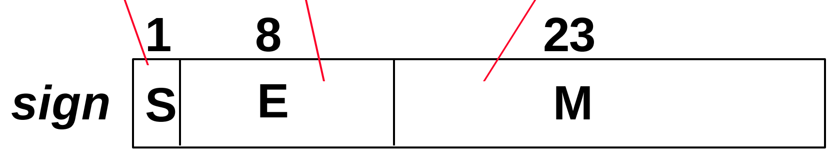
Representation of Floating Point Numbers in
Single Precision *IEEE 754 Standard*

Value = N = $(-1)^S \times 2^{E-127} \times (1.M)$



exponent: excess 127 binary integer added	*mantissa:* sign + magnitude, normalized binary significand with a hidden integer bit: 1.M

$0 < E < 255$
Actual exponent is:
$e = E - 127$

Example: 0 = 0 00000000 0 . . . 0 -1.5 = 1 01111111 10 . . . 0

Magnitude of numbers that can be represented is in the range: $2^{-126}(1.0)$ to $2^{127}(2 - 2^{-23})$

Which is approximately: 1.8×10^{-38} to 3.40×10^{38}

Representation of Floating Point Numbers in Double Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^S \times 2^{E-1023} \times (1.M)$$

	1	11	52
sign	S	E	M

$0 < E < 2047$
Actual exponent is:
$e = E - 1023$

exponent:
excess 1023
binary integer
added

mantissa:
sign + magnitude, normalized
binary significand with
a hidden integer bit: 1.M

Example: 0 = 0 00000000000 0 . . . 0 -1.5 = 1 01111111111 10 . . . 0

Magnitude of numbers that can be represented is in the range: $2^{-1022}(1.0)$ to $2^{1023}(2 - 2^{-52})$

Which is approximately: 2.23×10^{-308} to 1.8×10^{308}

IEEE 754 Format Parameters

	Single Precision	Double Precision
p (bits of precision)	24	53
Unbiased exponent e_{max}	127	1023
Unbiased exponent e_{min}	-126	-1022
Exponent bias	127	1023

IEEE 754 Special Number Representation

Single Precision		Double Precision		Number Represented
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	0	nonzero	Denormalized number[1]
1 to 254	anything	1 to 2046	anything	Floating Point Number
255	0	2047	0	Infinity[2]
255	nonzero	2047	nonzero	NaN (Not A Number)[3]

[1] May be returned as a result of underflow in multiplication

[2] Positive divided by zero yields "infinity"

[3] Zero divide by zero yields NaN "not a number"

Floating Point Conversion Example

- The decimal number $.75_{10}$ is to be represented in the *IEEE 754* 32-bit single precision format:

$-2345.125_{10} = 0.11_2$ (converted to a binary number)

$= 1.1 \times 2^{-1}$ (normalized a binary number)

Hidden

- The mantissa is positive so the sign S is given by:

  S = 0

- The biased exponent E is given by E = e + 127

  $E = -1 + 127 = 126_{10} = 01111110_2$

- Fractional part of mantissa M:

  M = .10000000000000000000000 (in 23 bits)

The *IEEE 754* single precision representation is given by:

0	01111110	10000000000000000000000
S	E	M
1 bit	8 bits	23 bits

Floating Point Conversion Example

- **The decimal number -2345.125_{10} is to be represented in the *IEEE 754* 32-bit single precision format:**

$-2345.125_{10} = -100100101001.001_2$ (converted to binary)

$= -1.00100101001001 \times 2^{11}$ (normalized binary)

Hidden

- **The mantissa is negative so the sign S is given by:**

  S = 1

- **The biased exponent E is given by E = e + 127**

  $E = 11 + 127 = 138_{10} = 10001010_2$

- **Fractional part of mantissa M:**

  M = .00100101001001000000000 (in 23 bits)

The *IEEE 754* single precision representation is given by:

1	10001010	00100101001001000000000
S	E	M
1 bit	8 bits	23 bits

Basic Floating Point Addition Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point addition: Result = X + Y = $(Xm \times 2^{Xe}) + (Ym \times 2^{Ye})$

involves the following steps:

(1) Align binary point:

- **Initial result exponent: the larger of Xe, Ye**
- **Compute exponent difference: Ye - Xe**
- **If Ye > Xe Right shift Xm that many positions to form $Xm\ 2^{Xe-Ye}$**
- **If Xe > Ye Right shift Ym that many positions to form $Ym\ 2^{Ye-Xe}$**

(2) Compute sum of aligned *mantissas*:

i.e $Xm2^{Xe-Ye} + Ym$ or $Xm + Xm2^{Ye-Xe}$

(3) If normalization of result is needed, then a normalization step follows:

- **Left shift result, decrement result exponent (e.g., if result is 0.001xx…) or**
- **Right shift result, increment result exponent (e.g., if result is 10.1xx…)**

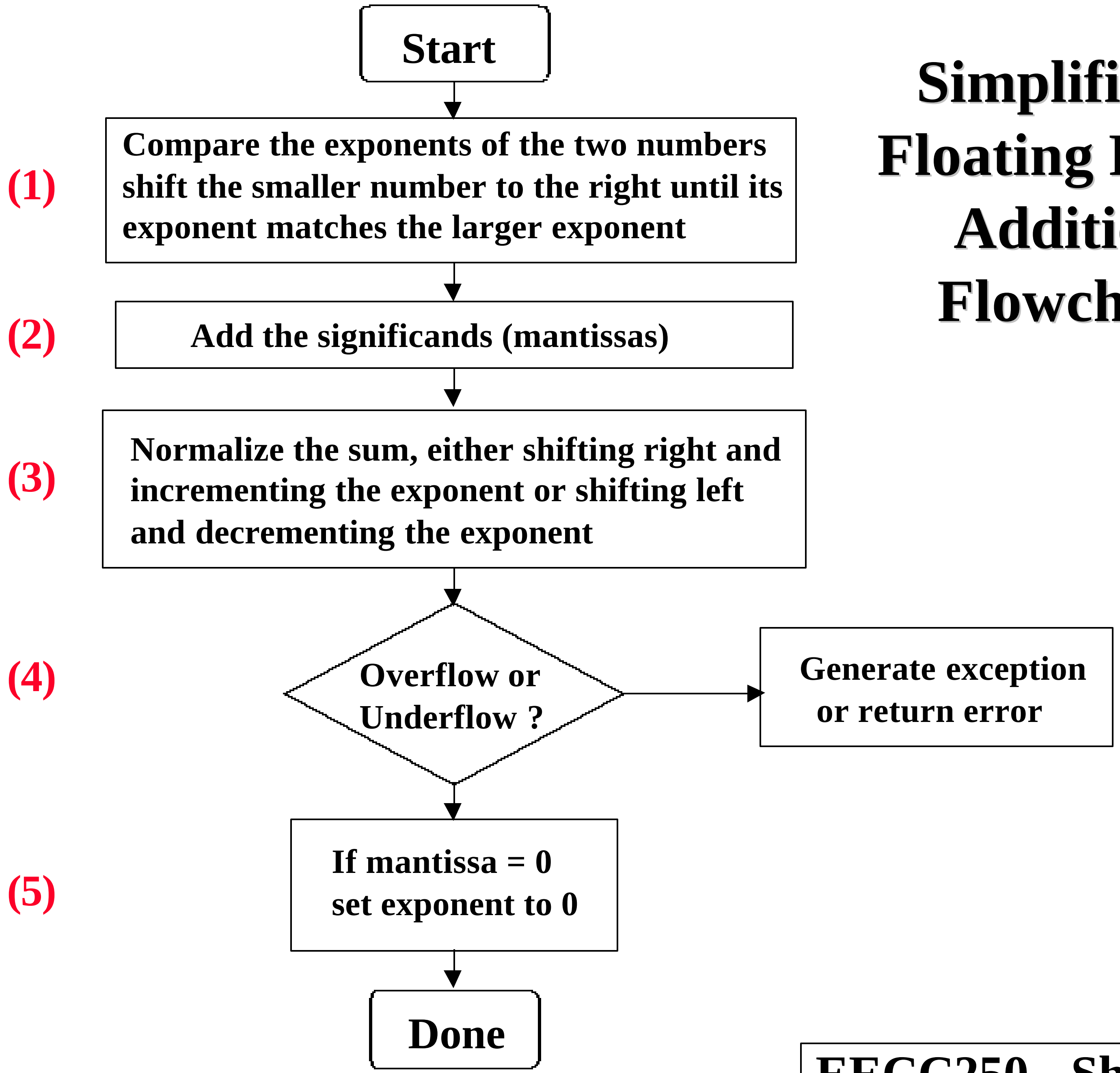
Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(4) Check result exponent:

- **If larger than maximum exponent allowed return exponent overflow**
- **If smaller than minimum exponent allowed return exponent underflow**

(5) If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.

