Implementation of Arithmetic
for the Data Flow Machine Processing Unit

by

Richard Tucker

(S.B. thesis, Department of Electrical Engineering and Computer Science, M.I.T.)

June 1980
Implementation of Arithmetic
for the Data Flow Machine Processing Unit

by

Richard Wesley Tucker

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering at the Massachusetts Institute of Technology, June 1980.

Abstract

The implementation of integer and floating point addition-subtraction and multiplication for the Processing Unit of the first prototype Data Flow Machine is described. A comparison is made among different implementations, the specifications given in proposals for an IEEE floating point standard for microprocessors, and the specified behavior in the applicative programming language VAL for these operations. The tradeoffs among program speed, program length, and desired abilities is discussed.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering
CONTENTS

Table of Figures ................................................................. 3
1. Introduction ................................................................. 4
  1.1 Limitations of the PU ............................................... 7
  1.2 Programming Conventions ........................................... 7
2. Number Representations .................................................. 8
  2.1 Error Values ............................................................ 8
  2.2 Integer Representation .............................................. 9
  2.3 Floating Point Representation ..................................... 10
  2.3.1 Special Values for Floating Point Operands .................. 12
    2.3.1.1 Overflows / Plus and Minus Infinity ..................... 12
    2.3.1.2 Underflows / Denormalized numbers ..................... 13
    2.3.1.3 Other Error Values / Not-a-Number ...................... 14
    2.3.1.4 Zeros .......................................................... 14
3. Implementation of Arithmetic ........................................... 15
  3.1 Integer Operations .................................................. 15
    3.1.1 Integer Addition ................................................. 15
    3.1.2 Integer Multiplication ....................................... 16
  3.2 Floating Point Operations .......................................... 16
    3.2.1 Floating Point Addition ..................................... 18
      3.2.1.1 Floating Point Addition - Coonen ...................... 18
      3.2.1.2 Floating Point Addition - Error Byte ................ 19
    3.2.2 Floating Point Multiplication ................................ 19
4. Conclusions and Suggestions .......................................... 20
Appendix A. Integer Add Program ........................................ 23
Appendix B. Integer Multiply Program ................................ 28
Appendix C. Floating Point Add Program / Coonen .................... 34
Appendix D. Floating Point Add Program / Error-Byte ............... 41
Appendix E. Floating Point Multiply Program ......................... 49
References ........................................................................... 58
Table of Figures

FIGURES

Figure 1. First Data Flow Machine Prototype .............................................. 4
Figure 2. Logical Diagram of the Processing Unit ........................................ 4
Figure 3. VAL’s error values ........................................................................ 8
Figure 4. Representations of Error Values, Error-Byte First Method ........... 9
Figure 5. Multiplication Algorithm ............................................................. 16
Figure 6. VAL’s specifications for real multiplication ................................. 20
1. Introduction

The Data Flow Machine being developed at MIT is designed with concurrency of instruction execution in mind. The desire in constructing a data flow machine is to attain a greater computation speed than that achieved by traditional machines, by taking advantage of parallelism in programs. Conventional computers perform instructions one at a time, in sequence, while the data flow machine is to perform an instruction as soon as it has received all of its operands, and has a number of independent functional units to do so. The machine identifies each instruction that has been enabled by the arrival of its operands, selects an available functional unit to execute it, and delivers the results to specified destination instructions. An applicative flow of instruction execution is thereby attained, driven by the availability of data. An applicative language, VAL, has been designed for use on the data flow machine; see [Ackerman-VAL].

In a practical form of a data flow processor, Instruction Cells are grouped into Cell Blocks [Dennis-Prototypes]. When an Instruction Cell is enabled by the arrival of all of its operands, an operation packet is sent to an Arbitration Network. The Arbitration Network dispatches the operation packet to an available functional unit appropriate for the operation code included in the packet. The functional units send result packets to a Distribution Network, which passes the result packets to proper Cell Block destinations.

The first data flow machine prototype for construction, shown in Figure 1, combines the actions of a Cell Block and functional unit into a Processing Unit (PU). The prototype consists of 4 PUs connected to a 4 by 4 Routing Network, which sends result packets to the proper PUs. The aim of this thesis is to describe how the arithmetic operations of addition-subtraction and multiplication might be implemented for the PU.

The PU is an 8 bit microprocessor which can be programmed to emulate any byte-serial packet communication module [Ackerman-PU]. A diagram of the PU’s data paths is given in Figure 2. The
TOPL supervisory routine specified in [Feridun-Module] acts as a scheduler to perform the actions of the Cell Block unit and functional unit. The data memory of the PU contains a block of cell-state information, a block of operation packets, and a bitmap of enabled cells. When the PU receives a result packet, it is delivered to the appropriate operation packet operand slot, and if that packet has received all of its operands (detected by examining the cell-state information), it is marked in the bitmap as enabled. When a functional unit is simulated, an enabled cell's operation is performed, and the results are transmitted to destination PUs via the router network, if possible.

Operands for operations are part of the operation packet stored in the PU's data memory. An operation program can access operands through the use of a pointer into the appropriate operation packet. This pointer is set up by a functional unit routine (known as OPER in Feridun-Module) which picks an enabled instruction cell, invokes the appropriate operation for it, delivers results to destinations specified in the operation packet, and sends acknowledge signals.
Figure 2. Logical Diagram of the Processing Unit
[Ackerman-PU & Fenidun-Module]

Immediate Memory — 65536 bytes of 8 bits, used as:

- Cell State Information 16000 bytes
- Operation Packets 48000 bytes
- Common Buffer — 1536 bytes
  (packet buffer, Bitmap, External Buffer)

All numbers are in decimal.
1.1 Limitations of the PU

The PU's program memory has a capacity of 4K (4096) 40 bit instructions. All the software to manage the cell-blocks and to perform arithmetic and other functions must fit within the 4K memory. This limitation requires a careful analysis of the decisions to be made in implementing a particular action. While it may be preferable to implement a particular function to VAL's specifications, it may be undesirable if PU program space is cramped. Certainly there are minimal necessities for an adequate implementation of any particular function. It might be a good idea to make changes to the PU hardware which would allow for shorter programs, e.g. the addition of a single instruction to increment or decrement the memory address register (MAR).

The PU's 16 scratch-pad registers and most of its operations work with 8 bit bytes. Integers and floating point numbers of single precision (32 bits) require operations on them to manipulate bytes among the registers and in an External Buffer (in the data memory) if necessary. Numbers of a greater fixed precision, or of an unfixed precision, are not easily handled within the 16 scratch-pad registers, particularly in multiplication. A method of handling multiple byte arithmetic in a signed-digit byte-serial fashion is described in [Feridun-Pipeline].

1.2 Programming Conventions

At invocation of an operation, the PU's scratch-pad registers 10. and 11. should contain the Bitmap pointer (high and low order bytes), registers 12. and 13. the External Buffer pointer, and registers 14. and 15. the Operation Packet pointer. The External Buffer is a section of the Data Memory available for various uses including as a scratch-pad area for operations. Therefore operations may use registers 0 through 9., and, if desired, the registers 10., 11., 14., and 15. may be saved in the External Buffer and restored before returning to the scheduler. The arithmetic operations by convention leave a 4 byte result in registers 7 (high order byte) through 4 (low order...
byte). The External Buffer pointer should be the same after an operation is finished as it was when invoked, i.e., nothing should be left in the External Buffer between operations. (For multiple precision operations, parts of a result might be left in the External Buffer.)

2. Number Representations

2.1 Error Values

In the data flow machine, there can be no interruption of program execution to handle exceptions, due to the concurrency of instruction execution. Therefore, operations produce error values for exceptional results. The error values used in VAL are described in Figure 3.

---

**Figure 3. VAL's error values**

*pos_over* and *neg_over* for results of a magnitude larger than can be represented (in single precision);

*pos_under* and *neg_under* for results of a magnitude smaller than can be represented (in single precision);

*unknown* for a result that cannot be calculated due to the limitation of representation capacity arising on a previous operation;

*undef* for a value that is not in the domain of an operator;

*miss_elt* for a missing element of an array within the array range; and

*zero_divide* for a result from a division by zero.

---

If a data flow program (in VAL) does not make explicit checks for error values, they will propagate. Tracing the data flow path that produced a particular error is likely to be difficult. It has been suggested [in McGraw-VAL] that each error value have an audit trail associated with it, to provide information regarding its origin and how it propagated. How any error tracing system could
interact with VAL is difficult to envision. When an error value results from an operation, some associated information could be transmitted upon a stream. The role of the functional unit operations when producing and propagating errors might be to encode extra information into an error value for error interceptors ahead. Any sort of error recording system is likely to be expensive in terms of its interfering with concurrent instruction execution. The value of any underlying error tracing effort would be in its ability to associate errors with their origins in a VAL program, and is outside the domain of this report.

2.2 Integer Representation

Single precision integers are represented in 4 bytes of 8 bits each, in two's complement form. Error values are represented in a manner suggested in [Aoki-Instruction Set]: the first bit of the high order byte is 1, and the rest of that byte can be decoded to identify the particular error value; see Figure 4. The implementation of integer arithmetic deals with error values in the same way specified by VAL.

The actual representation of integers is as follows:

\[
\begin{array}{cccccccc}
\text{high} & & & & & & & \\
RS & I & I & I & I & I & I & I \\
\text{low} & & & & & & & \\
\end{array}
\]

where R is the error bit; if R is on, then the rest of the high byte signifies the error code; if R is off, then S and the I bits represent the integer in two's complement, S indicating the sign.
Figure 4. Representations of Error Values, Error-Byte First Method

76543210 (bits)
10000000 unknown
10100000 pos_over
11000000 neg_over
10010000 pos_under (not applicable to integers)
11010000 neg_under (not applicable to integers)
10001100 zero_divide
10001000 miss_elt
10000100 undef

bit 7 on if an error value
bit 6 on if negative
bit 5 on if overflow (when bit 7 is on)
bit 4 on if underflow (when bit 7 is on)

2.3 Floating Point Representation

There are several proposed standards for floating point arithmetic under consideration by the IEEE Computer Society's Microprocessor Standards Subcommittee. None of the proposals yet has been deemed as officially approved by the IEEE, although one appears to have greater support than the others. That proposal is the one by Coonen, described in [Signum-Oct 1979] and [Coonen-Computer]. Payne & Strecker and Fraley & Walther also have submitted proposed standards.

The specifications of the Coonen standard include:

precisions: single, double, quad; single-extended, double-extended

results for add, subtract, multiply, etc.

rounding modes: round toward nearest, zero, plus infinity, minus infinity

infinity arithmetic modes: projective, affine

denormalized arithmetic modes: warning, normalizing

exceptions with optional traps: invalid_operation, overflow, underflow, division_by_zero, inexact_result
This proposed Floating Point Standard calls for a single precision floating point number to be represented by a bit string of 32 bits, with the leading bit indicating the sign of the number, the following 8 bits for the biased exponent, and the remaining 23 bits for the fractional part of the number's mantissa. A single precision floating point number requires 4 bytes of 8 bits each. A nonzero floating point number is ordinarily stored in a normalized format, and the leading bit is not kept but is by implication 1. I have implemented floating point addition-subtraction in two ways: according to Coonen's proposed standard, and using the first byte to indicate errors. The representations used by the two implementations are:

```
  high                  low
  SEEEEEEE EFFFFFFF FFFFFFFF FFFFFFFF .......... Coonen standard
  REEEEEEE EFFFFFFF FFFFFFFF FFFFFFFF .......... error-byte first
```

where S is the sign bit, the 8 E bits are for the biased exponent field, and the 23 F bits for the mantissa's fraction field, for the Coonen standard. For the error-byte first method, R is the error bit: if on, the first byte identifies the error value (see Figure 4), and the other three bytes can be ignored; if R is off, the S bit, the 8 E bits, and the 22 F bits represent the valid, in-range floating point number. The value of a nonzero normalized number in these representations is:

\[ (-1)^S \cdot 2^{E-127} \cdot (1.F) \]................. Coonen standard
\[ (-1)^S \cdot 2^{E-128} \cdot (.1F) \]................. error-byte first

where, in the Coonen standard, 127 is the bias for the exponent, and "1." is the implicit leading bit; Note that \( 1 \leq 1.F < 2 \). In the error-byte first implementation, \( 1/2 \leq .1F < 1 \), and 128 is the exponent's bias. The effective range of non-error normalized floating point numbers is:

\[ 2^{-126} \cdot 1.000... \ to 2^{-127} \cdot 1.111... \]................. Coonen standard
\[ 2^{-127} \cdot .1000... \ to 2^{-127} \cdot .1111... \]................. error-byte first

where the error-byte first format can represent numbers 4 times smaller than the Coonen standard's form, and 1/2 as large. The error-byte first format can therefore represent a slightly greater range of
normalized floating point numbers. As described below, this difference is due to the inclusion of denormalized numbers in the Coonen proposal.

2.3.1 Special Values for Floating Point Operands

In the Coonen standard, error and other special values are detected by testing a number’s exponent (for all zeroes or all ones), and for some, the fraction field as well. In [Aoki-Instruction Set], the first bit of the high order byte is used to indicate an error value, and if it is on, the rest of that byte encodes the error value. The latter method of encoding error values requires fewer PU program steps to detect some error values than would be required by adhering to the representation specified by the Coonen standard. It also reduces the precision of the fraction field by one bit. I have implemented floating point addition-subtraction using both methods.

