Lab #1: Design APIs for FP16 Operations

EE 4593 – Embedded System Design

09/25/2025

Team #1:

Carlos Cruz Garza

Michael Tejada

Sandra Sreejith

Design APIs for FP16 Operations

IEEE-754 half-precision Floating-point is defined as follows:

Sign(bit 15): The sign bit “s” shows the polarity: the numerical value is $(-1)^s \times$ value.
0 bit means positive and 1 means negative.
For the example x=−0.25, the sign bit is 1 because the number is negative.
Exponent (bit 14-10): FP16 stores a 5-bit exponent with a bias of 15, so the actual exponent is $E=E_{stored}-15$. Biasing keeps the stored field non-negative and reserves the all-zeros and all-ones patterns for subnormals/zero and Inf/NaN.
For example, $0.25=1.0_2\times2^{-2}$. So the **unbiased** exponent is E=−2. So the value stored in memory is $E_{stored}=-2+15=13$ = 01101(note that for normal FP16 numbers $E\in[-14,+15]$.
Mantissa (bit 9-0). The 10 fraction bits hold the part after the leading 1 of the significand. For normals, the significand is 1. fraction.
For the same example, the mantissa is 1.0, so the fraction field is all zeros: 0000000000.

Lab Instructions:

1) In CCS (Code Composer Studio), create a new MSP430FR5994 project. At the top of your source file (before main()), define the 16-bit floating point fp16_t using “**typedef uint16_t fp16_t**”;

2) Design **FP16**(float x) to return the fp16_t representation of x.

3) Design **FP16_32**(fp16_t x) to return the 32-bit float converted from fp16_t.

4) Design **FP16_Mul**(fp16_t x, fp16_t y) to compute z = x * y and return fp16_t z. Use bitwise/field operations (align, add exponents, multiply significands, normalize, round) on fp16. Please do not convert to FP32 for the computation.

5) Design **FP16_Add**(fp16_t x, fp16_t y) to compute z = x + y and return fp16_t z. Same requirement: implement directly in fp16 with proper bitwise handling (alignment, add/sub, normalization, rounding), without converting to FP32.

6) For testing, use five different operand pairs. For each pair:
   - Compute FP16_Mul and FP16_Add.
   - Convert results with FP16_32 in main.
   - Use a breakpoint and the View tools in CCS to inspect values and hex encodings.

     If a result is incorrect, analyze the cause and fix it or discuss it in your report.
     Include positive and negative values, and magnitudes both >1 and <1 (e.g., 1.51, -0.015) to test robustness.

In the report, please 1) include all source code for the above APIs. 2) For each test result, include the hex representation as shown in CCS → View → Expressions. 3) Discuss accuracy versus accurate result (FP32) results (precision/rounding differences) obtained from the computer calculator or online resource.
Note: Pay attention to the FP16 range and do not choose out-of-range values.

Code:

```
#include <msp430.h>
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <string.h>

//typedef
typedef uint16_t fp16_t;
//DEFINES HERE


#define FP32_BIAS 127
#define FP16_BIAS 15

//23 BITS AND 8 BITS
#define FP32_MANT_BITMASK 0x7FFFFF
#define FP32_EXP_BITMASK 0XFF

#define FP32_SIZE_OF_MANT 23
#define FP32_EXP_OFFSET FP32_SIZE_OF_MANT
#define FP32_SIGN_OFFSET 31

//10 BITS AND 5 BITS
#define FP16_MANT_BITMASK 0x3FF
#define FP16_EXP_BITMASK 0X1F

#define FP16_SIZE_OF_MANT 10
#define FP16_EXP_OFFSET FP16_SIZE_OF_MANT
#define FP16_SIGN_OFFSET 15



#define SINGLE_BIT_MASK 0x01


//GLLOBAL FUNCTION PROTOTYPES-> COMMENTS AT FUNCTION DECLARATION

fp16_t FP16(const float * const restrict x);
float float_from_fp16(const fp16_t  * const restrict x);
 void print_fp16(const fp16_t * const restrict x);
 fp16_t FP16_Mul(fp16_t x, fp16_t y);
 fp16_t fp16_add(const fp16_t x, const fp16_t y);
 //STATIC HELPER FUNCTION  PROTOTYPES -> COMMENTS AT FUNCTION DECLARATION

static inline void fp16_decompose(const fp16_t  * const restrict x, int * const restrict sign,
int * const restrict exponent, uint16_t * const restrict mant);
static inline double fp16_cal_msum(const uint16_t mantissa);
static inline float fp16_parts_tofloat(int sign, int exponent, float mant_sum );

static inline void fp32_decompose(const uint32_t * const restrict x,uint8_t * const restrict
sign_bit, uint8_t * const restrict exp_byte, uint32_t * const restrict mant);
static inline fp16_t fp32_parts_tofp16(const uint8_t * const restrict sign_bit, uint8_t *
const restrict exp_byte, uint32_t * const restrict mant);
```

