# APPLICATION NOTE

## AN701
SP floating point math with XA

Author: Santanu Roy

1995 Jul 28

Philips Semiconductors

PHILIPS

# SP floating point math with XA

Author: Santanu Roy, MCO Applications Group, Sunnyvale, California

## IEEE SINGLE PRECISION FLOATING POINT ARITHMETIC WITH XA

### Introduction

This application note is intended to implement Single Precision Floating Point Arithmetic package using the new Philips Semiconductors XA microcontroller. The goal is to have this package as a part of the run-time math library for the XA when the cross-compiler is developed. The package is based upon the IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985). This package, however, is not a conforming implementation of the said standard. The differences between the XA implementation and the standard are listed later in this report. Also, this package does not include routines for conversion between Integers to Floating Point and vice versa.

The following four standard Single Precision (SP) arithmetic operations have been implemented in this package:

1. **FPADD** Addition of two SP floating point numbers.
2. **FPSUB** Subtraction of two SP floating point numbers.
3. **FPMUL** Multiplication of two SP floating point numbers.
4. **FPDIV** Division of two SP floating point numbers.

The following section discusses the representation of FLP numbers. Then the differences between the XA implementation and the IEEE standard are described. This is followed by a description of the algorithms used in the computations. Appendix A is a user reference section for converting a floating point number into IEEE format and Appendix B is a listing of the code.

Note that this application note assumes that the reader is familiar with the IEEE Binary Floating-Point standard.

### IEEE Floating Point Formats

The basic format sizes for floating-point numbers, 32 bits and 64 bits, were selected for efficient calculation of array elements in byte-addressable memories. For the 32-bit format, precision was deemed the most important criterion, hence the choice of radix 2 instead of octal or hexadecimal. Other characteristics include not representing the leading significand bit in normalized numbers, a minimally acceptable exponent range which uses 8 bits, and exponent bias which allows the reciprocal of all normalized numbers to be represented without overflow. For the 64-bit format, the main consideration was range.

### Representation of FLP Number

The IEEE binary floating point number is represented in the following format:

$FP = \pm \text{Significand} \times \text{Base}^{\text{Characteristic}}$

The specification of a binary FP number involves two parts:
A Significand or Mantissa
and a Characteristic or Exponent.

The Mantissa is signed fixed point number and the Exponent is a signed integer. Mantissa or Significand is that component of a binary FLP number which consists of an explicit or implicit leading bit to the left of its binary point and a fraction field to the right of the binary point. Exponent signifies the power to which 2 is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent. The IEEE standard specifies that a single precision Floating Point number should be represented in 32 bits as shown in Figure 1.

| SIGN 1-bit | EXPONENT 8-bits | MANTISSA 23-bits |
|---|---|---|

Figure 1.

The significance of each of these fields is as follows:

1. **SIGN** — This 1–bit field is the sign of the Mantissa. ‘0’ indicates a *positive* and ‘1’ indicates a *negative* number.
2. **EXPONENT —** This is a 8–bit field. The width of this field determines the range of the FP number. The exponent is represented as a biased value with a bias of 127 decimal. The bias value is added to exponents in order to keep them always positive and is represented by

   $2^{n-1} - 1$,

   where n = number of bits in the binary exponent.

3. **MANTISSA** — This is a 23-bit field representing the fractional part. The width of this field determines the precision for the FP number. For normalized FP numbers (see below), a MSB of ‘1’ is assumed and not represented. Thus, for normalized numbers, the value of the mantissa is 1.Mantissa. This provides an effective precision of 24–bits for the mantissa.

   If dealt with normalized numbers only (as the XA implementation does), then the MSB of the Mantissa need not be explicitly represented as per IEEE standard specification.The normalized significand lies in the range shown below.

   1.0 < Normalized Mantissa < 2.0

   Given the values of Sign, Exponent, and Mantissa, the value of the FP number is obtained as follows:

(i) If 0 < Exp < 255, then

$FP = (-1)^{SIGN} \times 2^{EXP - 127} \times 1.\text{MANTISSA}$

(ii) If Exp = 0, then FP = 0

(iii) If Exp = 255, and Mantissa =/= 0, then FP = Invalid Number (NaN or Not a Number).

The above format for single precision binary FP numbers provides for the representation in the range $-3.4 \times 10^{38}$ to $-1.75 \times 10^{-38}$, 0, and $1.75 \times 10^{-38}$ to $3.4 \times 10^{38}$. The accuracy is between 7 and 8 decimal digits.

### Differences with the IEEE Standards

The IEEE standard specifies a comprehensive list of operations and representations for FLP numbers. Since an implementation that fully conforms to this standard would lead to an excessive amount of overhead, a number of features in the standard were omitted. This section describes the differences between the implemented package and the standard.

1. **Omission of –0** — The IEEE standard requires that both + and – 0 be represented, and arithmetic carried out using both. The implementation does not represent –0.
2. **Omission of infinity arithmetic** — The IEEE standard provides for the representation of + and – infinity, and requires that valid arithmetic operations be carried out on infinity.
3. **Omission of Quiet NaN** — The IEEE standard provides for both Quiet and Signalling NaNs. A signalling NaN can be produced as

the result of overflow during an arithmetic operation. If the NaN is passed as input to further FLP routines, then these routines would produce another NaN as output. The routines will also set the Invalid Operation Flag, and call the user FLP error trap routine at address FPTRAP.

4. **Omission of denormalized numbers** — These are FLP numbers with a biased exponent, E of zero and non-zero mantissa F. Such denormalized numbers are useful in providing gradual underflow to 0. These are not represented in the XA implementation. Instead, if the result of a computation cannot be represented as a normalized number within the allowable exponent range, then an underflow is signalled, the result is set to 0, and the user FLP error trap routine at address FPTRAP is called.
5. **Omission of Inexact Result Exponent** — The IEEE standard requires that an Inexact Result Exception be signalled when the round result of an operation is not exact, or it overflows without an overflow trap. This feature is not provided.
6. **Biased Rounding to Nearest** — The IEEE standard requires that rounding to the nearest be provided as the default rounding mode. Further, the rounding is required to be unbiased. The XA implementation provides biased rounding to nearest only, e.g., suppose the result of an operation is .b1b2b3nnn and needs to be rounded to 3 binary digits. Then if nnn is 0YY, the round to nearest result is .b1b2b3. If nnn is 1YY, with at least one of the Y's being 1, then the result is .b1b2b3 + 0.001. Finally, if nnn is 100, it is a tie situation. In such a case, the IEEE standard requires that the rounded result be such that its LSB is 0. The XA implementation, on the other hand, will round the result in such a case to .b1b2b3 + 0.001.

## DESCRIPTION OF ALGORITHMS

### General Considerations

The XA implementation of the SP floating point package consists of a series of subroutines. The subroutines have been written and tested using Microsoft C however, not been tested with the XA C cross-compiler and also not optimized for code efficiency. The executable could be run under DOS in any IBM compatible PC. It is a menu driven routine that enables the user to select any of the 4 Floating Point routines. The menu also includes a “HELP” item designed to provide some standard SP floating point numbers, their operations and results as a quick reference.