2.3.1.1 Overflows / Plus and Minus Infinity

In the Coonen standard, Infinity is represented by an exponent field equal to the maximum (all ones), and the mantissa’s fraction field as zero. The sign bit represents the sign of Infinity. When a result overflows the range of representable numbers, the default action to be taken, since exception handling traps do not exist, is to call the result infinity. However, this reserved operand Infinity does not act like VAL’s pos_over or neg_over in a number of cases. For example, in VAL, pos_over * 1/2 produces unknown, whereas according to Coonen, +\( \infty \) * 1/2 produces +\( \infty \); and pos_over * 0.0 produces 0.0, while in the Coonen standard, +\( \infty \) * 0.0 produces an Invalid-Operation. It is clear that an overflow is not mathematically the same as Infinity. VAL’s approach seems more mathematically sound.
2.3.1.2 Underflows / Denormalized numbers

In the Coonen standard, floating point numbers which cannot be normalized because they are too small are represented by denormalized numbers, in which the exponent is equal to the minimum (all zeroes) and the fraction field is nonzero; the implied leading bit in this case is 0 rather than 1 as in normalized numbers. A denormalized number has the value:

$$(-1)^{S} \times 2^{-126} \times (0.F).$$

The use of denormalized numbers is aimed at deferring an occurrence of an underflow at the sacrifice of precision [Coonen-Computer], while slightly extending the range of representation. There is fairly strong disagreement by proposers of other standards for floating point arithmetic that the use of denormalized numbers is worth the effort to implement them [Signum-Oct 1979, pages 22 to 23, and Signum-Mar 1979, pages 100 to 108].

For VAL's sake, either any denormalized number or the minimum denormalized number could be considered an underflow for the error values pos_under and neg_under. Operations on "slightly" denormalized numbers can still produce meaningful results, although care must be taken. If denormalized numbers are to be implemented at all, then they should not be considered as underflows, except for the minimum one, in which the least significant bit is 1, and all other bits (other than the sign bit) are 0. However, in denormalizing a preliminary result of an operation, the result may turn into a Zero. (In the Payne proposal, underflows are also converted to Zero.) It is undoubtedly unacceptable in VAL for an underflow result automatically to become Zero. If denormalized numbers were not implemented, while other specifications of Coonen's proposed standard were adhered to, then either (a) the exponent range can be increased by 1, and the exponent bias incremented; or (b) a zero exponent field would represent the number Zero, to reduce program steps in a test for Zero. In the error-byte first implementation, each value must be normalized, zero, or an error value; underflows are represented as error values.
2.3.1.3 Other Error Values / Not-a-Number

In the Coonen standard, Not-a-Number (NaN) is used to represent default results of various invalid-operations. It is to be represented by an exponent field equal to the maximum (all ones), and the fraction field as something other than zero. The fraction field is intended to be used for diagnostic or other coded information indicating why NaN was produced as a result of a floating point operation. NaN could be used to indicate the error values zero_divide, miss elt, unknown, and undef in VAL. However, Coonen's specifications of results of operations on NaNs do not agree with VAL's specifications of what to do with those error values. For example, NaN * <any non-NaN> produces the same NaN, while in VAL, unknown * 0.0 produces 0.0, and miss elt * 0.0 produces undef. Also, the result of Infinity divided by zero would be Infinity; for VAL, any division by zero would result in zero_divide. The NaN construct might be considered extensible, so that exceptions not covered in the Coonen standard could be encoded in the fraction field, and be handled separately. However, the error-byte first method handles all errors in a uniform way, and is likely to require fewer programming steps to identify each error value.

2.3.1.4 Zeros

In the Coonen standard, Zero is represented by an exponent field equal to the minimum (all zeroes) and the fraction field all zeroes, with the implied leading bit taken to be 0. The sign bit is used to indicate a signed Zero. In the error-byte first implementation, an exponent field which is zero indicates Zero; the fraction field is ignored. In the Payne proposal, Zero is unsigned; if the sign bit is 1 for a zero exponent field, the number represents a reserved operand. Having the ability to test just one byte to determine whether a number is Zero certainly saves program steps.
3. Implementation of Arithmetic

The implementations of the addition and multiplication operations, floating point and integer, error byte and according to Coonen, are given in the appendices. The tack taken in implementing each operation is described in the comments, for the most part. Points of interest are set forth below.

3.1 Integer Operations

3.1.1 Integer Addition

The implementation of an addition-subtraction operation for single precision (4 byte) two's complement operands is not complicated. The operands are first checked for the error values. If either operand is undef, miss_eit, or zero_divide, the result is set to be undef. If either is unknown, the result is set to unknown. If either is pos_over, the result is set to pos_over only if the other operand is greater than or equal to zero; else it is set to unknown. If either is neg_over, the result is set to neg_over only if the other operand is less than or equal to zero; else it is set to unknown. If both operands are not error values, then the addition is performed byte-wise, starting with the least significant bytes. The carry from each corresponding byte addition is added to the next most significant bytes added. Since the leading bit of the most significant byte of the result is the error bit, and the next bit is for the sign, a check is made to prevent the addition from overflowing into those bits. If an overflow is detected, the result is set to pos_over or neg_over. The implementation corresponds exactly with VAL's prescribed behavior. See Appendix A for the PU integer addition program.
3.1.2 Integer Multiplication

In the implementation of integer multiplication, operations on error values are checked first. As in VAL, if any operand is undef, miss_elt, or zero_divide, the result produced is undef. If either is Zero, so is the result. If either is unknown, so is the result. If one operand is pos_over, the result is the same if the other is positive; else neg_over. If one is neg_over, the results are similar, with the signs opposite. If neither operand is zero, then the operands are converted to their magnitudes, and multiplied as shown in Figure 5. The high 4 bytes are tested to see if the result has overflowed; if so the result becomes pos_over or neg_over, according to the original signs. Otherwise, the low 4 bytes are retrieved from the external buffer and become the result (an overflow may still result). This result is two's complemented if the original operand signs warrant, and the result is left in registers 7 through 4 for the caller to deliver.

3.2 Floating Point Operations

The Coonen standard refers to enabling traps when operations encounter error conditions, and also to checking user-settable choices among rounding methods and between infinity arithmetic systems. However, traps cannot exist on the data flow machine, and the limitation that the PU program must fit in 4K words of program memory probably prohibits niceties such as allowing for settable options. VAL has no provisions for setting such options, anyway.

The rounding method used in the implementation is round to nearest. Since a preliminary result of an operation often has more nonzero significant bits than would fit in a single precision destination, this rounding method chooses one of the two single precision numbers that bracket the preliminary result. The number that is chosen is the one that is nearest the preliminary result; if they are equally near, then the one with the least significant bit of 0 is selected. The other three rounding modes mentioned by Coonen are: round toward zero, which truncates a number, used when converting a
Figure 5. Multiplication Algorithm

Each \(<c_{ij}\) refers to the high byte of the result of multiplying \(<b_j\) and \(<a_i>\); \(<d_{ij}\) to the low byte. Each byte is added into the preliminary result K bytes as produced. After each multiplication of all of A's bytes by each multiplier \(<b_j\) (in subroutine IMABMult), the preliminary result is found in the bytes \(<K_{ik}\), and the lowest K byte is shifted into the external buffer. After all bytes have been multiplied, the 4 low K bytes (r3 to r0) are retrieved from the external buffer, if necessary.

```
  a3  a2  a1  a0
  b3  b2  b1  b0
  c00 d00
  c01 d01
  c02 d02
  c03  d03
  [K04  K03  K02  K01  K00]
  c10  d10
  c11  d11
  c12  d12
  c13  d13
  [K14  K13  K12  K11  K10]
  c20  d20
  c21  d21
  c22  d22
  c23  d23
  [K24  K23  K22  K21  K20]
  c30  d30
  c31  d31
  c32  d32
  c33  d33
  [K34  K33  K32  K31  K30]
  r7  r6  r5  r4 | r3  r2  r1  r0
```

number to an integer a'la Fortran; round to \(-\infty\), in which the lesser bracketing number is chosen; and round to \(+\infty\), in which the greater one is chosen. While these three rounding modes are not difficult to implement (they are easier than round to nearest), they have been omitted here, as no method of requesting a particular rounding mode exists.

The Coonen standard defines the use of Infinity in two infinity arithmetic systems, Projective and Affine. Coonen states that Projective mode should be the default. In Projective mode, the sign of Infinity is ignored. For example, an addition of two infinities results in an invalid operation; also,
Infinity cannot be compared to any value other than itself. In Affine mode, Infinity can be compared to all values except NaNs.

3.2.1 Floating Point Addition

3.2.1.1 Floating Point Addition - Coonen

In the implementation of floating point addition, according to the Coonen proposal, operands are first checked to see if they are NaNs. If so, the result is NaN. If either is Infinity, but not both, the result is Infinity with the appropriate sign; if both are, assuming as default the Projective infinity arithmetic mode, the result is NaN. If just one operand is Zero, the result is the other operand; if both are Zero, the appropriate sign is included. Otherwise, an addition is performed. First, the binary points of the operands are aligned, by shifting the lesser until its exponent equals the greater one (with a shift limit equal to the precision). The magnitudes are then added (or subtracted). If the addition overflows, the carry is shifted right into the result, and the exponent is incremented. If the operation was a subtraction, the result is tested for Zero. In any case, the result is normalized: the magnitude is shifted left until the explicit first bit is 1, while the exponent is decremented. While normalizing it may be obvious that the result cannot be normalized. The number would then be denormalized, as described earlier, which in essence reflects an underflow. The number is then rounded to fit the precision of the destination, which is single precision here, by the round to nearest method, described earlier. For all results, the number is repacked in the stated representation, and left in registers 7 through 4.
3.2.1.2 Floating Point Addition - Error Byte

In the implementation of floating point addition using the error byte first method, the first byte of each operand is first checked to see if the error bit is on. As in VAL, if either is undef, miss_eft, or zero divide, the result in undef. If either is unknown, so is the result. If both are underflows or overflows: if they have different signs, the result is unknown; else if one is an overflow so is the result, else underflow. If just one is an underflow, it is the result if the other is Zero; else the other operand is the result. If just one is an overflow, it is the result if both operands have the same signs, or if the other is Zero; otherwise the result is unknown. Otherwise, the two operands are added or subtracted, normalized, and rounded in the same manner as described in the previous section, except underflow error values are produced instead of denormalized numbers.

3.2.2 Floating Point Multiplication

The implementation of floating point multiplication given in an appendix follows Coonen’s specifications. If either operand is NaN, so is the result. If either is Infinity, and the other not Zero, the result is Infinity with appropriate sign. If one is Infinity and the other Zero, the result is NaN. If either is Zero, the result is Zero with the appropriate sign. Otherwise, the numbers are multiplied. The exponents are added, and the magnitudes multiplied in a fashion similar to the way integer multiplication was done, though with fewer significant bits. If the operands were both normalized, the result is either normalized or needs one right shift to be normalized, since each operand would be less than 2, as explained earlier. If the result exponent is an underflow, the associated denormalized value is left in registers 7 through 4. Otherwise, the result is rounded and left there, although it is checked for an overflow first. The behavior for multiplication involving error values in VAL is given in Figure 6, for comparison.

If one of the operands was denormalized, matters are complicated. The implementation given in
Appendix E does not handle denormalized numbers. What normally would be done would be to normalize all denormalized numbers prior to multiplication. The inclusion of denormalized numbers would complicate the program and add many more steps.

Figure 6. VAL's specifications for real multiplication
[from Ackerman-VAL, pages 25-26]

When either operand is undef, miss elt, or zero divide, the result is undef. For other error values, the results are produced as follows:

X is any real number other than undef, miss elt, or zero divide.

4a. \( X \times \text{pos\_over} = \text{neg\_over} \text{ if } X \leq -1.0 \text{ or } X = \text{neg\_over}, \)
\( \text{pos\_over} \text{ if } X \geq 1.0 \text{ or } X = \text{pos\_over}, \)
\( 0.0 \text{ if } X = 0.0, \)
\( \text{unknown otherwise} \)

4b. \( X \times \text{neg\_over} = -(X \times \text{pos\_over}) \)

4c. \( X \times \text{pos\_under} = \text{neg\_under} \text{ if } -1.0 \leq X < 0.0 \text{ or } X = \text{neg\_under}, \)
\( \text{pos\_under} \text{ if } 0.0 < X \leq 1.0 \text{ or } X = \text{pos\_under}, \)
\( 0.0 \text{ if } X = 0.0, \)
\( \text{unknown otherwise} \)

4d. \( X \times \text{neg\_under} = -(X \times \text{pos\_under}) \)

4e. \( X \times \text{unknown} = 0.0 \text{ if } X = 0.0, \)
\( \text{unknown otherwise} \)

4. Conclusions and Suggestions

While Coonen's proposed standard may be approved by the IEEE Microprocessor Standards Committee, it has a number of features which do not go well with the aims of the data flow machine project and the language VAL. The use of denormalized numbers complicates the programming of floating point operations; it requires a fair number of extra programming steps in every operation that
deals with them, which is undesirable due to the ultimate limitation that all programs in the Processing Unit fit within a 4K Program Memory. The reserved operands for special values and error values at first appear to resemble the error values used in VAL, but in most cases they are used differently. The Coonen standard takes into account the presence of traps to deal with exceptional results; and actions to be taken when they are disabled or don't exist. However, the actions taken can mean turning an underflow into a Zero. The reserved operand Infinity does not act like VAL's pos_over or neg_over. NaN corresponds roughly to undef, but there is no element corresponding to unknown, although NaN could encode the meaning of any error value (even those not used or handled in the standard, such as miss_ell and zero_divide, perhaps) and have each function act differently upon different encodings.