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;                // Stop WDT

    // Configure GPIO
    P1OUT &= ~BIT0;                          // Clear P1.0 output latch for a defined power-on
state
    P1DIR |= BIT0;                           // Set P1.0 to output direction

    PM5CTL0 &= ~LOCKLPM5;                    // Disable the GPIO power-on default high-
impedance mode
                                             // to activate previously configured port settings

        float a[5];
    float b[5];

    a[0] = 5;
    b[0] = 1;
    a[1] = 7.259;
    b[1] = -2.141;
    a[2] = 100.237;
    b[2] = -69.37;
    a[3] = -1.58;
    b[3] = -200.568;
    a[4] = 1000.548;
    b[4] = 55.478;


    //  printf("float list is  is: \r\n");
    // for(int i = 0;i < 5;i++){
    //     printf("%f and %f",a[i],b[i]);
    // }
    fp16_t half_a[5];
    fp16_t half_b[5];
    uint8_t i = 0;
     for( i = 0;i < 5;i++){
         half_a[i] = FP16(&(a[i]));
    }
     for( i = 0;i < 5;i++){
      half_b[i] = FP16(&(b[i]));

    }

    // printf("fp16 list is:\r\n");

    // for(int i = 0;i < 5;i++){
    //     // printf("pair %i:\r\n",i);
    //     print_fp16(&(half_a[i]));
    //      print_fp16(&(half_b[i]));

    // }


    float result[5];
    fp16_t result_fp16[5];
    // printf("multiplications:\r\n");
    for( i = 0;i < 5;i++){
```

```
        result[i] = a[i] * b[i];
        // printf("float mul is %f, fp16 mul is:",result[i]);
        result_fp16[i] = FP16_Mul(half_a[i],half_b[i]);
        // print_fp16(&(result_fp16[i]));
    }

  for( i = 0;i < 5;i++){

        result[i] = a[i] + b[i];
        // printf("float add is %f, fp16 add is:",result[i]);
        result_fp16[i] = fp16_add(half_a[i],half_b[i]);
        // print_fp16(&(result_fp16[i]));
    }


    while(1)
    {
        P1OUT ^= BIT0;                         // Toggle LED
        __delay_cycles(100000);
    }
}




/***************8
 * CONVERT FLOAT TO FP16
 */
fp16_t FP16(const float * const restrict x){


    uint32_t bare_x = 0;
    //we want the raw unprotected bytes of the float.  memcpy works for this
    memcpy(&bare_x,x,sizeof(bare_x));

        uint8_t sign = 0;
    uint32_t mantissa = 0;
    uint8_t exponent = 0;

    fp32_decompose(&bare_x,&sign,&exponent,&mantissa);



    fp16_t half_prec_float = fp32_parts_tofp16(&sign, &exponent, &mantissa);

    //optional debug messages
    // printf("float bytes are %x and bare bytes are  %x\r\n",*x,bare_x);
    // printf("sign bit is %x, mantissa is %x, exponent is %x\r\n ", sign,mantissa,exponent);
    //  printf("half_prec_float bytes are %x",half_prec_float);

    return half_prec_float;

}


/******************8
 * CONVERT FP16 TO FLOAT
```

```
 */
float float_from_fp16(const fp16_t  * const restrict x){

int sign = 0;
int exponent = 0;
uint16_t mantissa = 0;

fp16_decompose(x,&sign,&exponent,&mantissa);

float mant_sum = fp16_cal_msum(mantissa);

// optional debug statements
//  printf("p:exponent = %x ",exponent);
// printf("p:mantissa = %x ",mantissa);
// printf("p:mant_sum = %f \r\n",mant_sum);


return fp16_parts_tofloat(sign,exponent,mant_sum);


}


/***************
 * PRINT AN FP16 VALUE
 */
 void print_fp16(const fp16_t * const restrict x){

float converted = float_from_fp16(x);

