DESIGN NOTE: Multiply and Divide Formats
DATE: November 14, 1969

from: Bruce Delagi

Reviewed by:

K-Project

Henry W. Spencer
11 Programming

11 Engineering Manager

15 Engineering Manager

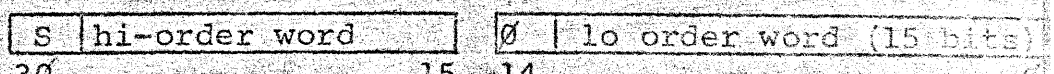
ABSTRACT

The effect of various data formats on signed and unsigned, double precision and floating point, addition, subtraction, multiplication, and division are examined. A rationale is presented for the choice of the non-zero format and signed (as opposed to unsigned) multiply and divide operations.

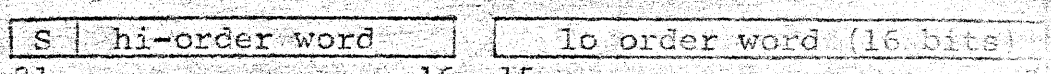
1.Ø Scope - This design note describes some alternate operations and formats for the extended arithmetic element instructions: multiply and divide.

2.Ø. Double Precision Data Format - "Zero" and "Non-Zero" Formats. Two alternative formats for signed double precision integers can be considered:

zero format:



non-zero format:



2.1. Formats in Double Precision Multiply - The zero format facilitates simulation of double precision signed multiplication on machines equipped with hardware, single precision signed multiply. Presuming that the double precision result of a multiplication is in the destination and the word following (in the case of registers - in the next register), a double precision multiplication sequence might be:

```
;full double precision signed multiply (4-word result) zero forma
;multiply (RØ,R1) by (R4,R5) and leave result in (RØ,R1, R2, R3)
;RØ and R4 are the low order words - result of MUL is in zero for

MULD:   MOV R4,-(SP)      ;save R4
        MOV R1,R2         ;multiply high order parts
        MUL R5,R2         ;R3 holds bits 45-6Ø
        MOV R2,-(SP)      ;R2 held bits 3Ø-44
        MUL R4,R1         ;get one cross product
        MOV R2,-(SP)      ;bits 3Ø-44
        MOV R1,-(SP)      ;bits 15-29
        MOV RØ, R1
        MUL R5,R1         ;get the other
        MOV R1,-(SP)      ;
        MUL R4,RØ         ;multiply lo-order parts
PSUMS:  ADD(SP)+,R1       ;add partial products
        ADD(SP)+,R1
        CLR R4
        ADD(SP)+,R2
        BPL .+4
        DEC R4
        ADD(SP)+R2
```

```
PCARRY:  BPL                    ;take care of carries in partials
         ADD R4,R3
PCNEXT:  TST R1
         BPL .+6
         INC R2
         ADC R3
         BIC #1ØØØØØ,R1         ;preserve format
         BIC #1ØØØØØ,R2
         MOV(SP)+,R4
         RTS PC
```

This works because in two's complement notation the double word integer A = (α,a) in zero format z is equal to $2^{k-1}\alpha+a$ (where k is the word length in bits) when A and a are taken as signed two's complement integers. The non-zero format representation of the unsigned integer B, (β,b) equals $2^k\beta+b$ where B and b are taken as unsigned integers. Thus the sequence from MULD to PSUMS would work for unsigned multiply in non-zero format if the MUL operation was taken as an unsigned multiply. However, the sequence beginning with PSUMS would be:

```
         ADD (SP)+,R1
         ADC R2
         ADC R3
         ADD (SP)+,R1
         ADC R2
         ADC R3
         ADD (SP)+,R2
         ADC R3
         ADD (SP)+,R2
         ADC R3
         RTS PC
```

which is a slightly (4 wds) shorter sequence. Thus it may be concluded that non-zero format double precision unsigned multiplication simulated with unsigned multiply instructions is about as efficient as zero format simulated with signed multiply instructions.

2.11 Signed vs Unsigned Multiply - The efficiency of zero and non-zero formats in simulation of double precision multiplication depends on whether the multiplication and the multiply operator are taken as unsigned. It is necessary to

to consider the requirement for both signed and unsigned multiplication.

The uses of unsigned multiplication lie chiefly in address arithmetic (e.g. the calculations to find the variable A(k,j) where j and the range of k are known). In address arithmetic, only the lo-order word of the product is of interest (2^{16} is the size of the address space) and it can be shown that the lo-order product of any two numbers is the same whether the multiplication is taken as signed or unsigned. Thus only the minority of applications for unsigned multiplication could not be as well met by signed multiplication.