Simplified Floating Point Addition Flowchart



Floating Point Addition Example

- Add the following two numbers represented in the *IEEE 754* single precision format: $X = 2345.125_{10}$ represented as:

0	10001010	00100101001001000000000

to $Y = .75_{10}$ represented as:

0	01111110	10000000000000000000000

(1) Align binary point:

- Xe > Ye initial result exponent = Ye = 10001010 = 138_{10}
- Xe - Ye = 10001010 - 01111110 = 00000110 = 12_{10}
- Shift Ym 12_{10} postions to the right to form

$Ym\ 2^{Ye-Xe} = Ym\ 2^{-12} =$ 0.00000000000110000000000

(2) Add mantissas:

$Xm + Ym\ 2^{-12} =$ 1.00100101001001000000000

+ 0.00000000000110000000000 =

1. 00100101001111000000000

(3) Normailzed? Yes

(4) Overflow? No. Underflow? No **(5)** zero result? No

Result

0	10001010	00100101001111000000000

IEEE 754 Single precision Addition Notes

- If the exponents differ by more than 24, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa.
  - The sum will then equal the larger number.
  - Such truncation errors occur when the numbers differ by a factor of more than 2^{24} , which is approximately 1.6×10^7 .
  - Thus, the precision of IEEE single precision floating point arithmetic is approximately 7 decimal digits.
- Negative mantissas are handled by first converting to 2's complement and then performing the addition.
  - After the addition is performed, the result is converted back to sign-magnitude form.
- When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude.
  - Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a large loss of accuracy.
- Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.

Basic Floating Point Subtraction Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point addition: Result = X - Y = $(Xm \times 2^{Xe}) - (Ym \times 2^{Ye})$

involves the following steps:

(1) Align binary point:

- **Initial result exponent: the larger of Xe, Ye**
- **Compute exponent difference: Ye - Xe**
- **If Ye > Xe Right shift Xm that many positions to form $Xm\ 2^{Xe-Ye}$**
- **If Xe > Ye Right shift Ym that many positions to form $Ym\ 2^{Ye-Xe}$**

(2) Subtract the aligned *mantissas*:

i.e $Xm2^{Xe-Ye} - Ym$ or $Xm - Xm2^{Ye-Xe}$

(3) If normalization of result is needed, then a normalization step follows:

- **Left shift result, decrement result exponent (e.g., if result is 0.001xx…) or**
- **Right shift result, increment result exponent (e.g., if result is 10.1xx…)**

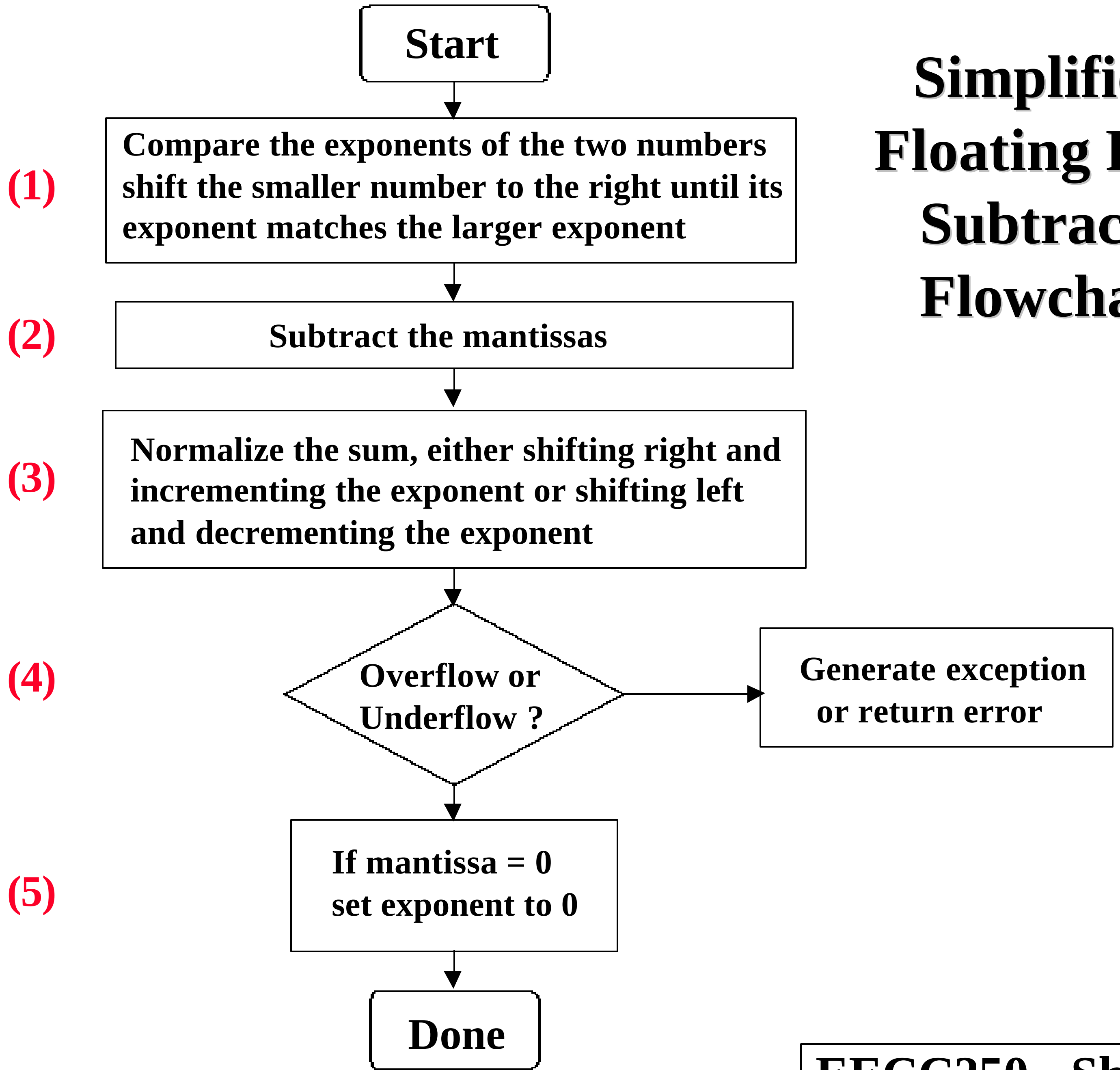
Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(4) Check result exponent:

- **If larger than maximum exponent allowed return exponent overflow**
- **If smaller than minimum exponent allowed return exponent underflow**

(5) If result mantissa is 0, may need to set the exponent to zero by a special step to return a proper zero.

Simplified Floating Point Subtraction Flowchart



Basic Floating Point Multiplication Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point multiplication:

$$\text{Result} = R = X * Y = (-1)^{Xs} (Xm \times 2^{Xe}) * (-1)^{Ys} (Ym \times 2^{Ye})$$

involves the following steps:

(1) If one or both operands is equal to zero, return the result as zero, otherwise:

(2) Compute the sign of the result Xs XOR Ys

(3) Compute the mantissa of the result:

- Multiply the mantissas: Xm * Ym
- Round the result to the allowed number of mantissa bits

(4) Compute the exponent of the result:

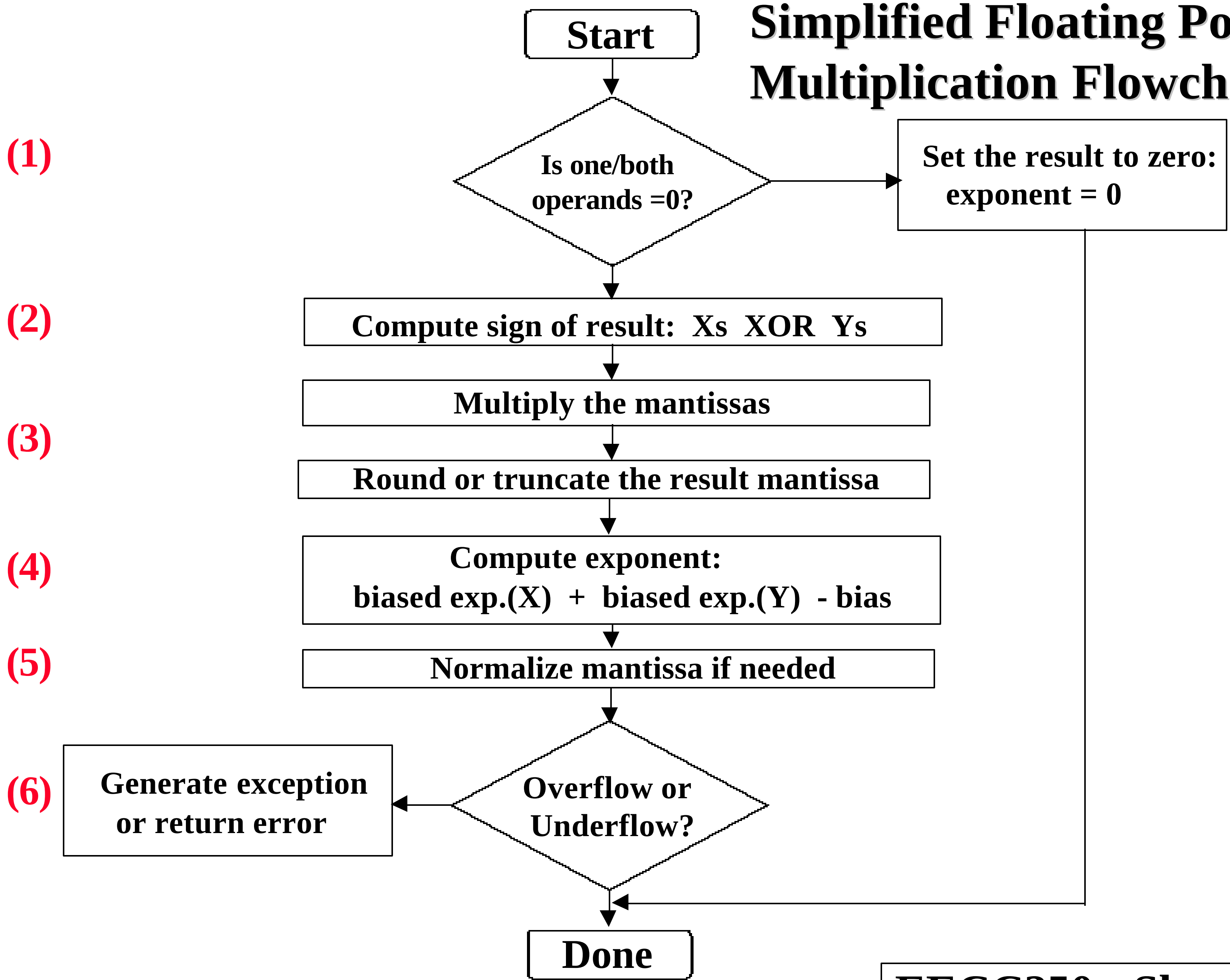
Result exponent = biased exponent (X) + biased exponent (Y) - bias