The Arithmetic subroutines that compute F1 (Op) F2, where Op is +, –, *, or / expect that F1 and F2 are in IEEE format. Each of F1 and F2 consists of two 16-bit words organized as follows:

Fn–HI : Sign(1) Biased exponent(7) MSB of Mantissa(8)

Fn–LO : Least Significant word of Mantissa(16)

### Exception Handling

The following types of exception can occur during the course of computation.

**Invalid Operand—**This exception occurs if one of the input is a NaN.

**Exponent Overflow—**This occurs if the result of a computation is such that its rounded result is finite and not an invalid result but its exponent is too large to represent in the floating point format, i.e., exponent has a biased value of 255 or more.

**Exponent Underflow—**This occurs if the result of a computation is such that its exponent is 0 or less.

**Divide-by-zero—**This exception occurs if the FLP divide routine is called with F2 being 0.

The package signals exceptions in 2 ways. First, a word at address ERRFLG is maintained that registers the history of the exception conditions. Bits 0–3 of this word are used for the same.

Bit 0 – Exponent overflow detect

Bit 1 – Exponent Underflow detect ERRFLG

Bit 2 – Illegal Operand detect

Bit 3 – Divide–by–0 detect

**ERRFLG**

| * | * | * | * | DBZ | IOP | EUF | EOV |
|---|---|---|---|---|---|---|---|

This bits are never cleared by the FLP package, and can be examined by the user software to determine the exception conditions that occurred during the course of a computation. If it is the responsibility of the user software to initialize this word before calling any of the floating point routines.

The second method that the package uses to signal exceptions is to call a user floating point exception handler whenever such conditions occurs. The corresponding exception bit in ERRFLG is set before calling the handler. The starting address of the handler should be defined by the symbol FPTRAP.

### Unpacked FLP Format

The IEEE standard FLP format described earlier is very cumbersome to deal with during computation. This is primarily because of splitting of the mantissa between the 2 words. The subroutine in the package unpack the input IEEE FLP numbers into an internal representation, do the computations using this representation, and finally pack the result into the IEEE format before returning to the calling program. The unpacking is done by the subroutine FUNPAK and the packing by the subroutine FPAK. The unpacked format consists of 3 words and is organized as follows:

**Unpacked FLP**

| Fn-Exponent (8-bit biased) | Fn-Sign (extended to 8-bits) |
|---|---|
| MS 16-bits of Mantissa (implicit 1 is present as MSB) | |
| LS 8-bits of Mantissa | Eight 0's |

Since all computations are carried out in this format, note that the result is actually known to 32 bits. This 32-bit mantissa is rounded to 24 bits before packed to the IEEE format.

### Algorithms

All the arithmetic algorithms first check for easy cases when either F1 or F2 is zero or a NaN. The result in these cases is immediately available. The description of the algorithms below is for those cases when neither F1 or F2 is 0 or a NaN. Also, in order to keep the algorithm description simple, the check for underflow/overflow at the various stages is not shown. The documentation in the program, the flowcharts given below, and the theory as described in the references should allow these programs to be easily maintained.

## FPADD AND FPSUB

Before a floating point add/subtract instruction is executed, the 2 operands in normalized form

The processing steps are as follows:

1. Compare the 2 exponents.
2. Align the mantissas by equalizing their exponents.
3. Compute result sign as the XOR of the signs of the 2 numbers.
4. Add/Subtract the mantissas.

   For subtract, FP2 is complemented and added with FP1, i.e., FSUB = FP1 + (–FP2).
5. Normalize the resulting sum/difference.
6. Pack the exponent, sign and mantissa in IEEE format and return.

## FMULT = FP1 * FP2

Floating-point multiplication is accomplished by multiplying the mantissas of the 2 operands and adding their corresponding exponents. Exponent overflow or underflow may occur when true addition is performed on 2 exponents of the same sign.

The processing steps are as follows:

1. Add the 2 exponents and subtract 7FH (IEE bias of $127_{10}$) to yield the result exponent.

   Result Exponent = FP1_EXP + FP2_EXP – 127
2. XOR the sign bits to get the result sign.

   Result Sign = FP1_SIGN XOR FP2_SIGN
3. Compute FP1_HI x FP2_HI = C1_HI.C1_LO.
4. Compute FP1_HI x FP2_LO = C0_HI.C0_LO.
5. Add C0_HI + C1_LO = C2_LO.
   If more than 16-bits, then C1_HI += 1.
6. Compute FP1_LO x FP2_HI = C3_HI.C3_LO.
7. Add C3_HI + C2_LO = C4_LO.
   If more than 16-bits, then C1_HI += 1.
8. Normalize mantissa. If MSB of C1_HI =/= 1,
   then result exponent += 1 else left shift C1_HI.C4_LO.
9. Round C1_HI.C4_LO to get result mantissa.
10. Pack the exponent, sign and mantissa in IEEE format and return.

