AN IEEE FLOATING POINT ARITHMETIC IMPLEMENTATION

by Kari Johnsen

Norwegian Computing Center (NCC)
Oslo, Norway

ABSTRACT

This article describes some of the methods and algorithms used in an implementation of floating point arithmetic following (almost) the IEEE standard defined in (1). The description is more directly algorithm-oriented than the 'Implementation Guide' for this standard (2), since the latter does not treat an actual implementation.

The article consists of two parts. One concerns the problem of getting the 'preliminary result' from each of the arithmetic operations, this result is the basis for a correct rounding.

The other part treats the multiword arithmetic, i.e. the routines to perform an m x n bit operation, using the corresponding n bit operation supplied by the hardware. Only multiplication and division are described, since the add/subtract routines are trivial. For division also a method for getting an n bit inverse is included, since the hardware in the case had no division operation.

INTRODUCTION

The project and the hardware

The floating point arithmetic implementation is part of the Mach-S project at the Norwegian Computing Center. The aim of this project is to build a fast work station dedicated to the execution of SIMULA programs. The prototype hardware is the S-2000 terminal developed by the company Simulation Excel (SIM-X) A.S in Oslo. It is designed as a page composer, but is easily adapted to other purposes both through microprogramming and through modification of the hardware. It is reasonably fast (200 ns microcycle) and internally the machine contains a number of cooperating processing units, which serve to speed up the execution. The arithmetic routines are microprogrammed on the two 'bottom levels', called the P- and M-level. Regarded from the next level (F) they are single instructions which form part of the 'Slang' instruction set on this level. Slang is an expression-stack and data-object oriented instruction set, which is tailored for execution of 'S-code', which again is the intermediate language between the analysing part and the machine specific code generator in a semi-portable SIMULA system.

The P- and M-level also contain routines for format-conversion, integer arithmetic, comparisons and transport of values to/from the F-level. The sequencing is done on P-level, while the M-level does the actual arithmetic work. The M-level processor contains a 16-bit Am 2901 ALU, an Am 29517 16x16 bit multiplier, two registerfiles, a RAM that holds the arithmetic value stack and a 16-bit databus to communicate between these devices. The microinstruction word has separate bits to control the ALU and the multiplier, so that these can operate in parallell.

The IEEE standard, and value representation

The implementation does not in every respect conform to the IEEE standard as this is defined in (1). The differences all concern what might be called the 'exception cases'. We do not use the concepts NaN (Not A Number), infinity or denormalized numbers, and accordingly do not define operations on these, as IEEE does. However, in normal calculations where no overflow, underflow or divide by zero occurs, we produce the results prescribed by IEEE. When these exceptions occur, they are trapped and handled by a higher level in the machine.

All the four rounding modes: truncate, round to nearest, toward +infinity and toward -infinity are implemented.

Both the single and double basic format are supported, and also a single and double extended format. The table below shows the width in bits of the various fields of these formats.

	Sign (S)	Exponent (E)	Int. part (J)	Fraction (F)
Single basic	1	8	not repr.	23
Double ,,	1	11	,,	52
Single ext.	1	15	1	31
Double ,,	1	15	1	63

For the basic formats these widths are defined in IEEE, for the extended they may be chosen by the implementor, as long as the extended format is wider than the corresponding basic format. Our extended formats are chosen so that the mantissa, which consists of the J an F fields, is contained in a whole number of 16-bit words, two for reals and four for long reals.

In all the formats the exponent is biased with the value 2**(w-1) - 1, where w is the width of the E-field. In the extended formats the sign and the exponent together occupy one 16-bit word.

The interpretation of the value represented in these formats will differ slightly from IEEE since we do not represent NaN's, infinities and denormalized numbers. Bit patterns that in IEEE would represent these quantities will here be interpreted as normal numbers in the basic formats, sometimes not used at all in the extended. Of course IEEE's normal numbers are interpreted as such.