(5) Normalize if needed, by shifting mantissa right, incrementing result exponent.

(6) Check result exponent for overflow/underflow:

- If larger than maximum exponent allowed return exponent overflow
- If smaller than minimum exponent allowed return exponent underflow

Simplified Floating Point Multiplication Flowchart



Floating Point Multiplication Example

- Multiply the following two numbers represented in the *IEEE 754* single precision format: X = -18_{10} represented as:

1	10000011	00100000000000000000000

and Y = 9.5_{10} represented as:

0	10000010	00110000000000000000000

(1) Value of one or both operands = 0? No, continue with step 2

(2) Compute the sign: S = Xs XOR Ys = 1 XOR 0 = 1

(3) Multiply the mantissas: The product of the 24 bit mantissas is 48 bits with two bits to the left of the binary point:

(01).0101011000000….000000

Truncate to 24 bits:

hidden → (1).01010110000000000000000

(4) Compute exponent of result:

Xe + Ye - 127_{10} = 1000 0011 + 1000 0010 - 0111111 = 1000 0110

(5) Result mantissa needs normalization? No

(6) Overflow? No. Underflow? No

Result

1	10000110	01010101100000000000000

IEEE 754 Single precision Multiplication Notes

- **Rounding occurs in floating point multiplication when the mantissa of the product is reduced from 48 bits to 24 bits.**
  - **The least significant 24 bits are discarded.**
- **Overflow occurs when the sum of the exponents exceeds 127, the largest value which is defined in bias-127 exponent representation.**
  - **When this occurs, the exponent is set to 128 (E = 255) and the mantissa is set to zero indicating + or - infinity.**
- **Underflow occurs when the sum of the exponents is more negative than -126, the most negative value which is defined in bias-127 exponent representation.**
  - **When this occurs, the exponent is set to -127 (E = 0).**
  - **If M = 0, the number is exactly zero.**
  - **If M is not zero, then a denormalized number is indicated which has an exponent of -127 and a hidden bit of 0.**
  - **The smallest such number which is not zero is 2^{-149}. This number retains only a single bit of precision in the rightmost bit of the mantissa.**

Basic Floating Point Division Algorithm

Assuming that the operands are already in the IEEE 754 format, performing floating point multiplication:

$$\text{Result} = R = X / Y = (-1)^{Xs} (Xm \times 2^{Xe}) / (-1)^{Ys} (Ym \times 2^{Ye})$$

involves the following steps:

(1) If the divisor Y is zero return "Infinity", if both are zero return "NaN"

(2) Compute the sign of the result Xs XOR Ys

(3) Compute the mantissa of the result:

- **The dividend mantissa is extended to 48 bits by adding 0's to the right of the least significant bit.**
- **When divided by a 24 bit divisor Ym, a 24 bit quotient is produced.**

(4) Compute the exponent of the result:

Result exponent = [biased exponent (X) - biased exponent (Y)] + bias

(5) Normalize if needed, by shifting mantissa left, decrementing result exponent.

