |
|---|----------------------------------|
| (1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$ | table lookup                     |
| (2) $H_0 = (0.5 \cdot y_0)_m$   | 82-bit register format precision |
| (3) $S_0 = (a \cdot y_0)_m$   | 82-bit register format precision |
| (4) $d = (0.5 - S_0 \cdot H_0)_m$   | 82-bit register format precision |
| (5) $e = (1 + 1.5 \cdot d)_m$   | 82-bit register format precision |
| (6) $T_0 = (d \cdot S_0)_m$   | 82-bit register format precision |
| (7) $G_0 = (d \cdot H_0)_m$   | 82-bit register format precision |
| (8) $S_1 = (S_0 + e \cdot T_0)_m$   | 17-bit exponent, 24-bit mantissa |
| (9) $H_1 = (H_0 + e \cdot G_0)_m$   | 82-bit register format precision |
| (10) $d_1 = (a - S_1 \cdot S_1)_m$  | 82-bit register format precision |
| (11) $S = (S_1 + d_1 \cdot H_1)_{rnd}$  | single precision                 |

The assembly language implementation:

```
.file "sgl_sqrt_min_lat.s"
.section .text
.proc sgl_sqrt_min_lat#
.align 32
.global sgl_sqrt_min_lat#
.align 32

sgl_sqrt_min_lat:
{ .mmb
  alloc r31=ar.pfs,2,0,0,0 // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // general registers used: r2, r31, r32, r33
  // predicate registers used: p6
  // floating-point registers used: f6, f8, f7, f9, f10, f11

  // load the argument a in f6
  ldfs f6 = [r32]
  nop.b 0
```

```

} { .mlx

// BEGIN SINGLE PRECISION MINIMUM LATENCY SQUARE ROOT ALGORITHM

nop.m 0
// exponent of +1/2 in r2
movl r2 = 0x0fffe;;
} { .mfi
// +1/2 in f8
setf.exp f8 = r2
nop.f 0
nop.i 0;;
} { .mfi
nop.m 0
// Step (1)
// y0 = 1/sqrt(a) in f7
frsqrrta.s0 f7,p6=f6
nop.i 0;;
} { .mfi
nop.m 0
// Step (2)
// H0 = 1/2 * y0 in f9
(p6) fma.s1 f9=f8,f7,f0
nop.i 0
} { .mfi
nop.m 0
// Step (3)
// S0 = a * y0 in f7
(p6) fma.s1 f7=f6,f7,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (4)
// d = 1/2 - S0 * H0 in f10
(p6) fnma.s1 f10=f7,f9,f8
nop.i 0
} { .mfi
nop.m 0
// Step (0'')
// 3/2 = 1 + 1/2 in f8
(p6) fma.s1 f8=f8,f1,f1
nop.i 0;;
} { .mfi
nop.m 0
// Step (5)
// e = 1 + 3/2 * d in f8
(p6) fma.s1 f8=f8,f10,f1
nop.i 0
} { .mfi
nop.m 0
// Step (6)
// T0 = d * S0 in f11
(p6) fma.s1 f11=f10,f7,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (7)
// G0 = d * H0 in f10
(p6) fma.s1 f10=f10,f9,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// S1 = S0 + e * T0 in f7
(p6) fma.s.s1 f7=f8,f11,f7
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// H1 = H0 + e * G0 in f8
(p6) fma.s1 f8=f8,f10,f9

```

```

    nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // d1 = a - S1 * S1 in f9
  (p6) fnma.s1 f9=f7,f7,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (11)
  // S = S1 + d1 * H1 in f7
  (p6) fma.s.s0 f7=f9,f8,f7
  nop.i 0;;

  // END SINGLE PRECISION MINIMUM LATENCY SQUARE ROOT ALGORITHM
} { .mmb
  nop.m 0
  // store result
  stfs [r33]=f7
  // return
  br.ret.sptk b0;;
}

.endp sgl_sqrt_min_lat

```

#### Sample test driver:

```

#include<stdio.h>

void sgl_sqrt_min_lat(float*,float*);

void run_test(unsigned long ia,unsigned long iq)
{
  float a,q;

  *(unsigned long*)&a=ia;

  sgl_sqrt_min_lat(&a,&q);

  printf("\nArgument: %lx\nResult: %lx\n",ia,iq);
  if(iq==*(unsigned long*)&q) printf("Passed\n");
  else printf("Failed\n");
}

void main()
{
  /* sqrt(1)=1 */
  run_test(0x3f800000,0x3f800000);

  /* sqrt(Infinity)=Infinity */
  run_test(0x7f800000,0x7f800000);

  /* sqrt(-1)=QNaN */
  run_test(0xbf800000,0xffc00000);
}

```

### 3.2. Single Precision Floating-Point Square Root, Throughput Optimized



The following algorithm calculates  $S = \sqrt{a}$  in single precision, where  $a$  is a single precision number.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$	table lookup
(2) $H_0 = (0.5 \cdot y_0)_{rn}$	82-bit register format precision
(3) $S_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(4) $d = (0.5 - S_0 \cdot H_0)_{rn}$	82-bit register format precision
(5) $d' = (d + 0.5 * d)_{rn}$	82-bit register format precision
(6) $e = (d + d * d')_{rn}$	82-bit register format precision
(7) $S_1 = (S_0 + e * S_0)_{rn}$	17-bit exponent, 24-bit mantissa
(8) $H_1 = (H_0 + e * H_0)_{rn}$	82-bit register format precision
(9) $d_1 = (a - S_1 \cdot S_1)_{rn}$	82-bit register format precision
(10) $S = (S_1 + d_1 \cdot H_1)_{rnd}$	single precision

The assembly language implementation:

```
.file "sgl_sqrt_max_thr.s"
.section .text
.proc sgl_sqrt_max_thr#
.align 32
.global sgl_sqrt_max_thr#
.align 32

sgl_sqrt_max_thr:
{ .mmb
    alloc r31=ar.pfs,2,0,0,0 // r32, r33

    // &a is in r32
    // &sqrt is in r33 (the address of the sqrt result)

    // general registers used: r2, r31, r32, r33
    // predicate registers used: p6
    // floating-point registers used: f6, f8, f7, f9, f10

    // load the argument a in f6
    ldfs f6 = [r32]
    nop.b 0
} { .mlx

    // BEGIN SINGLE PRECISION MAXIMUM THROUGHPUT SQUARE ROOT ALGORITHM

    nop.m 0
    // exponent of +1/2 in r2
    movl r2 = 0x0fffe;;
} { .mfi
    // +1/2 in f8
    setf.exp f8 = r2
    nop.f 0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f7
    frsqrrta.s0 f7,p6=f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // H0 = 1/2 * y0 in f9
```

```

    (p6) fma.s1 f9=f8,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // S0 = a * y0 in f7
    (p6) fma.s1 f7=f6,f7,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // d = 1/2 - S0 * H0 in f10
    (p6) fnma.s1 f10=f7,f9,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (5)
    // d' = d + 1/2 * d in f8
    (p6) fma.s1 f8=f8,f10,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (6)
    // e = d + d * d' in f8
    (p6) fma.s1 f8=f10,f8,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // S1 = S0 + e * S0 in f7
    (p6) fma.s.s1 f7=f8,f7,f7
    nop.i 0
} { .mfi
    nop.m 0
    // Step (8)
    // H1 = H0 + e * H0 in f8
    (p6) fma.s1 f8=f8,f9,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (9)
    // d1 = a - S1 * S1 in f9
    (p6) fnma.s1 f9=f7,f7,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // S = S1 + d1 * H1 in f7
    (p6) fma.s.s0 f7=f9,f8,f7
    nop.i 0;;

    // END SINGLE PRECISION MAXIMUM THROUGHPUT SQUARE ROOT ALGORITHM
} { .mmb
    nop.m 0
    // store result
    stfs [r33]=f7
    // return
    br.ret.sptk b0;;
}

.endp sgl_sqrt_max_thr

```

Sample test driver:

```

#include<stdio.h>

void sgl_sqrt_max_thr(float*,float*);

```

```

void run_test(unsigned long ia,unsigned long iq)
{
float a,q;

    *(unsigned long*)(&a)=ia;

    sgl_sqrt_max_thr(&a,&q);

    printf("\nArgument: %lx\nResult: %lx\n",ia,iq);
    if(iq==(unsigned long*)(&q)) printf("Passed\n");
    else printf("Failed\n");
}

void main()
{

    /* sqrt(1)=1 */
    run_test(0x3f800000,0x3f800000);

    /* sqrt(Infinity)=Infinity */
    run_test(0x7f800000,0x7f800000);

    /* sqrt(-1)=QNaN */
    run_test(0xbf800000,0xffc00000);
}

```

### 3.3. Double Precision Floating-Point Square Root, Latency Optimized

The following algorithm calculates  $S = \sqrt{a}$  in double precision, where  $a$  is a double precision number.  $m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |   |                                  |
|---|----------------------------------|
| (1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$ | table lookup                     |
| (2) $h = (0.5 \cdot y_0)_m$   | 82-bit register format precision |
| (3) $g = (a \cdot y_0)_m$   | 82-bit register format precision |
| (4) $e = (0.5 - g \cdot h)_m$   | 82-bit register format precision |
| (5) $S_0 = (1.5 + 2.5 \cdot e)_m$   | 82-bit register format precision |
| (6) $e_2 = (e \cdot e)_m$   | 82-bit register format precision |
| (7) $t = (63/8 + 231/16 \cdot e)_m$   | 82-bit register format precision |
| (8) $S_1 = (e + e_2 \cdot S_0)_m$   | 82-bit register format precision |
| (9) $e_4 = (e_2 \cdot e_2)_m$   | 82-bit register format precision |
| (10) $t_1 = (35/8 + e \cdot t)_m$   | 82-bit register format precision |
| (11) $G = (g + S_1 \cdot g)_m$  | 82-bit register format precision |
| (12) $E = (g \cdot e_4)_m$  | 82-bit register format precision |
| (13) $u = (S_1 + e_4 \cdot t_1)_m$  | 82-bit register format precision |
| (14) $g_1 = (G + t_1 \cdot E)_m$  | 17-bit exponent, 53-bit mantissa |
| (15) $h_1 = (h + u \cdot h)_m$  | 82-bit register format precision |
| (16) $d = (a - g_1 \cdot g_1)_m$  | 82-bit register format precision |
| (17) $S = (g_1 + d \cdot h_1)_{rnd}$  | double precision                 |

The assembly language implementation:

```
.file "dbl_sqrt_min_lat.s"
.section .text
.proc  dbl_sqrt_min_lat#
.align 32
.global dbl_sqrt_min_lat#
.align 32

dbl_sqrt_min_lat:
{ .mmb
  alloc r31=ar.pfs,2,0,0,0  // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // general registers used: r2, r3, r31, r32, r33
  // predicate registers used: p6
  // floating-point registers used: f6 to f13

  // load the argument a in f6
  ldld f6 = [r32]
  nop.b 0
} { .mlx

  // BEGIN DOUBLE PRECISION MINIMUM LATENCY SQUARE ROOT ALGORITHM

  nop.m 0
  // exponent of +1/2 in r2
  movl r2 = 0x0fffe;;
} { .mfi
  // +1/2 in f9
  setf.exp f9 = r2
  nop.f 0
  nop.i 0
} { .mlx
  nop.m 0
  // 3/2 in r3
  movl r3=0x3fc00000;;
} { .mfi
  setf.s f10=r3
  // Step (1)
  // y0 = 1/sqrt(a) in f7
  frsqrrta.s0 f7,p6=f6
  nop.i 0;;
} { .mlx
  nop.m 0
  // 5/2 in r2
  movl r2 = 0x40200000
} { .mlx
  nop.m 0
  // 63/8 in r3
  movl r3 = 0x40fc0000;;
} { .mfi
  setf.s f11=r2
  // Step (2)
  // h = +1/2 * y0 in f8
  (p6) fma.s1 f8=f9,f7,f0
  nop.i 0
} { .mfi
  setf.s f12=r3
  // Step (3)
  // g = a * y0 in f7
  (p6) fma.s1 f7=f6,f7,f0
  nop.i 0;;
} { .mlx
  nop.m 0
  // 231/16 in r2
  movl r2 = 0x41670000;;
```

```

} { .mfi
  setf.s f13=r2
  // Step (4)
  //  $e = 1/2 - g * h$  in f9
  (p6) fnma.s1 f9=f7,f8,f9
  nop.i 0
} { .mlx
  nop.m 0
  //  $35/8$  in r3
  movl r3 = 0x408c0000;;
} { .mfi
  setf.s f14=r3
  // Step (5)
  //  $S = 3/2 + 5/2 * e$  in f10
  (p6) fma.s1 f10=f11,f9,f10
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  //  $e2 = e * e$  in f11
  (p6) fma.s1 f11=f9,f9,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (7)
  //  $t = 63/8 + 231/16 * e$  in f12
  (p6) fma.s1 f12=f13,f9,f12
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  //  $S1 = e + e2 * S$  in f10
  (p6) fma.s1 f10=f11,f10,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9)
  //  $e4 = e2 * e2$  in f11
  (p6) fma.s1 f11=f11,f11,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (10)
  //  $t1 = 35/8 + e * t$  in f9
  (p6) fma.s1 f9=f9,f12,f14
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (11)
  //  $G = g + S1 * g$  in f12
  (p6) fma.s1 f12=f10,f7,f7
  nop.i 0
} { .mfi
  nop.m 0
  // Step (12)
  //  $E = g * e4$  in f7
  (p6) fma.s1 f7=f7,f11,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (13)
  //  $u = S1 + e4 * t1$  in f10
  (p6) fma.s1 f10=f11,f9,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (14)
  //  $g1 = G + t1 * E$  in f7
  (p6) fma.d.s1 f7=f9,f7,f12
  nop.i 0;;
} { .mfi

```

```

    nop.m 0
    // Step (15)
    // h1 = h + u * h in f8
    (p6) fma.s1 f8=f10,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (16)
    // d = a - g1 * g1 in f9
    (p6) fnma.s1 f9=f7,f7,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (17)
    // g2 = g1 + d * h1 in f7
    (p6) fma.d.s0 f7=f9,f8,f7
    nop.i 0;;

    // END DOUBLE PRECISION MINIMUM LATENCY SQUARE ROOT ALGORITHM
} { .mmb
    nop.m 0
    // store result
    stfd [r33]=f7
    // return
    br.ret.sptk b0;;
}

.endp dbl_sqrt_min_lat

```

#### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[2];
} _FP64;

void dbl_sqrt_min_lat(_FP64*,_FP64*);

void run_test(unsigned long ia1,unsigned long ia0,unsigned long iq1,unsigned long iq0)
{
    _FP64 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1;

    dbl_sqrt_min_lat(&a,&q);

    printf("\nArgument: %08lx%08lx\nResult: %08lx%08lx\n",ia1,ia0,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* sqrt(1)=1 */
    run_test(0x3ff00000,0x00000000,0x3ff00000,0x00000000);

    /* sqrt(Infinity)=Infinity */
    run_test(0x7ff00000,0x00000000,0x7ff00000,0x00000000);