Our interpretation of the value V is the following:

Basic format:

```
if E=0 & F=0
then V = (-1)**S * 0  (signed zero)
else V = (-1)**S * 2**(E-bias) * 1.F
```

Extended format:

```
if E=0 & J=0 & F=0
then V = (-1)**S * 0 (signed zero)
else if  J=1
     then V = (-1)**S * 2**(E-bias) * 1.F
     else the combination is not in use
```

On the internal stack in the arithmetic processor the values are always represented in the extended formats. Conversion between single and double is done by special functions, if a mixed mode operation is to be performed, the F-level will first call one of these, so that the arithmetic function will operate either on two real or on two long real operands. Thus each arithmetic function will need one routine for reals and another for long reals. These are of course similar with respect to method and algorithms.

ACCURACY AND ROUNDING

The IEEE standard requires the arithmetic operations to be performed 'as if correct to infinite precision, then rounded according to the specifications'. However, to do this rounding, it is of course not necessary to know every bit of the infinitely precise result. The 'Implementation Guide' (2) defines a 'preliminary result', which is a compressed form of the infinitely precise result, containing enough information to round correctly. Beyond the LSB (the least significant bit to go into the destination format) the preliminary result contains three bits: the guard bit G, the round bit R and the sticky bit S. The two first are equal to the corresponding bits in the infinitely precise result, the last is the 'or' of all the following bits.

To round correctly, in fact only one correct bit is needed between LSB and S. The guard bit is supplied since the preliminary result will sometimes have to be shifted one bit to the left to be normalized, if so, we still have the correct round bit R beyond LSB.

An exact R is needed only in the default rounding mode, round to nearest, where the result shall differ from the infinitely precise result by at most half a unit in the LSB position. The sticky bit S is then needed to decide whether this difference is exactly a half, in which case round to even is required. The three other rounding modes only need to know if the difference is zero or not, thus the GRS-bits are sufficient to round correctly in these modes too.

Though the preliminary result is defined in terms of the infinitely precise result, it is not always possible to compute it this way, since it may be impractical (or impossible) to compute the latter. The method for getting a correct preliminary result will differ for each arithmetic operation, hence we shall look at each operation separately.

Addition/subtraction.

When the exponents of the two operands are different, one operand must be unnormalized, i.e. shifted N bits to the right, where N is the exponent difference. When N is great, it would at least take some complicated programming to handle such very long operands, which may occupy far more than the available number of registers. Instead, the following procedure will work:

> During the operand shift, put a 'barrier' behind the sticky bit position, which shall mean that each bit that would have been shifted beyond this barrier, instead is 'or'ed into S. In this way however large N may be, this 'compressed operand' just occupies three bits more than the original operand, these are contained in one 16-bit word. The other operand is supplied with trailing zeroes, then the addition or subtraction is performed, giving a result of one word extra length. This is the correct preliminary result.

To see this, we shall look at the operation performed in both ways. The asterisk marks the LSB position, 'V' is the overflow bit that may be generated by addition.

With infinite precision:

```
                          *
Unshifted operand:    xxxxxxxx00000
Shifted operand:           yyyyyzzz
Inf. precise result:  VrrrrrrrrGRsss
Preliminary result:   VrrrrrrrrGRS        S=or(sss)
```

With compressed operand:

```
                          *
Unshifted operand:    xxxxxxxx000
Shifted operand:           yyyyyZ         Z=or(zzz)
Result:               rrrrrrrrGRS
```

In both cases we have the equivalences:

(zzz <> 0)

<=>

(S = 1)

<=>

(borrow from R-position in case of subtraction)

which shows that the last result is equal to the preliminary result.

This preliminary result may then have to be normalized. In case of addition, if V=1, the result must be shifted one bit to the right. Ideally the R-bit should be 'or'ed into S. Actually, since the GRS-bits are contained in a 16-bit word, S will not be shifted out by a one-bit shift. This is sufficient for the rounding routine, which is only interested in the leftmost bit of this word, and whether the remaining 15 bits are all zero or not.

After a subtraction, more than one of the leftmost bits may become zero, and the result must then be shifted a number of bits to the left to become normalized. One might ask: What if the sticky bit, which is inexact, is shifted into the significant part of the result? We shall see that if so, the sticky bit is not inexact, in fact it will be zero and no bits have been 'or'ed into it.