(6) Check result exponent for overflow/underflow:

- **If larger than maximum exponent allowed return exponent overflow**
- **If smaller than minimum exponent allowed return exponent underflow**

IEEE 754 Error Rounding

- **In integer arithmetic, the result of an operation is well-defined:**
  - **Either the exact result is obtained or overflow occurs and the result cannot be represented.**
- **In floating point arithmetic, rounding errors occur as a result of the limited precision of the mantissa. For example, consider the average of two floating point numbers with identical exponents, but mantissas which differ by 1. Although the mathematical operation is well-defined and the result is within the range of representable numbers, the average of two adjacent floating point values cannot be represented exactly.**
- **The IEEE FPS defines four rounding rules for choosing the closest floating point when a rounding error occurs:**
  - **RN - Round to Nearest. Break ties by choosing the least significant bit = 0.**
  - **RZ - Round toward Zero. Same as truncation in sign-magnitude.**
  - **RP - Round toward Positive infinity.**
  - **RM - Round toward Minus infinity. Same as truncation in integer 2's complement arithmetic.**
- **RN is generally preferred and introduces less systematic error than the other rules.**

Floating Point Error Rounding Observations

- **The absolute error introduced by rounding is the actual difference between the exact value and the floating point representation.**
- **The size of the absolute error is proportional to the magnitude of the number.**
  - **For numbers in single Precision IEEE 754 format, the absolute error is less than 2^{-24}.**
  - **The largest absolute rounding error occurs when the exponent is 127 and is approximately 10^{31} since $2^{-24}.2^{127} = 10^{31}$**
- **The relative error is the absolute error divided by the magnitude of the number which is approximated. For normalized floating point numbers, the relative error is approximately 10^{-7}**
- **Rounding errors affect the outcome of floating point computations in several ways:**
  - **Exact comparison of floating point variables often produces incorrect results. Floating variables should not be used as loop counters or loop increments.**
  - **Operations performed in different orders may give different results. On many computers, a+b may differ from b+a and (a+b)+c may differ from a+(b+c).**
  - **Errors accumulate over time. While the relative error for a single operation in single precision floating point is about 10^{-7}, algorithms which iterate many times may experience an accumulation of errors which is much larger.**