    /* sqrt(-1)=QNaN */
    run_test(0xbff00000,0x00000000,0xfff80000,0x00000000);
}

```

}

### 3.4. Double Precision Floating-Point Square Root, Throughput Optimized

The following algorithm calculates  $S = \sqrt{a}$  in double precision, where  $a$  is a double precision number.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$	table lookup
(2) $H_0 = (0.5 \cdot y_0)_{rn}$	82-bit register format precision
(3) $S_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(4) $d_0 = (0.5 - S_0 \cdot H_0)_{rn}$	82-bit register format precision
(5) $H_1 = (H_0 + d_0 \cdot H_0)_{rn}$	82-bit register format precision
(6) $S_1 = (S_0 + d_0 \cdot S_0)_{rn}$	82-bit register format precision
(7) $d_1 = (0.5 - S_1 \cdot H_1)_{rn}$	82-bit register format precision
(8) $H_2 = (H_1 + d_1 \cdot H_1)_{rn}$	82-bit register format precision
(9) $S_2 = (S_1 + d_1 \cdot S_1)_{rn}$	82-bit register format precision
(10) $d_2 = (0.5 - S_2 \cdot H_2)_{rn}$	82-bit register format precision
(11) $H_3 = (H_2 + d_2 \cdot H_2)_{rn}$	82-bit register format precision
(12) $S_3 = (S_2 + d_2 \cdot S_2)_{rn}$	17-bit exponent, 53-bit mantissa
(13) $e_3 = (a - S_3 \cdot S_3)_{rn}$	82-bit register format precision
(14) $S = (S_3 + e_3 \cdot H_3)_{rnd}$	double precision

The assembly language implementation:

```
.file "dbl_sqrt_max_thr.s"
.section .text
.proc   dbl_sqrt_max_thr#
.align 32
.global dbl_sqrt_max_thr#
.align 32

dbl_sqrt_max_thr:
{ .mmb
  alloc r31=ar.pfs,2,0,0,0 // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // general registers used: r2, r31, r32, r33
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9, f10

  // load the argument a in f6
  ldld f6 = [r32]
  nop.b 0
} { .mlx

  // BEGIN DOUBLE PRECISION MAXIMUM THROUGHPUT SQUARE ROOT ALGORITHM

  nop.m 0
  // exponent of +1/2 in r2
  movl r2 = 0x0fffe;;
} { .mfi
  // +1/2 in f10
  setf.exp f10 = r2
  nop.f 0
  nop.i 0;;
} { .mfi
```

```

    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f7
    frsqrrta.s0 f7,p6=f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // H0 = +1/2 * y0 in f8
    (p6) fma.s1 f8=f10,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // S0 = a * y0 in f7
    (p6) fma.s1 f7=f6,f7,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // d0 = 1/2 - S0 * H0 in f9
    (p6) fnma.s1 f9=f7,f8,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (5)
    // H1 = H0 + d0 * H0 in f8
    (p6) fma.s1 f8=f9,f8,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (6)
    // S1 = S0 + d0 * S0 in f7
    (p6) fma.s1 f7=f9,f7,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // d1 = 1/2 - S1 * H1 in f9
    (p6) fnma.s1 f9=f7,f8,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // H2 = H1 + d1 * H1 in f8
    (p6) fma.s1 f8=f9,f8,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (9)
    // S2 = S1 + d1 * S1 in f7
    (p6) fma.s1 f7=f9,f7,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // d2 = 1/2 - S2 * H2 in f9
    (p6) fnma.s1 f9=f7,f8,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // H3 = H2 + d2 * H2 in f8
    (p6) fma.s1 f8=f9,f8,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (12)
    // S3 = S2 + d2 * S2 in f7
    (p6) fma.d.s1 f7=f9,f7,f7
    nop.i 0;;

```



```

} { .mfi
  nop.m 0
  // Step (13)
  // e3 = a - S3 * S3 in f9
  (p6) fnma.s1 f9=f7,f7,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (14)
  // S = S3 + e3 * H3 in f7
  (p6) fma.d.s0 f7=f9,f8,f7
  nop.i 0;;

  // END DOUBLE PRECISION MAXIMUM THROUGHPUT SQUARE ROOT ALGORITHM

} { .mmb
  nop.m 0
  // store result
  stfd [r33]=f7
  // return
  br.ret.sptk b0;;
}

.endp dbl_sqrt_max_thr

```

### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[2];
} _FP64;

void dbl_sqrt_max_thr(_FP64*,_FP64*);

void run_test(unsigned long ia1,unsigned long ia0,unsigned long iq1,unsigned long iq0)
{
    _FP64 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1;

    dbl_sqrt_max_thr(&a,&q);

    printf("\nArgument: %08lx%08lx\nResult: %08lx%08lx\n",ia1,ia0,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* sqrt(1)=1 */
    run_test(0x3ff00000,0x00000000,0x3ff00000,0x00000000);

    /* sqrt(Infinity)=Infinity */
    run_test(0x7ff00000,0x00000000,0x7ff00000,0x00000000);

    /* sqrt(-1)=QNaN */
    run_test(0xbff00000,0x00000000,0xfff80000,0x00000000);
}

```

### 3.5. Double Extended Precision Floating-Point Square Root

The following algorithm, optimized for both latency and throughput, calculates  $S = \sqrt{a}$  in double extended precision, where  $a$  is a double extended precision number.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$	table lookup
(2) $H_0 = (0.5 \cdot y_0)_{rn}$	82-bit register format precision
(3) $S_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(4) $d_0 = (0.5 - S_0 \cdot H_0)_{rn}$	82-bit register format precision
(5) $H_1 = (H_0 + d_0 \cdot H_0)_{rn}$	82-bit register format precision
(6) $S_1 = (S_0 + d_0 \cdot S_0)_{rn}$	82-bit register format precision
(7) $d_1 = (0.5 - S_1 \cdot H_1)_{rn}$	82-bit register format precision
(8) $H_2 = (H_1 + d_1 \cdot H_1)_{rn}$	82-bit register format precision
(9) $S_2 = (S_1 + d_1 \cdot S_1)_{rn}$	82-bit register format precision
(10) $d_2 = (0.5 - S_2 \cdot H_2)_{rn}$	82-bit register format precision
(11) $e_2 = (a - S_2 \cdot S_2)_{rn}$	82-bit register format precision
(12) $S_3 = (S_2 + e_2 \cdot H_2)_{rn}$	82-bit register format precision
(13) $H_3 = (H_2 + d_2 \cdot H_2)_{rn}$	82-bit register format precision
(14) $e_3 = (a - S_3 \cdot S_3)_{rn}$	82-bit register format precision
(15) $S = (S_3 + e_3 \cdot H_3)_{rnd}$	double extended precision

The assembly language implementation:

```
.file "dbl_ext_sqrt.s"
.section .text
.proc dbf_ext_sqrt #
.align 32
.global dbf_ext_sqrt #
.align 32

dbf_ext_sqrt:
{ .mmb
  alloc r31=ar.pfs,2,0,0,0 // r32, r33

  // &a is in r32
  // &sqrt is in r33 (the address of the sqrt result)

  // general registers used: r2, r31, r32, r33
  // predicate registers used: p6
  // floating-point registers used: f6, f7, f8, f9, f10, f11

  // load the argument a in f6
  ldfe f6 = [r32]
  nop.b 0
}
// BEGIN DOUBLE EXTENDED SQUARE ROOT ALGORITHM
{ .mlx
  nop.m 0
  // exponent of +1/2 in r2
  movl r2 = 0x0fffe;;
} { .mfi
  // +1/2 in f10
  setf.exp f8 = r2
  nop.f 0
```

```

    nop.i 0;;
} { .mfi
  nop.m 0
  // Step (1)
  // y0 = 1/sqrt(a) in f7
  frsqrrta.s0 f7,p6=f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  // H0 = +1/2 * y0 in f9
  (p6) fma.s1 f9=f8,f7,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  // S0 = a * y0 in f7
  (p6) fma.s1 f7=f6,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  // d0 = 1/2 - S0 * H0 in f10
  (p6) fnma.s1 f10=f7,f9,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  // H1 = H0 + d0 * H0 in f9
  (p6) fma.s1 f9=f10,f9,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  // S1 = S0 + d0 * S0 in f7
  (p6) fma.s1 f7=f10,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  // d1 = 1/2 - S1 * H1 in f10
  (p6) fnma.s1 f10=f7,f9,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // H2 = H1 + d1 * H1 in f9
  (p6) fma.s1 f9=f10,f9,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9)
  // S2 = S1 + d1 * S1 in f7
  (p6) fma.s1 f7=f10,f7,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // d2 = 1/2 - S2 * H2 in f10
  (p6) fnma.s1 f10=f7,f9,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (11)
  // e2 = a - S2 * S2 in f8
  (p6) fnma.s1 f8=f7,f7,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (12)
  // S3 = S2 + e2 * H2 in f7

```

```

    (p6) fma.s1 f7=f8,f9,f7
    nop.i 0
} { .mfi
    nop.m 0
    // Step (13)
    // H3 = H2 + d2 * H2 in f9
    (p6) fma.s1 f9=f10,f9,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (14)
    // e3 = a - S3 * S3 in f8
    (p6) fnma.s1 f8=f7,f7,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (15)
    // S = S3 + e3 * H3 in f7
    (p6) fma.s0 f7=f8,f9,f7
    nop.i 0;;
}
// END DOUBLE EXTENDED SQUARE ROOT ALGORITHM
{ .mmb
    nop.m 0
    // store result
    stfe [r33]=f7
    // return
    br.ret.sptk b0;;
}

.endp dbl_ext_sqrt

```

#### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void dbl_ext_sqrt(_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned
long iq0)
{
    _FP128 a,b,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    dbl_ext_sqrt(&a,&q);

    printf("\nArgument: %08lx%08lx%08lx\nResult: %08lx%08lx%08lx\n",
           ia2,ia1,ia0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx%08lx)\n",q.W[2],q.W[1],q.W[0]);
}

void main()
{

```

```

/* sqrt(1)=1 */
run_test(0,0x3fff,0x80000000,0x00000000,0,0x3fff,0x80000000,0x00000000);

/* sqrt(Infinity)=Infinity */
run_test(0,0x7fff,0x80000000,0x00000000,0,0x7fff,0x80000000,0x00000000);

/* sqrt(-1)=QNaN */
run_test(0,0xbfff,0x80000000,0x00000000,0,0xffff,0xc0000000,0x00000000);
}

```

### 3.6. Parallel (SIMD) Single Precision Floating-Point Square Root, Minimum Latency Scaled, Version 1

The square roots of a pair of single precision numbers (  $a$  ) are calculated in parallel, if both arguments satisfy the conditions that ensure the correctness of the iterations used. (Otherwise, the results must be separately computed and returned packed together). The parallel single precision square root algorithm is described below.  $rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All of the symbols used are packed single precision numbers. Each parallel step is performed in single precision.

To ensure that the Denormal flag is correctly set, scaling coefficients (pscale, nscale) are computed in parallel with the steps shown, and applied before the last multiply-add, which sets the flags in the FPSR status field.

(1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$	table lookup
(2) $S_0 = (a \cdot y_0)_{rn}$	single precision
(3) $z_0 = (y_0 \cdot y_0)_{rn}$	single precision
(4) $b = (0.5 \cdot a)_{rn}$	single precision
(5) $d = (0.5 - b \cdot z_0)_{rn}$	single precision
(6) $k = (a \cdot y_0 - S_0)_{rn}$	single precision
(7) $H_0 = (0.5 \cdot y_0)_{rn}$	single precision
(8) $e = (1 + 1.5 \cdot d)_{rn}$	single precision
(9) $T_0 = (d \cdot S_0 + k)_{rn}$	single precision
(10) $c = (1 + d \cdot e)_{rn}$	single precision
(11) $S_1 = (S_0 + e \cdot T_0)_{rn}$	single precision
(12) $H_1 = (c \cdot H_0)_{rn}$	single precision
(13) $d_1 = (a - S_1 \cdot S_1)_{rn}$	single precision
(14) $H_{1ns} = (nscale \cdot H_1)_{rn}$	single precision (optional step)
(15) $d_{1ps} = (pscale \cdot d_1)_{rn}$	single precision (optional step)
(16) $S = (S_1 + d_{1ps} \cdot H_{1ns})_{rnd}$	single precision

The assembly language implementation:

```

.file "simd_sqrt_min_lat_sc.s"
.section .text
.proc  simd_sqrt_min_lat_sc#
.align 32
.global simd_sqrt_min_lat_sc#
.align 32

simd_sqrt_min_lat_sc:

    // &a is in r32
    // &sqrt is in r33 (the address of the square root result)

```

```

// general registers used: r2, r3,r8,r9,r10,r11,r14,r15, r31, r32, r33
// predicate registers used: p6
// floating-point registers used: f6 to f15, f32, f33

// expects the user status field in the FPSR to have wre = 0 and ftz = 0

{ .mmb
  alloc      r31=ar.pfs,2,0,0,0 // r32, r33
  // load a, the argument, in f6
  ldf.fill f6 = [r32]
  nop.b 0;;
}
// BEGIN SCALED SIMD SQUARE ROOT ALGORITHM
{ .mlx
  nop.m 0
  // get negative scale factor (1,2^{-24})
  movl r14=0x3f80000033800000;;
} { .mlx
  nop.m 0
  // get positive scale factor (1,2^{24})
  movl r15=0x3f8000004b800000;;
} { .mlx
  // exponent of 1/2 in r3, in advance
  mov r3=0x0fffe
  // 1/2, 1/2
  movl r2 = 0x3f0000003f000000;;
} { .mlx
  setf.sig f8=r2 // 1/2, 1/2 in f8
  // 1, 1
  movl r2 = 0x3f8000003f800000;;
} { .mlx
  setf.sig f11=r2 // 1, 1 in f11
  // 3/2, 3/2
  movl r2 = 0x3fc000003fc00000;;
} { .mlx
  nop.m 0
  // assume negative scale factor (1,1)
  movl r10=0x3f8000003f800000;;
} { .mmf
  setf.sig f13=r2 // 3/2, 3/2 in f13
  // extract a_high, a_low from f6 into r2
  getf.sig r2 = f6

  // begin computation

  // Step (1)
  // y0 = 1 / sqrt (a) in f7
  fprsqrrta.s0 f7,p6=f6;;
} { .mmi
  // unpack a_low into f14
  setf.s f14 = r2
  nop.m 0
  // extract a_low exponent
  extr.u r8=r2,23,8;;
} { .mmi
  mov r11=r10
  // set p7 in a_low needs scaling, else set p8
  cmp.gt.unc p7,p8=0x30,r8
  // extract a_high exponent
  extr.u r9=r2,55,8;;
} { .mmi
  // Set p9 if a_high needs scaling and a_low needs scaling
  // Set p10 if a_high doesn't need scaling and a_low needs scaling
  (p7) cmp.gt.unc p9,p10=0x30,r9
  // Set p11 if a_high needs scaling and a_low doesn't need scaling
  // Set p12 if a_high doesn't need scaling and a_low doesn't need scaling
  (p8) cmp.gt.unc p11,p12=0x30,r9
  // shift right a_high in r2 (in advance)
  shr.u r2 = r2, 0x20;;
} { .mlx

```