Let the mantissas be A and B, and let B be shifted K bits to the right before the subtraction. The result is then A - 2**(-K)*B. Since A and B are normalized mantissas, with the binary point to the right of the leftmost bit, we have 1 <= A,B < 2. Let us assume K >= 2. This implies:

A - 2**(-K)*B > 1 - 2**(-2)*2 = 1/2.

So if the sticky bit is really inexact, which means that K >= 4, then the result will have to be shifted at most one bit to the left. The guard bit G will move into the significant part of the result, while the R and S bits stay beyond. This is sufficient to perform the rounding correctly.

Multiplication

As for the construction of a preliminary result, multiplication is the most straightforward operation, since the infinitely precise result can be constructed in this case. The product of the two mantissas has a length which is the double of the mantissa length, it is computed accurately, the upper half is the significant part, and the lower half is compressed into the GRS-bits. The result may have to be shifted one bit to the left, so the guard bit G must be present.

Division

The division algorithm, which will be described later, contains a procedure DIGIT that takes a 'remainder' as input, and delivers 16 correct bits of the quotient as output, together with a new remainder. Called twice for reals, four times for long reals, DIGIT supplies enough bits for the significant part of the quotient mantissa.

The quotient may have an infinite number of bits, so that the infinitely precise result is impossible to construct. However, after each DIGIT-call the remainder is zero if and only if the quotient bits not yet computed are all zero, so that the 'or' of the remaining bits is equal to the 'or' of the bits in the remainder. Thus we can easily get a sticky bit when we have G and R, the question is if we have to execute DIGIT one time extra to get these two bits. We shall see that this is not necessary.

The dividend U and the divisor V are both normalized mantissas, thus, as described under subtraction, we have 1 <= U,V < 2, and therefore: 1/2 < U/V < 2, so that U/V will have one of the forms:

U/V = 1.xxxxxxxx (for U >= V)

U/V = 0.1xxxxxxx (for U < V)

The division algorithm regards U and V as integers, hence in the result the bit left of the binary point will be the least significant bit of a machine word.

We note that the quotient 2U/V differs from U/V only in the exponent. This suggests the following approach:

Set U' = U if U >= V
= 2U if U < V

Then U'/V will always have the form: 1.xxxxxxxx

and U'/V - 1 = (U' - V)/V will be: 0.xxxxxxxx

We therefore start the division with a remainder R = U' - V, compute the 'xxxxxxxx' to the length of a normal mantissa, then shift this value one bit right with a '1' shifted in from the left. This gives the normalized mantissa of U/V with one extra correct bit beyond the LSB, thus we have got the R-bit. Since this mantissa will not be left-shifted, we do not need the G-bit. Finally we get S from the remainder, as described above, and we have the normalized preliminary result, ready for rounding.

It may be mentioned that the code to set the initial remainder is rather simple:

```
R := U - V ;
if R < 0 then R := R + U, decrement exponent;
```

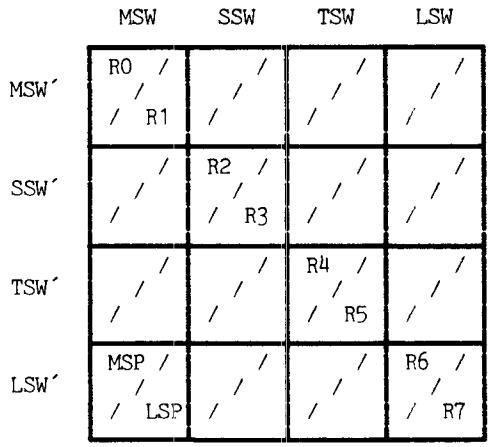
The division algorithm requires the initial remainder to be less than the divisor, R fulfills this condition, since R/V = 0.xxxxxxxx < 1.

THE MULTIWORD MANTISSA OPERATIONS

We shall study the algorithms for multiword arithmetic only for multiplication and division, for addition and subtraction the method with carry transfer is well-known. The long real case with 64-bit mantissas in four 16-bit registers is used as example in both cases.