A single precision signed multiplication may be performed with an unsigned multiply operator as follows:

```
        CLR SIGN
        MOV A,-(SP)
        BPL CHKB
        NEG (SP)
        INC SIGN
CHKB:   TST B
        BPL MULT
        NEG B
        DEC SIGN            ;sign is zero if product positive
MULT:   MUL (SP)+,B
        TST SIGN            ;if zero, multiplication done
        BEQ DONE
        NEG B+2             ;do double precision negate.
        NEG B
        SBC B+2
DONE:   ...
```

Similarly an unsigned multiplication may be performed with a signed multiply operator as follows: ($A = 2^{15} p+a$, $B=2^{15}q+b$)

```
        MOV B,-(SP)         ;push B
        MOV A,-(SP)         ;push A
        ASL (SP)
        ROR BITS            ;p = bit 15
        ROR (SP)            ;"a" on top of stack
        ASL B
        ROR BITS            ;p = bit 14, q = bit 15
        ROR B
        MUL A,B
```

```
        ASL B           ;double precision left shift
        ROL B+2
        ASL BITS
        BCS .+4         ;skip if q=1
        CLR (SP)
        ADD (SP)+,B+2   ;add in "a" (there can be no carry)
        ASL BITS        ;now checkp
        BCS .+4
        CLR (SP)
        ADD (SP)+,B+2
        ROR B+2         ;double precision shift with carry
        ROR B
```

It can be seen that it will take about 50% longer to do an unsigned single precision multiplication with a signed multiply operation than to do a signed single precision multiplication with an unsigned multiply. However, if only the lo-order word of the result is important or if the operands are known to be less than 2^{15}, signed and unsigned multipliers are equivalent.

operation \ operator	signed multiply	unsigned multiply
signed multiplication	roughly 8 us	roughly 26 us
unsigned multiplication	roughly 37 us	roughly 8 us

If signed and unsigned multiplication (exluding address arithmetic) were equally common, the results above would lead to choosing the unsigned multiply as more important than the signed multiply operation. However, it is felt that signed multiplication is a far more frequently used operation (excluding-address-arithmetic). Therefore, if a choice need be made, signed multiply is the preferred operation.

We will therefore consider signed double precision multiplication in zero and non-zero format as performed with a signed multiply instruction.

2.12 Double Precision Signed Multiplication in Non-Zero Format (MUL operation leaves two word product in Non-Zero Form)

If the only multiply operator is the signed-non-zero-format variety, the best approach to double precision signed multiplication is to convert to zero format before calculating

cross products. The partial products so computed will, however, not be aligned with one another so some shifting is required before adding them.

```
;full dbl prec signed multiply (4-word result) non-zero format
;multiply (RØ,R1) by (R4,R5) and leave result in(RØ,R1,R2,R3)
;RØ and R4 are the low order words-result of MUL is in non-zero form

MULDNZ: MOV R4,-(SP)      ;save old R4 and R5
        MOV R5,-(SP)
        ASL R4            ;convert (R4,R5) to zero-format
        ROL R5
        BVS OFLØ          ;(R4,R5) too large to convert to 31
        CLC                         bit zero format
        ROR R5
        ASL RØ            ;convert (RØ,R1) to zero format
        ROL R1
        BVS OFL1
        CLC
        ROR R1
MULD:   MOV R1,R2         ;proceed as in 2.1
        MUL R5,R2         ;R3 holds partial on bits 46-6Ø
        MOV R2,-(SP)      ;R2 holds partial on bits 3Ø-45
        MUL R4,R1         ;R1 holds partial on bits 15-3Ø
        MOV R2,-(SP)      ;R2 holds partial on bits 31-44
        MOV R1,-(SP)
        MOV RØ,R1
        MUL R5,R1         ;R2 holds partial on bits 31-44
        MOV R1,-(SP)      ;R1 holds partial on bits 15-3Ø
        MUL R4,RØ         ;R1 holds partial on bits 16-29
PSUMS:  ASL RØ            ;align lo-lo product with cross product
        ROR R1            ;a double precision left shift
        ADD (SP)+,R1      ;get lo order partial on cross product
        ADC R2            ;add carry into hi-order of cross prod.
        ADD (SP)+,R1      ;add other cross product (lo)
        ADC R2
        ADD (SP)+,R2      ;and hi
        ASL R1            ;align hi of cross partial into lo of h
        ROR R2            ;carry bit holds sign
        SBC R3            ;extend sign into R4
        ADD (SP)+,R2      ;add lo order part of hi-hi
        ADC R3
```

```
            ASR R3              ;now shift right to non-zero format
            ROR R2
            ROR R1
            ASR R3
            ROR R2
            ROR R1
            ROR RØ
            MOV (SP)+,R5        ;restore (R4,R5)
            MOV (SP)+,R4
            ADD #2,(SP)         ;go to normal return
            RTS PC

OFLØ:       ASR R4              ;handle overflow
            ROR R5
            RTS PC              ;go to error return

OFL1:       ASR RØ
            ROR R1
            RTS PC
```

Thus while zero-format double precision multiplication with a signed operator takes approximately 65us (presuming a 1 usec per memory reference processor with a 7 usec multiply) the non-zero format requires roughly twice the core and 85 usec - roughly a 3Ø% increase in time.