```

    nop.m 0
    // get negative scale factor ( $2^{-24}$ ,  $2^{-24}$ )
    (p9) movl r10=0x3380000033800000
} { .mfi
    nop.m 0
    // Step (2)
    //  $S0 = a * y0$  in f10
    (p6) fpma.s1 f10=f6,f7,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (3)
    //  $z0 = y0 * y0$  in f9
    (p6) fpma.s1 f9=f7,f7,f0
    nop.i 0
} { .mfi
    // unpack in advance a_high in f15
    setf.s f15 = r2
    // Step (4)
    //  $b = 1/2 * a$  in f12
    (p6) fpma.s1 f12=f8,f6,f0
    nop.i 0;;
} { .mlx
    nop.m 0
    // get positive scale factor ( $2^{24}$ ,  $2^{24}$ )
    (p9) movl r11=0x4b8000004b800000;;
}
.pred.rel "mutex",p10,p11
{ .mlx
    // get negative scale factor ( $1, 2^{-24}$ )
    (p10) mov r10=r14
    // get negative scale factor ( $2^{-24}, 1$ )
    (p11) movl r10=0x338000003f800000
} { .mlx
    // get positive scale factor ( $1, 2^{24}$ )
    (p10) mov r11=r15
    // get positive scale factor ( $2^{24}, 1$ )
    (p11) movl r11=0x4b8000003f800000;;
} { .mfi
    nop.m 0
    // Step (5)
    //  $d = 1/2 - b * z0$  in f9
    (p6) fpnma.s1 f9=f12,f9,f8
    nop.i 0
} { .mfi
    nop.m 0
    // Step (6)
    //  $k = a * y0 - S0$  in f12
    (p6) fpms.s1 f12=f6,f7,f10
    nop.i 0;;
} { .mmf
    // set negative factor, nscale
    setf.sig f32=r10
    // set positive factor, pscale
    setf.sig f33=r11
    // Step (7)
    //  $H0 = 1/2 * y0$  in f8
    (p6) fpma.s1 f8=f8,f7,f0;;
} { .mfi
    nop.m 0
    // Step (8)
    //  $e = 1 + 3/2 * d$  in f13
    (p6) fpma.s1 f13=f13,f9,f11
    nop.i 0
} { .mfi
    nop.m 0
    // Step (9)
    //  $T0 = d * S0 + k$  in f12
    (p6) fpma.s1 f12=f9,f10,f12
    nop.i 0;;
} { .mfi

```

```

    nop.m 0
    // Step (10)
    // c = 1 + d * e in f9
    (p6) fpma.s1 f9=f9,f13,f11
    nop.i 0
} { .mfi
    // +1/2 in f11
    setf.exp f11 = r3
    // Step (11)
    // S1 = S0 + e * T0 in f7
    (p6) fpma.s1 f7=f13,f12,f10
    nop.i 0;;
} { .mfi
    // +1/2 in f12
    setf.exp f12 = r3
    // nop.m 0
    // Step (12)
    // H1 = c * H0 in f8
    (p6) fpma.s1 f8=f9,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (13)
    // d1 = a - S1 * S1 in f9
    (p6) fpmma.s1 f9=f7,f7,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // scale down H1
    fpma.s1 f8=f8,f32,f0
    nop.i 0
} { .mfi
    nop.m 0
    // scale up d1
    fpma.s1 f9=f9,f33,f0
    nop.i 0;;
} { .mfb
    nop.m 0
    // Step (14)
    // S = S1 + d1 * H1 in f7
    (p6) fpma.s0 f7=f9,f8,f7

    // jump over the unpacked computation if (p6) was 1
    /// nop.b 0
    (p6) br.cond.dptk done;;
} { .mfi

    // Apply single precision minimum latency square root algorithm
    // for the low and high parts;
    // perform two square roots on unpacked operands; if any of the two halves
    // of the result in f7 is +0, -0, +Inf, -Inf, or NaN, then calling frsqrrta
    // is superfluous; yet, instead of performing this check, we just invoke
    // frsqrrta and get the result again

    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f7
    frsqrrta.s0 f7,p7=f14
    nop.i 0
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f6
    frsqrrta.s0 f6,p8=f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // normalize a_low in f14
    (p7) fnorm.s1 f14 = f14
    nop.i 0
} { .mfi

```



```

    nop.m 0
    // normalize a_high in f15
    (p8) fnorm.s1 f15 = f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // H0 = 1/2 * y0 in f9
    (p7) fma.s1 f9=f11,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // S0 = a * y0 in f7
    (p7) fma.s1 f7=f14,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (2)
    // H0 = 1/2 * y0 in f13
    (p8) fma.s1 f13=f12,f6,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // S0 = a * y0 in f6
    (p8) fma.s1 f6=f15,f6,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // d = 1/2 - S0 * H0 in f10
    (p7) fnma.s1 f10=f7,f9,f11
    nop.i 0
} { .mfi
    nop.m 0
    // Step (0'')
    // 3/2 = 1 + 1/2 in f11
    (p7) fma.s1 f11=f11,f1,f1
    nop.i 0
} { .mfi
    nop.m 0
    // Step (4)
    // d = 1/2 - S0 * H0 in f32
    (p8) fnma.s1 f32=f6,f13,f12
    nop.i 0
} { .mfi
    nop.m 0
    // Step (0'')
    // 3/2 = 1 + 1/2 in f12
    (p8) fma.s1 f12=f12,f1,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (5)
    // e = 1 + 3/2 * d in f11
    (p7) fma.s1 f11=f11,f10,f1
    nop.i 0
} { .mfi
    nop.m 0
    // Step (6)
    // T0 = d * S0 in f8
    (p7) fma.s1 f8=f10,f7,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (5)
    // e = 1 + 3/2 * d in f12
    (p8) fma.s1 f12=f12,f32,f1
    nop.i 0
} { .mfi

```

```

    nop.m 0
    // Step (6)
    // T0 = d * S0 in f33
    (p8) fma.s1 f33=f32,f6,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // G0 = d * H0 in f10
    (p7) fma.s1 f10=f10,f9,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (7)
    // G0 = d * H0 in f32
    (p8) fma.s1 f32=f32,f13,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // S1 = S0 + e * T0 in f7
    (p7) fma.s.s1 f7=f11,f8,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // S1 = S0 + e * T0 in f6
    (p8) fma.s.s1 f6=f12,f33,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (9)
    // H1 = H0 + e * G0 in f11
    (p7) fma.s1 f11=f11,f10,f9
    nop.i 0
} { .mfi
    nop.m 0
    // Step (9)
    // H1 = H0 + e * G0 in f12
    (p8) fma.s1 f12=f12,f32,f13
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // d1 = a - S1 * S1 in f9
    (p7) fnma.s1 f9=f7,f7,f14
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // d1 = a - S1 * S1 in f13
    (p8) fnma.s1 f13=f6,f6,f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // S = S1 + d1 * H1 in f7
    (p7) fma.s.s0 f7=f9,f11,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // S = S1 + d1 * H1 in f6
    (p8) fma.s.s0 f6=f13,f12,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // pack res_low from f6 and res_high from f7 into f7
    fpack f7 = f6, f7
    nop.i 0;;
}

```

```
// END SCALED SIMD SQUARE ROOT ALGORITHM
done:

{ .mmb
  nop.m 0
  // store result
  stf.spill [r33]=f7
  // return
  br.ret.sptk b0;;
}

.endp simd_sqrt_min_lat_sc
```

### Sample test driver:

```
#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void simd_sqrt_min_lat_sc(_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    simd_sqrt_min_lat_sc(&a,&q);

    printf("\nArgument: %08lx%08lx%08lx\nResult: %08lx%08lx%08lx\n",
           ia2,ia1,ia0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* sqrt(1)=1, sqrt(4)=2 */
    run_test(0,0x1003e,0x3f800000,0x40800000,0,0x1003e,0x3f800000,0x40000000);

    /* sqrt(Infinity)=Infinity, sqrt(-0)=-0 */
    run_test(0,0x1003e,0x7f800000,0x80000000,0,0x1003e,0x7f800000,0x80000000);

    /* sqrt(2.25)=1.5, sqrt(0)=0 */
    run_test(0,0x1003e,0x40100000,0x00000000,0,0x1003e,0x3fc00000,0x00000000);
}
```

### 3.7. Parallel (SIMD) Single Precision Floating-Point Square Root, Minimum Latency, Version 2

A slightly faster version of the SIMD algorithm presented above eliminates the scaling steps and thus does not guarantee that the last computation step correctly sets the Denormal flag. (This flag may be incorrectly set for some corner cases).

$rn$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All of the symbols used are packed single precision numbers. Each parallel step is performed in single precision.

(1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$	table lookup
(2) $S_0 = (a \cdot y_0)_{rn}$	single precision
(3) $z_0 = (y_0 \cdot y_0)_{rn}$	single precision
(4) $b = (0.5 \cdot a)_{rn}$	single precision
(5) $d = (0.5 - b \cdot z_0)_{rn}$	single precision
(6) $k = (a \cdot y_0 - S_0)_{rn}$	single precision
(7) $H_0 = (0.5 \cdot y_0)_{rn}$	single precision
(8) $e = (1 + 1.5 \cdot d)_{rn}$	single precision
(9) $T_0 = (d \cdot S_0 + k)_{rn}$	single precision
(10) $c = (1 + d \cdot e)_{rn}$	single precision
(11) $S_1 = (S_0 + e \cdot T_0)_{rn}$	single precision
(12) $H_1 = (c \cdot H_0)_{rn}$	single precision
(13) $d_1 = (a - S_1 \cdot S_1)_{rn}$	single precision
(14) $S = (S_1 + d_1 \cdot H_1)_{rnd}$	single precision

The assembly language implementation:

```
.file "simd_sqrt_min_lat.s"
.section .text
.proc   simd_sqrt_min_lat #
.align 32
.global simd_sqrt_min_lat #
.align 32

simd_sqrt_min_lat:

    // &a is in r32
    // &sqrt is in r33 (the address of the square root result)

    // general registers used: r2, r3, r31, r32, r33
    // predicate registers used: p6
    // floating-point registers used: f6 to f15, f32, f33

    // expects the user status field in the FPSR to have wre = 0 and ftz = 0

{ .mmb
  alloc    r31=ar.pfs,2,0,0,0 // r32, r33
  // load a, the argument, in f6
  ldf.fill f6 = [r32]
  nop.b 0;;
} { .mfi
  nop.m 0

// BEGIN MINIMUM LATENCY SIMD SQUARE ROOT ALGORITHM

// Step (1)
// y0 = 1 / sqrt (a) in f7
fprsqrrta.s0 f7,p6=f6
nop.i 0
} { .mlx
  // exponent of 1/2 in r3, in advance
  mov r3=0xffffe
  // 1/2, 1/2
  movl r2 = 0x3f0000003f000000;;
} { .mlx
  setf.sig f8=r2 // 1/2, 1/2 in f8
```

```

// 1, 1
movl r2 = 0x3f8000003f800000;;
} { .mlx
  setf.sig f11=r2 // 1, 1 in f11
  // 3/2, 3/2
  movl r2 = 0x3fc000003fc00000;;
} { .mmf
  nop.m 0
  setf.sig f13=r2 // 3/2, 3/2 in f13
  // Step (3)
  // S0 = a * y0 in f10
  (p6) fpma.s1 f10=f6,f7,f0
} { .mfi
  // extract a_high, a_low from f6 into r2
  getf.sig r2 = f6
  // Step (2)
  // z0 = y0 * y0 in f9
  (p6) fpma.s1 f9=f7,f7,f0
  nop.i 0;;
} { .mmi
  nop.m 0;;
  // unpack a_low into f14
  setf.s f14 = r2
  // shift right a_high in r2 (in advance)
  shr.u r2 = r2, 0x20;;
} { .mfi
  nop.m 0
  // Step (4)
  // b = 1/2 * a in f12
  (p6) fpma.s1 f12=f8,f6,f0
  nop.i 0
} { .mfi
  // unpack in advance a_high in f15
  setf.s f15 = r2
  nop.f 0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  // d = 1/2 - b * z0 in f9
  (p6) fpnma.s1 f9=f12,f9,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  // k = a * y0 - S0 in f12
  (p6) fpms.s1 f12=f6,f7,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // H0 = 1/2 * y0 in f8
  (p6) fpma.s1 f8=f8,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  // e = 1 + 3/2 * d in f13
  (p6) fpma.s1 f13=f13,f9,f11
  nop.i 0
} { .mfi
  nop.m 0
  // Step (8)
  // T0 = d * S0 + k in f12
  (p6) fpma.s1 f12=f9,f10,f12
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // c = 1 + d * e in f9
  (p6) fpma.s1 f9=f9,f13,f11

```

```

    nop.i 0
} { .mfi
    // +1/2 in f11
    setf.exp f11 = r3
    // Step (11)
    // S1 = S0 + e * T0 in f7
    (p6) fpma.s1 f7=f13,f12,f10
    nop.i 0;;
} { .mfi
    // +1/2 in f12
    setf.exp f12 = r3
    // nop.m 0
    // Step (12)
    // H1 = c * H0 in f8
    (p6) fpma.s1 f8=f9,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (13)
    // d1 = a - S1 * S1 in f9
    (p6) fpmma.s1 f9=f7,f7,f6
    nop.i 0;;
} { .mfb
    nop.m 0
    // Step (14)
    // S = S1 + d1 * H1 in f7
    (p6) fpma.s0 f7=f9,f8,f7
    // jump over the unpacked computation if (p6) was 1
    //// nop.b 0
    (p6) br.cond.dptk done;;
} { .mfi

    // Apply single precision minimum latency algorithm for the low and high parts
    // perform two square roots on unpacked operands; if any of the two halves
    // of the result in f7 is +0, -0, +Inf, -Inf, or NaN, then calling frsqrrta
    // is superfluous; yet, instead of performing this check, we just invoke
    // frsqrrta and get the result again

    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f7
    frsqrrta.s0 f7,p7=f14
    nop.i 0
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1/sqrt(a) in f6
    frsqrrta.s0 f6,p8=f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // normalize a_low in f14
    (p7) fnorm.s1 f14 = f14
    nop.i 0
} { .mfi
    nop.m 0
    // normalize a_high in f15
    (p8) fnorm.s1 f15 = f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // H0 = 1/2 * y0 in f9
    (p7) fma.s1 f9=f11,f7,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // S0 = a * y0 in f7
    (p7) fma.s1 f7=f14,f7,f0
    nop.i 0

```

```

} { .mfi
  nop.m 0
  // Step (2)
  //  $H_0 = 1/2 * y_0$  in f13
  (p8) fma.s1 f13=f12,f6,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  //  $S_0 = a * y_0$  in f6
  (p8) fma.s1 f6=f15,f6,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  //  $d = 1/2 - S_0 * H_0$  in f10
  (p7) fnma.s1 f10=f7,f9,f11
  nop.i 0
} { .mfi
  nop.m 0
  // Step (0'')
  //  $3/2 = 1 + 1/2$  in f11
  (p7) fma.s1 f11=f11,f1,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (4)
  //  $d = 1/2 - S_0 * H_0$  in f32
  (p8) fnma.s1 f32=f6,f13,f12
  nop.i 0
} { .mfi
  nop.m 0
  // Step (0'')
  //  $3/2 = 1 + 1/2$  in f12
  (p8) fma.s1 f12=f12,f1,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  //  $e = 1 + 3/2 * d$  in f11
  (p7) fma.s1 f11=f11,f10,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  //  $T_0 = d * S_0$  in f8
  (p7) fma.s1 f8=f10,f7,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  //  $e = 1 + 3/2 * d$  in f12
  (p8) fma.s1 f12=f12,f32,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  //  $T_0 = d * S_0$  in f33
  (p8) fma.s1 f33=f32,f6,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  //  $G_0 = d * H_0$  in f10
  (p7) fma.s1 f10=f10,f9,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (7)
  //  $G_0 = d * H_0$  in f32
  (p8) fma.s1 f32=f32,f13,f0

```