Multiplication

The figure below may help to vizualize the word-by-word mantissa multiplication:



The mnemonics MSW, SSW, TSW, LSW denote Most, Second, Third and Least Significant Word of the mantissas of the two factors. Each square represents a 32-bit product, consisting of the two parts MSP (most significant 16 bit) and LSP (least significant 16 bit) as shown in the leftmost bottom square. Those MSP´s and LSP´s that lie on the same diagonal, between two slashed lines, will be added to the same accumulator register, this register is marked on the figure for each diagonal. The final product will then reside in the eight-register ´accumulator´ R0 - R7.

We have the following problem: Each time an LSP or MSP is added to an accumulator, a carry may be generated, this should be added to the next accumulator (above to the left), this addition may again generate a carry, and so on. We need a method that will ´swallow´ these carries without extra addition. The one chosen is taken from (3), page 233. The basis is the following procedure for one single 16 x 16 bit multiplication:

```
procedure M(F´, F, Rin, Rout, Qin, Qout);
begin
  P:=F´*F + Rin + Qin;
  Rout:=LSP(P);
  Qout:=MSP(P);
end M;
```

- F, F´ are the two 16-bit factors.
- Rin, Rout both refer to the accumulator register belonging to LSP of this product.
- Qin, Qout are 16-bit ´carries´. Qin is the MSP from some previous product, Qout is the MSP of the current product.

The procedure assumes that P is a 32-bit value, so that it may be parted into an MSP(P) and LSP(P). The proof is simple, since all the righthand values are 16-bit, we have:

```
P <= (2**16-1)*(2**16-1)+(2**16-1)+(2**16-1)
  = (2**32-1)
```

showing that P is a 32-bit value. (Knuth´s proof)

We may reformulate M in a form a little ´closer to the machine´:

```
procedure M(.....  );
begin
  Rout:=LSP(F´*F) + Rin + Qin;
  Qout:=MSP(F´*F) + carry + carry;
end M;
```

The carries are from the previous additions.

The above proof shows that MSP of the product will ´swallow´ the two carries without generating any further carry. These carry-additions need not be separate instructions. In the actual implementation Qin and Qout each consists of both a 16-bit value and a carry-condition, thus one ´+ carry´ is not performed, and the ´+ Qin´ is actually an ´add Qin plus carry´-instruction.

Referring to the above figure, if we now do the multiplications in each column from bottom to top, we see that Qout from one product may be used as Qin in the next, since they belong to the same register. At the bottom Qin=0, at the top Qout is a register where nothing yet has been accumulated, if we do the columns from right to left. The total multiplication then becomes:

```
M(LSW´, LSW, R7, R7, 0, Q );    Rightmost
M(TSW´, LSW, R6, R6, Q, Q );    column
M(SSW´, LSW, R5, R5, Q, Q );
M(MSW´, LSW, R4, R4, Q, R3);

M(LSW´, TSW, R6, R6, 0, Q );    Next to
M(TSW´, TSW, R5, R5, Q, Q );    rightmost
M(SSW´, TSW, R4, R4, Q, Q );    column
M(MSW´, TSW, R3, R3, Q, R2);

M(LSW´, SSW, R5, R5, 0, Q );    Next to
M(TSW´, SSW, R4, R4, Q, Q );    leftmost
M(SSW´, SSW, R3, R3, Q, Q );    column
M(MSW´, SSW, R2, R2, Q, R1);

M(LSW´, MSW, R4, R4, 0, Q );    Leftmost
M(TSW´, MSW, R3, R3, Q, Q );    column
M(SSW´, MSW, R2, R2, Q, Q );
M(MSW´, MSW, R1, R1, Q, R0);
```

Before this program starts, the registers R4-R7 must be initiated to zero.

