SOME TRICKS OF THE (FLOATING-POINT) TRADE

J.B. Gosling

Dept. of Computer Science, University of Manchester
Manchester, M13 9PL England

Abstract.

In designing a floating-point arithmetic unit there are a number of places where the characteristics of the operations and the operands permit simplifications in the logical design. These have not been well documented, with the result that each new generation of designers has made the same mistakes as their predecessors. This paper describes some of these simplifications as they affect the mantissa section of a floating-point addition and subtraction unit. The areas covered include normalisation and rounding.

1. Introduction.

When designing any system, one comes across places where things seem to be working to ones advantage. This paper discusses some of these areas. None of them individually is very startling, and they are probably not original. So much so that most of them have never been published, certainly not in one place. As a result, each new generation of floating-point hardware designers has to re-invent the wheel. The purpose of this contribution is to reduce the effort, and, more importantly, ensure as many as possible of the helpful effects may be used in new designs. It does not claim to be complete. In a recent period of three months the author 'discovered' many of the points made, inspite of 17 years in the 'business'. Any contributions that can be made by others will be gladly added to the list.

The discussion will be restricted to addition and subtraction. It will also be limited to numbers represented in the sign and modulus notation, because the proposed floating-point standard says so. And it will be limited to the mantissa area for reasons of space. Within these limitations, subtraction can be regarded as a special case of addition. If a subtraction is encountered, the sign of the 'second' operand is changed, and an addition is performed. Using similar rules
POINT 1: It is posible to include within the discusion all four of the operations A + B, A - B, -A + B and -A - B, where A and B are two signed numbers.
This is one of the advantages of the sign and modulus representation as compared to twos complement, since in twos complement the last of the four operations cannot be implemented easily.

2. Basic addition.

The basic algorithm for sign and modulus addition is [1]
If the signs of the operands are the same, add; goto end.
  Else invert one of the operands and add.
    If the carry from the most significant bit is a one, result := sum, sign := sign of the non-inverted operand.
    Else (if the carry from the most significant bit is zero), result := inverse of the sum, sign := sign of the inverted operand.
End.
The carry from the most significant bit must be added to the least significant bit, and is known as the End Around Carry (EAC).

When considering floating-point arithmetic, one of the operands will be shifted right relative to the other. In general, the bits 'spilt' will not take part in arithmetic. However, the EAC must be added at the correct point relative to this spill. Further, if the EAC = 0 the spill bits must be inverted before calculating whether to round or not. Things begin to look bad! But do not despair.

3. G, R and S bits.

POINT 2: It can be shown quite easily [1,2] that if the prearithmetic alignment shift is greater than one place, then the maximum normalisation that can be required is one place. If the alignment shift is zero or one place, then a large normalisation shift may be necessary, but
POINT 3: A maximum of one non-zero digit can be brought into the required result.
This digit is called the Guard digit, G. Its size is dependent on the exponent base, being 4 bits on ICL or IBM machines with their exponent base of 16. For binary exponents it is one bit. Thus, arithmetic should include this guard digit.

The next bit below G is known as the Round bit, R. If G is normalised into the result, then R indicates that the remaining bits of the pre-rounded result are greater than or equal to half the least significant bit. To determine whether this is exactly one half, the OR of all remaining bits of the spill is used to form the Spill bit, S. As indicated earlier,
POINT 4: R and S must be calculated taking into

account any inversions that may be performed on the operand, or the sum, or both.

As G is included in the arithmetic, consider, first of all, R and S alone. Addition of like-signed numbers is straight forward, so consider only the case of numbers with opposite signs. Suppose that the number which is inverted is the one which is NOT shifted. All bits beyond the "real" adder must be ones (zeros inverted), and hence the EAC is added into the least significant bit of the adder. It would seem that R and S could be formed from the spill. NOT TRUE.

```
          1.0110                      1.0110
          1.1101                      1.1101
Invert 1st 0.1001 111..               0.1001 111..
Shift R 2nd 0.0011 1010.  No shift R  1.1101 000..
                                    1 0.0111 0000.

          (a)                         (b)
```

Fig.1. Subtraction - non shifted operand inverted. (a) Align shift of 3; (b) No align shift.