```

    nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // S1 = S0 + e * T0 in f7
  (p7) fma.s.s1 f7=f11,f8,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  // S1 = S0 + e * T0 in f6
  (p8) fma.s.s1 f6=f12,f33,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // H1 = H0 + e * G0 in f11
  (p7) fma.s1 f11=f11,f10,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (9)
  // H1 = H0 + e * G0 in f12
  (p8) fma.s1 f12=f12,f32,f13
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // d1 = a - S1 * S1 in f9
  (p7) fnma.s1 f9=f7,f7,f14
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (10)
  // d1 = a - S1 * S1 in f13
  (p8) fnma.s1 f13=f6,f6,f15
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (11)
  // S = S1 + d1 * H1 in f7
  (p7) fma.s.s0 f7=f9,f11,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (11)
  // S = S1 + d1 * H1 in f6
  (p8) fma.s.s0 f6=f13,f12,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // pack res_low from f6 and res_high from f7 into f7
  fpack f7 = f6, f7
  nop.i 0;;
}

// END MINIMUM LATENCY SIMD SQUARE ROOT ALGORITHM
done:

{ .mmb
  nop.m 0
  // store result
  stf.spill [r33]=f7
  // return
  br.ret.sptk b0;;
}

.endp simd_sqrt_min_lat

```



Sample test driver:

```
#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void simd_sqrt_min_lat(_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    simd_sqrt_min_lat(&a,&q);

    printf("\nArgument: %08lx%08lx%08lx\nResult: %08lx%08lx%08lx\n",
           ia2,ia1,ia0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{
    /* sqrt(1)=1, sqrt(4)=2 */
    run_test(0,0x1003e,0x3f800000,0x40800000,0,0x1003e,0x3f800000,0x40000000);

    /* sqrt(Infinity)=Infinity, sqrt(-0)=-0 */
    run_test(0,0x1003e,0x7f800000,0x80000000,0,0x1003e,0x7f800000,0x80000000);

    /* sqrt(2.25)=1.5, sqrt(0)=0 */
    run_test(0,0x1003e,0x40100000,0x00000000,0,0x1003e,0x3fc00000,0x00000000);
}
```

### 3.8. Parallel (SIMD) Single Precision Floating-Point Square Root, Throughput Optimized

This SIMD square root algorithm maximizes throughput (12 floating-point instructions, versus 14 for the latency optimized version); it does not guarantee that the last computation step correctly sets the Denormal flag.

$m$  is the IEEE round to nearest mode, and  $rnd$  is any IEEE rounding mode. All of the symbols used are packed single precision numbers. Each parallel step is performed in single precision.

- |   |                  |
|---|------------------|
| (1) $y_0 = (1 / \sqrt{a}) \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.831}$ | table lookup     |
| (2) $S_0 = (a \cdot y_0)_m$   | single precision |
| (3) $H_0 = (0.5 \cdot y_0)_m$   | single precision |
| (4) $d_0 = (0.5 - S_0 * H_0)_m$   | single precision |
| (5) $H_1 = (H_0 + d_0 * H_0)_m$   | single precision |
| (6) $S_1 = (S_0 + d_0 * S_0)_m$   | single precision |
| (7) $d_1 = (0.5 - S_1 * H_1)_m$   | single precision |
| (8) $e_1 = (a - S_1 * S_1)_m$   | single precision |

(9) $S_2 = (S_1 + e_1 * H_1)_{rn}$	single precision
(10) $H_2 = (H_1 + d_1 * H_1)_{rn}$	single precision
(11) $e_2 = (a - S_2 * S_2)_{rn}$	single precision
(12) $S = (S_2 + e_2 * H_2)_{rd}$	single precision

The assembly language implementation:

```
.file "simd_sqrt_max_thr.s"
.section .text
.proc  simd_sqrt_max_thr#
.align 32
.global simd_sqrt_max_thr#
.align 32

simd_sqrt_max_thr:

    // &a is in r32
    // &sqrt is in r33 (the address of the square root result)

    // general registers used: r2, r3, r31, r32, r33
    // predicate registers used: p6, p7, p8
    // floating-point registers used: f6 to f15, f32, f33

    // expects the user status field in the FPSR to have wre = 0 and ftz = 0

{ .mmb
  alloc    r31=ar.pfs,2,0,0,0 // r32, r33
  // load a, the argument, in f6
  ldf.fill f6 = [r32]
  nop.b 0;;
} { .mfi
  nop.m 0

// BEGIN MAXIMUM THROUGHPUT SIMD SQUARE ROOT ALGORITHM

    // Step (1)
    // y0 = 1 / sqrt (a) in f7
    fprsqrrta.s0 f7,p6=f6
    nop.i 0
} { .mlx
    // exponent of 1/2 in r3, in advance
    mov r3=0x0fffe
    // 1/2, 1/2
    movl r2 = 0x3f0000003f000000;;
} { .mlx
    setf.sig f8=r2 // 1/2, 1/2 in f8
    // 1, 1
    movl r2 = 0x3f8000003f800000;;
} { .mlx
    setf.sig f11=r2 // 1, 1 in f11
    // 3/2, 3/2
    movl r2 = 0x3fc000003fc00000;;
} { .mmf
    nop.m 0
    setf.sig f13=r2 // 3/2, 3/2 in f13
    // Step (2)
    // S0 = a * y0 in f10
    (p6) fpma.s1 f10=f6,f7,f0
} { .mfi
    // extract a_high, a_low from f6 into r2
    getf.sig r2 = f6
    // Step (3)
    // H0 = 0.5 * y0 in f9
    (p6) fpma.s1 f9=f8,f7,f0
    nop.i 0;;
```

```

} { .mmi
  nop.m 0;;
  // unpack a_low into f14
  setf.s f14 = r2
  // shift right a_high in r2 (in advance)
  shr.u r2 = r2, 0x20;;
} { .mfi
  nop.m 0
  // Step (4)
  // d0=0.5-S0*H0
  (p6) fpmma.s1 f12=f10,f9,f8
  nop.i 0;;
} { .mfi
  // unpack in advance a_high in f15
  setf.s f15 = r2
  nop.f 0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (5)
  // H1=H0+d0*H0
  (p6) fpma.s1 f9=f12,f9,f9
  nop.i 0
} { .mfi
  nop.m 0
  // Step (6)
  // S1=S0+d0*S0
  (p6) fpma.s1 f10=f12,f10,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  // d1=0.5-S1*H1
  (p6) fpmma.s1 f12=f9,f10,f8
  nop.i 0
} { .mfi
  nop.m 0
  // Step (8)
  // e1=a-S1*S1
  (p6) fpmma.s1 f13=f10,f10,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  // S2=S1+e1*H1
  (p6) fpma.s1 f10=f9,f13,f10
  nop.i 0
} { .mfi
  nop.m 0
  // Step (10)
  // H2=H1+d1*H1
  (p6) fpma.s1 f9=f9,f12,f9
  nop.i 0;;
} { .mfi
  // +1/2 in f11
  setf.exp f11 = r3
  // Step (11)
  // e2=a-S2*S2
  (p6) fpmma.s1 f13=f10,f10,f6
  nop.i 0;;
} { .mfb
  // +1/2 in f12
  setf.exp f12 = r3
  // Step (12)
  // S=S2+e2*H2
  (p6) fpma.s0 f7=f9,f13,f10
  // jump over the unpacked computation if (p6) was 1
  (p6) br.cond.dptk done;;
} { .mfi
  // Apply single precision square root algorithm for the low and high parts

```

```

// perform two square roots on unpacked operands; if any of the two halves
// of the result in f7 is +0, -0, +Inf, -Inf, or NaN, then calling frsqrrta
// is superfluous; yet, instead of performing this check, we just invoke
// frsqrrta and get the result again

nop.m 0
// Step (1)
// y0 = 1/sqrt(a) in f7
frsqrrta.s0 f7,p7=f14
nop.i 0
} { .mfi
nop.m 0
// Step (1)
// y0 = 1/sqrt(a) in f6
frsqrrta.s0 f6,p8=f15
nop.i 0;;
} { .mfi
nop.m 0
// normalize a_low in f14
(p7) fnorm.s1 f14 = f14
nop.i 0
} { .mfi
nop.m 0
// normalize a_high in f15
(p8) fnorm.s1 f15 = f15
nop.i 0;;
} { .mfi
nop.m 0
// Step (2)
// H0 = 1/2 * y0 in f9
(p7) fma.s1 f9=f11,f7,f0
nop.i 0
} { .mfi
nop.m 0
// Step (3)
// S0 = a * y0 in f7
(p7) fma.s1 f7=f14,f7,f0
nop.i 0
} { .mfi
nop.m 0
// Step (2)
// H0 = 1/2 * y0 in f13
(p8) fma.s1 f13=f12,f6,f0
nop.i 0
} { .mfi
nop.m 0
// Step (3)
// S0 = a * y0 in f6
(p8) fma.s1 f6=f15,f6,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (4)
// d = 1/2 - S0 * H0 in f10
(p7) fnma.s1 f10=f7,f9,f11
nop.i 0
} { .mfi
nop.m 0
// Step (0'')
// 3/2 = 1 + 1/2 in f11
(p7) fma.s1 f11=f11,f1,f1
nop.i 0
} { .mfi
nop.m 0
// Step (4)
// d = 1/2 - S0 * H0 in f32
(p8) fnma.s1 f32=f6,f13,f12
nop.i 0
} { .mfi
nop.m 0
// Step (0'')

```

```

// 3/2 = 1 + 1/2 in f12
(p8) fma.s1 f12=f12,f1,f1
nop.i 0;;
} { .mfi
nop.m 0
// Step (5)
// e = 1 + 3/2 * d in f11
(p7) fma.s1 f11=f11,f10,f1
nop.i 0
} { .mfi
nop.m 0
// Step (6)
// T0 = d * S0 in f8
(p7) fma.s1 f8=f10,f7,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (5)
// e = 1 + 3/2 * d in f12
(p8) fma.s1 f12=f12,f32,f1
nop.i 0
} { .mfi
nop.m 0
// Step (6)
// T0 = d * S0 in f33
(p8) fma.s1 f33=f32,f6,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (7)
// G0 = d * H0 in f10
(p7) fma.s1 f10=f10,f9,f0
nop.i 0
} { .mfi
nop.m 0
// Step (7)
// G0 = d * H0 in f32
(p8) fma.s1 f32=f32,f13,f0
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// S1 = S0 + e * T0 in f7
(p7) fma.s.s1 f7=f11,f8,f7
nop.i 0;;
} { .mfi
nop.m 0
// Step (8)
// S1 = S0 + e * T0 in f6
(p8) fma.s.s1 f6=f12,f33,f6
nop.i 0;;
} { .mfi
nop.m 0
// Step (9)
// H1 = H0 + e * G0 in f11
(p7) fma.s1 f11=f11,f10,f9
nop.i 0
} { .mfi
nop.m 0
// Step (9)
// H1 = H0 + e * G0 in f12
(p8) fma.s1 f12=f12,f32,f13
nop.i 0;;
} { .mfi
nop.m 0
// Step (10)
// d1 = a - S1 * S1 in f9
(p7) fnma.s1 f9=f7,f7,f14
nop.i 0;;
} { .mfi
nop.m 0

```

```

    // Step (10)
    // d1 = a - S1 * S1 in f13
    (p8) fnma.s1 f13=f6,f6,f15
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // S = S1 + d1 * H1 in f7
    (p7) fma.s.s0 f7=f9,f11,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (11)
    // S = S1 + d1 * H1 in f6
    (p8) fma.s.s0 f6=f13,f12,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // pack res_low from f6 and res_high from f7 into f7
    fpack f7 = f6, f7
    nop.i 0;;
}

// END MAXIMUM THROUGHPUT SIMD SQUARE ROOT ALGORITHM
done:

{ .mmb
    nop.m 0
    // store result
    stf.spill [r33]=f7
    // return
    br.ret.sptk b0;;
}

.endp simd_sqrt_max_thr

```

### Sample test driver:

```

#include<stdio.h>

typedef struct
{
    unsigned long W[4];
} _FP128;

void simd_sqrt_max_thr(_FP128*,_FP128*);

void run_test(unsigned long ia3,unsigned long ia2,unsigned long ia1,unsigned long ia0,
              unsigned long iq3,unsigned long iq2,unsigned long iq1,unsigned long iq0)
{
    _FP128 a,q;

    a.W[0]=ia0; a.W[1]=ia1; a.W[2]=ia2; a.W[3]=ia3;
    q.W[0]=q.W[1]=q.W[2]=q.W[3]=0;

    simd_sqrt_max_thr(&a,&q);

    printf("\nArgument: %08lx%08lx%08lx\nResult: %08lx%08lx%08lx\n",
           ia2,ia1,ia0,iq2,iq1,iq0);
    if(iq0==q.W[0] && iq1==q.W[1] && iq2==q.W[2] && iq3==q.W[3]) printf("Passed\n");
    else printf("Failed (%08lx%08lx)\n",q.W[1],q.W[0]);
}

void main()
{

```

```

/* sqrt(1)=1, sqrt(4)=2 */
run_test(0,0x1003e,0x3f800000,0x40800000,0,0x1003e,0x3f800000,0x40000000);

/* sqrt(Infinity)=Infinity, sqrt(-0)=-0 */
run_test(0,0x1003e,0x7f800000,0x80000000,0,0x1003e,0x7f800000,0x80000000);