The error-byte first representation allows programs to detect error values more easily, and can handle all those used in VAL. It is a simple format, with an error value encoded in one place, though at the expense of one bit of significance. The implementations follow VAL's specifications of the results of operations involving error values, since they seem more appropriate than Coonen's in some cases. For example, pos_over * 1/2 produces unknown rather than pos_over. Coonen converts most overflows and divisions-by-zero into Infinity, and some underflows to Zeroes. However, an overflow is not exactly analogous to a mathematical infinity, and probably should not be considered so unless a program wishes to use it as such. The error byte could encode a value for Infinity, separately from an Overflow, if desired. The error-byte method appears to be more extensible than the various proposals to the IEEE, particularly for a machine which cannot have traps for exceptions, and which must take good care of error results as they propagate.

The conversions of error values between floating point and integer formats would be quite direct if the same error representation, error-byte first, were used. For integers, there is no alternative but to reserve a bit somewhere to mark a number as a special value. For floating point numbers, using certain exponent values to mark reserved operands is an obvious choice, used by Coonen. The
error-byte method, however, uses a simpler, though mildly drastic, ploy for floating point numbers, reserving one bit to denote an error value.

The programming language for the Processing Unit is fairly rich in its expressiveness. There are some common actions requiring two or more program steps for which new instructions could be added to the processor to be done in fewer instructions, such as for incrementing, decrementing, or adding/subtracting from the memory address register. As suggested in [Feridun-Module], the use of a 16 bit processor would be beneficial in reducing program steps and increasing program speed, or doubling precision capabilities. In the 8 bit PU, the programming of greater precision arithmetic operations with the same sort of error handling care cannot be done without the cost of much greater execution time due to accessing the external buffer in data memory.
Appendix A - Integer Add Program

: Representation =
RSAAAAA IIIIIII IIIIIIIII IIIIIIIII
if R = 1
then # is an error value
S = sign
SAAAAAA IIIIII IIIIIIII IIIIIIIII is number in two's complement.
else # is a representable integer
end

Error values:
76543210 bit7 on if error; bit6 on if neg; bit5 on if over; bit4 on if undef;
(bit3 or bit2) on if undef, miss_elt, or zero_divide.
10100000 pos_over
11000000 neg_over
10010000 pos_under (Not applicable to Integers)
11010000 neg_under (ditto)
10000000 unknown
10000100 undef
10001000 miss_elt
10001100 zero_divide

VAL Behaviour (J = any int):
J + undef --> undef
J + miss_elt --> undef
J + zero_divide --> undef
J + pos_over --> pos_over if J >= 0 or J = pos_over
--> unknown otherwise
J + neg_over --> neg_over if J <= 0 or J = neg_over
--> unknown otherwise
J + unknown --> unknown

: adding a3a2a1a0 and b3b2b1b0, a(i) & b(i) are bytes.
: result is left in r7 (high order) through r4.
: assumes packet ptr in r14 (low) r15 (high)
: assumes r10, r11, r12, r13 are not to be clobbered.

::: equates

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
r7 = 7
r8 = 8.
r9 = 9.
r10 = 10.
r11 = 11.
r12 = 12.
r13 = 13.
r14 = 14.
r15 = 15.

: gets a0
: a1
: a2
: a3
: b0 : result 0 - low order byte
: b1 : result 1
: b2 : result 2
: b3 : result 3 - high order byte

: operation packet ptr - low
: ditto - high

Bit7 = 200
Bit6 = 100
Bit5 = 40
Integer Add Program

```
MiscErrs = 14
ErUndefined = 204

; ----------------------debugging setup----------------------
Start: srci 0, r14 ; setup packet pointer low
zero r15 ; & high
jmpl IAAdd ; to point just before 1st argument

PktPtrMAR: ; Increment operation packet pointer (low byte) in r14, and put
addi warr 1, r14 ; result in r14 and MAR right
dstc warr rtn r15 ; carry propagate for high byte in r15 &
; put in MAR left - return

geta210: ; read in a2, a1, a0; packet pointer assumed to be pointing to a3
jsr PktPtrMAR ; r2 <- a2
dst mr r2
jsr PktPtrMAR ; r1 <- a1
dst mr r1
jsr PktPtrMAR ; r0 <- a0
dst mr rtn r0

getb210: ; read in b2, b1, b0; packet pointer assumed to be pointing to b3
jsr PktPtrMAR ; r6 <- b2
dst mr r6
jsr PktPtrMAR ; r5 <- b1
dst mr r5
jsr PktPtrMAR ; r4 <- b0
dst mr rtn r4

IAAdd: jsr PktPtrMAR ; incr packet ptr, store in MAR
dst mr r3
andi n bit7, r3 ; is error bit on?
jmp eq IAAnotErr
andi n MiscErrs, r3 ; is A undef, miss_elt, or zero_divide?
jmp eq IAAnotMiscErrs ; if not, go test for other conditions

SetUndefined: srci ErUndefined, r7
Zerest: zero r6
zero r5
zero r4
jmpl Deliver

IAAnotMiscErrs: ; A is an error, not Undefined, Miss_elt, or Zero_divide
; so A is unknown or pos/neg_over
; if B undef, miss_elt, or zero_divide, result <-- undef
; elseif B unknown then result <-- unknown
; elseif A unknown then result <-- unknown
; else A is pos/neg_over and B is same or a number
; (get b2 - b0)
; if B = 0 then result <-- A
; elseif sign (A) * sign (B) then result <-- A
; else result <-- unknown
; endif

; get b3
addi warr 4, r14 ; result in r14 and MAR right
dstc warr r15 ; carry propagate for high byte in r15 &
dst mr r7 ; r7 <- b3
andi n bit7, r7 ; is B's error bit on?
jmp eq IAAnotErr
andi n MiscErrs, r7 ; is B undef, miss_elt, or zero_divide?
```

Appendix A
Integer Add Program

jmp ne SetUnd                   : if so, go set result to undef

::: B is unknown, pos_over, or neg_over; A is also
jmp IAABunkow

IAABnotErr: ::: B not an error, but A is unknown or pos/neg_over

::: Is A unknown?
xori n bit7, r3
jmp eq SetUnk                   : jump if A is Unknown, & set result Unknown

::: A is pos/neg_over; do A & B have the same sign?
eqv r3, r7                       : compare b3 & a3
and1 bit6, r7                   : for sign bit
jmp ne IAAovDifSign             : if different signs, check if B is 0
src r3, r7                      : else result <-- A
jmp Zerest

IAAovDifSign: ::: A is pos/neg_over; B is not an error;
::: A & B have different signs; test if B is 0

dst n r7                        : is b3 zero?
jmp ne SetUnk                   : if not, set result unknown

::: go get b2, b1, b0
::: op packet pointer is pointing at b3
jsr getB210

dst n r6                        : is b2 zero?
jmp ne SetUnk                   : is b1 zero?

dst n r6                        : is b0 zero?
jmp ne SetUnk

::: B is 0, so result <-- A
src r3, r7                      :
jmp Zerest

IAABnotErr: ::: A is not an error value
::: Check for error values of b3
::: op packet pointer is pointing to a3
add1 warr 4, r14                 : result in r14 and WAR right
dstc warl r16                    : carry propagate for high byte in r16 &

dst mr r7                       : r7 <-- b3

and1 n bit7, r7                 : is B's error bit on?
jmp eq IAABnotErr               : if so, go set result to undef

::: B is unknown, pos_over, or neg_over; A is not an error

::: Is B Unknown?
xori n bit7, r7
jmp eq SetUnk                   : jump if B is Unknown, & set result Unknown

::: B is pos/neg_over; do A & B have the same sign?
eqv r3, r7                       : compare b3 & a3
and1 bit6, r7                   : for sign bit
jmp ne IAAovDifSign             : if different signs, check if A is 0
jmp Zerest

IAAovDifSign: ::: B is pos/neg_over; A is not an error;
::: A & B have different signs; test if A is 0

dst n r3                        : is b3 zero?
Integer Add Program

jmp ne SetUnk : if not, set result unknown

::: go get a2, a1, a0
::: op packet pointer is pointing at b3
addi warr -4, r14 ; decrement packet ptr
rsbci wari rtn 0, r15 ; borrow propagate
jsr getA210

dst n r2 ; is a2 zero?
jmp ne SetUnk
dst n r1 ; is a1 zero?
jmp ne SetUnk
dst n r0 ; is a0 zero?
jmp ne SetUnk

::: A is 0, so result <-- B
jmp Zerest

IAABunkv: ::: A is unknown or pos/neg_over; and so is B

::: Is A unknown?
xori n bit7, r3
jimp eq SetUnk ; jump if A is Unknown & set result Unknown

::: Well, is B unknown?
xori n bit7, r7
jimp eq SetUnk ; jump if B is Unknown & set result Unknown

::: A & B are pos/neg_over
eqv r3, r7 ; compare b3 & a3
andi bit6, r7 ; for sign bit
jimp eq Zerest ; if same signs, result <-- B (= A)
else ... SetUnk: srci bit7, r7 ; Set result to be Unknown
jimp Zerest

IAABnotErr: ::: A & B are not error values
::: op packet pointer is pointing at b3
addi warr -4, r14 ; decrement packet ptr
rsbci wari rtn 0, r15 ; borrow propagate
jsr getA210 ; get a2, a1, a0 in r2, r1, r0
addi 1, r14 ; increment pointer to point at b3
dstc r15 ; carry propagate for high byte in r16
jsr getB210 ; get b2, b1, b0 in r6, r5, r4

iaabadd: ::: Neither A nor B are error values, so add them.
dst ls r7 ; left shift so can check for overflow in add
dst ls r3 ; - since bit7 is error bit
zero r8
add r0, r4
addc r1, r5
addc r2, r6
dstc ls r8 ; put carry bit in bit1 of r8
add r3, r7
jimp vc WoOver
add r8, r7
jimp vc WoOver

::: addition overflowed
andi neg bit7, r3 ; put sign bit in Q
qreg r3 a r7 ; shift sign bit back to proper place (bit6)
or bit7 & bit5, r7 ; turn on error bit (redundant) & overflow bit
jmp zerest ; then zero r6 to r4
NoOVer: dst rs r7 ; shift back r7 - bit7 (error bit) gets 0

Deliver: :: then Deliver result
::: Should invoke delivery routine, or just return to caller
::: who will deliver.
sin
Appendix B - Integer Multiply Program

; -*-PU-*- Thursday 22 May 1980 6:35:39 am
; Integer Multiplication for the PU

; A number of labels in IMULT are identical to labels in
; TADO, FADO, FMULT. They perform identical functions in each.

; Representation =
; RSAAAAAA I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I
; if R = 1
; then # is an error value
; SAAAAAA encodes the error value; I's ignored.
; else # is a representable integer
; S = sign
; SAAAAAA I I I I I I I I I I I I is number in twu's complement.
; end

; Error values;
; 76643210 bit7 on if error; bit6 on if neg; bit5 on if over; bit4 on if under;
; (bit3 or bit2) on if undef, miss埃, or zero divide.
; 10100000 pos_over
; 11100000 neg_over
; 10010000 pos_under (Not applicable to Integers)
; 11010000 neg_under (ditto)
; 10000000 unknown
; 10000100 undef
; 10001000 miss埃
; 10001100 zero_divide

; Adding a3a2a1a0 and b3b2b1b0, a(1) & b(1) are bytes.
; Result is left in r7 (high order) through r4.
; Assumes packet pt: in r14 (low) r15 (high);
; Assumes r10, r11, r12, r13 are not to be clobbered.

; VAL Behaviour (J = any int):
; J * undef --> undef
; J * miss埃 --> undef
; J * zero_divide --> undef
; J * pos_over --> neg_over If J <= -1 or J = neg_over
; pos_over If J > 1 or J = pos_over
; 0 if J = 0
; J * neg_over --> -(J * pos_over)
; J * unknown --> 0 if J = 0
; unknown otherwise

::: equates

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
r7 = 7
r8 = 8
r9 = 9
r10 = 10.
r11 = 11.
r12 = 12.
r13 = 13.
r14 = 14.
r15 = 15.