Including both signed and unsigned multiply operators does not help since the cross partials need to be taken as the product of signed integers in any case.

4.13 Double Precision Formats in Mantissa Calculations for Floating Point Multiply Simulation.

Full double precision multiply (i.e. with a 4 word result) is probably not as common as double precision multiplication in which the result is taken as the high order double word - the mantissa of the product in a floating point multiply. We will compare the zero and non-zero formats for this case as well.

```
;31 bit mantissa of the product in a floating point multiplicatio
;zero format multiply (RØ,R1) by (R4,R5)-result in (RØ,R1)
```

```
MULM:   MOV R4,-(SP)
        MOV R1, R2
        MUL R5,RØ          ;R1 holds hi of cross product
        MOV R2,RØ
        MUL R4,R2          ;R3 holds hi of other cross
        CLR R4
        ADD R1, R3
        BPL .+4
        DEC R4
        MUL R5,RØ
        ADD R3,RØ
        BPL .+4
        ADD R4,R1
        BIC #1ØØØØØ,RØ
        MOV (SP)+,R4
        RTS PC

;non-zero format multiply (RØ,R1) by (R4,R5)-result in (RØ,R1)

MULMNZ: CLC                ;convert R4-R5 to zero format
        ROR R4             ;R4-R5 represents a fraction
        CLC
        ROR RØ
        MOV R1, R2
        MUL R5, RØ         ;R1 holds hi of cross product
        MOV R2,RØ
        MUL R4,R2          ;R3 holds hi of other cross
        ADD R1, R3
        MUL R5,RØ
        ASL R3
        SBC R1
        ADD R3,RØ
        ADC R1
        ASL R4             ;restore (R4,R5)
        RTS PC
```

Thus the mantissa operations in a floating point multiply for zero format will take approximately 4Ø usec and for non-zero format will take approximately 35 usec - actually shorter. This is chiefly so because doing double precision addition is more convenient in non-zero format - unless a special double precision add instruction is postulated as well.

2.2 Formats in Double Precision Add and Subtract -

The non-zero format facilitates double precision addition and subtraction:

```
;double precision addition - non-zero format
        ADD AØ,BØ          ;add lo-order
        ADC B1             ;add carry
        ADD A1, B1         ;add hi-order

;double precision subtraction - non-zero format
        SUB AØ,BØ
        SBC B1
        SUB A1, B1

;double precision addition - zero format
        ADD AØ,BØ
ADCARY: BPL ADDHI
        BIC #1ØØØØØ,BØ
        INC B1
ADDHI   ADD A1, B1
```

and similarly for subtraction. Note that a special instruction to replace the code between ADCARY and ADDHI would need to be a double operand instruction - an inefficient use of op code space. Thus while non-zero format permits double precision add in 3 usec (on our hypothetical processor), zero format requires 5 usec - a 6Ø% increase. The same is true for subtraction.

2.21 Formats in Floating Add and Subtract - Alignment of operands is the chief operation in floating addition and subtraction. Double precision shifting would be an instruction added by EAE in any case so either format would be acceptable - however, the zero format shift is special purpose and might not be as useful in shifts done for other than arithmetic operations. Further, the non-zero format makes for easier floating adds in machines with no EAE; a double precision shift in non-zero format on a nonEAE configuration is:

```
;double precision shift - non-EAE - non-zero format
        ASL AØ             ;left shift - lo part
        ROL A1             ; hi part

        ASR A1             ;right shift - hi part
        ROR AØ             ;lo part
```

```
;double precision shift - non-EAE - zero format
        ASL AØ          ;left
        ASL AØ
        ROL A1
        CLC
        ROR AØ

        ASL AØ          ;right
        ASR A1
        ROR AØ
        CLC
        ROR AØ
```

The overhead in the zero format shift is so great (approximately 250%) that a better plan for long shifts would be to convert zero to non-zero shift and then reconvert. Different conversion/reconversion routines would be used for floating point mantissas (where the double precision word is considered as a fraction) than for integer shifts.

```
;convert from zero to non-zero - fraction
        ASL AØ          ;change lo part

;convert from non-zero to zero - fraction
        CLC
        ROR AØ

;convert from zero to non-zero - integer
        ASL AØ
        ASR A1
        ROR AØ

;convert from non-zero to zero - integer
        ASL AØ
        ROL A1
        BVS OFLO        ;can't convert
        CLC
        ROR AØ
```

Thus the overhead for zero formats in shifts and therefore for floating point addition and subtraction in non-EAE machines ranges from Ø to 25Ø% - 25Ø% if the operands were within 1 place of alignment in floating add or subtract.