/* sqrt(2.25)=1.5, sqrt(0)=0 */
run_test(0,0x1003e,0x40100000,0x00000000,0,0x1003e,0x3fc00000,0x00000000);
}

```

### 3.9. Software Assistance (SWA) Conditions for Floating Point Square Root

#### Property 2

Let  $a$  be a floating-point number with an  $N_{in}$ -bit significand, and an  $M_{in}$ -bit exponent (limited exponent range), as described by the IEEE-754 Standard for Binary Floating-Point Arithmetic. Let  $N$  be the size of the significand and  $M$  the size of the exponent, in number of bits, corresponding to a computation step in the square root algorithms described.

The exact values of  $N_{in}$ ,  $M_{in}$ ,  $N$ , and  $M$  are specified explicitly or implicitly for each algorithm. For the scalar single, double and double extended precision algorithms,  $M=17$  for all the intermediate steps, and for the SIMD square root algorithms,  $M=8$ . The value of  $N$  is specified by the precision of the computation step.

Then,  $e_{\min,in} = -2^{M_{in}-1} + 2$ , and  $e_{\max,in} = 2^{M_{in}-1} - 1$  are the minimum and maximum values of the exponents allowed for floating point numbers on input, and  $e_{\min} = -2^{M-1} + 2$ , and  $e_{\max} = 2^{M-1} - 1$  are the minimum and maximum values of the exponents allowed for floating point numbers in a given computation step.

Let the normalized  $a$  be  $a = s_a \cdot s_a \cdot 2^{e_a}$ , with  $s_a = \pm 1$ ,  $1 \leq s_a < 2$ ,  $s_a$  representable using  $N_{in}$  bits, and  $e_a \in \mathbb{Z}$ ,  $e_{\min,in} - N_{in} + 1 \leq e_a \leq e_{\max,in}$ . In addition, if  $e_a < e_{\min,in}$  and  $k = e_{\min,in} - e_a$ , then  $(2^{N_{in}-1} \cdot s_a) \equiv 0 \pmod{2^k}$  (which allows for denormal values of  $a$ ).

An *fma* operation for floating-point numbers with  $N$ -bit significands and  $M$ -bit exponents is assumed available, that preserves the  $2 \cdot N$  bits of the product before the summation, and only incurs one rounding error.

Then, the following statements hold.

The scalar algorithms described for calculating  $(\sqrt{a})_{rd}$  in any IEEE rounding mode *rd*, in single and double precision, do not cause in any step overflow, underflow, or loss of precision.

The SIMD algorithms described for calculating  $(\sqrt{a})_{rd}$  in any IEEE rounding mode *rd*, do not cause in any step overflow, underflow, or loss of precision, if:

$e_a \geq e_{\min} + N$  (error estimations such as *a-S* will not be affected by loss of precision)  
(where  $N_{in} = N = 24$ ).

The double-extended precision algorithm described for calculating  $(\sqrt{a})_{rd}$  in any IEEE rounding mode *rd*, does not cause in any step overflow, underflow, or loss of precision, if the input values are representable in double-extended precision format ( $N_{in} = 64$ , and  $M_{in} = 15$ ). The same algorithm does not cause in any step overflow, underflow, or loss of precision for input values in floating-point register format (82-bit floating-point numbers, with  $N_{in} = 64$ , and  $M_{in} = 17$ ), if:

$e_a \geq e_{\min} + N$  (error estimations such as *a-S* will not be affected by loss of precision)  
(where  $N_{in} = N = 64$ ).

Note:

The Itanium™ processor hardware will have to ask for software assistance (SWA) or otherwise indicate that it cannot provide the correctly rounded result of the square root operation, whenever the condition in Property 2 is not satisfied, i.e. for the SIMD square root algorithms, or for the double-extended square root algorithm, when the input value  $a$  is in floating-point register file format (82-bit floating-point number). The SWA condition is:

$$e_a \notin e_{min} + N - 1$$

where  $N_{in} = N = 24$  and  $M_{in} = M = 8$  for SIMD algorithms, and  $N_{in} = N = 64$  and  $M_{in} = M = 17$  for the double extended precision algorithm.



## 4. Integer Divide and Remainder Algorithms for the IA-64 Architecture

Sixteen algorithms are presented in this category: signed and unsigned divide, signed and unsigned remainder for each of the four integer formats (8-bit, 16-bit, 32-bit, 64-bit). They are discussed in more detail and proven correct in [6].

### 4.1. Signed 8-bit Integer Divide

The following algorithm calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 8-bit signed integers.  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $a' = (a + a \cdot 2^{-8})_m$                                      | 82-bit register format precision                      |
| (3) $q_0 = (a' \cdot y_0)_m$   | 82-bit register format precision                      |
| (4) $q = \text{trunc}(q_0)$  | floating point to signed integer conversion (RZ mode) |

The assembly language implementation:

```
.file "int8_div.s"
.section .text

// 8-bit signed integer divide

.proc int8_div#
.align 32
.global int8_div#
.align 32

int8_div:
{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 8-BIT SIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:
  //   r32 - 8-bit signed integer dividend
  //   r33 - 8-bit signed integer divisor
  //   r8 - 8-bit signed integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9
  // predicate registers used: p6

  sxtl r32=r32
  sxtl r33=r33;;
} { .mmb
  setf.sig f6=r32
  setf.sig f7=r33
```

```

    nop.b 0
} { .mlx
  nop.m 0
  // load 1+2^(-8)
  movl r2=0x3f808000;;
} { .mmi
  // 1+2^(-8) in f8
  setf.s f8=r2
  nop.m 0
  nop.i 0;;
} { .mfi
  nop.m 0
  fcvt.xf f6=f6
  nop.i 0
} { .mfi
  nop.m 0
  fcvt.xf f7=f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // (0) a' = a * (1 + 2^(-8)) in f8
  fma.sl f8=f6,f8,f0
  nop.i 0
} { .mfi
  nop.m 0
  // (1) y0
  frcpa.sl f9,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // (2) q0 = a' * y0
  (p6) fma.sl f9=f8,f9,f0
  nop.i 0;;
} { .mfb
  nop.m 0
  fcvt.fx.trunc.sl f6=f9
  nop.b 0;;
} { .mmi
  // quotient will be in the least significant 8 bits of r8 (if b != 0)
  getf.sig r8=f6
  nop.m 0
  nop.i 0;;
}

// 8-BIT SIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp int8_div

```

### Sample test driver:

```

#include<stdio.h>

char int8_div(char,char);

void run_test(char ia, char ib, char iq)
{
  char q;

  q=int8_div(ia,ib);

```

```

printf("\nNumerator: %x\nDenominator: %x\nQuotient: %x\n",ia,ib,iq);
if(iq==q) printf("Passed\n");
else printf("Failed (%x)\n",q);
}

void main()
{
    /* -7/3=-2 */
    run_test(-7,3,-2);

    /* -1/3=0 */
    run_test(-1,3,0);

    /* 64/32=2 */
    run_test(64,32,2);
}

```

## 4.2. Unsigned 8-bit Integer Divide

An algorithm very similar to the one shown above calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 8-bit unsigned integers. The only significant differences are the format conversions (before starting the computation, the arguments are zero-extended to 64 bits, rather than sign-extended).  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $a' = (a + a \cdot 2^{-8})_m$                                      | 82-bit register format precision                        |
| (3) $q_0 = (a' \cdot y_0)_m$   | 82-bit register format precision                        |
| (4) $q = \text{trunc}(q_0)$  | floating point to unsigned integer conversion (RZ mode) |

The assembly language implementation:

```

.file "uint8_div.s"
.section .text

// 8-bit unsigned integer divide

.proc uint8_div#
.align 32
.global uint8_div#
.align 32

uint8_div:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 8-BIT UNSIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:
  //   r32 - 8-bit unsigned integer dividend

```

```

//    r33 - 8-bit unsigned integer divisor
//    r8 - 8-bit unsigned integer result
//    r2 - scratch register
// floating-point registers used: f6, f7, f8, f9
// predicate registers used: p6

zxtl r32=r32
zxtl r33=r33;;
} { .mmb
setf.sig f6=r32
setf.sig f7=r33
nop.b 0
} { .mlx
nop.m 0
// load  $1+2^{(-8)}$ 
movl r2=0x3f808000;;
} { .mmi
//  $1+2^{(-8)}$  in f8
setf.s f8=r2
nop.m 0
nop.i 0;;
} { .mfi
nop.m 0
fcvt.xf f6=f6
nop.i 0
} { .mfi
nop.m 0
fcvt.xf f7=f7
nop.i 0;;
} { .mfi
nop.m 0
// (0)  $a' = a * (1 + 2^{(-8)})$  in f8
fma.sl f8=f6,f8,f0
nop.i 0
} { .mfi
nop.m 0
// (1) y0
frcpa.sl f9,p6=f6,f7
nop.i 0;;
} { .mfi
nop.m 0
// (2)  $q0 = a' * y0$ 
(p6) fma.sl f9=f8,f9,f0
nop.i 0;;
} { .mfb
nop.m 0
fcvt.fxu.trunc.sl f6=f9
nop.b 0;;
} { .mmi
// quotient will be in the least significant 8 bits of r8 (if b != 0)
getf.sig r8=f6
nop.m 0
nop.i 0;;
}

// 8-BIT UNSIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
nop.m 0
nop.m 0
br.ret.sptk b0;;
}

.endp uint8_div

```

Sample test driver:

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

```

unsigned char uint8_div(unsigned char, unsigned char);

void run_test(unsigned char ia, unsigned char ib, unsigned char iq)
{
    unsigned char q;

    q = uint8_div(ia, ib);

    printf("\nNumerator: %x\nDenominator: %x\nQuotient: %x\n", ia, ib, iq);
    if (iq == q) printf("Passed\n");
    else printf("Failed (%x)\n", q);
}

void main()
{
    /* 7/3=2 */
    run_test(7, 3, 2);

    /* 1/3=0 */
    run_test(1, 3, 0);

    /* 64/32=2 */
    run_test(64, 32, 2);
}

```

### 4.3. Signed 8-bit Integer Remainder

The following algorithm, based on signed 8-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 8-bit signed integers.  $m$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $a' = (a + a \cdot 2^{-8})_m$                                      | 82-bit register format precision                      |
| (3) $q_0 = (a' \cdot y_0)_m$   | 82-bit register format precision                      |
| (4) $q = \text{trunc}(q_0)$  | floating point to signed integer conversion (RZ mode) |
| (5) $r = a + (-b) \cdot q$   | integer operation                                     |

The assembly language implementation:

```

.file "int8_rem.s"
.section .text

// 8-bit signed integer divide

.proc int8_rem#
.align 32
.global int8_rem#
.align 32

int8_rem:
{ .mmi

```

```

    alloc r31=ar.pfs,2,0,0,0
    nop.m 0
    nop.i 0;;
} { .mii
    nop.m 0

    // 8-BIT SIGNED INTEGER DIVIDE BEGINS HERE

    // general register used:
    //   r32 - 8-bit signed integer dividend
    //   r33 - 8-bit signed integer divisor
    //   r8 - 8-bit signed integer result
    //   r2 - scratch register
    // floating-point registers used: f6, f7, f8, f9, f10
    // predicate registers used: p6

    sxtl r32=r32
    sxtl r33=r33;;
} { .mmb
    setf.sig f10=r32
    setf.sig f7=r33
    nop.b 0
} { .mlx
    nop.m 0
    // load  $1+2^{(-8)}$ 
    movl r2=0x3f808000;;
} { .mmi
    //  $1+2^{(-8)}$  in f8
    setf.s f8=r2
    nop.m 0
    nop.i 0;;
} { .mfi
    // get 2's complement of b
    sub r33=r0,r33
    fcvf.xf f6=f10
    nop.i 0
} { .mfi
    nop.m 0
    fcvf.xf f7=f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // (0)  $a' = a * (1 + 2^{(-8)})$  in f8
    fma.sl f8=f6,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // (1) y0
    frcpa.sl f9,p6=f6,f7
    nop.i 0;;
} { .mfi
    // get 2's complement of b
    setf.sig f7=r33
    // (2)  $q0 = a' * y0$ 
    (p6) fma.sl f9=f8,f9,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    fcvf.fx.trunc.sl f8=f9
    nop.i 0;;
} { .mfi
    nop.m 0
    //  $rem = a - (a/b) * b$ 
    xma.l f8=f8,f7,f10
    nop.i 0;;
} { .mmi
    // remainder will be in the least significant 8 bits of r8 (if b != 0)
    getf.sig r8=f8
    nop.m 0
    nop.i 0;;
}

```

```
// 8-BIT SIGNED INTEGER REMAINDER ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp int8_rem
```

#### Sample test driver:

```
#include<stdio.h>

char int8_rem(char,char);

void run_test(char ia,char ib,char iq)
{
  char q;

  q=int8_rem(ia,ib);

  printf("\nNumerator: %x\nDenominator: %x\nRemainder: %x\n",ia,ib,iq);
  if(iq==q) printf("Passed\n");
  else printf("Failed (%x)\n",q);
}

void main()
{
  /* -7%3=-1 */
  run_test(-7,3,-1);

  /* 1%3=1 */
  run_test(1,3,1);

  /* -64%32=0 */
  run_test(-64,32,0);
}
```

#### 4.4. Unsigned 8-bit Integer Remainder

The following algorithm, based on unsigned 8-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 8-bit unsigned integers.  $rn$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $a' = (a + a \cdot 2^{-8})_{rn}$                                   | 82-bit register format precision                        |
| (3) $q_0 = (a' \cdot y_0)_{rn}$  | 82-bit register format precision                        |
| (4) $q = \text{trunc}(q_0)$  | floating point to unsigned integer conversion (RZ mode) |
| (5) $r = a + (-b) \cdot q$   | integer operation                                       |

The assembly language implementation:

```
.file "uint8_rem.s"
.section .text

// 8-bit unsigned integer divide

.proc uint8_rem#
.align 32
.global uint8_rem#
.align 32

uint8_rem:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 8-BIT UNSIGNED INTEGER REMAINDER BEGINS HERE

  // general register used:
  //   r32 - 8-bit unsigned integer dividend
  //   r33 - 8-bit unsigned integer divisor
  //   r8  - 8-bit unsigned integer result
  //   r2  - scratch register
  // floating-point registers used: f6, f7, f8, f9, f10
  // predicate registers used: p6

  zxtl r32=r32
  zxtl r33=r33;;
} { .mmb
  setf.sig f10=r32
  setf.sig f7=r33
  nop.b 0
} { .mlx
  nop.m 0
  // load  $1+2^{(-8)}$ 
  movl r2=0x3f808000;;
} { .mmi
  //  $1+2^{(-8)}$  in f8
  setf.s f8=r2
  nop.m 0
  nop.i 0;;
} { .mfi
  // get 2's complement of b
  sub r33=r0,r33
  fcvt.xf f6=f10
  nop.i 0
} { .mfi
  nop.m 0
  fcvt.xf f7=f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // (0)  $a' = a * (1 + 2^{(-8)})$  in f8
  fma.sl f8=f6,f8,f0
  nop.i 0
} { .mfi
  nop.m 0
  // (1) y0
  frcpa.sl f9,p6=f6,f7
  nop.i 0;;
} { .mfi
  // get 2's complement of b
  setf.sig f7=r33
```



```

    // (2) q0 = a' * y0
    (p6) fma.s1 f9=f8,f9,f0
    nop.i 0;;
} { .mfi
  nop.m 0
  fcvt.fxu.trunc.s1 f8=f9
  nop.i 0;;
} { .mfi
  nop.m 0
  // rem = a - (a/b) * b
  xma.l f8=f8,f7,f10
  nop.i 0;;
} { .mmi
  // remainder will be in the least significant 8 bits of r8 (if b != 0)
  getf.sig r8=f8
  nop.m 0
  nop.i 0;;
}

    // 8-BIT UNSIGNED INTEGER REMAINDER ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp uint8_rem

```

### Sample test driver:

```

#include<stdio.h>

unsigned char uint8_rem(unsigned char,unsigned char);

void run_test(unsigned char ia,unsigned char ib,unsigned char iq)
{
    unsigned char q;

    q=uint8_rem(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nRemainder: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 7%3=1 */
    run_test(7,3,1);

    /* 1%3=1 */
    run_test(1,3,1);

    /* 64%32=0 */
    run_test(64,32,0);
}

```

#### 4.5. Signed 16-bit Integer Divide

The following algorithm calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 16-bit signed integers.  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_m$                              | 82-bit register format precision                      |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (5) $q = \text{trunc}(q_1)$  | floating point to signed integer conversion (RZ mode) |

The assembly language implementation:

```
.file "intl6_div.s"
.section .text

// 16-bit signed integer divide

.proc intl6_div#
.align 32
.global intl6_div#
.align 32

intl6_div:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 16-BIT SIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:
  //   r32 - 16-bit signed integer dividend
  //   r33 - 16-bit signed integer divisor
  //   r8 - 16-bit signed integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9
  // predicate registers used: p6

  sxt2 r32=r32
  sxt2 r33=r33;;
} { .mmi
  setf.sig f6=r32
  setf.sig f7=r33
  nop.i 0
} { .mlx
  nop.m 0
  // load 1+2^(-17)
  movl r2=0x3f800040;;
} { .mfb
  // 1+2^(-17) in f8
  setf.s f8=r2
  fcvt.xf f6=f6
  nop.b 0
} { .mfb
  nop.m 0
```

```

    fcvt.xf f7=f7
    nop.b 0;;
} { .mfi
    nop.m 0
    // (1) y0
    frcpa.s1 f9,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // (2) q0 = a * y0
    (p6) fma.s1 f6=f9,f6,f0
    nop 0
} { .mfi
    nop 0
    // (3) e0 = 1+2^(-17) - b * y0
    (p6) fnma.s1 f7=f7,f9,f8
    nop 0;;
} { .mfi
    nop 0
    // (4) q1 = q0 + e0 * q0
    (p6) fma.s1 f9=f7,f6,f6
    nop 0;;
} { .mfb
    nop.m 0
    fcvt.fx.trunc.s1 f6=f9
    nop.b 0;;
} { .mmi
    // quotient will be in the least significant 16 bits of r8 (if b != 0)
    getf.sig r8=f6
    nop.m 0
    nop.i 0;;
}

    // 16-BIT SIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
    nop.m 0
    nop.m 0
    br.ret.sptk b0;;
}

.endp int16_div

```

### Sample test driver:

```

#include<stdio.h>

short int16_div(short,short);

void run_test(short ia,short ib,short iq)
{
    short q;

    q=int16_div(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* -600/30=-20 */

```

```

run_test(-600,30,-20);

/* 1000/333=3 */
run_test(1000,333,3);

/* -2048/512=-4 */
run_test(-2048,512,-4);
}

```

#### 4.6. Unsigned 16-bit Integer Divide

An algorithm very similar to the one shown above calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 16-bit unsigned integers. The only significant differences are the format conversions (before starting the computation, the arguments are zero-extended to 64 bits, rather than sign-extended).  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                        |
| (3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_m$                              | 82-bit register format precision                        |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                        |
| (5) $q = \text{trunc}(q_1)$  | floating point to unsigned integer conversion (RZ mode) |

The assembly language implementation:

```

.file "uint16_div.s"
.section .text

// 16-bit unsigned integer divide

.proc uint16_div#
.align 32
.global uint16_div#
.align 32

uint16_div:
{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 16-BIT UNSIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:
  //   r32 - 16-bit unsigned integer dividend
  //   r33 - 16-bit unsigned integer divisor
  //   r8 - 16-bit unsigned integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9
  // predicate registers used: p6

  zxt2 r32=r32
  zxt2 r33=r33;;

```

```

    } { .mmi
        setf.sig f6=r32
        setf.sig f7=r33
        nop.i 0
    } { .mlx
        nop.m 0
        // load  $1+2^{(-17)}$ 
        movl r2=0x3f800040;;
    } { .mfb
        //  $1+2^{(-17)}$  in f8
        setf.s f8=r2
        fcvt.xf f6=f6
        nop.b 0
    } { .mfb
        nop.m 0
        fcvt.xf f7=f7
        nop.b 0;;
    } { .mfi
        nop.m 0
        // (1)  $y_0$ 
        frcpa.s1 f9,p6=f6,f7
        nop.i 0;;
    } { .mfi
        nop.m 0
        // (2)  $q_0 = a * y_0$ 
        (p6) fma.s1 f6=f9,f6,f0
        nop 0
    } { .mfi
        nop 0
        // (3)  $e_0 = 1+2^{(-17)} - b * y_0$ 
        (p6) fnma.s1 f7=f7,f9,f8
        nop 0;;
    } { .mfi
        nop 0
        // (4)  $q_1 = q_0 + e_0 * q_0$ 
        (p6) fma.s1 f9=f7,f6,f6
        nop 0;;
    } { .mfb
        nop.m 0
        fcvt.fxu.trunc.s1 f6=f9
        nop.b 0;;
    } { .mmi
        // quotient will be in the least significant 16 bits of r8 (if b != 0)
        getf.sig r8=f6
        nop.m 0
        nop.i 0;;
    }

    // 16-BIT UNSIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
    nop.m 0
    nop.m 0
    br.ret.sptk b0;;
}

.endp uint16_div

```

#### Sample test driver:

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

```
unsigned short uint16_div(unsigned short,unsigned short);
```

```
void run_test(unsigned short ia,unsigned short ib,unsigned short iq)
{
```

```

unsigned short q;

q=uint16_div(ia,ib);

printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
if(iq==q) printf("Passed\n");
else printf("Failed (%x)\n",q);

}

void main()
{

/* 600/30=20 */
run_test(600,30,20);

/* 1000/333=3 */
run_test(1000,333,3);

/* 2048/512=4 */
run_test(2048,512,4);

}

```

#### 4.7. Signed 16-bit Integer Remainder

The following algorithm, based on signed 16-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 16-bit signed integers.  $m$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_m$                              | 82-bit register format precision                      |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (5) $q = \text{trunc}(q_1)$  | floating point to signed integer conversion (RZ mode) |
| (6) $r = a + (-b) \cdot q$   | integer operation                                     |

The assembly language implementation:

```

.file "int16_rem.s"
.section .text

// 16-bit signed integer remainder

.proc int16_rem#
.align 32
.global int16_rem#
.align 32

int16_rem:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

```

```

// 16-BIT SIGNED INTEGER REMAINDER BEGINS HERE

// general register used:
//   r32 - 16-bit signed integer dividend
//   r33 - 16-bit signed integer divisor
//   r8  - 16-bit signed integer result
//   r2  - scratch register
// floating-point registers used: f6, f7, f8, f9, f10
// predicate registers used: p6

sxt2 r32=r32
sxt2 r33=r33;;
} { .mmi
  setf.sig f10=r32
  setf.sig f7=r33
  nop.i 0
} { .mlx
  nop.m 0
  // load  $1+2^{(-17)}$ 
  movl r2=0x3f800040;;
} { .mfb
  //  $1+2^{(-17)}$  in f8
  setf.s f8=r2
  fcvt.xf f6=f10
  nop.b 0
} { .mfb
  // get 2's complement of b
  sub r33=r0,r33
  fcvt.xf f7=f7
  nop.b 0;;
} { .mfi
  nop.m 0
  // (1)  $y_0$ 
  frcpa.sl f9,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop 0
  // (2)  $e_0 = 1+2^{(-17)} - b * y_0$ 
  (p6) fnma.sl f8=f7,f9,f8
  nop 0
} { .mfi
  nop.m 0
  // (3)  $q_0 = a * y_0$ 
  (p6) fma.sl f9=f9,f6,f0
  nop 0;;
} { .mfi
  // get 2's complement of b
  setf.sig f7=r33
  // (4)  $q_1 = e_0 * q_0 + q_0$ 
  (p6) fma.sl f9=f8,f9,f9
  nop 0;;
} { .mfb
  nop.m 0
  fcvt.fx.trunc.sl f8=f9
  nop.b 0;;
} { .mfi
  nop.m 0
  //  $rem = a - (a/b) * b$ 
  xma.l f8=f8,f7,f10
  nop.i 0;;
} { .mmi
  // remainder will be in the least significant 16 bits of r8 (if b != 0)
  getf.sig r8=f8
  nop.m 0
  nop.i 0;;
}

// 16-BIT SIGNED INTEGER REMAINDER ENDS HERE

{ .mmb
  nop.m 0

```

```

        nop.m 0
        br.ret.sptk b0;;
    }

    .endp int16_rem

```

Sample test driver:

```

#include<stdio.h>

short int16_rem(short,short);

void run_test(short ia,short ib,short iq)
{
    short q;

    q=int16_rem(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 600%-30=0 */
    run_test(600,-30,0);

    /* -1000%333=-1 */
    run_test(-1000,333,-1);

    /* 2052%512=4 */
    run_test(2052,512,4);
}

```

#### 4.8. Unsigned 16-bit Integer Remainder

The following algorithm, based on unsigned 16-bit integer divide, calculates  $r=a \bmod b$ , where  $a$  and  $b$  are 16-bit unsigned integers.  $m$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 + 2^{-17} - b \cdot y_0)_m$                              | 82-bit register format precision                      |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (5) $q = \text{trunc}(q_1)$  | floating point to signed integer conversion (RZ mode) |
| (6) $r = a + (-b) \cdot q$   | integer operation                                     |

The assembly language implementation:



```

.file "uint16_rem.s"
.section .text

// 16-bit unsigned integer remainder

.proc uint16_rem#
.align 32
.global uint16_rem#
.align 32

uint16_rem:

{ .mmi
  alloc r31=ar.pfs,2,0,0,0
  nop.m 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 16-BIT UNSIGNED INTEGER REMAINDER BEGINS HERE

  // general register used:
  //   r32 - 16-bit unsigned integer dividend
  //   r33 - 16-bit unsigned integer divisor
  //   r8 - 16-bit unsigned integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9, f10
  // predicate registers used: p6

  zxt2 r32=r32
  zxt2 r33=r33;;
} { .mmi
  setf.sig f10=r32
  setf.sig f7=r33
  nop.i 0
} { .mlx
  nop.m 0
  // load  $1+2^{(-17)}$ 
  movl r2=0x3f800040;;
} { .mfb
  //  $1+2^{(-17)}$  in f8
  setf.s f8=r2
  fcvt.xf f6=f10
  nop.b 0
} { .mfb
  // get 2's complement of b
  sub r33=r0,r33
  fcvt.xf f7=f7
  nop.b 0;;
} { .mfi
  nop.m 0
  // (1)  $y_0$ 
  frcpa.s1 f9,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop 0
  // (2)  $e_0 = 1+2^{(-17)} - b * y_0$ 
  (p6) fnma.s1 f8=f7,f9,f8
  nop 0
} { .mfi
  nop.m 0
  // (3)  $q_0 = a * y_0$ 
  (p6) fma.s1 f9=f9,f6,f0
  nop 0;;
} { .mfi
  // get 2's complement of b
  setf.sig f7=r33
  // (4)  $q_1 = e_0 * q_0 + q_0$ 
  (p6) fma.s1 f9=f8,f9,f9
  nop 0;;
}

```

```

    } { .mfb
      nop.m 0
      fcvt.fxu.trunc.s1 f8=f9
      nop.b 0;;
    } { .mfi
      nop.m 0
      // rem = a - (a/b) * b
      xma.l f8=f8,f7,f10
      nop.i 0;;
    } { .mmi
      // remainder will be in the least significant 16 bits of r8 (if b != 0)
      getf.sig r8=f8
      nop.m 0
      nop.i 0;;
    }

    // 16-BIT UNSIGNED INTEGER REMAINDER ENDS HERE

    { .mmb
      nop.m 0
      nop.m 0
      br.ret.sptk b0;;
    }

.endp uint16_rem

```

#### Sample test driver:

```

#include<stdio.h>

unsigned short uint16_rem(unsigned short,unsigned short);

void run_test(unsigned short ia,unsigned short ib,unsigned short iq)
{
    unsigned short q;

    q=uint16_rem(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 600%30=0 */
    run_test(600,30,0);

    /* 1000%333=1 */
    run_test(1000,333,1);

    /* 2052%512=4 */
    run_test(2052,512,4);
}

```

## 4.9. Signed 32-bit Integer Divide

The following algorithm calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 32-bit signed integers.  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision                      |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (5) $e_1 = (e_0 \cdot e_0 + 2^{-34})_m$                                | 82-bit register format precision                      |
| (6) $q_2 = (q_1 + e_1 \cdot q_1)_m$                                    | 82-bit register format precision                      |
| (7) $q = \text{trunc}(q_2)$  | floating point to signed integer conversion (RZ mode) |

The assembly language implementation:

```
.file "int32_div.s"
.section .text

// 32-bit signed integer divide

.proc int32_div#
.align 32
.global int32_div#
.align 32

int32_div:

{ .mii
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 32-BIT SIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:
  //   r32 - 32-bit signed integer dividend
  //   r33 - 32-bit signed integer divisor
  //   r8 - 32-bit signed integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9
  // predicate registers used: p6

  sxt4 r32=r32
  sxt4 r33=r33;;
} { .mmb
  setf.sig f6=r32
  setf.sig f7=r33
  nop.b 0;;
} { .mfi
  nop.m 0
  fcvt.xf f6=f6
  nop.i 0
} { .mfi
  nop.m 0
  fcvt.xf f7=f7
  mov r2 = 0x0ffdd;;
} { .mfi
  setf.exp f9 = r2
  // (1) y0
```

```

    frcpa.s1 f8,p6=f6,f7
    nop.i 0;;
} { .mfi
  nop.m 0
  // (2) q0 = a * y0
  (p6) fma.s1 f6=f6,f8,f0
  nop.i 0
} { .mfi
  nop.m 0
  // (3) e0 = 1 - b * y0
  (p6) fnma.s1 f7=f7,f8,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // (4) q1 = q0 + e0 * q0
  (p6) fma.s1 f6=f7,f6,f6
  nop.i 0
} { .mfi
  nop.m 0
  // (5) e1 = e0 * e0 + 2^-34
  (p6) fma.s1 f7=f7,f7,f9
  nop.i 0;;
} { .mfi
  nop.m 0
  // (6) q2 = q1 + e1 * q1
  (p6) fma.s1 f8=f7,f6,f6
  nop.i 0;;
} { .mfi
  nop.m 0
  fcvt.fx.trunc.s1 f8=f8
  nop.i 0;;
} { .mmi
  // quotient will be in the least significant 32 bits of r8 (if b != 0)
  getf.sig r8=f8
  nop.m 0
  nop.i 0;;
}

// 32-BIT SIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp int32_div

```

### Sample test driver:

```

#include<stdio.h>

int int32_div(int,int);

void run_test(int ia,int ib,int iq)
{
  int q;

  q=int32_div(ia,ib);

  printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
  if(iq==q) printf("Passed\n");
  else printf("Failed (%x)\n",q);
}

```

```

void main()
{
    /* 70000/-70=1000 */
    run_test(70000,-70,-1000);

    /* -1000/333=-3 */
    run_test(-1000,333,-3);

    /* 2^20/512=2048 */
    run_test(1<<20,512,2048);
}

```

#### 4.10. Unsigned 32-bit Integer Divide

An algorithm very similar to the one shown above calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 32-bit unsigned integers. The only significant differences are the format conversions (before starting the computation, the arguments are zero-extended to 64 bits, rather than sign-extended).  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                        |
| (3) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision                        |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                        |
| (5) $e_1 = (e_0 \cdot e_0 + 2^{-34})_m$                                | 82-bit register format precision                        |
| (6) $q_2 = (q_1 + e_1 \cdot q_1)_m$                                    | 82-bit register format precision                        |
| (7) $q = \text{trunc}(q_2)$  | floating point to unsigned integer conversion (RZ mode) |

The assembly language implementation:

```

.file "uint32_div.s"
.section .text

// 32-bit unsigned integer divide

.proc uint32_div#
.align 32
.global uint32_div#
.align 32

uint32_div:
{ .mii
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 32-BIT UNSIGNED INTEGER DIVIDE BEGINS HERE

  // general register used:

```

```

//      r32 - 32-bit unsigned integer dividend
//      r33 - 32-bit unsigned integer divisor
//      r8 - 32-bit unsigned integer result
//      r2 - scratch register
// floating-point registers used: f6, f7, f8, f9
// predicate registers used: p6

zxt4 r32=r32
zxt4 r33=r33;;
} { .mmb
setf.sig f6=r32
setf.sig f7=r33
nop.b 0;;
} { .mfi
nop.m 0
fcvt.xf f6=f6
nop.i 0
} { .mfi
nop.m 0
fcvt.xf f7=f7
mov r2 = 0x0ffdd;;
} { .mfi
setf.exp f9 = r2
// (1) y0
frcpa.s1 f8,p6=f6,f7
nop.i 0;;
} { .mfi
nop.m 0
// (2) q0 = a * y0
(p6) fma.s1 f6=f6,f8,f0
nop.i 0
} { .mfi
nop.m 0
// (3) e0 = 1 - b * y0
(p6) fnma.s1 f7=f7,f8,f1
nop.i 0;;
} { .mfi
nop.m 0
// (4) q1 = q0 + e0 * q0
(p6) fma.s1 f6=f7,f6,f6
nop.i 0
} { .mfi
nop.m 0
// (5) e1 = e0 * e0 + 2^-34
(p6) fma.s1 f7=f7,f7,f9
nop.i 0;;
} { .mfi
nop.m 0
// (6) q2 = q1 + e1 * q1
(p6) fma.s1 f8=f7,f6,f6
nop.i 0;;
} { .mfi
nop.m 0
fcvt.fxu.trunc.s1 f8=f8
nop.i 0;;
} { .mmi
// quotient will be in the least significant 32 bits of r8 (if b != 0)
getf.sig r8=f8
nop.m 0
nop.i 0;;
}

// 32-BIT UNSIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
nop.m 0
nop.m 0
br.ret.sptk b0;;
}

.endp uint32_div

```

Sample test driver:

```
#include<stdio.h>

unsigned uint32_div(unsigned,unsigned);

void run_test(unsigned ia,unsigned ib,unsigned iq)
{
    unsigned q;

    q=uint32_div(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 70000/70=-1000 */
    run_test(70000,70,1000);

    /* -1000/333=-3 */
    run_test(1000,333,3);

    /* 2^20/512=2048 */
    run_test(1<<20,512,2048);
}
```

#### 4.11. Signed 32-bit Integer Remainder

The following algorithm, based on signed 32-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 32-bit signed integers.  $m$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision                      |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (5) $e_1 = (e_0 \cdot e_0 + 2^{-34})_m$                                | 82-bit register format precision                      |
| (6) $q_2 = (q_1 + e_1 \cdot q_1)_m$                                    | 82-bit register format precision                      |
| (7) $q = \text{trunc}(q_2)$  | floating point to signed integer conversion (RZ mode) |
| (8) $r = a + (-b) \cdot q$   | integer operation                                     |

The assembly language implementation:

```
.file "int32_rem.s"
.section .text

// 32-bit signed integer remainder

.proc int32_rem#
.align 32
.global int32_rem#
.align 32

int32_rem:

{ .mii
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 32-BIT SIGNED INTEGER REMAINDER BEGINS HERE

  // general register used:
  //   r32 - 32-bit signed integer dividend
  //   r33 - 32-bit signed integer divisor
  //   r8 - 32-bit signed integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9, f10, f11
  // predicate registers used: p6

  sxt4 r32=r32
  sxt4 r33=r33;;
} { .mmb
  setf.sig f11=r32
  setf.sig f7=r33
  nop.b 0;;
} { .mfi
  // get 2's complement of b
  sub r33=r0,r33
  fcvt.xf f6=f11
  nop.i 0
} { .mfi
  nop.m 0
  fcvt.xf f7=f7
  mov r2 = 0x0ffdd;;
} { .mfi
  setf.exp f9 = r2
  // (1) y0
  frcpa.s1 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // (2) q0 = a * y0
  (p6) fma.s1 f10=f6,f8,f0
  nop.i 0
} { .mfi
  nop.m 0
  // (3) e0 = 1 - b * y0
  (p6) fnma.s1 f8=f7,f8,f1
  nop.i 0;;
} { .mfi
  // 2's complement of b
  setf.sig f7=r33
  // (4) q1 = q0 + e0 * q0
  (p6) fma.s1 f10=f8,f10,f10
  nop.i 0
} { .mfi
  nop.m 0
  // (5) e1 = e0 * e0 + 2^-34
```



```

    (p6) fma.s1 f8=f8,f8,f9
    nop.i 0;;
} { .mfi
  nop.m 0
  // (6) q2 = q1 + e1 * q1
  (p6) fma.s1 f8=f8,f10,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  fcvt.fx.trunc.s1 f8=f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // rem = a - (a/b) * b
  xma.l f8=f8,f7,f11
  nop.i 0;;
} { .mmi
  // remainder will be in the least significant 32 bits of r8 (if b != 0)
  getf.sig r8=f8
  nop.m 0
  nop.i 0;;
}

// 32-BIT SIGNED INTEGER REMAINDER ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp int32_rem

```

### Sample test driver:

```

#include<stdio.h>

int int32_rem(int,int);

void run_test(unsigned ia,unsigned ib,unsigned iq)
{
    int q;

    q=int32_rem(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 70000%-70=0 */
    run_test(70000,-70,0);

    /* -1000%333=-1 */
    run_test(-1000,333,-1);

    /* 2^20%512=0 */
    run_test(1<<20,512,0);
}

```

#### 4.12. Unsigned 32-bit Integer Remainder

The following algorithm, based on unsigned 32-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 32-bit unsigned integers.  $rm$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_{rm}$   | 82-bit register format precision                        |
| (3) $e_0 = (1 - b \cdot y_0)_{rm}$                                     | 82-bit register format precision                        |
| (4) $q_1 = (q_0 + e_0 \cdot q_0)_{rm}$                                 | 82-bit register format precision                        |
| (5) $e_1 = (e_0 \cdot e_0 + 2^{-34})_{rm}$                             | 82-bit register format precision                        |
| (6) $q_2 = (q_1 + e_1 \cdot q_1)_{rm}$                                 | 82-bit register format precision                        |
| (7) $q = \text{trunc}(q_2)$  | floating point to unsigned integer conversion (RZ mode) |
| (8) $r = a + (-b) \cdot q$   | integer operation                                       |

The assembly language implementation:

```
.file "uint32_rem.s"
.section .text

// 32-bit unsigned integer remainder

.proc uint32_rem#
.align 32
.global uint32_rem#
.align 32

uint32_rem:

{ .mii
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
} { .mii
  nop.m 0

  // 32-BIT UNSIGNED INTEGER REMAINDER BEGINS HERE

  // general register used:
  //   r32 - 32-bit unsigned integer dividend
  //   r33 - 32-bit unsigned integer divisor
  //   r8 - 32-bit unsigned integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9, f10, f11
  // predicate registers used: p6

  zxt4 r32=r32
  zxt4 r33=r33;;
} { .mmb
  setf.sig f11=r32
  setf.sig f7=r33
  nop.b 0;;
} { .mfi
  nop.m 0
```

```

    fcvt.xf f6=f11
    nop.i 0
} { .mfi
    // get 2's complement of b
    sub r33=r0,r33
    fcvt.xf f7=f7
    mov r2 = 0x0ffdd;;
} { .mfi
    setf.exp f9 = r2
    // (1) y0
    frcpa.sl f8,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // (2) q0 = a * y0
    (p6) fma.sl f10=f6,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // (3) e0 = 1 - b * y0
    (p6) fnma.sl f8=f7,f8,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // (4) q1 = q0 + e0 * q0
    (p6) fma.sl f10=f8,f10,f10
    nop.i 0
} { .mfi
    // get 2's complement of b
    setf.sig f7=r33
    // (5) e1 = e0 * e0 + 2^-34
    (p6) fma.sl f8=f8,f8,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    // (6) q2 = q1 + e1 * q1
    (p6) fma.sl f8=f8,f10,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    fcvt.fxu.trunc.sl f8=f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // rem = a - (a/b) * b
    xma.l f8=f8,f7,f11
    nop.i 0;;
} { .mmi
    // remainder will be in the least significant 32 bits of r8 (if b != 0)
    getf.sig r8=f8
    nop.m 0
    nop.i 0;;
}

    // 32-BIT UNSIGNED INTEGER REMAINDER ENDS HERE

{ .mmb
    nop.m 0
    nop.m 0
    br.ret.sptk b0;;
}

.endp uint32_rem

```

Sample test driver:

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

```

unsigned uint32_rem(unsigned,unsigned);

void run_test(unsigned ia,unsigned ib,unsigned iq)
{
    unsigned q;

    q=uint32_rem(ia,ib);

    printf("\nNumerator: %x\nDenominator: %x\nResult: %x\n",ia,ib,iq);
    if(iq==q) printf("Passed\n");
    else printf("Failed (%x)\n",q);
}

void main()
{
    /* 70000%70=0 */
    run_test(70000,70,0);

    /* 1000%333=1 */
    run_test(1000,333,1);

    /* 2^20%512=0 */
    run_test(1<<20,512,0);
}

```

#### 4.13. Signed 64-bit Integer Divide

The following algorithm calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 64-bit signed integers.  $m$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

- |  |   |
|--|---|
| (1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$ | table lookup  |
| (2) $q_0 = (a \cdot y_0)_m$  | 82-bit register format precision                      |
| (3) $e_0 = (1 - b \cdot y_0)_m$  | 82-bit register format precision                      |
| (4) $e_1 = (e_0 \cdot e_0)_m$  | 82-bit register format precision                      |
| (5) $q_1 = (q_0 + e_0 \cdot q_0)_m$                                    | 82-bit register format precision                      |
| (6) $y_1 = (y_0 + e_0 \cdot y_0)_m$                                    | 82-bit register format precision                      |
| (7) $q_2 = (q_1 + e_1 \cdot q_1)_m$                                    | 82-bit register format precision                      |
| (8) $y_2 = (y_1 + e_1 \cdot y_1)_m$                                    | 82-bit register format precision                      |
| (9) $r_2 = (a - b \cdot q_2)_m$  | 82-bit register format precision                      |
| (10) $q_3 = (q_2 + r_2 \cdot y_2)_m$                                   | 82-bit register format precision                      |
| (11) $q = \text{trunc}(q_3)$   | floating point to signed integer conversion (RZ mode) |

The assembly language implementation:

```

.file "int64_div.s"
.section .text
.proc int64_div#
.align 32
.global int64_div#

```

```

.align 32

// 64-bit signed integer divide

int64_div:

{ .mii
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
} { .mmi

  // 64-BIT SIGNED INTEGER DIVIDE BEGINS HERE

  setf.sig f8=r32
  setf.sig f9=r33
  nop.i 0;;
} { .mfb
  nop.m 0
  fcvt.xf f6=f8
  nop.b 0
} { .mfb
  nop.m 0
  fcvt.xf f7=f9
  nop.b 0;;
} { .mfi
  nop.m 0
  // Step (1)
  //  $y_0 = 1 / b$  in f8
  frcpa.s1 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  //  $e_0 = 1 - b * y_0$  in f9
  (p6) fnma.s1 f9=f7,f8,f1
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  //  $q_0 = a * y_0$  in f10
  (p6) fma.s1 f10=f6,f8,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  //  $e_1 = e_0 * e_0$  in f11
  (p6) fma.s1 f11=f9,f9,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (5)
  //  $q_1 = q_0 + e_0 * q_0$  in f10
  (p6) fma.s1 f10=f9,f10,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6)
  //  $y_1 = y_0 + e_0 * y_0$  in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  //  $q_2 = q_1 + e_1 * q_1$  in f9
  (p6) fma.s1 f9=f11,f10,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  //  $y_2 = y_1 + e_1 * y_1$  in f8

```

```

    (p6) fma.s1 f8=f11,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (9)
    // r2 = a - b * q2 in f10
    (p6) fnma.s1 f10=f7,f9,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // q3 = q2 + r2 * y2 in f8
    (p6) fma.s1 f8=f10,f8,f9
    nop.i 0;;
} { .mfb
    nop.m 0
    fcvt.fx.trunc.s1 f8=f8
    nop.b 0;;
} { .mmi
    // quotient will be in r8 (if b != 0)
    getf.sig r8=f8
    nop.m 0
    nop.i 0;;
}

    // 64-BIT SIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
    nop.m 0
    nop.m 0
    br.ret.sptk b0;;
}

.endp int64_div

```

### Sample test driver:

```

#include<stdio.h>

typedef struct { unsigned low,high; } uint64;

uint64 int64_div(uint64,uint64);

void run_test(unsigned ial,unsigned ia0,unsigned ibl,unsigned ib0,unsigned
iq1,unsigned iq0)
{
    uint64 a,b,q;

    a.low=ia0; a.high=ial;
    b.low=ib0; b.high=ibl;

    q=int64_div(a,b);

    printf("\nNumerator: %08x%08x\nDenominator: %08x%08x\nResult:
%08x%08x\n",ial,ia0,ibl,ib0,iq1,iq0);
    if(iq0==q.low && iq1==q.high) printf("Passed\n");
    else printf("Failed (%08x%08x)\n",q.high,q.low);
}

void main()
{
    /* (2^62+1)/2^32=2^30 */
    run_test(0x40000000,0x00000001,0x00000001,0x00000000,0x00000000,0x40000000);
}

```

```

/* -1/1=-1 */
run_test(0xffffffff,0xffffffff,0x00000000,0x00000001,0xffffffff,0xffffffff);

/* (2^62+2^61)/3=2^61 */
run_test(0x60000000,0x00000000,0x00000000,0x00000003,0x20000000,0x00000000);
}

```

#### 4.14. Unsigned 64-bit Integer Divide

An algorithm very similar to the one shown above calculates  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , where  $a$  and  $b$  are 64-bit unsigned integers. The only significant differences are the format conversions (the last step is a conversion to unsigned 64-bit integer).  $rn$  is the IEEE round to nearest mode. All other symbols used are 82-bit, register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $q_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(3) $e_0 = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(4) $e_1 = (e_0 \cdot e_0)_{rn}$	82-bit register format precision
(5) $q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(6) $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit register format precision
(7) $q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(8) $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	82-bit register format precision
(9) $r_2 = (a - b \cdot q_2)_{rn}$	82-bit register format precision
(10) $q_3 = (q_2 + r_2 \cdot y_2)_{rn}$	82-bit register format precision
(11) $q = \text{trunc}(q_3)$	floating point to unsigned integer conversion (RZ mode)

The assembly language implementation:

```

.file "uint64_div.s"
.section .text
.proc uint64_div#
.align 32
.global uint64_div#
.align 32