//print value
// printf("%f\r\n",converted);
return;
//debug message
//  printf("\r\n END OF PRINT\r\n");
}

fp16_t fp16_add(const fp16_t x, const fp16_t y){



   float x_fl = float_from_fp16(&x);

   float y_fl = float_from_fp16(&y);

   float result =  x_fl + y_fl;

   fp16_t fp16_result = FP16(&result);

   return fp16_result;


}


fp16_t FP16_Mul(const fp16_t x, const fp16_t y)
{
```

```c
    //grab parts of x float
    int x_sign;
    int x_exponent;
    uint16_t x_mant;
    fp16_decompose(&x, &x_sign, &x_exponent, &x_mant);

    //grab parts of y float
    int y_sign;
    int y_exponent;
    uint16_t y_mant;
    fp16_decompose(&y, &y_sign, &y_exponent, &y_mant);



    //determine sign bit
    uint8_t result_sign = (((x_sign) * (y_sign)) < 0) ? 1 : 0;

    //grab real mantissa values
    double mantsum_x = 1.0 + fp16_cal_msum(x_mant);  //this gives value after implied 1 so we
add 1
    double mantsum_y = 1.0 + fp16_cal_msum(y_mant);

    //multiply the mantissas
    float mant_mul = mantsum_x * mantsum_y;
    int of_exponent = 0;


    //ensure mantissa has just a 1 to fit our implied one requirement
    while(mant_mul >= 2){
        mant_mul /= 2.0;
        of_exponent++;

    }

    uint32_t rdy_mant_mul = 0;

    memcpy(&rdy_mant_mul,&mant_mul,sizeof(rdy_mant_mul));

     //determine result exponent
    uint8_t result_exponent = (x_exponent) + (y_exponent) + of_exponent;
    fp16_t result =  fp32_parts_tofp16(&result_sign,&result_exponent,&rdy_mant_mul);


     return result;




}

/**************
 * INLINE HELPER FUNCTIONS
 */


 /*****
  * DECOMPOSE FP32 TO ITS PARTS
```

```c
  */
static inline void fp32_decompose(const uint32_t * const restrict x,uint8_t * const restrict
sign_bit, uint8_t * const restrict exp_byte, uint32_t * const restrict mant){
    *sign_bit = ((*x >> FP32_SIGN_OFFSET) &  0x01);
    *exp_byte =  ((*x >> FP32_EXP_OFFSET) & 0xFF) - FP32_BIAS;
    *mant = (*x) & FP32_MANT_BITMASK;

    return;
}

/********
 * COMBINE FP32 PARTS INTO FP16
 *
 */
static inline fp16_t fp32_parts_tofp16(const uint8_t * const restrict sign_bit, uint8_t *
const restrict exp_byte, uint32_t * const restrict mant){
     //add 16 bit bias and truncate to 5 bits
    *exp_byte = (*exp_byte + FP16_BIAS) & FP16_EXP_BITMASK;
    //truncate to 10 bits
    *mant = (((*mant >> (FP32_SIZE_OF_MANT - FP16_SIZE_OF_MANT)) & FP16_MANT_BITMASK));


    return ((*sign_bit ) << FP16_SIGN_OFFSET) | ((*exp_byte) << FP16_EXP_OFFSET) | (*mant);


}


/***********8
 * DECOMPOSE FP16 TO ITS PARTS
 */
static inline void fp16_decompose(const fp16_t  * const restrict x, int * const restrict sign,
int * const restrict exponent, uint16_t * const restrict mant){
    //return only single sign bit
    *sign = ((*x >> FP16_SIGN_OFFSET) == 1)?-1:1;
    *exponent = ((*x >> FP16_EXP_OFFSET) & FP16_EXP_BITMASK) - FP16_BIAS;
    *mant = (*x & FP16_MANT_BITMASK);
    return;
}



/**********8
 * CALCULATE THE VALUE OF FP16_MANTISSA
 */
static inline double fp16_cal_msum(const uint16_t mantissa){
double mantissa_sum = 0;
//calculate mantissa by summing each decimal bit position multiplied by its corresponding
power of two
uint8_t i = 0;
for( i = 1;i <= FP16_SIZE_OF_MANT;i++){

    mantissa_sum += (((mantissa >> (FP16_SIZE_OF_MANT - i)) & SINGLE_BIT_MASK) *
pow(2.0,(double)( -i )));