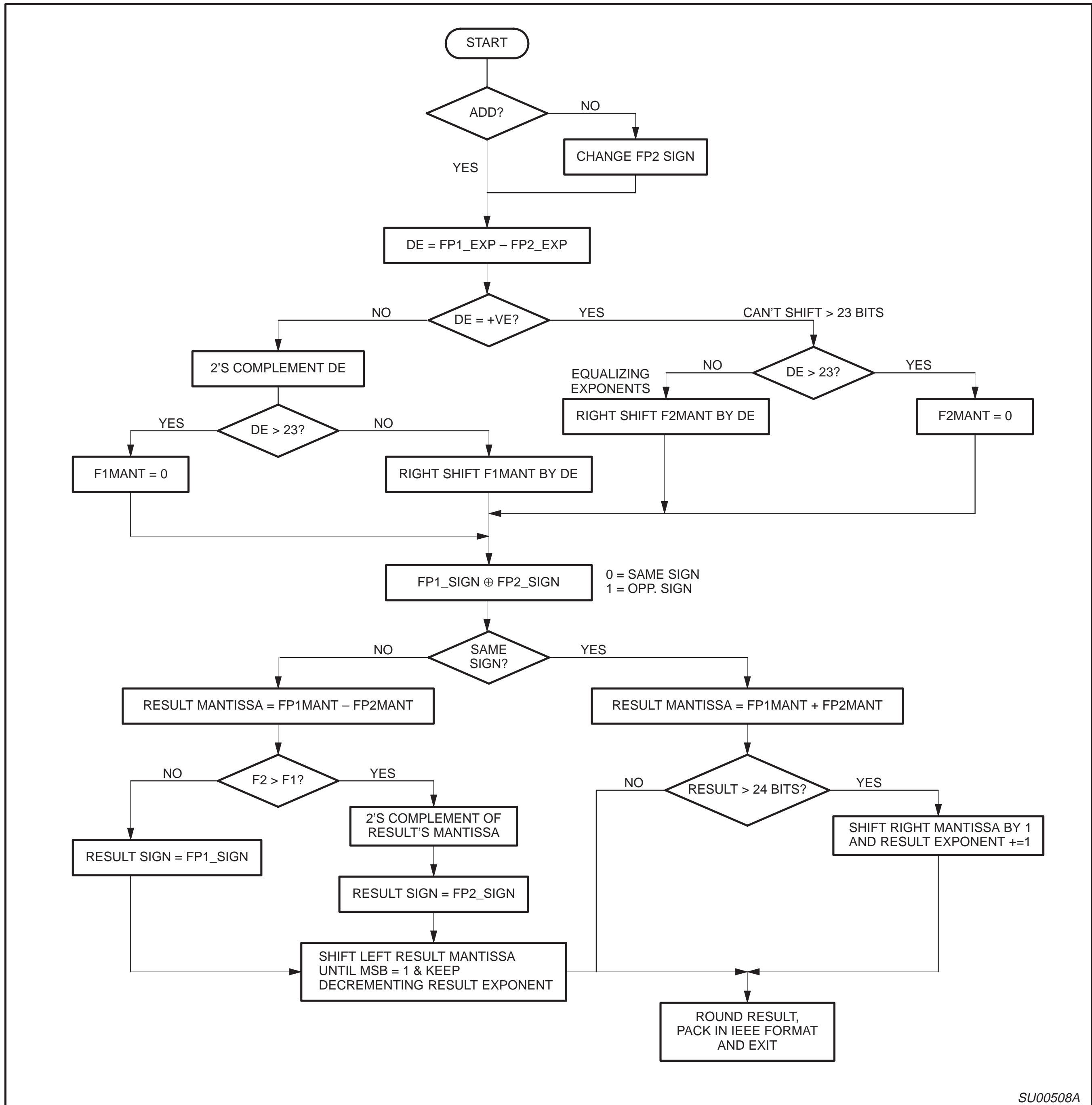
## FPDIV = FP1/FP2

The way a floating-point DIVIDE instruction is executed is analogous to that of a Floating Point Multiply, except that mantissa multiplication is replaced by mantissa division and the exponent addition by exponent subtraction. Exponent overflow or underflow may occur when true addition is performed on the 2 exponents of opposite signs. The scheme must avoid the situation of having a divisor which is smaller than dividend mantissa, including the special case of a 0 divisor. With this constraint, the post normalization is unnecessary in FLP division as long as pre-normalization was conducted to avoid quotient overflow.

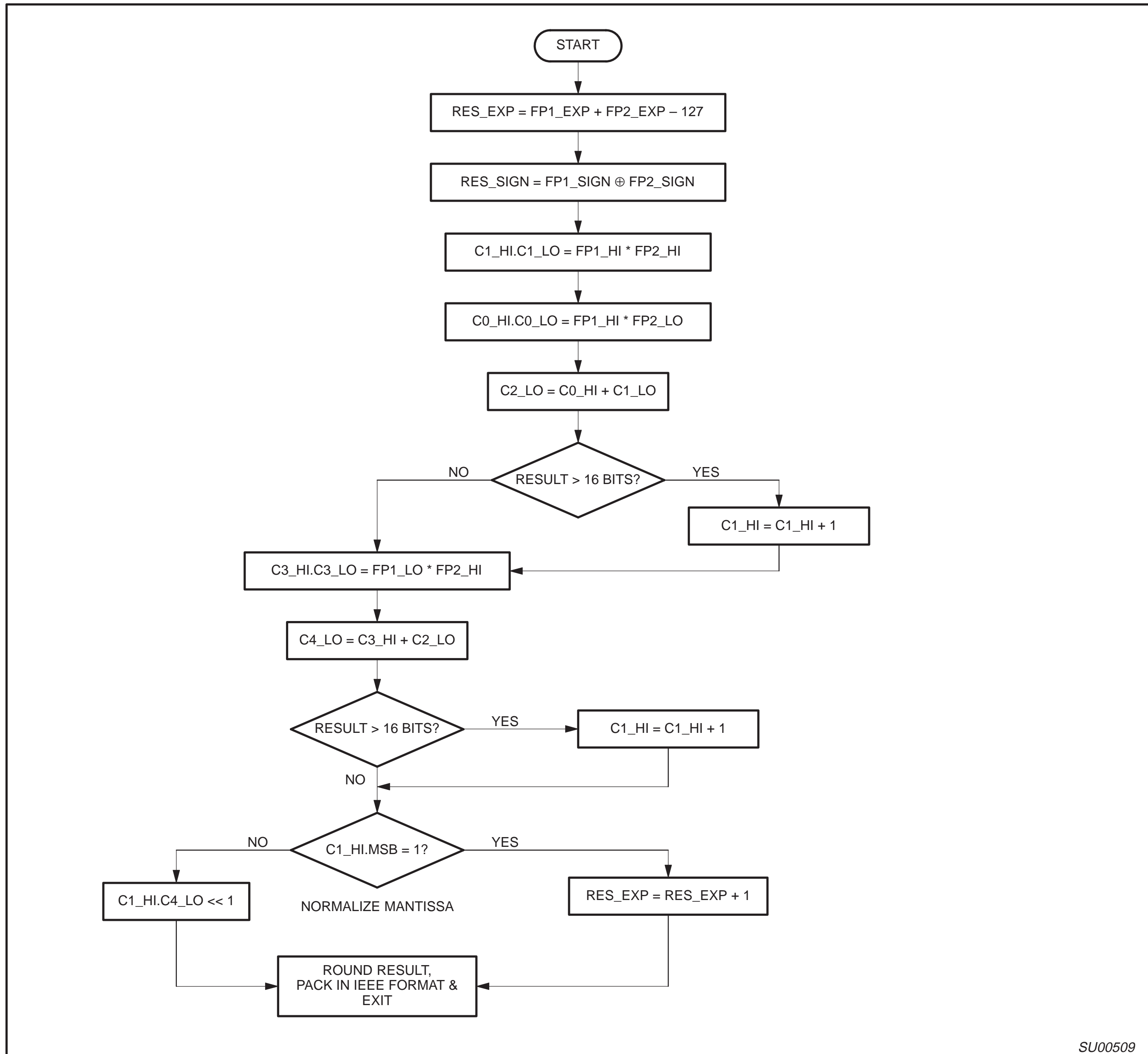
The processing steps are as follows:

1. Compare FP1_HI and FP2_HI.
   If FP2_HI > FP1_HI, then go to step 3, else go to step 2.
2. Shift right FP1 and FP1_EXP += 1.
3. Compute FP1_EXP – FP2_EXP + 127 to get C_EXP.
4. Compute FP1_SIGN XOR FP2_SIGN to get result sign, i.e., C_SIGN
5. Compute FP1_HI x FP2_LO = M1_HI.M1_LO.
6. Divide M1_HI.M1_LO / FP2_HI = M2_HI (Quotient)
7. Do a true subtract FP1_LO – M2_HI = M3_LO.
   If result –ve then go to step 8 else FP1_HI –= 1 and go to step 8.
8. Divide FP1_HI.M3_LO / FP2_HI = C1_HI (Quotient) + R1 (remainder)
9. Divide R1.0000 / FP2_HI = C1_LO (Quotient)
10. If MSB of C1_HI = 1, then go to step 11, else shift left C1_HI.C1_LO, C_EXP –= 1, and go to step 11.
11. Round C1_HI.C1_LO to get C_HI.C_LO, go to step 12
12. Pack the exponent, sign and mantissa in IEEE format and return.