Consider two normalised operands, one of which is to be shifted right, introducing zeros at the more significant end, Fig.1(a). The other operand is then inverted. The two most significant bits in the addition are zero, and there can be no EAC. To get an EAC there must be no alignment shift, and hence there can be no spill, Fig.1(b). When both operands are denormalised, the EAC may be 1 or 0, but the exponents are equal, and hence R = S = 0.
POINT 5:If the non-shifted operand is inverted, and EAC = 1, then R and S are both zero.
By a similar set of arguments it is found possible to derive
POINT 6:If the shifted operand is inverted, and the EAC = 0, then R and S are both zero, Fig.2.

```
          1.0110                      1.0110
          1.1101                      1.1101
Shift 2nd and invert      Invert 2nd
          1.1100 0101..               0.0010 111..
        1 1.0010 0110..             0 1.1000
                          Invert      ..... 000..

          (a)                         (b)
```

Fig.2. Subtraction - shifted operand inverted. (a)Align shift of 3; (b)No align shift.

R	S bits	"S"	R'	S' bits	R_f	S_f	Cin
0	0....0	0	1	1....1	0	0	0
0	d	1	0	d	1	1	1
0	0...01	1	0	0....0	1	1	1
1	0....0	0	0	1....1	1	0	1
1	d	1	1	d	0	1	1
1	0...01	1	1	0....0	0	1	1

Table 1. Generation of R and S: non-shifted operand inverted and EAC = 0.
d implies at least one 1 and at least one 0.

Table 1 shows what happens when the EAC = 0. A string of ones must be added to the spill. This spill may be zero, so there may be no carry to the adder. In this case R' = S' = 1 (the string of 1's from the inverted operand). In Table 1, "S" represents the S bit from the original operand, and R' and S' are the result of adding a string of 1's to R and S bits. The final R_f and S_f are derived from the inverse of R' and S', since the sum must be inverted when EAC = 0. From Table 1 it is seen that S will be 0 only when the original spill is zero, and the carry to the adder, Cin, is always one except when the original R and S are both zero.

When the shifted operand is inverted, R = S = 0 if EAC = 0. Table 2 shows what happens when EAC = 1. The EAC must be added at the bottom of the spill. The condition for EAC to reach the 'real' adder is shown. Again, S is zero only when the initial spill is all zeros.

R	S bits	"S"	R'	S' bits	R_f	S_f	Cin
0	0....0	0	1	1....1	0	0	1
0	d	1	1	d	1	1	0
0	1....1	1	1	0....0	1	1	0
1	0....0	0	0	1....1	1	0	0
1	d	1	0	d	0	1	0
1	1....1	1	0	0....0	0	1	0

Table 2. R and S generation; shifted operand inverted and EAC =1.

4.Generation of S.

So far it has been shown how R and S might be calculated. The question arises as to what happens if the exponent difference is very large, so that the alignment shift is very large. A lot of hardware is needed to collect the spill. Software implementations can cope, as can very slow hardware using shift registers, since it is possible to look at the spill as it goes past. For fast hardware, where semi-infinite logical shifters are too expensive, the following technique, due to Zurawski [3], is useful.

Suppose that the mantissa is p bits long, and the exponent difference is d. The mantissa of the smaller operand is shifted right by d places for the addition. To obtain the S bit
POINT 7: the smaller mantissa is shifted left by p + 2 - d places.
A shift of p - d places would place the total spill on the shifter output. The extra 2 places is to remove the G and R bits. Fig.3 illustrates this. If d is greater than p + 2 (i.e. a negative shift is demanded) then the shift becomes a circular shift. All mantissa bits are in the S area of the spill, and must be retained.

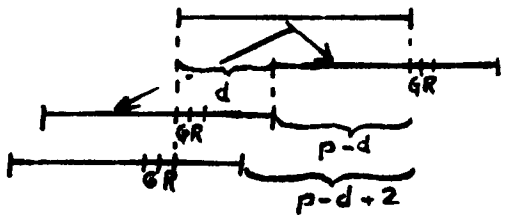

Fig.3. Derivation of S in a fast unit.