The code for each column is quite similar, only different registers are involved. Code may be saved by using a common code sequence for the columns, this is possible by changing the actual registers associated with each accumulator, for each column. We note that Rout from line N in one column is Rin in line N-1 in the next column. Choosing the registers where the values are finally to end up, we construct the following procedure to handle one column:

```
procedure MM(F, Rout);
begin
  M(LSW', F, R3, Rout, 0, Q );
  M(TSW', F, R2, R3,   Q, Q );
  M(SSW', F, R1, R2,   Q, Q );
  M(MSW', F, R0, R1,   Q, R0);
end MM;
```

The total multiplication then becomes:

```
MM(LSW, R7);
MM(TSW, R6);
MM(SSW, R5);
MM(MSW, R4);
```

With this method, R0-R3 must be initiated to zero.

After the computation, the significant part of the product is in R0-R3, while R4-R7 is compressed into the GRS-bits, as described in 'Accuracy and rounding'.

Division

The division method is based on an algorithm taken from (3), page 237. We shall regard the dividend U, the divisor V and the quotient Q as consisting of 16-bit 'digits': U=u1,u2,u3,u4, V=v1,v2,v3,v4, Q=q1,q2,q3,q4, each digit then occupies one register.

The basis of the method is a procedure DIGIT, which generates one digit qi of Q. DIGIT operates on a remainder R, both as input and output. This remainder has one digit more than U, V: R=r0,r1,r2,r3,r4.

```
procedure DIGIT(R, qi);
begin qhat is a 16-bit value;
  r0:=r1;          multiply R by 2**16
  r1:=r2;               ,,
  r2:=r3;               ,,
  r3:=r4;               ,,
  r4:=0 ;               ,,
  guess a 'trial digit' qhat;
  R:=R -V*qhat;
  while R<0 do begin
    R:=R+V;
    qhat:=qhat-1;
  end;
  qi:=qhat;
end DIGIT;
```

Knuth's 'guess' of trial digit is the following:

```
qhat:=FLOOR(r0r1/v1);
if qhat >= 2**16 then qhat:=2**16-1;
```

'FLOOR(x)' means the integer part of x, i.e. the biggest integer not greater than x. Later we shall also use 'CEILING(x)', which denotes the smallest integer not less than x.

This trial digit then comes from dividing the two most significant digits of the remainder by the most significant digit of the divisor. Knuth shows that this qhat is never smaller than the correct digit, thus the while-loop in DIGIT only counts qhat down. He also shows that qhat is not more than 2 too big, provided that the most significant bit of v1 is '1', which is fulfilled in a normalized mantissa. Therefore the loop will execute twice at most, with this choice of qhat.

We now have the following problem: Knuth assumes the existence of a 16-bit division operation to compute r0r1/v1, but our machine has no such hardware operation. The quotient is computed 4 times, but we see that the divisor is the same, v1, each time. If we then had the inverse of v1, we could multiply by r0r1 to get qhat. The multiplication is simplest if the inverse is contained in 16 bits. The position of the binary point should be chosen so that these contain as many significant bits as possible. Regarding v1 as an integer, we have for all v1<>2**15:

```
2**15+1 <= v1 < 2**16
```

and so:

```
2**15 < 2**31/v1 <= 2**31/(2**15+1)
                  = 2**16 - 2 + 2/(2**15+1)
```

2**31/v1 is not an integer. Our multiplication factor must not produce a qhat less than Knuth's, if we are not to alter DIGIT and test for both too small and too big qhat. Thus we must use the CEILING of the value, and the above inequalities show that we have:

```
2**15 < CEILING(2**31/v1) <= 2**16-1
```

which means that CEILING(2**31/v1) is contained in 16 bits, and its most significant bit is '1', so we have got the maximum precision possible in 16 bits. For v1=2**15 it is simple to compute r0r1/v1, for all the other v1 we want a way of computing the value CEILING(2**31/v1). The method chosen is an iterative process often used to construct a division-operation, when a multiplication is available, see (4). The mathematical basis can shortly be described:

We regard the function $f(x)=2^{31}/x - v1$

and will determine x so that f(x)=0. We have $f'(x)= -2^{31}/x^2$ Given an initial guess x0 of x, we approximate f(x) by the tangent at x0, and set this to zero:

$f(x0) + f'(x0)*(x-x0) = 0$

Substituting f(x0) and f'(x0), we finally arrive at:

$x = x0*(2^{16} - x0*v1/2^{16})/2^{15}$

This gives a quickly converging iterative formula for x, as may be seen most easily from a function diagram of f(x) and the tangent. The formula is implemented by the following steps:

1. Multiply x0 and v1, discard least significant word of result.

2. Negate result (equal to 2**16 - result in two's complement arithm.)

3. Multiply by x0.

4. Shift result one bit left, keep upper 16 bit as final result.

The initial x0 is taken from a table, where the index consists of some (currently 7) of the most significant bits of v1. It seemed difficult to deduce in a purely mathematical way how many iterations would be necessary and how much the final x would differ from CEILING(2**31/v1). Also the steps in the calculation above are not all exact.

For safety reasons a test routine was constructed. For each v1<>2**15 this routine computes the inverse x by the iterative process described, then multiplies x by v1, and finds how much this product differs from 2**31 in multiples of v1, this gives the difference x - CEILING(2**31/v1). The routine also counts for how many v1's this difference is negative, 0, 1, 2, .. and so on. Thus one can experiment to find out how many iterations are necessary, and also check that the inverting procedure gives no values smaller than CEILING(2**31/v1). If the value is bigger, it may cause the while-loop in DIGIT to run some extra times. It is of interest to estimate an upper limit for the number of times this loop will run.

Knuth's qhat=FLOOR(r0r1/v1) may cause it to run twice, as proved by him. The value we use is qhat=FLOOR(r0r1*v1inv/2**31), where v1inv is the result from the iterative inverting procedure. The current routine, which uses 3 iterations (it will be changed later, when a better table for start values is available) is showed by the test routine to produce 1501 v1inv's that are one too big, the rest are equal to the value CEILING(2**31/v1). This latter value may be nearly one greater than 2**31/v1, thus:

$$0 < v1inv - 2^{31}/v1 < 2$$

Since $r0r1 < 2^{32}$, we get:

$$0 < r0r1*v1inv/2^{31} - r0r1/v1 <= 4$$

$$0 <= FLOOR(r0r1*v1inv/2^{31}) - FLOOR(r0r1/v1) <= 4$$

showing that our qhat is no more than 4 bigger than Knuth's, so the while-loop in DIGIT cannot run more than 6 times. (As yet, it has never been seen to run more than 3 times, usual has been one or zero in the examples tried till now.)

We may now describe the total division process, some of these steps were explained in 'Accuracy and rounding':

1. Given a dividend U=u1,u2,u3,u4 and a divisor V=v1,v2,v3,v4.

2. Get the inverse v1inv of v1, by the iterative inverting procedure.

3. Set the initial remainder R:
   R:=U-V if U >= V
   R:=2*U-V if U < V

4. Call DIGIT 4 times to get 4 x 16 bits of the quotient. The 'guess' of qhat is now:
   qhat:=FLOOR(r0r1*v1inv/2**31)
   if qhat >= 2**16 then qhat:=2**16-1;

5. Shift the result 1 bit to the right, with '1' in from the left. Set sticky bit equal to 'or' of the bits of the final remainder.

6. Round.

We see that actually two division algorithms are used:

The iterative gives an n-bit division, based on an n-bit multiplication.

Knuth's gives an m x n-bit division, based on an n-bit division.

It was considered to use only the first method to get the whole 64-bit inverse. This would however involve long composite multiplications, whereas in Knuth's algorithm just one 64 x 16-bit multiplication is needed in each DIGIT-call. Also the control of the accuracy seemed easier with the latter method, so it was finally decided to use both algorithms, as described.

REFERENCES

(1) A Proposed Standard for Binary Floating-Point Arithmetic.
Draft 8.0 of IEEE Task P754
COMPUTER, March 1981

(2) An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic
Jerome T. Coonen,
University of California at Berkeley
COMPUTER, January 1980

(3) Donald Knuth: The Art of Computer Programming, vol. 2: Seminumerical Algorithms

(4) Morris & Ibbett: The MU5 Computer System
Macmillan 1979