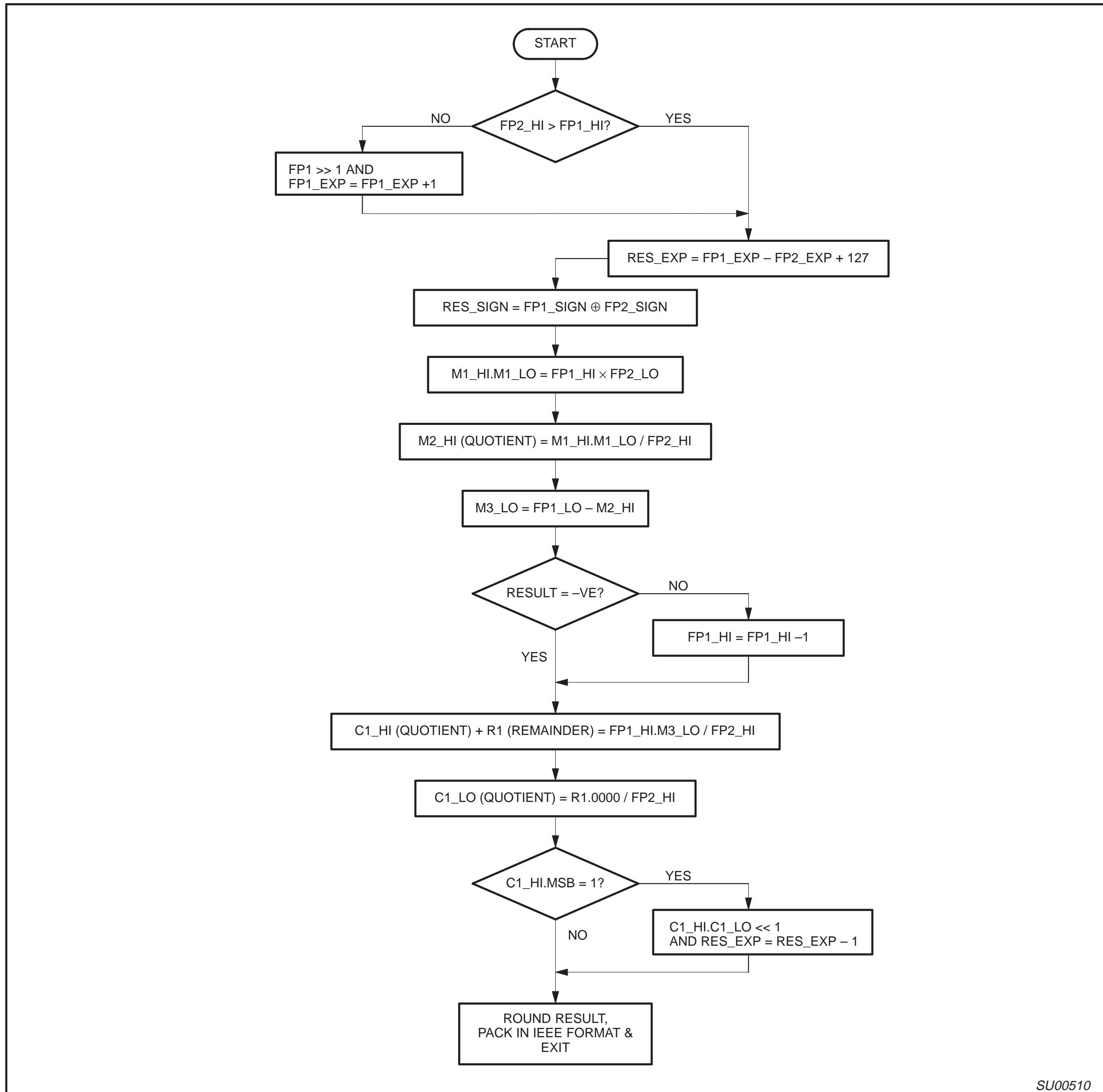
## FPADD AND FPSUB



## FPMULT



## FPDIV



## APPENDIX A

### Conversion of Floating Point Numbers

In general IEEE FP = +– mantissa X 2 $^{exponent}$

Format = Sign bit (1). Biased Exponent bits (8). Normalized Mantissa bits (23), e.g., convert 1.0 to a 32–bit IEEE Floating Point:

1 = 1.0 X $2^0$;
Biased Exponent = 0 + 127 = 127 = 7F Hex
The 24-bit normalized mantissa = 100000000000000000000000;
Sign = positive = 0;

So, IEEE 1.0 = 0 0111 1111 00000000000000000000000

which is 3f80 0000 hex & so on.

### Some Floating Point Numbers are given below for user reference:

–1.0 = BF80 0000;
+1.0 = 3F80 0000

–0.25 = BE80 0000;
+0.25 = 3E80 0000

–0.50 = BF00 0000;
+0.50 = 3F00 0000

6.250 = 40C8 0000;
1.625 = 3FD0 0000;

12.0 = 4140 0000

### References

1. IEEE Draft 8.0 on *A Proposed Standard for Binary Floating-point Arithmetic, 1981*
2. K.Hwang, *Computer Arithmetic, John–Wiley and Sons, 1979*
3. *Microprocessor System Design Concepts, Nikitas A. Alexandridis, Computer Sc.Press, 1984*
4. *Microprocessors and Digital Systems, Douglas V. Hall, McGraw-Hill, 1980*