Bit7 = 200
Cbit7 = 177

; 10000000
; 01111111
Integer Multiply Program

Bit6 = 100
Bit5 = 40
MiscErrs = 14
ErUndef = 204

: : : Increment operation packet pointer (low byte) in r14, and put
addr
warr 1, r14 ; result in r14 and MAR right
dstc
warr rtn r15 ; carry propagate for high byte in r15 &

DecPktPtrMAR: : : Decrement
addi
warr -1, r14 ; decrement packet ptr
rsbici
warr rtn 0, r15 ; borrow propagate

DecExtBufMAR: : : ditto for External Buffer pointer
addi
warr -1, r12 ; decrement external buffer pointer
rsbici
warr rtn 0, r13 ; borrow propagate & set MAR

geta210: : : read in a2, a1, a0; packet pointer assumed to be pointing to a3
jr
IncPktPtrMAR
dst
mr r2 ; r2 <- a2
jr
IncPktPtrMAR
dst
mr r1 ; r1 <- a1
dst
mr rtn r0 ; r0 <- a0

getb210: : : read in b2, b1, b0; packet pointer assumed to be pointing to b3
jr
IncPktPtrMAR
dst
mr r6 ; r6 <- b2
jr
IncPktPtrMAR
dst
mr r5 ; r5 <- b1
dst
mr rtn r4 ; r4 <- b0

IMGetB: : : Reads in a byte; if r11 says B was negative, then propagate
: : the two's complement, and save C bit for next GetB.
dst
mr r8 ; r8 <- b1
dst
n r11 ; was B negative?
rtm eq
: : return if it wasn't
ldc
r11 ; else set C bit from bit 0 of r11
cdstc
r8 ; two's complement propagate
ori
rcc rtn bit7, r11 ; put condition codes in r11 & return

Save1011: : : Save registers 10 & 11 in external buffer
: : external buffer pointer is assumed to be pointing to last item
: : stored in exbuf; if none there, is 1 less than available spot
addi
warr 1, r12 ; increment external buffer ptr
dstc
warr r13 ; ... & set MAR
dst
n wmm r10 ; r10 -> exbuf
addi
warr 1, r12 ; increment external buffer ptr
dstc
warr r13 ; ... & set MAR
dst
n wmm rtn r11 ; r11 -> exbuf

Restore1011: : : restore registers 10 & 11 from exbuf
: : External buffer points to last entry put there; if none there,
: : is 1 less than available spot.
dst
mr r11 ; retrieve r11
jr
DecExtBufMAR
dst
mr rtn r10 ; retrieve r10

IMult: jsr
IncPktPtrMAR
dst
mr r3 ; r3 <- a3

: : Is A undef, miss_elit, or zero Divide?
and
n bit7, r3 ; is A's error bit on?
jmp eq
IManotErr
and i n MiscErrs, r3 : is A undef, miss elt, or zero divide?
jmp eq IMANotMiscErrs : if not, go test for other conditions

result is Undefined since A is one of (undef, miss elt, zero divide).

SetUnd: src ErUndefined, r7
What is in r6-r4 is ignored when number is an error value.
jmp Deliver

IMANotMiscErrs: : : A is an error, not Undefined, Miss elt, or Zero divide
so A is Unknown, Pos_over, or Neg_over
: if B undefined, miss elt, or zero divide, result <= undef
: else (get b2 - b0) if B = 0 then result <= 0
: else if B unknown then result <= unknown
: else if A unknown then result <= unknown
: else result <= over with xor of signs
: endall

: get b3
addi dsrc warr 4, r14 : result in r14 and MAR right
dst src warr r16 : carry propagate for high byte in r16 &
dst mr r7 : r7 <- b3
and i n bit7, r7 : is B's error bit on?
jmp eq IMANotErr
and i n MiscErrs, r7 : is B undef, miss elt, or zero divide?
jmp ne SetUnd : if so, go set result to undef

: B is unknown, pos_over, or neg_over; A is also
jmp IMABunkov

IMANotErr: : : A is unknown, pos_over, or neg_over; B is not known
jsr getB110 : get b2, b1, b0 in r6, r5, r4
dst n r7 : is b3 zero?
jmp ne IMANotEO

IMANotEO: : : A is unknown, pos_over, or neg_over; B is not an error, is <= 0
: exchange A & B high order bytes and have tests made elsewhere
: dst nq r7 : // this exchange also
: src r3, r7 : // is 1n
: qreg r3 : // IADD
jmp IMANotEOBerr

IMANotEOBerr:
jsr getA110 : get a2, a1, a0 in r2, r1, r0

: Check for error values of b3
jsr IncPtrlPtrMAR
dst mr r7 : r7 <- b3

: Is B undefined, miss elt, or zero divide?
and i n bit7, r7 : is B's error bit on?
jmp eq IMANotErr
and i n MiscErrs, r7 : is B undefined, miss elt, or zero divide?
jmp ne SetUnd : if so, go set result to undef

: B is an error value, but not undefined, miss elt, zero divide

: B is unknown, pos_over, or neg_over; test if A is 0
dst n r3 : is a3 zero?
jmp ne IMANotEOBerr : [A is not an error or zero; B is an error]
dst n r2 : is a2 zero?
Integer Multiply Program

jmp ne IMAnotEZBerr
dst n r1 ; is a1 zero?
jmp ne IMAnotEZBerr
dst n r0 ; is a0 zero?
jmp IMAnotEZBerr

;;; A is 0, so is result
zero r7
jmp Deliver ; zero r6, r5, r4 and return

Setunk: src bit7, r7 ; Set result to be Unknown
jmp Deliver

IMABunkov: ;; A is unknown or pos/neg_over; and so is B

;;; Is A unknown?
xori n bit7, r3
jmp eq Setunk ; jump if A is Unknown, & set result Unknown

IMAnotEZBerr: ;; A is either [not an error, and is not zero] or [pos/neg_over]
;;; B is unknown, pos_over, or neg_over

;;; Well, is B unknown?
xori n bit7, r7
jmp eq Setunk ; jump if B is Unknown, & set result Unknown

;;; B is pos/neg_over, and A is either [pos/neg_over] or [number <- 0]
;;; so result <-- an overflow with xor of the signs of A & B
xor r3, r7
andi bit6, r7 ; put sign bit in r7
ori bit7 & bit6, r7 ; put in error bit & overflow bit
jmp Deliver

IMABnotErr: ;; A & B are not error values

;; To simplify the multiplication, the magnitudes
;; will be multiplied rather than the two's complement values.
;; b3 is in r7.
jsr Save1011 ; Save r10 & r11 in the external buffer

;;; Save signs of A & B
src ls r7, r11 ; put b3 left-shifted into r11
andi bit7, r11 ; extract sign
src ls r3, r10 ; ditto for a3
andi bit7, r10
jmp eq Apos

;;; A was negative, so make positive
ndst r0 ; two's complement
cdstc r1 ; propagate
cdstc r2 ;
cdstc r3 ;
andi cbit7, r3 ; remove top bit

Apos: ;; operation pkt pointer is pointing to b3
addi war3, r14 ; -- get low order B byte, b0 --
dstc war1 r16
dst mr r8 ; r8 <- b0
dst n r11 ; was B negative?
jmp eq Bpos0
ndst r8 ; begin two's complement
ori rcc bit7, r11 ; put condition codes in r11 (C in bit 0)

BPos0: jsr IMABMul

jsr DecPktPtrMAR ; move pointer from b0 to b1
jsr IMGetB ; read in b1; if was negative, propagate
jsr IMABMul
Integer Multiply Program

```assembly
jsr DecPktPtrMAR
jsr IMGetB : read in b2; if was negative, propagate
two’s complement.
jsr IMABMul

jsr DecPktPtrMAR
jsr IMGetB : read in b3; if was negative, propagate
two’s complement. If C bit still set,
is not relevant.
andi cbit7, r8 : remove top bit
jsr IMABMul

::: leaves 4 MSBs in r7 to r4
::: and 4 LSBs in <ExtBuf - 3> to <ExtBuf>

::: If any of r7 to r4 are nonzero, result is an overflow.
dst n r7
jmp ne IMResOv

dst n r6
jmp ne IMResOv

dst n r5
jmp ne IMResOv

dst n r4
jmp ne IMResOv

::: So result isn’t an overflow (yet), so retrieve low 4 bytes.
dst warr r12 : move external buffer ptr to MAR
dst warl r13

dst mr r7 : get next byte of result from external buffer
andi n bit7, r7 : is high bit on?
jmp ne IMResOv2 : then overflow (this test precludes the
 inclusion of -2^30; range of result
 is -2^30 + 1 to 2^30 - 1)

jsr DecExtBufMAR

dst mr r5
jsr DecExtBufMAR

dst mr r6
jsr DecExtBufMAR

dst mr r4

::: Need to reset the external buffer pointer to initial state
addi -1, r12 : decrement external buffer pointer
rsubici 0, r13 : borrow propagate & set MAR

andi r10, r11 : AND saved sign bits of A & B
andi n bit7, r11 : just want the sign bit, no carry bits
jmp eq PreDeliver : if result is positive, done

::: else have to two’s complement the result.
ndst r4
cdstc r5
cdstc r6
cdstc r7
jmp PreDeliver

IMResOv2: ::: Overflow after retrieving some from external buffer.
::: Reset external buffer pointer
addi -3, r12 : decrement external buffer pointer
jmp IMResCont

IMResOv: ::: The result of the actual multiplication overflowed. Set
::: r7 - r4 for an overflow, with sign
::: Reset external buffer pointer
addi -4, r12 : decrement external buffer pointer
jmp IMResCont

IMResCont:
rsubici 0, r13 : borrow propagate
andi rs r10, r11 : AND saved sign bits of A & B
```
Integer Multiply Program

```
  andi  bit6, r11  ; just want the sign bit, no condition bits
  orl  bit7 & bit6, r11  ; OR in the error & overflow bits
  src  r11, r7  ; put in the conventional place
  ; ; for now, the values of r6-r4 are ignored for error values.
  jmp  Deliver

IMABMult:
  dst  nq r0    ; q <- r0 (a0)
  zero r9
  lsetup 7
  umpy d 1pct r8, r9    ; b[i] * a0 [8 times]; MSB to r9; LSB to q
    b[i] = MSB (c[i]) -> r9, LSB (d[i]) -> q
  addi  warr 1, r12    ; increment external buffer ptr
  ddst  warr1 r13    ; ; & set MAR
  addq  n wmm r4, r9    ; q {LSB} + prev. low byte -> ext buf
  srcr  r5, r4    ; r5 + carry -> r4
  srcr  r6, r5    ; r6 + carry -> r3
  srcr  r7, r6    ; r7 + carry -> r2
  srci  0, r7    ; carry -> r7
  add  r9, r4    ; r9 + r4 -> r4 [r9 = MSB]
  dstc r5    ; carry propagate
  dstc r6    ; =
  dstc r7    ; =
  dst  nq r1    ; q <- r1 (a1)
  zero r9
  lsetup 7
  umpy d 1pct r8, r9    ; MSB -> r9, LSB -> q
  addq  r4, r4    ; d[i3] + r4 -> r4
  dstc r5    ; carry propagate
  dstc r6    ; =
  dstc r7    ; =
  add  r9, r5    ; c[i3] + r5 -> r6
  dstc r6    ; carry propagate
  dstc r7    ; =
  dst  nq r2    ; q <- r2 (a2)
  zero r9
  lsetup 7
  umpy d 1pct r8, r9    ; MSB -> r9, LSB -> q
  addq  r6, r6    ; d[i2] + r6 -> r5
  dstc r6    ; carry propagate
  dstc r7    ; =
  add  r9, r6    ; c[i2] + r6 -> r6
  dstc r7    ; =
  dst  nq r3    ; q <- r3 (a3)
  zero r9
  lsetup 7
  umpy d 1pct r8, r9    ; MSB -> r9, LSB -> q
  addq  r6, r6    ; d[i3] + r6 -> r6
  dstc r7    ; =
  add  rtn r9, r7    ; c[i3] + r7 -> r7
  ; ; carry should = 0 for unsigned multiplication.

PreDeliver:  ; ; Restore saved registers
  jsr  Restore1011

Deliver:  ; ; then Deliver result
  ; ; Should invoke delivery routine, or just return to caller
  ; ; who will deliver.
  rtn
```
As per Coonen's proposed IEEE floating point standard
Described in Signum Newsletter special issue, October 1979; and
Computer (IEEE), January 1980
- single precision only
- without exception traps or signals
- with denormalized numbers
- using round to nearest
- using projective infinity arithmetic (+Infinity = -Infinity)

Floating point numbers should arrive in the following format:

S EEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE
so they are unpacked to be as:

a3 - Exponent --- first
a2 - sign bit, Fract MSB 7 bits
a1 - Fract 8 bits
a0 - Fract LSB 8 bits
b3 - like a3
b2 - like a2
b1 - like a1
b0 - like a0 --- last

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
r7 = 7
r8 = 8.
r9 = 9.
r10 = 10.
r11 = 11.
r12 = 12.
r13 = 13.
r14 = 14.
r15 = 15.

cbit7 = 177
allbits = -1

Subroutines

IncrementPtrMAR: Increment operation packet pointer (low byte) in r14, and put
addi  warr 1, r14  ; result in r14 and MAR right
dstd  war1 rtx r16  ; carry propagate for high byte in r16 &
                  ; put in MAR left - return

Restore10111415: Restore registers 10, 11, 14, and 15 from extbuf
                  External buffer points to last entry put there; if none there,
                  is I less than available spot.
dst  mr r15  ; retrieve r16
addi  warr -1, r12
rsbuci  warl 0, r13
dst  mr r14  ; retrieve r14
addi  warr -1, r12
rsbuci  warl 0, r13
dst  mr r11  ; retrieve r11
addi  warr -1, r12
rsbuci  warl 0, r13
dst  mr rtx r10  ; retrieve r10
Save10111415:  ; ; ; ; Save registers 10, 11, 14, 15 in external buffer
; ; ; ; external buffer pointer is assumed to be pointing to last item
; ; ; ; stored in extbuf; if none there, is 1 less than available spot
addi  warr 1, r12  ; ; increment external buffer ptr
dstc  warl r13    ; ; & set MAR
dst  n wmm r10    ; ; r10 -> extbuf
addi  warr 1, r12  ; ; increment external buffer ptr
dstc  warl r13    ; ; & set MAR
dst  n wmm r11    ; ; r11 -> extbuf
addi  warr 1, r12  ; ; increment external buffer ptr
dstc  warl r13    ; ; & set MAR
dst  n wmm r14    ; ; r14 -> extbuf
addi  warr 1, r12  ; ; increment external buffer ptr
dstc  warl r13    ; ; & set MAR

NaNFr:  ; ; ; ; Produce NaN by setting fraction field to something diagnostic.
; ; ; ; Actually, the caller should indicate what sort of problem
; ; ; ; there was so NaNFr can produce something meaningful.
; ; ; ; But (for ifAdd at least) NaN is produced for only
; ; ; ; improper infinity arithmetic, and even so there are no plans
; ; ; ; for using any encoded information, so it doesn't matter what the
; ; ; ; fract field is as long as it is nonzero.
srcl  rtm allbits, r9  ; ; r9 is the first fraction byte, which
; ; ; ; when repacked, is put in r6, without 1st bit.