The cost of a second shifter is still prohibitive, and it is often difficult and/or

expensive to provide shared access or control. To reduce the cost, first perform an OR on groups of four bits in the mantissa to be shifted (four input gates are easily available). Now perform the left shift of p/4 bits by (p + 2 - d)/4 places. Unfortunately this figure is not an integer. Fig.4 shows how the end effects can be handled using specific numbers of bits. The required shift is calculated as

$$1 + p/4 - \lfloor 0.5 + d/4 \rfloor$$

Fig.4 shows the original shifting, where the above number is multiplied by 4. Three extra bits must be saved and OR-ed into the S generated from the shifter alone as will be seen underlined. Sometimes some of the bits are ORed in twice, which is acceptable.

Bit position	0	1	2	3	4	5	6	7	8	9	10	11	G	R	14	15	16
shift 8									0	1	2	3	4	5	6	7	8
left 8	8	9	10	11													
shift 9										0	1	2	3	4	5	6	7
left 8	8	9	10	11													
shift 10												0	1	2	3	4	5	6
left 4	4	5	6	7	8	9	10	11									
shift 11													0	1	2	3	4	5
left 4	4	5	6	7	8	9	10	11									

Fig.4. Cheaper derivation of S.

An interesting aside to this is that where a unit must handle several formats, as required by the proposed floating-point standard [4],
POINT 8: changing the value of p is sufficient to give the correct value of S in all cases.

5.Rounding.

To complete the discussion of rounding, the G, R and S bits must be modified by the normalisation requirements. In this process G disappears, and only R and S remain. When normalisation is not required, S := R V S, and R := G. If R and S are non-zero, then the biggest possible normalisation is one place (see para 3). For this one place shift, G is taken into the result, and R and S are unaltered. Finally, when the mantissa goes 'supernormal' it is shifted right by one place, and hence S := S V R V G, and R := LSBA, where LSBA is the least significant bit at the ADDER output.

Given the new values of R and S, directed roundings are straight forward. For positive sums and round to plus infinity, add one to the normalised sum if R V S. If the sum is negative, truncate. The reverse is true for round to minus infinity. For round to zero truncation is used regardless of sign.

Round to nearest even (RNE) is more difficult. Clearly a one must be added if R & S is true, and zero if R is false. If R & $\bar{S}$, then the sum must increase by 0.5 LSB if LSB is a one, or the error will be 1.5 LSBs. If the LSB is zero, however, the sum must be left alone. Hence the condition for adding one to the sum is R(S V LSBN), where LSBN is the least significant bit after normalisation. Note the difference between LSBA and LSBN.

Should this rounding add cause the mantissa to become too large, a further right shift will be necessary. However, the only case where this can occur is when the mantissa is cleared to zero except for the most significant bit.
POINT 9: Thus the new 'R' and 'S' must be zero.

6.Detection of zero.

Consider how a result of zero can be obtained. For the usual case of two normal numbers, zero can be obtained only if there is complete cancellation. This can only occur if there is no alignment shift. If one operand is denormalised, cancellation cannot occur. If both are denormalised, then the exponents are again equal. Hence
POINT 10: Detection of a zero mantissa is sufficient to detect a zero result.
Further, this can be done after the adder, rather than after normalisation (though it does not matter).

7.Underflow.

Underflow is defined to occur when a result is denormalised and inexact [4]. Cancellation of mantissae requires that the alignment shift should be zero or one. In either case the normalisation process would cause the result to be exact, and, therefore, not underflowed. This is also the case when both operands are denormalised. The limiting case is when the exponents are -125 and -126 (IEEE single). It is easily shown that this also gives an exact result. Hence
POINT 11:Addition and subtraction cannot underflow.

This paper has covered a number of topics usually left out of published work as being 'not worth publishing'. This writer (obviously) disagrees. It would be a service to arithmetic unit designers if this were to be extended to the many areas deliberately omitted. Perhaps one day.....

References.

[1] Gosling, J.B. 'Design of Arithmetic Units for Digital Computers' Macmillan, 1980.
[2] Sterbenz, P. 'Floating point Computation' Prentice-Hall, 1974.
[3] Zurawski, J.H.P. 'High performance evaluation of division and other elementary functions' PhD Thesis, University of Manchester, 1980.
[4] 'A proposed standard for binary floating point arithmetic' Draft 10.0 Dec 1982