## APPENDIX B

```
#include <stdio.h>
#include ”fpp.h”




void main(void)

{

unsigned int fprtn;
unsigned short j;
FILE *fp;


start:
       while(1)       // clear RAM
 {

                  for(j=0; j<6;j++)
                  {
                      tmp1[j] = 0;
                      tmp2[j] = 0;
                  }


                  // Menu

                  printf(”\n\n\n\n”);
                  printf(”             XA FLOATING POINT ARITHMETIC FUNCTION MENU\n”);
                  printf(”            -------------------------------------------\n”);
                  printf(”\n”);
                  printf(”              A ..............Floating Point Add\n”);
                  printf(”              B ..............Floating Point Subtract\n”);
                  printf(”              C ..............Floating Point Multiply\n”);
                  printf(”              D ..............Floating Point Divide\n”);
                  printf(”              H ..............Help File\n”);
                  printf(”              J ..............Error Register Status\n”);
                  printf(”              Q ..............Exit Menu\n\n\n\n\n”);
                  printf(”              ==>”);


                  fprtn = getche();   /* wait for user i/p */

                  getch();   // wait for <CR>
                  printf(”\n\n”);


                  /* Select Floating Point Routine */

                  switch(fprtn)
                  {
                      case ’A’ :
                      case ’a’ :
                                    printf(”Floating Point Addition in Progrss....\n\n\n”);
                                    getnum();
                                    fpadd();
                                    break;
```

```
                case 'B' :
                case 'b' :
                              printf("Floating Point Subtraction in Progrss....\n\n\n");
                              printf("\n\n\n\n");
                              getnum();
                              fpsub();
                              break;

                case 'C':
                case 'c':
                              printf("Floating Point Multiplication in Progress....\n\n\n");
                              printf("\n\n\n\n");
                              getnum();
                              fpmult();
                              break;



                case 'D':
                case 'd':
                              printf("Floating Point Divide in Progress....\n\n\n");
                              printf("\n\n\n\n");
                              getnum();
                              fpdiv();
                              break;

                case 'H':
                case 'h':
                              if( (fp = fopen("help","r")) == NULL)
                              printf("can't open flpdat for read\n");

                              fcopy(fp,stdout);
                              fclose(fp);
                              printf("Hit any key to continue ...");
                              getch();
                              goto start;


                case 'J':
                case 'j':
                              show_err_reg();
                              printf("Hit any key to continue ...");
                              getch();
                              goto start;
                case 'Q':
                case 'q':
                              printf("******Hit Ctrl+C to exit ....!!!!*******\n");
                              getch();


                default:
                              printf("   *****************************************************\n");
                              printf("   *  UNKNOWN COMMAND, GOODBYE!!                       *\n");
                              printf("   *****************************************************\n");
                              goto start;

  }

 }

}
```

```
/* Exception Handling Routines */

void divbyz (void)

{
                ERRFLG |= 8;            /* set the DIVBY0 bit (#3) */
                fp_lo = 0;                          /* Return  NaN */
                fp_hi = NANH;
                fptrap();               /*  exception handler */
                return;
}


/* Illegal Operand – one of the numbers is a INVALID. */

void fnan(void)

{

                ERRFLG |= 4;            /* set the NAN operand bit */
                fp_lo = NANL;
                fp_hi = NANH;           /* return NAN in fp_lo and fp_hi */
                fptrap();
                return;
}



void undflw(void)               /* exponent underflow */

{
                ERRFLG |= 2;            /* set the exponent underflow bit */
                fp_lo = 0;              /* clear FP */
                fp_hi = 0;
                fptrap();
                return;
}

void ovflw(void)                /* exponent overflow */

{
                ERRFLG |= 1;            /* set the exponent overflow bit */
                fp_lo = 0;
                fp_hi = NANH;
                fptrap();
                return;
}


/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'INVALID'
   Returns 0 if number != INVALID and
   Returns 1 if Number == INVALID
*/

unsigned int  fnanchk()

/* Subroutine to check if a SP floating point number is a NaN
   Returns 1, if YES, and 0 if NOT */

{
                long tmp;

                tmp = fp_hi;
                tmp = tmp >> 1;         /* Shift left fp_hi by 1 */
                if(tmp > 0xfeff)        /* feff + 1 = ff00 */
                return 1;               /* biased exp >= 255 & f =/= 0 */
                return                  /* OK */
```

```
}
/* Subroutine to check if a single precision FLP # stored in the
   IEEE flp format in registers fp_lo and fp_hi is 'ZERO'
   Returns 0 if number != and
   Returns 1 if Number == 0
   Note : fp "0" = 1.0 x 2 e -127  i.e if biased exp = 0, fp = 0
*/

char zchk(void)

{
                unsigned long tmp;
                tmp = fp_hi;

                tmp = tmp << 1;        /* Shift left fp_hi by 1 */

                if(tmp > 0x00ff)
                return 0;
                return 1;
}


/* subroutine to unpack a SP IEEE formatted FP # and held in regs.fp_hi
   and fp_lo.  The unpacked format occupies 3 words & is organized as
   follows:

   WORD2 :  eeeeeeee ssssssss     -> biased-exp.sign
   WORD1 :  1mmmmmmm mmmmmmmm     -> 16 MSB of Mantissa (m23 : m16)
   WORD0 :  mmmmmmmm 00000000     -> 8 LSB of Mantissa.zeros

   e7:0 - 8-bit exponent in excess-127 format
   s7:0 - sign bit -> 0x00 = +ve, 0xff = -ve ;
   m23:0 - normalized mantissa i.e 1.0000...
*/

char* funpak(fparray)                // returns ptr. to a character array to fpadd
int *fparray;                        //pointer to the flp. array passed by fpadd

{

                char  sign;
                unsigned short i= 0,j = 0;

                     if(k=2) k=0;
                     else
                     k ++;
                     flpar[i] = 0;   // clear lo-byte of fp0



                flpar[++i] = fparray[j] & 0x00ff; /* fp0_hi = m0:m7 */
                flpar[++i] = (fparray[j++] & 0xff00) >> 8; /* fp1_lo = m8:m15 */


                flpar[++i] = fparray[j] & 0x007f; /* m16:m22 */


                flpar[i] |= 0x80;                 /* set bit7 -> normlz. bit */


                sign = (fparray[j] & 0x8000) >> 15; /* check sign bit */
                                                    /* fp2_lo = 8 sign bits */

                  if (sign)                         /*  -ve number ? */

                    flpar[++i] = 0xff;
                  else                              /* yes */
                    flpar[++i] = 0x00;              /* no */

      flpar[++i] = (fparray[j] & 0x7f80) >> 7;      /**/
                return (flpar);                     /* return the ptr to the array */
}
```

```
/* subroutine to pack a SP held in 3 words flpar0:3 into IEEE format.
   The packed format is stored in fp_hi & fp_lo as follows:
   fp_hi :  seeeeee  emmmmmmm     -> sign.biased-exp.7 MSBs of mantissa
   fp_lo :  mmmmmmmm mmmmmmmm     -> 16 LSBs of Mantissa

                Result Stored in tmp2[i]; i=0:5