FAdd:  jsr  IncPktPtrMAR  ; ; incr packet ptr, store in MAR
dst  mr r3   ; ; r3 <- a3
jsr  IncPktPtrMAR
dst  mr r2   ; ; r2 <- a2
dst  ls c r2  ; ; take off low order exp bit
dst  ls rc r3  ; ; put on r3, take off sign
dst  rs rc r2  ; ; put sign on r2

; ; ; ; At this point, could test for error values of A rather than
; ; ; ; reading in the rest of A
jsr  IncPktPtrMAR
dst  mr r1   ; ; r1 <- a1
jsr  IncPktPtrMAR
dst  mr r0   ; ; r0 <- a0
src  r2, r6
andd  CBIT7, r6  ; ; get rid of sign bit
nandd  n ALLBITS, r3  ; ; test if A = NaN part 1 - is Max E?
jmp  ne  A1sN
dst  n r6
jmp  ne  ANaN
dst  n r1
jmp  ne  A1sN
dst  n r0
jmp  eq  A1sN  ; ; |A| = Infinity but need to test if B is NaN

ANaN:  ; ; ; ; A is NaN, so result <- A
src  r3, r7
src  r2, r6
src  r1, r5
src  r0, r4
jmp  FARepackX

A1sN:  jsr  IncPktPtrMAR
dst  mr r7   ; ; r7 <- b3
jsr  IncPktPtrMAR
dst  mr r6   ; ; r6 <- b2
dst  ls c r6  ; ; take off low order exp bit
dst  ls rc r7  ; ; put on r7, take off sign
dst  rs rc r6  ; ; put sign on r6

; ; ; ; At this point, could test for error values of B rather than
; reading in the rest of B
jsr IncPtrMAR
dst mr r6 : r5 <-- b1
jsr IncPtrMAR
dst mr r4 : r4 <-- b0
src r6, r9
andi CB177, r9 : get rid of sign bit
andi n ALLBITS, r7 : test if B = NaN part 1
jmp ne BisN
dst n r9
jmp ne BNaN : test if NaN
dst n r5
jmp eq BNaN ; |B| = Infinity so bypass Zero test

dst n r4
BNaN: ; B is NaN, so result <- B
jump FAREpackX

BisN: ; Test for |A|=0=|B|
; Actually, this test is unneeded, although specified by
; Coonen to allow "narrow rounding precision" to occur
; (Computer p76).
dst n r3
jmp ne IsAInf ; test if Exp[A] = 0 (Min E)
dst n r7
jmp ne ABAdd ; test if Exp[B] = 0
dst n r8
jmp ne ABAdd ; test if MSByte[A] = 0
dst n r9
jmp ne ABAdd ; test if MSByte[B] = 0
dst n r1
jmp ne ABAdd ; [A]
dst n r6
jmp ne ABAdd ; [B]
dst n r0
jmp ne ABAdd ; [A]
dst n r4
jmp ne ABAdd ; [B]

; |A| = 0 = |B|
and r2, r5 ; AND sign bits: assume Round to Nearest
jump FAREpackX

BisInf: nandi n allbits, r3 ; know |B| = Infinity via NaN test
jmp ne FAREpackX ; test if |A| = Infinity; if not, premim
; result already in r7 to r4
; already tested for A NaN, so |A| = Infinity

; |A| = Infinity = |B|
; if Affine ...
eqv n r2, r6 ; if r2 & r6 have same sign
jmp ne ProjTest ; then valid for Affine
jsr NaNFr ; else not, so produce NaN [by filling fract
; field with (non-)diagnostic info of some
; sort]
jump FAREpackX

ProjTest: ; Coonen's standard suggests the projective infinity arithmetic
; system as default, so it is used here:
jsr NaNFr ; so make NaN (set Fract = 0)
jump FAREpackX

IsAInf: ; |B| = Infinity: test if |A| = Infinity
nandi n allbits, r3
jmp ne ABAdd ; if |A| = Infinity go add A & B
src r3, r7 ; |A| = Infinity so put it in r7 to r4
src r5, r6
src r1, r6
src r0, r4
Floating Point Add Program / Coonen

jmp FARunpackX

ABAdd: 0 = |A|, |B| < Infinity; but if |A| = 0 then |B| → 0
Cases b, c, d, e on page 75 of Computer January 1980
Put (previously implicit) leading bit in place of sign,
put A & B MSBs in r8 & r9
src 1s r2, r8
dst n r3
jmp eq AleadBit ; check if lead bit should = 0
sec

AleadBit:
dst rs rc r8 ; shift in lead bit for A
src 1s r6, r9 ; move B's MSByte to r9, shift out sign
dst n r7
jmp eq BleadBit ; check if lead bit should = 0
sec

BleadBit:
dst rs rc r9 ; shift in lead bit for B
jsr Save10111115 ; save registers 10, 11, 14, 15 in ExtBuf
zero r10
zero r11

... Align binary points by coercing exponents to whichever is larger.
... and shifting mantissas.
src r3, r14 ; r14 ← Exp[A]
rsub r7, r14 ; r14 ← Exp[A] - Exp[B]
... subtraction of unsigned numbers
jmp eq Aligned ; jump if exponents same
jmp hi ExpAgkB ; jump if Exp[A] > Exp[B]

subi 0, r14 ; r14 ← Exp[B] - Exp[A] ... positive
src r7, r3 ; Exp[A] ← Exp[B]

ExtFrl = 37 ... 37 (octal) for extended format
... don't want to shift forever, so maximum shift = length of
... fraction field (extended)
rsubi n ExtFrl, r14 ; subtract length of extended fract field
jmp los ARSSetup ; jump if exponent difference << fract length
src ExtFrl, r14 ; r14 ← max fract length

ARSSetup:
dst n r14
ldct reg ; load addr/count reg with what's in r14

ARSLoop:
dst rs un r8 ; shift right A's MSByte
dst rs rc r1 ; propagate shift to A's 2nd byte
dst rs rc r0 ; to least (non-extended) byte
dst rs rc r10 ; to least-extended byte
... don't use sticky bit
count ARSSloop ; decrement, loop if count ≠ 0
jmp Aligned

ExpAgkB:
src r3, r7 ; Exp[B] ← Exp[A]
rsubi n ExtFrl, r14 ; subtract length of fract field
jmp los BRSSetup ; jump if Exp difference << fract length
src ExtFrl, r14 ; r14 ← max fract length

BRSSetup:
dst n r14
ldct reg ; load addr/count reg with what's in r14

BRSLoop:
dst rs un r9 ; shift right B MSByte
dst rs rc r5 ; propagate shift B 2nd byte
dst rs rc r4 ; to least (non-extended) byte
dst rs rc r11 ; to least-extended byte
... don't use sticky bit
count BRSLoop ; decrement, loop if ≠ 0
Aligned:

:: Binary points are aligned
:: A B
:: r3 r7 exponent
:: r14 r15 exponent-extended (not used for Add)
:: r8 r9 MSByte
:: r1 r5 2nd
:: r0 r4 least
:: r10 r11 least-extended
:: r2 r8 1st bit = sign

eqv n c r2, r6 ; find same bits, set c bit = sign
jmp cc DifSign  ; jump if Sign[A] == Sign[B]

:: Same sign for Add
add r10, r11 ; add least-extended bytes
addc r0, r4 ; least
addc r1, r6 ; 2nd
addc r8, r9 ; most
jmp cc Normalize ; jump if no carry
dst rs rc r9 ; shift result to put carry (load bit) in
dst rs rc r5 ; propagate
dst rs rc r4 ;
dst rs rc r11 ;

zero ls u r3 ; put right-shift carry-out bit in bit 0 of r3
or r3, r11 ; OR shifted out bit into r11 (sticky bit)

inc r7 ; increment result exponent since lead bit
; shifted into MSB. Should not set C, since
; Max E = all 1's ones is reserved for
; Infinity & NaN previously caught.
; However, if exponent now = the max,
; there is an overflow, caught later

:: The explicit leading bit (now in r9) will be thrown away
:: when normalizing the number
jmp Normalize

DifSign:: A & B have different signs
:: subtract B from A
sub r10, r11 ; C is 0 if r10 < r11
subic r0, r4 ; C is 0 if r0 < r4
subic r1, r5
subic r8, r9
jmp cs SignofA ; jump if |A| > |B|
:: |A| < |B| -- de-negate result
ndst r11 ; two's complement
cdstc r4 ; propagate
cdstc r5

cdstc r9 ;
:: a carry from here is not relevant
:: result gets sign of B, which is already in r6
jmp ZeroCheck

SignofA:: result gets sign of A
src r2, r6 ; move r2 to r6 for sign only

ZeroCheck::

dst n r9 ; is msbyte zero?
jmp ne Normalize ; no - jump
dst n r6
jmp ne Normalize
dst n r4
jmp ne Normalize
dst n r11 ; is least-extended byte zero?
jmp ne Normalize ; no - jump
:: result is zero
zero r7 ; set exp for minimum
zero r8 ; sign + (assuming Round to Nearest)
Floating Point Add Program / Coonen

```assembly
: : result now in r7 to r4
jmp FARepackX

Normalize:
: : Convert result to the normal form
: : which here means that r9 (high order mantissa byte) gets
: : the 1st 7 bits of mantissa fraction field, and
: : msb(r9)=leading bit;
: : r5 and r4 get 8 bits each. Note that the explicit leading
: : bit will become implicit when unpacked.

Normalloop:

dst n c r9 : is msb of mantissa's msbyte 1?
jmp ne Round : yes - time to round
rsubi 0, r7 : decrement exponent; is exp < zero?
jmp lo Denorm : yes - go denormalize result
dst ls c r11 : left shift lsbyte extended of result
dst ls rc r4 : propagate
dst ls rc r6 : 
dst ls rc r9 : " to msbyte
jmp NormalLoop : loop

Denorm:
: : Can't fit as normalized, shifted left as far as can
: : zero r7
: : Result in r7 (exponent): r9 (msbyte), r6, r4 (lsbyte);
: : and lower significant bits in r11; sign in r6
: : so round the result to fit.

Round:
: : There is no need to check for underflow as long as the
: : destination is single precision for single precision operands.

: : Assume Round to Nearest (RW)
: : r11 contains the extra bits
: : r4 bit 0 is LSBit

Cases:
: : r4 bit 0
: : Case: r11 Do this
: : 0 0 same [exact] case 1
: : 1 0 ** same case 2
: : 0 < 100.. same [truncate] case 3
: : 1 0 ** same case 4
: : 0 100.. same [LSB 0] case 5
: : 1 0 ** add 1[LSB 0] case 6
: : 0 > 100.. add 1[Round up] case 7
: : 1 0 ** case 8

: : So to get desired results,
: : - add MSB(r11) to r4 except when LSB(r4) = 0 = Left_Shift(r11)

dst n r11 : is least-extended byte 07
jmp eq Exact : yes - no need to round (cases 1, 2)
zero nq
dst ls rd r11 : MSB(r11) -> LSB(q); r11 shifted left
dst un r4 : LSB(r4) -> C bit
dst rs rdc r11 : shift that C bit into MSB(r11); LSB(q) -> C
jmp los Inexact : [los = ~C | Z]
: [Z bit on:] if original 1sb(r4) &
: left_shift(original r11) = 0, then
: jump, as r4 etc. stays same. (case 5)
: [C bit off:] if original msb(r11)
: zero, no need to add (cases 3, 4)

dstc r4 : else add C bit to low byte (cases 6, 7, 8)
dstc r6 & propagate
dstc r9 : 
jmp cc Inexact : jump if no carry
: : if carry here, then have to increment exponent, and shift
: : r9, r5, and r4 right.
inc r7 : increment exponent (overflow caught later)
sec
dst rs rc r9 : shift in carry-out which required the
: exponent incremented
```
Floating Point Add Program / Coonen

\begin{verbatim}

dst rs rc r6       ; & propagate
dst rs rc r4       ;
zero ls u r3       ; put right-shift carry-out bit in bit 0 of r3
or r3. r4          ; OR shifted out bit into r4 (sticky bit)

Inexact: ... fall through

Exact:

... Check for overflow
xorw m slfbits. r7  ; is exp all ones?
jmp ne FARepack    ; no - jump
zero r9             ; fract field all zeroes indicates infinity
zero r6             ; (sign field retained, from r6)
zero r4             ;

FARepack:

dst ls r9           ; throw away explicit leading bit
dst n c r5           ; get sign bit
dst rs rc r7         ; put sign bit on r7, take off exp low bit
src rs rc r9, r6     ; put low exp bit in top bit of 2nd highest byte, put result in r6, so

... entire single precision result is in r7 through r4.