2.3 Formats in double precision division - By far the largest use for double precision division will be for the mantissa calculations in emulation of floating divide. There, the mantissa of dividend, divisor, and quotient are all taken as double word fractions. The signed divide instruction (DIV) will be considered as dividing a two word signed dividend by a one word signed divisor and resulting in a one word signed quotient preceeded by a one word remainder. The divisor will be the source operand. The quotient will replace the hi-order word of the dividend at the destination address + 2. The remainder will replace the lo-order word of the dividend at the word preceeding the quotient. (In the preceeding general register if the destination is a register.)

The procedure for double precision divide is to compute the first terms of the series expansion of $A/B = (a+2^{-15}\alpha)/(b+2^{-15}\beta)$ where $|a|$, $|\alpha|$, $|b|$, and $|B|$ are all <1 and $\geq \frac{1}{2}$.

$$\frac{A}{B} = A(b+2^{-15}\beta)^{-1} = \frac{A}{b}\left\{1 - 2^{-15}\beta/b + 2^{-30}\beta^2/b^2 - 2^{-45}\beta^3/b^3 + \ldots\right\}$$

and so $A/B = \frac{A}{b}\left\{1 - 2^{-15}\beta/b + 2^{-30}\beta^2/b^2\right) + O(2^{-37})$

if A/b is considered as $q_0 + 2^{-15}r_0/b$ we may rewrite this as

$$A/B = q_0 + \frac{2^{-15}(r_0-\beta q_0)}{b} - \frac{2^{-30}\beta(r_0-\beta q_0)}{b^2} + O(2^{-37})$$

and writing $(r_0-\beta q_0)/b = q_1 + 2^{-15}r_1/b$

this becomes $A/B = q_0 + 2^{-15}q_1 - 2^{-30}\beta q_1/b + O(2^{-37})$.

The cross product βq_0 will retain accuracy only to 2^{-29} in our multiplication scheme (either zero or non-zero format). Thus, including the term $-2^{-30}\beta q_1/b$, which is always less than 2^{-29}, will not affect accuracy. Therefore we calculate

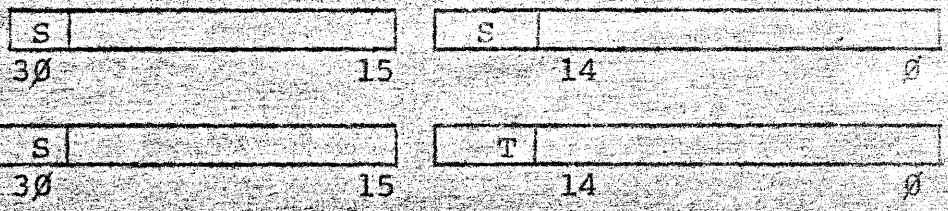
$A/B = q_0 + 2^{-15}q_1 + O(2^{-30})$.

```
;routine to divide the mantissa in (R2,R3) by (R4,R5) - non-
zero format
DIVM:   ASR R3          ;prevent overflow, divide A by 2
        ROR R3
        DIV R5,R2       ;get q0 /2 and r0/2, q0/2 in R3
        MOV R2,R0       save remainder
        MOV R4,R1       ;put B in zero format for cross multip.
```

```
        CLC
        ROR R1          ;-β in zero format
        NEG R1
        MUL R3,R1       ;βq₀/2 : R2 holds bits for 2^-17 to 2^-2
        ASL R1
        ROL R2
        ADD RØ,R2       ;(r₀ - Bq₀)
        CLR RØ
        DIV R5,R1       ;q₁ in R2
        SBC RØ
        ASL R2
        ROL RØ
        ADD RØ,R3       ;A/2β in (R2,R3)
        RTS PC
```

This procedure would take slightly longer in zero-format

2.4 Other formats - Two other formats have been considered: signed format and typed variable format. Both suffer from the same computational difficulties as the zero format but posses none of its advantages.



S is the sign bit, T is the type bit. (T determines whether a variable is fixed or floating.)

2.5 Conclusions -

a. While emulating unsigned multiplication with signed operators is substantially (~50%) more time consuming than calculating signed multiplication with unsigned operators, the incidence of signed multiplies is thought to be sufficiently great with respect to unsigned multiplies (excluding address arithmetic) that the choice is for a signed multiply operator.

b. Division is far more often perrormed with signed than unsigned operands - thus signed divide will be the divide operator.

c. The non-zero format is chosen because it facilitates non-EAE emulation of EAE operators, mantissa calculations

in floating multiply and divide, and double precision addition, subtraction, negation, normalization, and shifts. The advantage of zero format in full, 4 word, integer multiplication is not felt to be sufficient to outweigh this.