                tmp2[0] = 0x00; tmp[1] = m0:m7; tmp[2] = m8:m15;
                tmp[3] = 1.m16:m22; tmp[4] = sign7:0; tmp[5] = exp7:0
*/


void fpak(void)

{
                int i=1;
                unsigned int sign,tmpc;
                long ltmp1,ltmp2;


                fp_lo =  tmp2[i++];                 /* get m0:m7 */
                tmpc = tmp2[i++];
                fp_lo = fp_lo | (tmpc << 8);        /* get m8:m15 */


                fp_hi = tmp2[i++];                  /* get 1.m22:m16 */
                ltmp1 = (fp_hi & 0x007f);
                ltmp1 = ltmp1 << 16;                /* mask & shift */


                sign = tmp2[i++];                   /* save sign-bit(s) */
                ltmp2 = tmp2[i];                    /* exponent */
                ltmp2 = ltmp2 << 23;
                ltmp2 = ltmp2 | ltmp1;

                if(sign)
                ltmp2 = ltmp2 | 0x80000000;

                printf("\n\nThe IEEE packed result is:");
                printf("%lx\n",ltmp2);
}


/* This routine rounds up the 32-bit mantissa resulted from FP operations
   to a 24-bit number */

void fround(void)
{

    unsigned int i=3, ctmp;
    long tmpl;

                tmpl = tmp2[i--];   /* 1.m22:16 */
                tmpl = tmpl << 8;
                ctmp = tmp2[i--];   /* m15:8 */
                tmpl = tmpl|ctmp;
                tmpl = tmpl << 8;
                ctmp = tmp2[i--];   /* m7:0 */
                ctmp = tmp2[i];

                    if(ctmp & 0x80)   /* if bit 7 i not set in the LSB */

                                 tmpl += 0x0100;  /* Increase next byte by 1 */

                                 ctmp = tmp2[5];

                                   if(tmpl & 0x1000000)    /* carry out of MSB? */
                                   {
                                                              tmpl = tmpl >> 1;
                                                              ctmp++;           /* exp = exp+1 */

                                                                if(ctmp > 255)

                                                                             tmp2[5] = 0xff;
```

```
                                  }

                                                 tmp2[1] = (tmpl & 0x000000ff);
                                                 tmp2[2] = (tmpl & 0x0000ff00) >> 8;
                                                 tmp2[3] = (tmpl & 0x00ff0000) >> 16;
}

/* User supplied FLP Trap Routine */

void fptrap(void)

{
                printf("\n\nException Occurance !!!\n\n");

                printf("Error Flag Register = %x",ERRFLG);
                /* User exception handler

                ....
                ....
                ....
                ....
                ....
                ....
                ....
                */

}

void show_err_reg(void)

{
    printf("\n The Error Flag Register Bit Map is as follows :");
    printf("\n --------------------------------------\n\n");
    printf("        ---------------------------------------\n");
    printf("       | - | - | - | - | DBZ | IOP | EUF | EOV |\n");
    printf("        ---------------------------------------\n\n\n");
    printf(" where DBZ = Divide by Zero exception\n");
    printf("       IOP = NaN or Invlaid operand\n");
    printf("       EUF = Exponent Underflow\n");
    printf(" and   EOV = Exponent Overflow\n\n\n");


    printf("The status of the register after the operation is :");
    printf("0x%x\n\n",ERRFLG);
 }
/* FPADD - Floating Point Add
   It is assumed two  floating point numbers F1 & F2 are in IEEE format
   Format :
   Fn = fpnh (s.e7:0.m22:16) + fpnl (m15:0)  -> In registers
*/


int fpadd()

{               // 1

long dmant, flpm1, flpm2, lnfp, tlong;
int  *flpn, temp1, temp2;
char dexp, i=0,j=0,k=5,t=0, exp1, exp2;

                exp1 = tmp1[k];   /* get exponent of 1st */

                exp2 = tmp2[k];   /* get exponent of 2nd */

                dexp = (exp1 - exp2);           /* difference in exponent */

                printf("\n\n");
                printf ("DEXP =  %x\n", dexp);
```

```
/* CASE 1: */
                              if(dexp > 0 && dexp < 23)     /* f1exp > f2exp */
  {  //2


                              tmp = dexp;

                /* if exp > 23, can't shift mantissa
                                                       more than 23-bits */


                              while(tmp--)
                              {
                                for(i=0; i<=3; i++)
                                {
                                tmp2[i] = tmp2[i] >> 1;
                                }
                               }



                               i = 4;
                               temp1 = tmp1[i] & 0x00ff;  /* get the fp1 sign byte */
                               printf("\nFP1 SIGN BITS = %x", temp1);

                               temp2 = tmp2[i] & 0x00ff;  /* get the fp2 sign byte */
                               printf("\nFP2 SIGN BITS = %x", temp2);



loop_here:
                               i = 4;
                               rsign = (tmp1[i] ^ tmp2[i]);  /* ex-or sign bits */

                               if(rsign != 0)                /* if different sign */

                { //44
                               printf("\nFPs ARE OF DIFFERENT SIGNS!!\n");

                               flpm1 = getmant(0);
                               flpm2 = getmant(1);

                               dmant = flpm1 - flpm2;

                               if (flpm1 > flpm2)            /* F1man > F2man */

               {  //22

                               printf("\n");
                               printf("FP1 MANTISSA GREATER THAN FP2 MNATISSA\n");
                               printf("\n\n");

                               tmp2[4] = tmp1[4];           /* result sign = sign of F1 */
                               rsign = tmp1[4];             /* stored in FP2 sign byte */


                /* Shift left mantissa till MSB = 1 */

                         for(k=0; k <= 23  && ((dmant & 0x00800000) == 0); k++)
                              {
                                                         dmant = dmant << 1;
                                                         exp1 = exp1 - 1;
                              }


                         /* save result in tmp2[i] array */
                                           tmp2[5] = exp1;
                                           tmp2[4] = rsign;
                                           tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                           tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                           tmp2[1] = (dmant & 0x000000ff);
```

```
            printf("RESULT EXPONENT : %x\n", exp1);
            printf("RESULT MANTISSA (NORMLZD) = %lx\n", dmant);
            printf("RESULT SIGN = %x\n", rsign & 0x1);

                              fround();    /* round the result */
                                         fpak();                    /* Pak & leave */

                                         printf("Hit any key to continue ...");
                                         getch();
                                         return 0;

        } //22
            else if (flpm1<flpm2)              /* F2man > F1man */

        { //23
                              printf(" FP2 MANTISSA GREATER THAN FP1 MANTISSA \n");


                              dmant = ~dmant; /* 2'S COMPLEMENT */
                              dmant++;
                              exp2 = tmp2[5];    /* res.exp = F2.exp */
                              rsign = tmp2[4];
                              tlong = dmant;

            while(!(tlong & 0x800000))
                              { dmant = dmant << 1;
                                        tlong = dmant;
                                        exp2--;
                              }

                            dmant = dmant & 0xffffff;



                          /* save result in tmp2[i] array */
                             tmp2[5] = rexp;
                             tmp2[4] = rsign;
                             tmp2[3] = (dmant & 0x00ff0000) >> 16;
                             tmp2[2] = (dmant & 0x0000ff00) >> 8;