Pre Deliver:

jsr restore01111416   ; restore saves registers from external buf

Deliver: ... Results are left in r7-r4 (mbyte - 1sbyte)
rtn

FARepackX:

dst ls c r6         ; take off sign bit
dst rs rc r7         ; put on r7, take off low order exp bit
dst rs rc r6         ; put on r6
jmp Deliver
\end{verbatim}
Floating Point Add Program / Error-Byte

Appendix D - Floating Point Add Program / Error-Byte

Errors encoded in first byte - floating point Add

Floating point numbers should arrive in the following format:

RISEEEEE EEEEFF FF FF FFF FF FF FF FF FF FF FF FF FF FF FF FF FF

if R = 1
then # is an error value
1st byte encodes the error value; other bytes ignored.
else # is a representable integer
S = sign
8 E bits = exponent
22 F bits = fractional part of mantissa
if exponent = minimum (all zeroes)
then number is zero (i.e. no denormalized numbers)
else biased exponent can range from 1 to maximum (all ones)
(i.e. from 1 to 2^8-1 = 255); bias is 128, so true exponent ranges from -127 to +127.
endall

Error values:
76543210 bit7 on if error; bit6 on if neg; bit5 on if over; bit4 on if under;
(bit3 or bit2) on if undef, miss_eit, or zero_divide.
10100000 pos_over
11000000 neg_over
10010000 pos_under (Not applicable to Integers)
11010000 neg_under (ditto)
10000000 unknown
10000100 undef
10001000 miss_eit
10001100 zero_divide

Non-error valued numbers are unpacked to be as:

a3 - Exponent --- first
a2 - sign bit, fract MSB 6 bits
a1 - fract 8 bits
a0 - fract LSB 8 bits
b3 - like a3
b2 - like a2
b1 - like a1
b0 - like a0 --- last

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
r7 = 7
r8 = 8
r9 = 9
r10 = 10
r11 = 11
r12 = 12
r13 = 13
r14 = 14
r15 = 16.

Bit7 = 200
Bit6 = 100
Bit5 = 40
Bit4 = 20
CBit7 = 177
MiscErrs = 14
ErUndef = 204

---
Allbits = -1 ; 11111111

; subroutine

IncPktPrrMAR: ; increment operation packet pointer (low byte) in r14, and put
addi warr 1, r14 ; result in r14 and war right
addi warr 1, r15 ; carry propagate for high byte in r15 &
dstc warr 1, r15 ; put in MAR left - return

Restore10111415: ; restore registers 10, 11, 14, and 15 from extbuf
; external buffer points to last entry put there; if none there,
; is 1 less than available spot.
dst mr r15 ; retrieve r15
addi warr 1 -1, r12
rsbclci warr 1 0, r13
dst mr r14 ; retrieve r14
addi warr 1 -1, r12
rsbclci warr 1 0, r13
dst mr r11 ; retrieve r11
addi warr 1 -1, r12
rsbclci warr 1 0, r13
dst mr rtn r10 ; retrieve r10

Save10111415: ; save registers 10, 11, 14, 15 in external buffer
; external buffer pointer is assumed to be pointing to last item
; stored in extbuf; if none there, is 1 less than available spot
addi warr 1, r12 ; increment external buffer ptr
dstc warr 1, r13 ; & set MAR
dst n mm r10 ; r10 -> extbuf
addi warr 1, r12 ; increment external buffer ptr
dstc warr 1, r13 ; & set MAR
dst n mm r11 ; r11 -> extbuf
addi warr 1, r12 ; increment external buffer ptr
dstc warr 1, r13 ; & set MAR
dst n mm r14 ; r14 -> extbuf
addi warr 1, r12 ; increment external buffer ptr
dstc warr 1, r13 ; & set MAR
dst n mm rtn r15 ; r15 -> extbuf

geta210: ; read in a2, a1, a0; packet pointer assumed to be pointing to a3
jsr IncPktPrrMAR
dst mr r1 ; r2 <- a2
jsr IncPktPrrMAR
dst mr r1 ; r1 <- a1
jsr IncPktPrrMAR
dst mr rtn r0 ; r0 <- a0

getb210: ; read in b2, b1, b0; packet pointer assumed to be pointing to b3
jsr IncPktPrrMAR
dst mr r6 ; r6 <- b2
jsr IncPktPrrMAR
dst mr r6 ; r5 <- b1
jsr IncPktPrrMAR
dst mr rtn r4 ; r4 <- b0

UnpackA & UnpackB unpack from
; HEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE EEEEEEEE to
; r7 [exponent]
; r6 [sign: 7 bits]
; r5 [6 bits]
; r4 [7 bits]

UnpackA:
dst ls c r0 ; shift fraction left 1 bit through r2
dst ls rc r1 ;
dst ls rc r2 ; put fract bit in r2; remove exp bit (a1)
dst ls ur3 ; put e1 on r3; remove error bit (& throw away)
dst ls cr2 ; take off low order exponent bit
dst ls rc r3 ; put it on r3; take off sign bit
dst rs rc rthr2 ; put sign on r2

UnpackB:

dst ls cr4 ; shift fraction left 1 bit through r6
dst ls cr r5 ;
dst ls rc r6 ; put frac bit in r6; remove exp bit (e1)
dst ls ur7 ; put e1 on r7; remove error bit (& throw away)
dst ls cr r6 ; take off exponent lsb (e0)
dst ls rc r7 ; put it on r7; take off sign bit
dst rs rc rthr6 ; put sign on r6

EFRepack: ... rpacks from
... r7 [exponent byte]
... r6 [sign bit; 6 high fract bits]
... r5 [8 fr bits]
... r4 [8 low fr bits]
... to RSEEEEE EEEFFFF FFFFFF FF FFFFFF
... (Note that the format repacked from is different from the
... format unpacked to. This is because there is a conversion from
... one to the other for the rounding routine.)
dst ls cr r6 ; take off sign bit
dst rs dc r7 ; put on r7; take off exp lsb (e0) (-> msb(q))
dst lslg r r6 ; rotate unused bit of r6; e0 of q to lsb(q)
dst rs rd r6 ; put e0 on r6
dst rs un r7 ; take off exponent bit (e1)
dst rs rc r6 ; put on r6
jmp Deliver

Efadd: jsr IncPktPtrMAR
dst mr r3 ; r3 <= a3
andi n bit7, r3 ; Is A an error value?
jmp no EFAerr
; yes - jump
dst warl 4, r14
addi warl r16

dst mr r7 ; r7 <= b3
andi n bit7, r7 ; Is B an error value?
jmp no EFBerrAnerr
; yes - jump

; neither A nor B are error values, so get b2, b1, b0
jsr getb210
addi -7, r14 ; decrement packet ptr
rsublic 0, r15 ; borrow propagate
jsr geta210 ; - & get a2, a1, a0

; A & B are not error values; could be zeroes
; unpack exponent, move sign ...
jsr unpackA

; Now r3 = exponent; r2 has sign bit, unused bit, and 7 fract bits;
; r1 has 8 fract bits; r0 has 7 fract bits
jsr unpackB
jmp EFAAdd

EFAerr: ... A is an error value
andi n miscerrs, r3 ; is A undef, miss_elt, zero_divide?
jmp ne fSetundef
; yes - result -- undef
andi n cb177, r3 ; well, is A unknown?
jmp eq fSetunk
; yes - result -- unknown

; so A is an under or overflow - go get B
addi warl 4, r14
dst warl r16
dst    nr r7        ; r7 <- b3
andi   n bit7, r7  ; is B an error value?
jmp eq AvundBnerr  ; no - jump
andi   n miscerrs, r7 ; is B undef, miss handleClick, zero_divide?
jmp ne FSetundef  ; yes - result <- undef
andi   n cbit7, r7  ; well, is B unknown?
jmp eq FSetunk     ; yes - result <- unkown

::: so A & B are both underflows or overflows

::: if A & B have different signs
:::    then result <- unknown
:::    else if A is an overflow
:::        then result <- A
:::    else result <- B

eqv   nq r3, r7
andi   n b6,       ; are sign bits (bit 6) same?
jmp eq FSetunk     ; no - result <- unknown
andi   n bit5, r3  ; is A an overflow?
jmp ne FErrresA    ; yes - result <- A
jmp    Deliver      ; else result <- B

AvundBnerr:
::: A is an overflow or underflow, B is not an error
jsr    getb210
::: if A is an underflow
:::    then if B = 0.0
:::    then result =< B
:::    else result =< A
:::    else if A & B have same signs (A is overflow)
:::        then result =< A
:::        else if B = 0.0
:::        then result =< A
:::    else result =< unknown
:::    endif
andi   n bit4, r3  ; is A an underflow?
jmp eq EFAov       ; no - jump
dst    n r7         ; is b3 (exponent) zero? [if yes, then b = 0]
jmp ne Deliver      ; no -
::: well B is 0, so result =< A

EFresA: src r3, r7  ; result is A, so move to r7-r4.
src r2, r6	src r1, r5
src r0, r4
jmp    Deliver

EFAov: 
::: A is an overflow, B is not an error.
::: if same sign, result =< A
eqv   nq r2, r6
andi   n b6,       ; are sign bits (bit 6) same?
jmp ne FErrresA    ; yes - jump: result =< A

::: well, if B is zero, then result =< A
dst    n r7         ; is b3 (exponent) zero? [if yes, then b = 0]
jmp ne FSetunk
::: so result =< A

FEErrresA:
src r3, r7
::: r6 through r4 can be left with whatever they contain, since
::: with the error bit on, all bytes other than the error
::: byte are ignored.
jmp    Deliver

EfberrAnerr: ::: B is an error value, A is not.
andi   n miscerrs, r7 ; is B undef, miss handleClick, zero_divide?
Floating Point Add Program / Error-Byte

```
jmp ne FSetUndef ; yas - set result undefined
andl n cbit7, r7 ; well, is B unknown?
jmp eq FSetunk ; yas - set result unknown

; so B is {pos or neg} {under or over} flow
andl n bit4, r7 ; is B an underflow?
jmp eq EFBov ; no - jump
dst n r3 ; is a3 (exponent) zero? [if so, then A=0]
jmp ne EPresA ; no -

; so result <-- B
jmp Deliver

EFBov: ; B is an overflow, A is not an error.

; if same sign, result <-- B
eqv nq r2, r8
andq n bit6 ; are sign bits (bit 6) same?
jmp ne Deliver ; yes - jump: result <-- B

; well, if A is zero, then result <-- B
dst n r3 ; is a3 (exponent) zero? [if so, then A=0]
jmp ne FSetunck ; so result <-- B
jmp Deliver

FSetunk: ; Set result to be Unknown
srci bit7, r7
jmp Deliver

FSetUndef: ; Set result to be undefined
srci EUndef, r7
jmp Deliver

EFABAdd: ; 0 <= |A|, |B| < Pos_over
; A & B might both be 0.
; Put (previously implicit) leading bit in place of sign.
; put A & B MSBs in r8 & r9
srci rs r2, r8
dst n r3
jmp eq ALeadBit ; check if lead bit should = 0
sec

ALeadBit:
dst rs rc r8 ; shift in lead bit for A
srci rs r6, r9
_dst n r7
jmp eq BLEadBit ; check if lead bit should = 0
sec

BLEadBit:
dst rs rc r9 ; shift in lead bit for B
jsr Save010111415 ; save registers 10, 11, 14, 15 in ExtBuf
zero r10
zero r11

; Align binary points by coercing exponents to whichever is larger,
; and shifting mantissas.
srci r3, r14 ; r14 <-- Exp[A]
rsub r7, r14 ; r14 <-- Exp[A] - Exp[B]

; subtraction of unsigned numbers
jmp eq Aligned ; jump if exponents same
jmp hi ExpAgtB ; jump if Exp[A] > Exp[B]
subi 0, r14 ; r14 <-- Exp[B] - Exp[A] ... positive
srci r7, r3 ; Exp[A] <-- Exp[B]

Effxtfrl = 36 ; 36 (octal) for extended format
; don't want to shift forever, so maximum shift = length of
; fraction field (extended)
rsubi n EFExtfrl, r14 ; subtract length of extended fract field
```
jmp los ARSSsetup ; jump if exponent difference << Fract length
srci EffeXFrL, r14 ; r14 <= max fract length

ARSSsetup:
dst n r14
ldct reg ; load addr/count reg with what's in r14

ARSSloop:
dst rs un r8 ; shift right A's MSByte
dst rs rc r1 ; propagate shift to A's 2nd byte
dst rs rc r0 ; to least (non-extended) byte
dst rs rc r10 ; to least-extended byte
::: don't use sticky bit
count ARSSloop ; decrement, loop if != 0
jmp Aligned

ExpAgtB:
src r3, r7 ; Exp[B] <- Exp[A]
rsubi n EffeXFrL, r14 ; subtract length of fract field
jmp los ARSSsetup ; jump if Exp difference << Fract length
srci EffeXFrL, r14 ; r14 <= max fract length

BRSSsetup:
dst n r14
ldct reg ; load addr/count reg with what's in r14

BRSSloop:
dst rs un r9 ; shift right B MSByte
dst rs rc r5 ; propagate shift B 2nd byte
dst rs rc r4 ; least (non-extended) byte
dst rs rc r11 ; least-extended byte
::: don't use sticky bit
count BRSSloop ; decrement, loop if != 0

Aligned: ::: Binary points are aligned
::: A B
::: r3 r7 exponent
::: r14 r15 exponent-extended (not used for Add)
::: r9 r9 MSByte
::: r5 2nd
::: r0 r4 least
::: r10 r11 least-extended
::: r2 r6 1st bit = sign
eqv n c r2, r6 ; find same bits, set c bit = sign
jmp cc DiffSign ; jump if Sign[A] != Sign[B]

::: Same sign for Add
add r10, r11 ; add least-extended bytes
addc r0, r4 ; least
addc r1, r6 ; 2nd
addc r8, r9 ; most
jmp cc Normalize ; jump if no carry
dst rs rc r9 ; shift result to put carry (lead bit) in
dst rs rc r6 ; propagate
dst rs rc r4 ;
dst rs rc r11 ;

zero ls u r3 ; put right-shift carry-out bit in bit 0 of r3
or r3, r11 ; OR shifted out bit into r11 (sticky bit)
inc r7 ; increment result exponent since lead bit shifted into MSB.
jmp cc Normalize ; if increment didn't have a carry-out, then
Setov: ; normalize

overd:
srci bit7 & bit8, r7 ; put error and overflow bits in r7
and r6, r6 ; get sign bit in r6 as 0S000000
or r6, r7 ; put sign bit in r7
Floating Point Add Program / Error-Byte

Appendix D

jmp Deliver ; r7 has error byte; r6 - r4 has mantissa of number whose exponent overflowed (unrounded).