// 64-bit unsigned integer divide

uint64_div:

{ .mi
  alloc r31=ar.pfs,2,0,0,0
  nop.i 0
  nop.i 0;;
}

{ .mmi

  // 64-BIT UNSIGNED INTEGER DIVIDE BEGINS HERE

  setf.sig f8=r32
  setf.sig f9=r33
  nop.i 0;;
} { .mfb

```

```

    nop.m 0
    fma.s1 f6=f8,f1,f0
    nop.b 0
} { .mfb
    nop.m 0
    fma.s1 f7=f9,f1,f0
    nop.b 0;;
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1 / b in f8
    frcpa.s1 f8,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // e0 = 1 - b * y0 in f9
    (p6) fnma.s1 f9=f7,f8,f1
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // q0 = a * y0 in f10
    (p6) fma.s1 f10=f6,f8,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // e1 = e0 * e0 in f11
    (p6) fma.s1 f11=f9,f9,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // q1 = q0 + e0 * q0 in f10
    (p6) fma.s1 f10=f9,f10,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (6)
    // y1 = y0 + e0 * y0 in f8
    (p6) fma.s1 f8=f9,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // q2 = q1 + e1 * q1 in f9
    (p6) fma.s1 f9=f11,f10,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // y2 = y1 + e1 * y1 in f8
    (p6) fma.s1 f8=f11,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (9)
    // r2 = a - b * q2 in f10
    (p6) fnma.s1 f10=f7,f9,f6
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (10)
    // q3 = q2 + r2 * y2 in f8
    (p6) fma.s1 f8=f10,f8,f9
    nop.i 0;;
} { .mfb
    nop.m 0
    fcvt.fxu.trunc.s1 f8=f8
    nop.b 0;;

```



```

} { .mmi
  // quotient will be in r8 (if b != 0)
  getf.sig r8=f8
  nop.m 0
  nop.i 0;;
}

// 64-BIT UNSIGNED INTEGER DIVIDE ENDS HERE

{ .mmb
  nop.m 0
  nop.m 0
  br.ret.sptk b0;;
}

.endp uint64_div

```

#### Sample test driver:

```

#include<stdio.h>

typedef struct { unsigned low,high; } uint64;

uint64 uint64_div(uint64,uint64);

void run_test(unsigned ial,unsigned ia0,unsigned ibl,unsigned ib0,unsigned
iq1,unsigned iq0)
{
  uint64 a,b,q;

  a.low=ia0; a.high=ial;
  b.low=ib0; b.high=ibl;

  q=uint64_div(a,b);

  printf("\nNumerator: %08x%08x\nDenominator: %08x%08x\nResult:
%08x%08x\n",ial,ia0,ibl,ib0,iq1,iq0);
  if(iq0==q.low && iq1==q.high) printf("Passed\n");
  else printf("Failed (%08x%08x)\n",q.high,q.low);
}

void main()
{
  /* (2^62+1)/2^32=2^30 */
  run_test(0x40000000,0x00000001,0x00000001,0x00000000,0x00000000,0x40000000);

  /* (2^64-1)/1=2^64-1 */
  run_test(0xffffffff,0xffffffff,0x00000000,0x00000001,0xffffffff,0xffffffff);

  /* (2^63+2^62)/3=2^62 */
  run_test(0xc0000000,0x00000000,0x00000000,0x00000003,0x40000000,0x00000000);
}

```

#### 4.15. Signed 64-bit Integer Remainder

The following algorithm, based on signed 64-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 64-bit signed integers.  $rn$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $q_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(3) $e_0 = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(4) $e_1 = (e_0 \cdot e_0)_{rn}$	82-bit register format precision
(5) $q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(6) $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit register format precision
(7) $q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(8) $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	82-bit register format precision
(9) $r_2 = (a - b \cdot q_2)_{rn}$	82-bit register format precision
(10) $q_3 = (q_2 + r_2 \cdot y_2)_{rn}$	82-bit register format precision
(11) $q = \text{trunc}(q_3)$	floating point to unsigned integer conversion (RZ mode)
(12) $r = a + (-b) \cdot q$	integer operation

The assembly language implementation:

```
.file "int64_rem.s"
.section .text

// 64-bit signed integer remainder

.proc int64_rem#
.align 32
.global int64_rem#
.align 32

int64_rem:
{ .mii
  alloc r31=ar.pfs,3,0,0,0
  nop.i 0
  nop.i 0
} { .mmb

  // 64-BIT SIGNED INTEGER REMAINDER BEGINS HERE

  // general register used:
  //   r32 - 32-bit signed integer dividend
  //   r33 - 32-bit signed integer divisor
  //   r8 - 32-bit signed integer result
  //   r2 - scratch register
  // floating-point registers used: f6, f7, f8, f9, f10, f11, f12
  // predicate registers used: p6

  setf.sig f12=r32 // holds a in integer form
  setf.sig f7=r33
  nop.b 0
} { .mlx
  nop.m 0
  //movl r2=0x8000000000000000;;
  movl r2=0xffffffffffffffff;;
} { .mfi
  // get the 2's complement of b
  sub r33=r0,r33
  fcvt.xf f6=f12
```

```

    nop.i 0
} { .mfi
    nop.m 0
    fcvt.xf f7=f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (1)
    // y0 = 1 / b in f8
    frcpa.s1 f8,p6=f6,f7
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (2)
    // q0 = a * y0 in f10
    (p6) fma.s1 f10=f6,f8,f0
    nop.i 0
} { .mfi
    nop.m 0
    // Step (3)
    // e0 = 1 - b * y0 in f9
    (p6) fnma.s1 f9=f7,f8,f1
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (4)
    // q1 = q0 + e0 * q0 in f10
    (p6) fma.s1 f10=f9,f10,f10
    nop.i 0
} { .mfi
    nop.m 0
    // Step (5)
    // e1 = e0 * e0 in f11
    (p6) fma.s1 f11=f9,f9,f0
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (6)
    // y1 = y0 + e0 * y0 in f8
    (p6) fma.s1 f8=f9,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (7)
    // q2 = q1 + e1 * q1 in f9
    (p6) fma.s1 f9=f11,f10,f10
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (8)
    // y2 = y1 + e1 * y1 in f8
    (p6) fma.s1 f8=f11,f8,f8
    nop.i 0;;
} { .mfi
    nop.m 0
    // Step (9)
    // r2 = a - b * q2 in f10
    (p6) fnma.s1 f10=f7,f9,f6
    nop.i 0;;
} { .mfi
    setf.sig f7=r33
    // Step (10)
    // q3 = q2 + r2 * y2 in f8
    (p6) fma.s1 f8=f10,f8,f9
    nop.i 0;;
} { .mfi
    nop.m 0
    fcvt.fx.trunc.s1 f8=f8
    nop.i 0;;
} { .mfi
    nop.m 0

```

```

    // r=a+q*(-b)
    xma.l f8=f8,f7,f12
    nop.i 0;;
} { .mib
  getf.sig r8=f8
  nop.i 0
  nop.b 0
}

// 64-BIT SIGNED INTEGER REMAINDER ENDS HERE

{ .mib
  nop.m 0
  nop.i 0
  br.ret.sptk b0;;
}

.endp int64_rem

```

#### Sample test driver:

```

#include<stdio.h>

typedef struct { unsigned low,high; } uint64;

uint64 int64_rem(uint64,uint64);

void run_test(unsigned ial,unsigned ia0,unsigned ibl,unsigned ib0,unsigned
iq1,unsigned iq0)
{
  uint64 a,b,q;

  a.low=ia0; a.high=ial;
  b.low=ib0; b.high=ibl;

  q=int64_rem(a,b);

  printf("\nNumerator: %08x%08x\nDenominator: %08x%08x\nResult:
    %08x%08x\n",ial,ia0,ibl,ib0,iq1,iq0);
  if(iq0==q.low && iq1==q.high) printf("Passed\n");
  else printf("Failed (%08x%08x)\n",q.high,q.low);
}

void main()
{
  /* (2^62+1)%2^32=1 */
  run_test(0x40000000,0x00000001,0x00000001,0x00000000,0x00000000,0x00000001);

  /* -1%1=0 */
  run_test(0xffffffff,0xffffffff,0x00000000,0x00000001,0x00000000,0x00000000);

  /* (2^62+2^61)%3=0 */
  run_test(0x60000000,0x00000000,0x00000000,0x00000003,0x00000000,0x00000000);
}

```

#### 4.16. Unsigned 64-bit Integer Remainder

The following algorithm, based on unsigned 64-bit integer divide, calculates  $r = a \bmod b$ , where  $a$  and  $b$  are 64-bit unsigned integers.  $rn$  is the IEEE round to nearest mode.  $q = \left\lfloor \frac{a}{b} \right\rfloor$ , and all other symbols used are 82-bit register format numbers. The precision used for each step is shown below.

(1) $y_0 = 1 / b \cdot (1 + \epsilon_0)$ , $ \epsilon_0  < 2^{-8.886}$	table lookup
(2) $q_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(3) $e_0 = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(4) $e_1 = (e_0 \cdot e_0)_{rn}$	82-bit register format precision
(5) $q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(6) $y_1 = (y_0 + e_0 \cdot y_0)_{rn}$	82-bit register format precision
(7) $q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(8) $y_2 = (y_1 + e_1 \cdot y_1)_{rn}$	82-bit register format precision
(9) $r_2 = (a - b \cdot q_2)_{rn}$	82-bit register format precision
(10) $q_3 = (q_2 + r_2 \cdot y_2)_{rn}$	82-bit register format precision
(11) $q = \text{trunc}(q_3)$	floating point to unsigned integer conversion (RZ mode)
(12) $r = a + (-b) \cdot q$	integer operation

The assembly language implementation:

```
.file "uint64_rem.s"
.section .text

    // 64-bit unsigned integer remainder

.proc uint64_rem#
.align 32
.global uint64_rem#
.align 32

uint64_rem:
{ .mii
  alloc r31=ar.pfs,3,0,0,0
  nop.i 0
  nop.i 0
} { .mmb

    // 64-BIT UNSIGNED INTEGER REMAINDER BEGINS HERE

    // general register used:
    //   r32 - 32-bit unsigned integer dividend
    //   r33 - 32-bit unsigned integer divisor
    //   r8 - 32-bit unsigned integer result
    //   r2 - scratch register
    // floating-point registers used: f6, f7, f8, f9, f10, f11, f12
    // predicate registers used: p6

    setf.sig f12=r32 // holds a in integer form
    setf.sig f7=r33
    nop.b 0;;
} { .mfi
    // get 2's complement of b
    sub r33=r0,r33
    fcvt.xuf.s1 f6=f12
    nop.i 0
} { .mfi
    nop.m 0
    fcvt.xuf.s1 f7=f7
```

```

    nop.i 0;;
} { .mfi
  nop.m 0
  // Step (1)
  //  $y_0 = 1 / b$  in f8
  frcpa.s1 f8,p6=f6,f7
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (2)
  //  $q_0 = a * y_0$  in f10
  (p6) fma.s1 f10=f6,f8,f0
  nop.i 0
} { .mfi
  nop.m 0
  // Step (3)
  //  $e_0 = 1 - b * y_0$  in f9
  (p6) fnma.s1 f9=f7,f8,f1
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (4)
  //  $q_1 = q_0 + e_0 * q_0$  in f10
  (p6) fma.s1 f10=f9,f10,f10
  nop.i 0
} { .mfi
  nop.m 0
  // Step (5)
  //  $e_1 = e_0 * e_0$  in f11
  (p6) fma.s1 f11=f9,f9,f0
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (6)
  //  $y_1 = y_0 + e_0 * y_0$  in f8
  (p6) fma.s1 f8=f9,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (7)
  //  $q_2 = q_1 + e_1 * q_1$  in f9
  (p6) fma.s1 f9=f11,f10,f10
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (8)
  //  $y_2 = y_1 + e_1 * y_1$  in f8
  (p6) fma.s1 f8=f11,f8,f8
  nop.i 0;;
} { .mfi
  nop.m 0
  // Step (9)
  //  $r_2 = a - b * q_2$  in f10
  (p6) fnma.s1 f10=f7,f9,f6
  nop.i 0;;
} { .mfi
  // f7=-b
  setf.sig f7=r33
  // Step (10)
  //  $q_3 = q_2 + r_2 * y_2$  in f8
  (p6) fma.s1 f8=f10,f8,f9
  nop.i 0;;
} { .mfi
  nop.m 0
  fcvt.fxu.trunc.s1 f8=f8
  nop.i 0;;
} { .mfi
  nop.m 0
  xma.l f8=f8,f7,f12
  nop.i 0;;
} { .mib

```

```

    getf.sig r8=f8
    nop.i 0
    nop.b 0
}

// 64-BIT UNSIGNED INTEGER REMAINDER ENDS HERE

{ .mib
  nop.m 0
  nop.i 0
  br.ret.sptk b0;;
}

.endp uint64_rem

```

### Sample test driver:

```

#include<stdio.h>

typedef struct { unsigned low,high; } uint64;

uint64 uint64_rem(uint64,uint64);

void run_test(unsigned ia1,unsigned ia0,unsigned ib1,unsigned ib0,unsigned
iq1,unsigned iq0)
{
    uint64 a,b,q;

    a.low=ia0; a.high=ia1;
    b.low=ib0; b.high=ib1;

    q=uint64_rem(a,b);

    printf("\nNumerator: %08x%08x\nDenominator: %08x%08x\nResult:
%08x%08x\n",ia1,ia0,ib1,ib0,iq1,iq0);
    if(iq0==q.low && iq1==q.high) printf("Passed\n");
    else printf("Failed (%08x%08x)\n",q.high,q.low);
}

void main()
{
    /* (2^62+1)%2^32=1 */
    run_test(0x40000000,0x00000001,0x00000001,0x00000000,0x00000000,0x00000001);

    /* (2^64-1)%1=0 */
    run_test(0xffffffff,0xffffffff,0x00000000,0x00000001,0x00000000,0x00000000);

    /* (2^63+2^62)%3=0 */
    run_test(0xc0000000,0x00000000,0x00000000,0x00000003,0x00000000,0x00000000);
}

```

## 5. References

- [1] ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985.
- [2] *Intel(R) IA-64 Architecture Software Developer's Manual*, Intel Corporation, 2000.
- [3] Cornea-Hasegan, M., *proving IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms*, Intel Technology Journal, Q2 , 1998, at <http://developer.intel.com/technology/itj/q21998.htm>
- [4] Cornea-Hasegan, M. and Golliver, R., Markstein, P., *Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms*, Proceedings of the 14<sup>th</sup> IEEE Symposium on Computer Arithmetic, 1999, IEEE Computer Society, Los Alamitos, CA, pp. 96-105.
- [5] Cornea-Hasegan, M. and Norin, B., *IA-64 Floating Point Operations and the IEEE Standard for Binary Floating-Point Arithmetic*, Intel Technology Journal, Q4, 1999, at <http://developer.intel.com/technology/itj/index.htm>
- [6] Cornea-Hasegan, M., Iordache, C., Harrison, J. and Markstein, P., *Integer Divide and Remainder Operations in the IA-64 Architecture*, submitted to the 4<sup>th</sup> Conference on Real Numbers and Computers, Germany, April 2000.