68000 FLOATING POINT ADD/SUBTRACT (FFPADD/FFPSUB) Subroutine

```
*************************************************************
*                  FFPADD/FFPSUB                            *
*             FAST FLOATING POINT ADD/SUBTRACT              *
*                                                           *
*  FFPADD/FFPSUB - FAST FLOATING POINT ADD AND SUBTRACT     *
*                                                           *
*  INPUT:                                                   *
*      FFPADD                                               *
*          D6 - FLOATING POINT ADDEND                       *
*          D7 - FLOATING POINT ADDER                        *
*      FFPSUB                                               *
*          D6 - FLOATING POINT SUBTRAHEND                   *
*          D7 - FLOATING POINT MINUEND                      *
*                                                           *
*  OUTPUT:                                                  *
*          D7 - FLOATING POINT ADD RESULT                   *
*                                                           *
*  CONDITION CODES:                                         *
*          N - RESULT IS NEGATIVE                           *
*          Z - RESULT IS ZERO                               *
*          V - OVERFLOW HAS OCCURED                         *
*          C - UNDEFINED                                    *
*          X - UNDEFINED
```

```
*           REGISTERS D3 THRU D5 ARE VOLATILE                      *
*                                                                  *
*  CODE SIZE: 228 BYTES       STACK WORK AREA:  0 BYTES            *
*                                                                  *
*  NOTES:                                                          *
*    1) ADDEND/SUBTRAHEND UNALTERED (D6).                          *
*    2) UNDERFLOW RETURNS ZERO AND IS UNFLAGGED.                   *
*    3) OVERFLOW RETURNS THE HIGHEST VALUE WITH THE                *
*       CORRECT SIGN AND THE 'V' BIT SET IN THE CCR.               *
*                                                                  *
*  TIME: (8 MHZ NO WAIT STATES ASSUMED)                            *
*                                                                  *
*           COMPOSITE AVERAGE  20.625 MICROSECONDS                 *
*                                                                  *
*  ADD:         ARG1=0               7.75 MICROSECONDS             *
*               ARG2=0               5.25 MICROSECONDS             *
*                                                                  *
*          LIKE SIGNS  14.50 - 26.00  MICROSECONDS                 *
*                    AVERAGE   18.00  MICROSECONDS                 *
*         UNLIKE SIGNS 20.13 - 54.38  MICROCECONDS                 *
*                    AVERAGE   22.00  MICROSECONDS                 *
*                                                                  *
*  SUBTRACT:    ARG1=0               4.25 MICROSECONDS             *
*               ARG2=0               9.88 MICROSECONDS             *
*                                                                  *
*          LIKE SIGNS  15.75 - 27.25  MICROSECONDS                 *
*                    AVERAGE   19.25  MICROSECONDS                 *
*         UNLIKE SIGNS 21.38 - 55.63  MICROSECONDS                 *
*                    AVERAGE   23.25  MICROSECONDS                 *
*
```

```
*************************
* SUBTRACT ENTRY POINT *
*************************
FFPSUB   MOVE.B  D6,D4    TEST ARG1
         BEQ.S   FPART2   RETURN ARG2 IF ARG1 ZERO
         EOR.B   #$80,D4  INVERT COPIED SIGN OF ARG1
         BMI.S   FPAMI1   BRANCH ARG1 MINUS
* + ARG1
         MOVE.B  D7,D5    COPY AND TEST ARG2
         BMI.S   FPAMS    BRANCH ARG2 MINUS
         BNE.S   FPALS    BRANCH POSITIVE NOT ZERO
         BRA.S   FPART1   RETURN ARG1 SINCE ARG2 IS ZERO

********************
* ADD ENTRY POINT *
********************
FFPADD   MOVE.B  D6,D4    TEST ARGUMENT1
         BMI.S   FPAMI1   BRANCH IF ARG1 MINUS
         BEQ.S   FPART2   RETURN ARG2 IF ZERO

* + ARG1
         MOVE.B  D7,D5    TEST ARGUMENT2
         BMI.S   FPAMS    BRANCH IF MIXED SIGNS
         BEQ.S   FPART1   ZERO SO RETURN ARGUMENT1
```

```
* +ARG1 +ARG2
* -ARG1 -ARG2
FPALS    SUB.B   D4,D5    TEST EXPONENT MAGNITUDES
         BMI.S   FPA2LT   BRANCH ARG1 GREATER
         MOVE.B  D7,D4    SETUP STRONGER S+EXP IN D4