                             tmp2[1] = (dmant & 0x000000ff);

                          printf("\n\n");
                          printf("THE RESULT MANTISSA (NORMLZD) IS = %lx\n",dmant);
                          printf("THE RESULT EXPONENT IS = %x\n", exp2);
                          printf("THE RESULT SIGN IS = %x\n", rsign & 0x1);

                               fround();
                               fpak();
                               printf("Hit any key to continue ...");
                               getch();
                                                                return(0);
        } //23



    } // 44

        else if(rsign == 0) // same sign, so ex-OR is 0
```

```
    { //55
                    printf("\n");
                    printf("FPs ARE OF SAME SIGN!!\n");

                                   flpm1 = getmant(0);
                                   flpm2 = getmant(1);
                                   dmant = flpm1 + flpm2;
                                   rsign = tmp1[4];
                                   rexp = exp1;

                                   tlong = dmant;
                                   tlong = tlong & 0x01000000;// check if carry set


                                   if(tlong)
                                {
                                dmant = dmant >> 1;
                                rexp = rexp + 1;
                                }

                                /* save result in tmp2[i] array */
                                   tmp2[5] = rexp;
                                   tmp2[4] = rsign;
                                   tmp2[3] = (dmant & 0x00ff0000) >> 16;
                                   tmp2[2] = (dmant & 0x0000ff00) >> 8;
                                   tmp2[1] = (dmant & 0x0000ff);


                                printf("\n\n");
                                printf("THE RESULT MANTISSA(NORMLZD) IS = %lx\n",dmant);
                                printf("THE RESULT EXPONENT IS = %x\n", rexp);
                                printf("THE RESULT SIGN IS = %x\n", rsign & 1);


                                  fround();
                                  fpak();      /* pack in IEEE format */

                    printf("Hit any key to continue ...");
                                getch();

                                   return(0);
    } //55


} // 2



                    else if (dexp > 23)

{ //99
                                printf("DIFFERENCE IN EXPONENT IS  GREATER THAN 23\n");

                                for(i = 0; i<= 3; i++)
                                tmp2[i] = 0;
                                goto loop_here;
} //99



    else if (dexp < 0)
```

```
{//6
               printf("DIFFERENCE IN EXPONENT IS NEGETIVE\n");
               printf("FP2 EXPONENT GREATER THAN  FP1 EXPONENT\n");

               tmp = ~dexp ;      /* 2's complement */
               tmp++;

       printf("2's COMPLEMENT OF DEXP = %d\n", tmp);


                                 if(tmp < 23)
     {
                                 while(tmp--)
                                 {
                                   for(i = 0; i<= 3; i++)
                                   {
                                   tmp1[i] = tmp1[i] >> 1;
                                   }
                 }
                    goto loop_here;
     }

                                   else if(tmp > 23)               /* dexp > 23 */
    {
                                   for(i=0; i<= 3; i++)
                                   tmp1[i] = 0;                    /* F1mant = 0 */
                                   goto loop_here;
    }

  } //6
                    else if(dexp == 0)                             /* shift done */
                      printf("FPs GOT SAME EXPONENT!!\n");
                    goto loop_here;

} // 1


/* Get the mantissa for both FP in 24-bit format */


long getmant(val)
 unsigned short val;

{
    long lnfp, flpm;
    unsigned short i;
                                   i= 1;
                                   lnfp = 0;
                                   flpm = 0;

    if (!val)
                    {

                                   lnfp = tmp1[i];
                                   flpm |= (lnfp & 0x000000ff);

                                   lnfp = 0;
                                   i++;
                                   lnfp = tmp1[i];
                                   lnfp = lnfp << 8;
                                   flpm |= lnfp & 0x0000ff00;


                                   lnfp = 0;
                                   i++;
                                   lnfp = tmp1[i];
                                   lnfp = (lnfp << 16);
                                   flpm |= (lnfp & 0x00ff0000);


                                   flpm = flpm & 0x00ffffff;

                                   printf("\n FP1 MANTISSA = %lx\n", flpm);

                    }
```

```
                else
                   {
                              i=1;
                              flpm = 0;
                              lnfp = 0;
                              lnfp = tmp2[i++];
                              flpm |= (lnfp & 0x000000ff);

                              lnfp = 0;
                              lnfp = tmp2[i++];
                              lnfp = (lnfp << 8);
                              flpm |= (lnfp & 0x0000ff00);

                              lnfp = 0;
                              lnfp = tmp2[i];
                              lnfp = (lnfp << 16);
                              flpm |= (lnfp & 0x00ff0000);

                              flpm = (flpm & 0x00ffffff);

                              printf(" FP2 MANTISSA = %lx\n", flpm);

                   }
      return (flpm);
}


int fpsub()
{
   unsigned char fp2_sign;

   fp2_sign = tmp2[4];
   fp2_sign = ~fp2_sign;

   tmp2[4] = fp2_sign;
   fpadd();
}


void getnum()

{
unsigned int fp[3], px;
unsigned short i,j,k;

printf("Type the lo-word of FP#1 in IEEE format :");
                 scanf("%x",&px);
                 i= 0;
                 fp[i] = *&px;

                 printf("Type the hi-word of FP#1 in IEEE format :");
                 scanf("%x",&px);
                 fp[++i] = *&px;


                 funpak(fp);  // pass the array ptr. to unpack routine

                 for(k=0,j=0; k<=5; j++,k++)
                 {
                 tmp1[k] = flpar[j];
                 }
                 printf("\n");


                 i=0;
                 printf("\n\n");
                 printf("Type the lo-word of FP#2 in IEEE format :");
                 scanf("%x",&px);
                 fp[i] = *&px;
```

```
                  printf("Type the hi-word of FP#2 in IEEE format :");
                  scanf("%x",&px);
                  fp[++i] = *&px;
                  funpak(fp);

                  for(k=0,j=0; k<=5; j++,k++)
                  {
                  tmp2[k] = flpar[j];
                  }

}

void fpmult()
{

    int fp1_exp,fp2_exp, res_sign;
    unsigned int fp1_hi, fp1_lo, fp2_hi, fp2_lo, tmp;
    unsigned int c0_hi,c0_lo,c1_hi, c1_lo, c2_lo,c3_hi, c3_lo, c4_lo;
    int res_exp;
    long ltmp;


    fp1_exp = tmp1[5];
    fp2_exp = tmp2[5];

                  res_exp = (long)(fp1_exp + fp2_exp -127); // result exponent


                                  if(res_exp < 0)     // underflow
                                  undflw();


                                  if (res_exp > 255)
                                  ovflw();


                  res_sign = tmp1[4] ^ tmp2[4];  // XOR = result sign


                   tmp = tmp1[3] << 8;   /* 1.m22:8 = FP_HI*/
                   tmp = tmp & 0xff00;
                   tmp |= tmp1[2];

                   fp1_hi = tmp;
                   fp_hi = fp1_hi;

                   tmp = fnanchk();

                                  if(tmp)
                                  fnan();       // F1 is a NaN


                   tmp = tmp2[3] << 8;   /* 1.m22:8 = FP_LO */
                   tmp = tmp & 0xff00;
                   tmp |= tmp2[2];

                   fp2_hi = tmp;
                   fp_hi = fp2_hi;