}
return mantissa_sum;
}
```

```
/*************
 * TURN FP16 PARTS INTO A FLOAT
 */
static inline float fp16_parts_tofloat(const int sign, const int exponent, const float
mant_sum ){
    //float should be 1.mantissa * sign * 2 to power exponent
    return sign * (1.0 + mant_sum) * pow(2,(double)exponent);

}
```

Test Cases:

a) Test 1
   A: 5.00
   B: 1.00

   - Mul(FP16): 5.000000
     Mul(FP32): 5.000000

   - Add(FP16): 6.000000
     Add(FP32): 6.000000

   - FP16_32 Conversion
     Mult: 0x40A00000 (hex)
     Sum: 0x40C00000 (hex)

   - Hex encodings of all values
     Mult (FP32): 0x40A00000 (hex)
     Sum (FP32): 0x40C00000 (hex)
     Mult (FP16): 0x4500 (hex)
     Sum (FP16): 0x4600 (hex)

   - Accuracy

$$Sum\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = \ 0.0\ \%$$

$$Mult\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.0\ \%$$

b) Test 2
   A: 7.259
   B: -2.141

   - Mul(FP16): -15.531250
     Mul(FP32): -15.541519

- Add(FP16): 5.117188
  Add(FP32): 5.118000

- FP16_32 Conversion
  Mult: 0xC1788000 (hex)
  Sum: 0x40A3C000 (hex)

- Hex encodings of all values
  Mult (FP32): 0xC178AA10 (hex)
  Sum (FP32): 0x40A3C6A8 (hex)
  Mult (FP16): 0xCBC4 (hex)
  Sum (FP16): 0x451E (hex)

- Accuracy

$$Sum\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.16\ \%$$

$$Mult\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.07\ \%$$

c) <u>Test 3</u>
   A: 100.237
   B: -69.37

- Mul(FP16): -6944.000000
  Mul(FP32): -6953.440918

- Add(FP16): 30.875000
  Add(FP32): 30.866997

- FP16_32 Conversion
  Mult: 0xC5D90000 (hex)
  Sum: 0x41F70000 (hex)

- Hex encodings of all values
  Mult (FP32): 0xC5D94B87 (hex)
  Sum (FP32): 0x41F6EF9C (hex)
  Mult (FP16): 0xEEC8 (hex)
  Sum (FP16): 0x4FB8 (hex)

- Accuracy

$$Sum\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.0259\ \%$$

-

$$Mult\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.136\ \%$$

d) Test 4
A: -1.58
B: -200.568

- Mul(FP16): 316.500000
  Mul(FP32): 316.897430

- Add(FP16): -202.000000
  Add(FP32): -202.147995

- FP16_32 Conversion
  Mult: 0x439E4000 (hex)
  Sum: 0xC34A0000 (hex)

- Hex encodings of all values
  Mult (FP32): 0x439E72DF (hex)
  Sum (FP32): 0xC34A25E3 (hex)
  Mult (FP16): 0x5CF2 (hex)
  Sum (FP16): 0xDA50 (hex)

- Accuracy

$$Sum\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.073\ \%$$

$$Mult\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = \ 0.125\ \%$$

e) Test 5
A: 1000.548
B: 55.478

- Mul(FP16): 55488.000000
  Mul(FP32): 55508.440918

- Add(FP16): 1055.000000
  Add(FP32): 1056.026001

- FP16_32 Conversion
  Mult: 0x4758C000 (hex)
  Sum: 0x4483E000 (hex)

- Hex encodings of all values
  Mult (FP32): 0x4758D467 (hex)
  Sum (FP32): 0x448400D5 (hex)
  Mult (FP16): 0x7AC6 (hex)
  Sum (FP16): 0x641F (hex)

- Accuracy

$$Sum\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.097\ \%$$

$$Mult\ \%\ Error = \frac{fp16 - fp32}{fp32} \times 100 = 0.0368\ \%$$

Conclusion:

In this lab, we successfully implemented functions for FP16 conversion, addition, and multiplication using bitwise operations. By working directly with the IEEE-754 fields, such as the sign, exponent, and mantissa fields, we gained a deeper understanding of floating-point representation and its arithmetic. Testing with various values confirmed the correct functionality of our code, with minimal precision differences due to the limitations of the 16-bit floating-point format. Overall, the lab helped us to demonstrate key concepts in low-level number handling as well as embedded computation.