* ARG1EXP <= ARG2EXP
         CMP.B   #24,D5   OVERBEARING SIZE
         BCC.S   FPART2   BRANCH YES, RETURN ARG2
         MOVE.L  D6,D3    COPY ARG1
         CLR.B   D3       CLEAN OFF SIGN+EXPONENT
         LSR.L   D5,D3    SHIFT TO SAME MAGNITUDE
         MOVE.B  #$80,D7  FORCE CARRY IF LSB-1 ON
         ADD.L   D3,D7    ADD ARGUMENTS
         BCS.S   FPA2GC   BRANCH IF CARRY PRODUCED
FPARSR   MOVE.B  D4,D7    RESTORE SIGN/EXPONENT
         RTS              RETURN TO CALLER
```

```
* ADD SAME SIGN OVERFLOW NORMALIZATION
FPA2GC   ROXR.L  #1,D7     SHIFT CARRY BACK INTO RESULT
         ADD.B   #1,D4     ADD ONE TO EXPONENT
         BVS.S   FPA2OS    BRANCH OVERFLOW
         BCC.S   FPARSR    BRANCH IF NO EXPONENT OVERFLOW
FPA2OS   MOVEQ   #-1,D7    CREATE ALL ONES
         SUB.B   #1,D4     BACK TO HIGHEST EXPONENT+SIGN
         MOVE.B  D4,D7     REPLACE IN RESULT
*        OR.B    #$02,CCR SHOW OVERFLOW OCCURRED
         DC.L    $003C0002 ****ASSEMBLER ERROR****
         RTS               RETURN TO CALLER

* RETURN ARGUMENT1
FPART1   MOVE.L  D6,D7     MOVE IN AS RESULT
         MOVE.B  D4,D7     MOVE IN PREPARED SIGN+EXPONENT
         RTS               RETURN TO CALLER

* RETURN ARGUMENT2
FPART2   TST.B   D7        TEST FOR RETURNED VALUE
         RTS               RETURN TO CALLER
```

```
* -ARG1EXP > -ARG2EXP
* +ARG1EXP > +ARG2EXP
FPA2LT   CMP.B   #-24,D5  ? ARGUMENTS WITHIN RANGE
         BLE.S   FPART1   NOPE, RETURN LARGER
         NEG.B   D5       CHANGE DIFFERENCE TO POSITIVE
         MOVE.L  D6,D3    SETUP LARGER VALUE
         CLR.B   D7       CLEAN OFF SIGN+EXPONENT
         LSR.L   D5,D7    SHIFT TO SAME MAGNITUDE
         MOVE.B  #$80,D3  FORCE CARRY IF LSB-1 ON
         ADD.L   D3,D7    ADD ARGUMENTS
         BCS.S   FPA2GC   BRANCH IF CARRY PRODUCED
         MOVE.B  D4,D7    RESTORE SIGN/EXPONENT
         RTS              RETURN TO CALLER

* -ARG1
FPAMI1   MOVE.B  D7,D5    TEST ARG2'S SIGN
         BMI.S   FPALS    BRANCH FOR LIKE SIGNS
         BEQ.S   FPART1   IF ZERO RETURN ARGUMENT1
```

```
* -ARG1 +ARG2
* +ARG1 -ARG2
FPAMS     MOVEQ    #-128,D3  CREATE A CARRY MASK ($80)
          EOR.B    D3,D5     STRIP SIGN OFF ARG2 S+EXP COPY
          SUB.B    D4,D5     COMPARE MAGNITUDES
          BEQ.S    FPAEQ     BRANCH EQUAL MAGNITUDES
          BMI.S    FPATLT    BRANCH IF ARG1 LARGER
* ARG1 <= ARG2
          CMP.B    #24,D5    COMPARE MAGNITUDE DIFFERENCE
          BCC.S    FPART2    BRANCH ARG2 MUCH BIGGER
          MOVE.B   D7,D4     ARG2 S+EXP DOMINATES
          MOVE.B   D3,D7     SETUP CARRY ON ARG2
          MOVE.L   D6,D3     COPY ARG1
FPAMSS    CLR.B    D3        CLEAR EXTRANEOUS BITS
          LSR.L    D5,D3     ADJUST FOR MAGNITUDE
          SUB.L    D3,D7     SUBTRACT SMALLER FROM LARGER
          BMI.S    FPARSR    RETURN FINAL RESULT IF NO
   OVERFLOW
```

```
* MIXED SIGNS NORMALIZE
FPANOR   MOVE.B  D4,D5    SAVE CORRECT SIGN
FPANRM   CLR.B   D7       CLEAR SUBTRACT RESIDUE
         SUB.B   #1,D4    MAKE UP FOR FIRST SHIFT
         CMP.L   #$00007FFF,D7 ? SMALL ENOUGH FOR SWAP
         BHI.S   FPAXQN   BRANCH NOPE
         SWAP.W  D7       SHIFT LEFT 16 BITS REAL FAST
         SUB.B   #16,D4   MAKE UP FOR 16 BIT SHIFT
FPAXQN   ADD.L   D7,D7    SHIFT UP ONE BIT
         DBMI    D4,FPAXQN DECREMENT AND BRANCH IF POSITIVE
         EOR.B   D4,D5    ? SAME SIGN
         BMI.S   FPAZRO   BRANCH UNDERFLOW TO ZERO
         MOVE.B  D4,D7    RESTORE SIGN/EXPONENT
         BEQ.S   FPAZRO   RETURN ZERO IF EXPONENT
   UNDERFLOWED
         RTS              RETURN TO CALLER

* EXPONENT UNDERFLOWED - RETURN ZERO
FPAZRO   MOVEQ.L #0,D7    CREATE A TRUE ZERO
         RTS              RETURN TO THE CALLER
```

```
* ARG1 > ARG2
FPATLT   CMP.B   #-24,D5  ? ARG1 >> ARG2
         BLE.S   FPART1   RETURN IT IF SO
         NEG.B   D5       ABSOLUTIZE DIFFERENCE
         MOVE.L  D7,D3    MOVE ARG2 AS LOWER VALUE
         MOVE.L  D6,D7    SETUP ARG1 AS HIGH
         MOVE.B  #$80,D7  SETUP ROUNDING BIT
         BRA.S   FPAMSS   PERFORM THE ADDITION

* EQUAL MAGNITUDES
FPAEQ    MOVE.B  D7,D5    SAVE ARG1 SIGN
         EXG.L   D5,D4    SWAP ARG2 WITH ARG1 S+EXP
         MOVE.B  D6,D7    INSURE SAME LOW BYTE
         SUB.L   D6,D7    OBTAIN DIFFERENCE
         BEQ.S   FPAZRO   RETURN ZERO IF IDENTICAL
         BPL.S   FPANOR   BRANCH IF ARG2 BIGGER
         NEG.L   D7       CORRECT DIFFERENCE TO POSITIVE
         MOVE.B  D5,D4    USE ARG2'S SIGN+EXPONENT
         BRA.S   FPANRM   AND GO NORMALIZE

         END
```