                                  tmp = fnanchk();

                                  if(tmp)
                                  fnan();       // F2 is a NaN


                   tmp = zchk();  // Check for F2 = 0

                                  if(tmp)  {
                                  printf("\nResult is = 0 as Multiplier is 0\n");   // F2 = 0
                                  goto endprog; }

                   fp_hi = fp1_hi;
```

```
  tmp = zchk();  // Check for F1 = 0

                if(tmp) {
                printf("\nResult is = 0 as Multiplicand is 0\n");   // F1 = 0
                goto endprog;  }


  tmp = tmp1[1] << 8;   /* m7:0.0000 */
  tmp = tmp & 0xff00;
  tmp |= tmp1[0];

  fp1_lo = tmp;

  tmp = tmp2[1] << 8;
  tmp = tmp & 0xff00;
  tmp |= tmp2[0];

  fp2_lo = tmp;

  ltmp =  (long)fp1_hi * (long)fp2_hi;  /* FP1_HI * FP2_HI */


  c1_hi = (ltmp &  0xffff0000) >> 16;
  c1_lo = ltmp & 0x0000ffff;

  ltmp = (long)fp1_hi * (long)fp2_lo;
  c0_hi = (ltmp  & 0xffff0000) >> 16;
  c0_lo = ltmp &0x0000ffff;

  ltmp  = c0_hi + c1_lo;

              if(ltmp & 0x10000)
                  c1_hi++;

  c2_lo = ltmp & 0xffff;


  ltmp = (long)fp1_lo * (long)fp2_hi;

  c3_hi = (ltmp & 0xffff0000) >> 16;
  c3_lo = ltmp & 0x0000ffff;

  ltmp = c3_hi + c2_lo;

             if(ltmp & 0x10000)
                 c1_hi++;


  c4_lo = ltmp & 0xffff;


  ltmp = c1_hi;
  ltmp = ltmp <<8;
  ltmp = (ltmp & 0xffffff00) | c4_lo;

  if(!(c1_hi & 0x8000))
  ltmp = ltmp << 1;

  else
  res_exp++;

  if(res_exp > 254)
  ovflw();


/* save result in tmp2[i] array */
                tmp2[5] = res_exp;
                tmp2[4] = res_sign;
                tmp2[3] = (ltmp & 0x00ff0000) >> 16;
                tmp2[2] = (ltmp & 0x0000ff00) >> 8;
                tmp2[1] = (ltmp & 0x000000ff);
```

```
                  printf("\n\n");
                  printf("RESULT EXPONENT : %x\n", res_exp);
                  printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                  printf("RESULT SIGN = %x\n", res_sign & 1);

                  fround();


                /* final check of exponent */

                  tmp = tmp2[5];

                               if(!tmp)
                               undflw();  /* exponent underflow */

                               if (tmp > 254)
                               ovflw();

                               fpak();
endprog:
                  printf("Hit any key to continue ...");
                                   getch();
}


void fpdiv(void)

{
    unsigned int tmp, fp1_hi, fp2_hi, fp1_lo, fp2_lo, c1_hi, c1_lo;
    unsigned char i, c1_exp, c1_sign, fp1_sign, fp2_sign;
    long ltmp, rem, lfpm1, lfpm2, tmplng;
    unsigned int m1_hi, m1_lo, m2_hi, m2_lo, m3_hi, m3_lo;
    signed int fp1_exp, fp2_exp, dexp;

                  tmp = tmp1[3] << 8;
                  tmp = tmp & 0xff00;
                  tmp |= tmp1[2];

                  fp1_hi = tmp;
                  fp_hi = fp1_hi;
                  tmp = fnanchk();

                               if(tmp)
                               fnan();        // F1 is a NaN

                  tmp = tmp2[3] << 8;
                  tmp = tmp & 0xff00;
                  tmp |= tmp2[2];

                  fp2_hi = tmp;
                  fp_hi = fp2_hi;

                  tmp = fnanchk();

                               if(tmp)
                               fnan();        // F2 is a NaN


                               tmp = zchk();  // Check for F2 = 0

                               if(tmp)  {
                               divbyz();      // F2 = 0
                               printf("\nException as a result of Division by 0\n");
                               goto exit;
                               }

                               fp_hi = fp1_hi;
                               tmp = zchk();  // Check for F1 = 0

                               if(tmp)        // Result = 0
                               {
                               printf("\nResult of Division of a zero Dividend is = 0\n");
                               goto exit;
                               }
```

```
                tmp = tmp1[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp1[0];

                fp1_lo = tmp;


                tmp = tmp2[1] << 8;
                tmp = tmp & 0xff00;
                tmp |= tmp2[0];

                fp2_lo = tmp;


                lfpm1 = fp1_hi;
                lfpm1 = lfpm1 << 16;
                lfpm1 |= fp1_lo;

                lfpm2 = fp2_hi;
                lfpm2 = lfpm2 << 16;
                lfpm2 |= fp2_lo;


/* Exponent bits */
                fp1_exp = tmp1[5];

                fp2_exp = tmp2[5];


/* Sign bits */
                fp1_sign = tmp1[4];
                fp2_sign = tmp2[4];



    /* Ensure that fp2_hi > fp1_hi */


    if(fp1_hi > fp2_hi)          // compare fp1_hi & fp2_hi

                {
                lfpm1 >> 1;
                fp1_exp++;
                }

   /* else := good */


     fp1_hi = (lfpm1 & 0xffff0000) >> 16;
     fp1_lo = lfpm1 & 0x0000ffff;


     dexp = (fp1_exp - fp2_exp);  // difference in exponent (2's compl)


     c1_exp = dexp + 127;
     c1_sign = fp1_sign + fp2_sign;

     ltmp = (long)fp1_hi*(long)fp2_lo;  // m1_hi.m1_lo

     m1_hi = (ltmp & 0xffff0000) >> 15;
     m1_lo = (ltmp & 0x0000ffff);


     m2_hi = ltmp / (long)fp2_hi;   // Quotient
     m3_lo = fp1_lo - m2_hi;

     if( m2_hi > fp1_lo)    // B = 0 i.e C = 1
                fp1_hi--;
                ltmp = (unsigned long)fp1_hi;
                ltmp = ltmp << 16;
                ltmp = ltmp|m3_lo;
                tmplng = ltmp;
```

```
                  ltmp = ltmp /(unsigned long)fp2_hi;  // Quotient
                  c1_hi = ltmp & 0x0000ffff;
                  ltmp = tmplng;
                  ltmp =  ltmp % (unsigned long)fp2_hi; // remainder


                  ltmp = ltmp << 16;
                  ltmp = ltmp & 0xffff0000;
                  c1_lo = ltmp/(unsigned long)fp2_hi;


                  if(c1_hi & 0x8000)

                  {
                     c1_hi << 1;
                     c1_lo << 1;
                     c1_exp -= 1;
                  }


                     ltmp = c1_hi;
                     ltmp = ltmp << 16;
                     ltmp = ltmp | c1_lo;


                     /* save result in tmp2[i] array */
                                       tmp2[5] = c1_exp;
                                       tmp2[4] = c1_sign;
                                       tmp2[3] = (ltmp & 0xff000000) >> 24;
                                       tmp2[2] = (ltmp & 0x00ff0000) >> 16;
                                       tmp2[1] = (ltmp & 0x0000ff00) >> 8;
                                       tmp2[0] = ltmp & 0x000000ff;


                     printf("\n\n");
                     printf("RESULT EXPONENT : %x\n", c1_exp);
                     printf("RESULT MANTISSA (NORMLZD) = %lx\n", ltmp);
                     printf("RESULT SIGN = %x\n", c1_sign & 1);


                                       fround();   // round up results in IEEE format

 /* final check of exponent */

                     tmp = tmp2[5];

                                       if(!tmp)
                                       undflw();  /* exponent underflow */

                                       else if (tmp > 0xfe)
                                       ovflw;     /* exponent overflow */


                                         fpak();  // pack in IEEE format
exit:
                     printf("Hit any key to continue ...");
                                           getch();

}

void fcopy(FILE *ifp, FILE *ofp)
{
    int c;
    while ((c=getc(ifp)) != EOF)
                  putc(c,ofp);
}
```

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS
Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

**Philips Semiconductors**
**811 East Arques Avenue**
**P.O. Box 3409**
**Sunnyvale, California 94088–3409**
**Telephone 800-234-7381**

Philips Semiconductors and Philips Electronics North America Corporation
register eligible circuits under the Semiconductor Chip Protection Act.
© Copyright Philips Electronics North America Corporation 1994
All rights reserved. Printed in U.S.A.