DifSign: :: A & B have different signs
:: subtract B from A
sub r10, r11 ; C is 0 if r10 < r11 (unsigned); otherwise 1
subl r0, r4 ; C is 0 if r0 < r4
subl r1, r5
subl r8, r9
jmp cs SignofA ; jump if |A| > |B|
:: |A| < |B| -- de-negate result
ndst r11 ; two's complement
cdstc r4 ; propagate
cdstc r5 ; *
cdstc r9 ; *
:: carry from here is not relevant (only occurs if number is zero)
:: result gets sign of B, which is already in r6
jmp ZeroCheck

SignofA: :: result gets sign of A
src r2, r6 ; move r2 to r6 for sign only

ZeroCheck:
dst n r0 ; is msbyte zero?
jmp ne Normalize ; no - jump
dst n r6
jmp ne Normalize
dst n r4
jmp ne Normalize
dst n r11 ; is least-extended byte zero?
jmp ne Normalize ; no - jump
:: result is zero
zero r7 ; set exp for minimum
zero r6 ; sign + (assuming Round to Nearest)
:: result now in r7 to r4
jmp Deliver ; (for +0, repacking is unnecessary)

Normalize: :: Convert result to the normal form
:: which here means that r9 (high order mantissa byte) gets
:: the 1st 7 bits of mantissa fraction field, and
:: msb(r0)=leading bit;
:: r5 and r4 get 8 bits each. Note that the explicit leading
:: bit will become implicit when repacked.

Normalloop:
dst n c r9 ; is msbit of mantissa's msbyte 1?
jmp ne EFRound ; yes - time to round
rsub1 0, r7 ; decrement exponent; is exp < zero?
jmp lo Setunder ; yes - result is an underflow
dst ls c r11 ; left shift 1sbyte extended of result
dst ls rc r4 ; propagate
dst ls rc r5 ; *
dst ls rc r9 ; * to msbyte
jmp Normalloop ; loop

Setunder: :: result has underflowed (no denormalized numbers used)
src bit7 & bit4, r7 ; put error and underflow bits in r7
jmp ovund

EFRound: :: Since in EffAdd there are 22 fraction bits as compared with 23
:: in Add, and rounding must work the same in both, EffAdd has to
:: shift the fraction registers right to permit proper (and
:: easier) rounding.
dst rs un r9 ; top bit now zero
dst rs rc r6
dst rs rc r4 ; since want to round relative to 22nd bit
dst rs rc r11 ; 23rd bit to 30th
Round:  
There is no need to check for underflow as long as the  
destination is single precision for single precision operands.

Assume Round to Nearest (RN)  
r11 contains the extra bits  
r4 bit 0 is LSB

Cases:

- r4 bit 0    r11  Do this
  0    0  same [exact]  case 1
  1    "  "  "  case 2
  0    < 100  same [truncate]  case 3
  1    "  "  "  case 4
  0  > 100  same [LSB 0]  case 5
  1  add 1 [LSB 0]  "  case 6
  0  > 100  add 1 [Round up]  case 7
  1  "  "  "  case 8

So to get desired results,
- add MSB(r11) to r4 except when LSB(r4) = 0 = Left_Shift(r11)

dst   n r11  : is least-extended byte 0?
jmp eq Exact  : yes - no need to round (cases 1, 2)
zero  nq

dst   ls rd r11  : MSB(r11) -> LSB(q); r11 shifted left

dst   un r4  : LSB(r4) -> C bit

dst   rs rd rc r11  : shift that C bit into MSB(r11); LSB(q) -> C

jmp los Inexact  : [los = -C \oplus 2]

[jump on:] if original lsbr(r4) &
left_shift(original r11) = 0, then
jump, as r4 etc. stays same (case 6)
[C bit off:] if original msbr(r11)
zero, no need to add (cases 3, 4)
else add C bit to low byte (cases 6, 7, 8)

dst   r4  & propagate

dst   r5  :

dst   rc  :

jmp cc Inexact  : jump if no carry

[jump here, then have to increment exponent, and shift
r9, r5, and r4 right]

inc r7  : increment exponent (overflow caught later)

jmp cs Setover  : if increment set carry-out bit, exponent has

overflowed, so result is overflow

sec

dst   rs rc r9  : shift in carry-out which required the

 exponent incremented

dst   rs rc r5  & propagate

dst   rs rc r4  :

zero  ts wr r3  : put right-shift carry-out bit in bit 0 of r3
or  r3, r4  : OR shifted out bit into r4 (sticky bit)

Inexact:
Exact:

jmp  EfRepack  : fract bytes of 6 bits, 8, 8 --> Ef format

Deliver:

Entire single precision result is in r7 through r4.
Results are left in r7-r4 (msbyte - lsbyte)
jsr  restore10111415  : restore saves registers from external buf

rtn
Appendix E - Floating Point Multiply Program

; *PU*- Thursday 22 May 1980 11:15 am
; Floating Point Multiplication - Coonen proposal

; Denormalized operands are not handled.

; As per Coonen's proposed IEEE floating point standard
; Described in Signum Newsletter special issue, October 1970; and
; Computer (IEEE), January 1980
; - single precision only
; - without exception traps or signals
; - with denormalized numbers
; - using round to nearest
; - using projective infinity arithmetic (+Infinity = -Infinity)

; floating point numbers should arrive in the following format:
; SEEEEE EEEEEEEE EEEEE EEEEE EEEEE
; so they are unpacked to be as:
; a3 - Exponent --- first
; a2 - Sign bit, Fract MSB 7 bits
; a1 - Fract 8 bits
; a0 - Fract LSB 8 bits
; b3 - like a3
; b2 - like a2
; b1 - like a1
; b0 - like a0 --- last

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
r7 = 7
r8 = 8.
r9 = 9.
r10 = 10.
r11 = 11.
r12 = 12.
r13 = 13.
r14 = 14.
r15 = 16.

cbits7 = 177
a1bits = -1
b1bits = 200
bias = 177

DecExBufMAR: 
; Decrement external buffer pointer and write into MAR
addi   warr -1, r12
rsb  warr 0, r12
; decrement
borrow propagate

IncPktPtrMAR: 
; Increment operation packet pointer (low byte) in r14, and put
addi   warr 1, r14
; result in r14 and MAR right
dstmt warr r15
; carry propagate for high byte in r16 &
; put in MAR left - return

Restore10111415: 
; restore registers 10, 11, 14, and 15 from exbuf
; External buffer points to last entry put there; if none there,
; is 1 less than available spot.
dstd warr r15
addi   warr -1, r12
; retrieve r16
Floating Point Multiply Program

rsubci warl 0, r13  ; retrieve r14
dst   mr r14
addi warl -1, r12
rsubci warl 0, r13  ; retrieve r11
dst   mr r11
addi warl -1, r12
rsubci warl 0, r13  ; retrieve r10
dst   mr rtn r10

Save10111115:  ::: Save registers 10, 11, 14, 15 in external buffer
::: external buffer pointer is assumed to be pointing to last item
::: stored in extbuf; if none there, is 1 less than available spot
addi warl 1, r12  ; increment external buffer ptr
dstc  warl r13  ; ... & set MAR
dst n wmr r10  ; r10 -> extbuf
addi warl 1, r12  ; increment external buffer ptr
dstc  warl r13  ; ... & set MAR
dst n wmr r11  ; r11 -> extbuf
addi warl 1, r12  ; increment external buffer ptr
dstc  warl r13  ; ... & set MAR
dst n wmr r14  ; r14 -> extbuf
addi warl 1, r12  ; increment external buffer ptr
dstc  warl r13  ; ... & set MAR
dst n wmr rtn r15  ; r15 -> extbuf

NaFR: ::: Produce NaFR by setting fraction field to something diagnostic.
::: Actually, the caller should indicate what sort of problem
::: there was so NaFR can produce something meaningful.
::: But (for FAdd at least) NaFR is produced for only
::: improper infinity arithmetic, and even so there are no plans
::: for using any encoded information, so it doesn't matter what the
::: Fract field is as long as it is nonzero.
scl rtm allbits, r9  ; r9 is the first fraction byte, which
::: when re-packed, is put in r6, without 1st bit.

-------------------------------
FMult: jsr IncPktPtrMAR
dst   mr r3  ; r3 <= a3
jsr IncPktPtrMAR
dst   mr r2  ; r2 <= a2
dst   ls c r2, r2  ; take off e0 (low exponent bit)
dst   ls rc r3, r3  ; put e0 into r3, take off sign
dst   ls rc r2, r2  ; put sign on r2

::: At this point, could test for error values of A rather than
::: reading in the rest of A
jsr IncPktPtrMAR
dst   mr r1  ; r1 <= a1
jsr IncPktPtrMAR
dst   mr r0  ; r0 <= a0
src r2, r8
andi cb17, r8  ; get rid of sign bit
nan d n allbits, r3  ; test if A = NaN (part 1) - is Maxim E?
jmp ne AisN  ; no - jump
dst n r8
jmp ne ANan  ; significand field non-zero, so NaN, jump
dst n r1
jmp ne ANan
dst n r0
jmp eq AisInf  ; |A| = Infinity but need to test if B is NaN

ANan: ::: A is NaN so result <= A
src r3, r7
src r2, r6
src r1, r6
src r0, r4
jmp FMrepacc
AisInf:

AisN:  
:: A is not NaN, so get B and see if B is NaN

    jsr IncPktPtrMAR
    dst mr r7  : r7 <= b3
    jsr IncPktPtrMAR
    dst mr r6  : r6 <= b2
    dst ls c r6, r6  : take off e0 (low exponent bit)
    dst ls rc r7, r7  : put e0 in r7, take off sign
    dst ls rc r6, r6  : put sign on r6

:: At this point, could test for error values of A rather than
:: reading in the rest of B

    jsr IncPktPtrMAR
    dst mr r5  : r5 <= b1
    jsr IncPktPtrMAR
    dst mr r4  : r4 <= b0

src r6, r9
andi cbit7, r9  : get rid of sign bit
nandi n allbits, r7  : test if B is NaN (part 1): is exp = max exp?

jmp ne BisN  : no - jump

dst n r9
jtmp ne BNaN   : NS significand byte == 0, so result is NaN

jtmp ne BNaN

dst n r5  : 2nd "

jtmp eq BisInf  : jump - ([B] = Infinity)

BNaN:  
:: B is NaN so result <= B

    jmp FMPrepack

BisInf:  
:: |B| = Infinity so test if |A| = 0

    dst n r3

jtmp no BlnfAnotZ  : jump if A's exp is non-zero

    dst n r8

jtmp no BlnfAnotZ  : jump if A's MSByte non-zero

    dst n r1

jtmp no BlnfAnotZ

    dst n r0

jtmp no BlnfAnotZ

:: |B| = Infinity & |A| = 0 so Invalid Operation results.
:: Assuming no ability to handle traps, so
:: produce some NaN.
:: NOTE that in VAL, 0.0 * Pos_Over produces 0.0, so if pos_over
:: corresponds to + Infinity, the Coonen result specific is different.
:: There are other cases in which the results of operations
:: involving infinity operands do not concur with VAL's specs for
:: pos_over or neg_over, see below.

    jsr NaNFr    : produce some NaN

    jmp FMPrepack

BlnfAnotZ:
:: NOTE that in VAL,
:: Pos_Over * f [where 0.0 < |f| < 1.0] produces Unknown.
:: In Coonen standard, +Infinity * f produces Infinity with sign
:: of f.

:: Result <= B with signs XORed.

xor n c r2, r6

src rs rc r9, r6  : sign shifted in [r9 was 0]

    jmp FMPrepackX

BisN:  
:: B is not Infinity, but check if A is.

nandi n allbits, r3  : Since already checked if A is NaN.

jtmp no ABisN  : jump if |A| = Infinity (if exp >= max exp)

    dst n r7

jtmp ne AInifBnotZ  : jump if B's exp is non-zero

    dst n r9
Floating Point Multiply Program

; [A] = infinity & [B] = 0 so Invalid Operation
; assuming no ability to handle traps, so
; produce some appropriate NaN
; NOTI that in VAL, 0.0 * Pos_Over produces 0.0
; [see similar comment above]

jsr  NaNFr          ; produce some NaN
jmp   FMRepackX

AInfBnotZ:
; NOTE that in VAL,
; Pos_Over * f [where 0.0 < |f| < 1.0] produces Unknown.
; [see similar comment above]

; Result <= A with signs XORed.
xor   n c r2, r6
src   rs rc r8, r6          ; sign shifted in [r8 was 0]
src   r3, r7
src   r1, r5
src   r0, r4
jmp   FMRepackX

ABIsX:
; Both [A] & [B] are not NaNs, not Infinity,
; so let's see if they are zero.
dst   n r3
jmp   ne IsBZ               ; A's exp not zero, so go test B
dst   n r8          ; [if here, [A] is either 0 or denormalized]
jmp   ne IsBZ
dst   n r1
jmp   ne IsBZ
dst   n r0
jmp   ne IsBZ

; [A] is zero so Result <= Zero
xor   n c r2, r6          ; xor signs
src   rs rc r8, r6          ; put sign in high byte [r8 is 0]
zero  r7          ; exp
zero  r5          ; middle byte
zero  r4          ; low byte
jmp   FMRepackX

IsBZ:
; A not zero, so test if B is.
dst   n r7
jmp   ne ABFMult          ; B's exp not zero, so go multiply
dst   n r9          ; [if here, [B] is either 0 or denormalized]
jmp   ne ABFMult
dst   n r6
jmp   ne ABFMult
dst   n r4
jmp   ne ABFMult

; [B] is zero so Result <= Zero
xor   n c r2, r6          ; xor signs
src   rs rc r9, r6          ; put sign in high byte [r8 is 0]
    r7, r5, & r4 are already zero
jmp   FMRepackX

ABFMult:
; A & B are both representable non-zero numbers, multiply them.
; r7 - B exponent
; r9 - B most signific. byte with msb it 0
; r6 - B most signific. byte with msb it = sign
; r5 - B middle byte
Floating Point Multiply Program

;;; r4 - B least signific. byte
;;; r3 - A exponent
;;; r8 - A most signific. byte with msbit 0
;;; r2 - A most signific. byte with msbit = sign
;;; r1 - A middle byte
;;; r9 - A least signific. byte

;;; put leading bits in A & B's msbytes (r8 & r9)
dst n r7 ; if B denormalized,
  jmp eq TryA ; then jump since lead bit is 0
  ori bi7, r9 ; else put in lead bit of 1
TryA:
  jmp eq LeadBitsIn ; then jump since lead bit is 0
  ori bit7, r8 ; else put in lead bit of 1

LeadBitsIn:
  ;; multiplicant significand A in r8, r1, r0
  ;; multiplier significand B in r9, r5, r4
  jsr save10111415 ; save registers 10, 11, 14, 16 in ext buffer

  ;; add exponents
  src r7, r14
  add r3, r14 ; new exponent in r14 (with double bias)
  src 0, r15 ; & r15 for carry [we'll worry about
  ; overflow neg or pos later]
  dst ls r15 ; (Is for sign to be put in in the next
  ; few lines)

  ;; produce sign of result
  xor n c r2, r6 ; msbit = new sign
  dst rs rc r15 ; put sign in msbit of r15

  ;; registers now unneeded: r2, r3, r6, r7, r10, r11
  src r8, r2 ; so A significand is in r2, r1, r0
  src r9, r10
  src r5, r1
  src r4, r8 ; so B significand is in r10, r9, r8

  ;; registers now unneeded: r3, r4, r6, r5, r7, r8, r11
  zero r7
  zero r6
  zero r5
  zero r4

  ;; 000 Denormalized numbers complicate matters. Not handled.
  jsr FMAbMult ; B's LSBbyte * A
  src r9, r8
  jsr FMAbMult ; B's middle byte * A + previous result
  src r10, r8
  jsr FMAbMult ; B's MSByte * A + previous result

  ;; result should be in r6r5r4 & <ExtBuf> to <ExtBuf - 2>
  ;; so retrieve what is in ExtBuf
  dst war r12
  dst war1 r13
  dst mr r3
  jsr DecExtBufMR
  dst mr r2
  jsr DecExtBufMR
  dst mr r1
  jsr DecExtBufMR

  ;; Reset ext buf ptr to initial
  addi -1, r12 ; decrement
  rsubc i 0, r13 ; borrow propagate

  ;; Result of multiply is in r6r5r4r3r2r1 (most to least signific. bytes)
  ;; Result sign bit is in r16 top bit. Take off and put in
  ;; r0.
  zero r0
Floating Point Multiply Program

```assembly
dst    Is c r15 ; get sign bit
dst    rs rc r0 ; put on r0
dst    rs r15 ; put 0 in r16

;; "Normalize" result.
;; Now that the significands have been multiplied, the result
;; needs to be made to fit.
;; Exponent with double bias is in r14 (low) & r15 (bit 0 = high
;; bit of exp)
;; If the two operands were normalized, each was >= 1 and < 2,
;; so the result of the multiplication is >= 1 and < 4.
;; If the top bit of r6 is 1, then the interim result is >=2, so
;; shift that bit out and increment exponent. If the top bit of
;; r6 now isn't 1, then one of the operands was denormalized,
;; so the result must be specially handled.

dst    Is c r1 ; shift
dst    Is rc r2 ; and propagate
dst    Is rc r3
dst    Is rc r4
dst    Is rc r5
dst    Is rc r6
jmp cc normalch ; jump if carry out not on
inc    r14 ; increment exponent
dstc   r15 ; and propagate

Normalch: ;; If top bit of r6 is 1, number is normalized.
dst    n c r6
jmp cc Unnormal ; Uh-oh, an operand was denormalized

;; Don't need lowest 2 bytes since destination is single
;; precision, in which there are 23 significant bits.
;; However, the Coonen standard specifies the use of a sticky
;; bit into which all right shifts are ORed, to allow for
;; more exact rounding, so ...
;; The sticky bit is not necessarily the last bit of r3, but
;; it is easier to use that bit and later OR all bits up to
;; the appropriate sticky bit.
add    n r1, r2 ; are the last 2 bytes zero?
jmp eq Underflush ; yes - jump
or    1, r3 ; OR 1 into last bit of r3 (sticky bit)

Underflush: ;; Now the result must be checked for underflow
;; First convert the double-bias in the exponent to a single-bias
rsubi   bias, r14 ;
rsubi   0, r16 ; borrow propagate
jmp pl FMRound ; if r15 is pos, then 0 < exponent, so round
                ; if negative, then exponent underflowed

FMDenorm: ;; So we have to denormalize the number.
;; r14 contains the negative (biased) exponent.
;; So to denormalize, we shift the result right |r14|+1 times, to
;; get the biased exponent to 1. [The exponent will actually
;; be set to zero, though]
sub    1, r14 ; |r14| = r14, the # of shifts required

FMSignbits = 23.
rsubi  n FMSignbits, r14 ; but if r14 is < # of significant bits
jmp los FMDenormSetup
src    FMSignbits, r14 ; r14 < max # of shifts

;; You see how expensive denormalized numbers can be: the
;; maximum number of shifts possible is 23.

FMDenormSetup:
dst    n r14
```
ldct  reg  ; load addr/count reg with what's in r14

FMDeLoop:
dst  rs  un  r6  ; shift right result's MSByte
dst  rs  rc  r5  ; propagate shift
dst  rs  rc  r4
dst  rs  rc  r3
zero  ls  u  r0  ; put right-shift carry-out bit in bit 0 of r0
or  r0,  r3  ; OR shifted out bit into r3 (sticky bit)
lpc  FMDeLoop  ; keep on shifting

: So now we have a denormalized number in r6 to r3
: Set exponent to 0, which marks that the number is
: denormalized or zero.
: (also, a denormalized number may round to zero).
: zero  r14

FMRound:  : Round the result

: r6r5r4 and r3 have bits of interest; r3 previously had its
: low bit ORed with 1 if (r2 + r1 > 0)
: Assume Round to Nearest (RN)
: r3 contains the extra bits
: r4 bit 0 is LSB1
: Cases:
: r4 bit 0  r3  Do this
: 0  0  same [exact]  case 1
: 1  "  case 2
: 0  < 100.. same [truncate] case 3
: 1  "  case 4
: 0  100.. same [LSB 0] case 5
: 1  "  case 6
: 0  > 100.. add 1 [LSB 0] case 7
: 1  "  case 8
: So to get desired results,
: - add MSB(r3) to r4 except when LSB(r4) = 0 = Left_Shift(r3)

dst  n  r3  ; is least-extended byte 0?
jmp  eq  FMExact  ; yes - no need to round (cases 1, 2)
zero  nq
dst  ls  rd  r3  ; MSB(r3) -> LSB(q); r11 shifted left
dst  un  r4  ; LSB(r4) -> C bit
dst  rs  rdc  r3  ; shift that C bit into MSB(r3); LSB(q) -> C
jmpls  los  FMInexact  ; [los = -C \mid Z]
  ; [2 bit om:] if original lsb(r4) &
  ; left_shift(original r3) = 0, then
  ; jump, as r4 etc. stays same. (case 5)
  ; [C bit off:] if original msb(r3)
  ; zero, no need to add (cases 3, 4)
  ; else add C bit to low byte (cases 6,
  ; 7, 8)
dstc  r4
  ; & propagate
dstc  r5
  ;
jmp  cc  FMInexact  ; jump if no carry
  ; If carry here, then have to increment exponent, and shift
  ; r6, r5, and r4 right.
inc  r14  ; increment exponent (overflow caught later)
dstc  r15  ; carry propagate
sec
dst  rs  rc  r6  ; shift in carry-out which required the
  ; exponent incremented
dst  rs  rc  r5  ; & propagate
dst  rs  rc  r4
  ; no longer need r3, so...
zero  ls  u  r3  ; put right-shift carry-out bit in bit 0 of r3
or  r3,  r4  ; OR shifted out bit into r4 (sticky bit)

FMInexact:  : fail through
Floating Point Multiply Program

FMExact:

... Check for overflow

 dst  n r15 : is high byte of exponent nonzero?
 jmp ne FMSetov : yes. - an overflow
 xor n allbits, r14 : is exp all ones?
 jmp eq FMSetov
 src r14, r7 : move exponent
 andi cbit7, r6 : put sign bit in r6
 or r0, r6
 jmp Restpack

FMSetov:

... exponent has overflowed, so...

 xff r7 : exponent field all ones
 src r0, r6 : put sign (all other bits are zeroes) in r6
 zero r6 :
 zero r4 :

... Pack & Deliver

 jmp Restpack : restore save registers, then repack

Unnormal:

... Hack the result of multiplying with a denormalized
... operand. /// Not handled ///

FMA8nMult: ... Multiplies r2r1r0 by r8. Assumes r6r5r4 contains previous
... results of multiplies, which will be shifted into <ExtBuf>.
... clobbers r11.

dst  nq r0 : q <- r0 (a0)
zero r11
isetup 7
umpy d lpc r8, r11 : b[i] = a0 [8 times]: MSB to r11: LSB to q
 addi warr 1, r12 : increment external buffer ptr
 dstc war* r13 : ... & set MAR
 addq n wmm r4, : q (LSB) + prev. low byte -> ext buf
 src r5, r4 : r6 + carry -> r4
 src r6, r5 : r6 + carry -> r5
 srcct 0, r6 : carry -> r6
 add r11, r4 : r11 + r4 -> r4 [r11 = MSB]
dstc r5 : carry propagate
dstc r6 :

dst  nq r1 : q <- r1 (a1)
zero r11
isetup 7
umpy d lpc r8, r11 : MSB -> r11, LSB -> q
 addq r4, r4 : d[11] + r4 -> r4
 dstc r5 : carry propagate
dstc r6
 add r11, r6 : c[11] + r6 -> r6
dstc r6 : carry propagate
dst  nq r2 : q <- r2 (a2)
zero r11
isetup 7
umpy d lpc r8, r11 : MSB -> r11, LSB -> q
 addq r6, r5 : d[12] + r5 -> r5
dstc r6 : carry propagate
 add rtn r11, r6 : c[12] + r6 -> r6

Restpack:

jsr restore10111145 : restore saved registers from external buf

FMRepackX: ... repack but use r6 instead of r9, don't restore registers.

dst  ls c r6 : take off sign bit

dst rc r7 : put on r7, take off low order exp bit
Floating Point Multiply Program

\[\text{dst} \quad rs \quad rc \quad r6 \quad : \quad \text{put on r6}\]

Deliver: :: Results are left in r7-r4 (msbyte - l义务)

FMRepack: :: repack with fraction high byte r9, sign in r6; don't
:: restore registers
\[\text{dst} \quad ls \quad r9 \quad : \quad \text{throw away explicit leading bit}\]
\[\text{dst} \quad n \quad c \quad r6 \quad : \quad \text{get sign bit}\]
\[\text{dst} \quad rs \quad rc \quad r7 \quad : \quad \text{put sign bit on r7, take off exp low bit}\]
\[\text{src} \quad rs \quad rc \quad r6 \quad : \quad \text{put low exp bit in top bit of 2nd highest byte, put result in r6, so}\]
\[\text{jmp} \quad \text{Deliver}\]
REFERENCES

[Ackerman-Pu]

[Ackerman-Val]

[Aoki-Instruction Set]

[Coonen-Computer]

[Dennis-Prototypes]

[Feridun-Module]

[Feridun-Pipeline]

[Mcgraw-Val]

[Signum-Mar 1979]
SIGNUM Newsletter. Association for Computing Machinery, Special Interest Group on Numerical Mathematics, volume 14, number 1, March 1979, pages 12 to 27 and 96 to 108.

[Signum-Oct 1979]