ENB350 Real-Time Computer-Based Systems
Laboratory
Extended Precision Fixed-Point Arithmetic

Context

Sometimes embedded applications require arithmetic with precision beyond that offered by the word length and native instruction set of the microcontroller used. For example an embedded application on a 16 bit processor using graphics might need calculation of new co-ordinates from old ones and this may involve multiplication of 32 bit quantities on the 16 bit processor. Further, real numbers that have fractional parts may need to be represented without the use of a floating point processor. The extent to which a compiler will support arithmetic is also limited. Compilers have limits on the lengths of the data types they support and the operations permitted on them. If a compiler supports floating point operations on a system without a floating point co-processor, it will make use of computationally expensive library functions. Even if a 16 bit compiler permitted 32 bit double precision real arithmetic with such a library but what if you wanted higher precision or wanted the operations performed faster using fewer clock cycles? In such cases, the embedded systems programmer must select a suitable fixed point representation and develop a library of functions to support extended precision real arithmetic on that representation. Fixed precision arithmetic is actually just integer arithmetic. When the integer word length goes above the length that is supported on that processor, the programmer must write functions using algorithms to extend the word length of arithmetic operations – if a library is not already available from the vendor or other sources.

Learning Objectives:

After completing this laboratory, you should be able to

1. program embedded C applications with extended precision arithmetic
2. program embedded C applications with fixed point real numbers
3. implement a function to display fixed-point numbers in decimal format
4. implement a product function for extended precision fixed-point numbers

Reference Documentation:

Rabbit 4000 user manual (RC4000UM.pdf)

Dynamic C 10 user manual (DCPUM.pdf), section 12 Using Assembly Language.

Rabbit instructions manual (RabbitInstructions.pdf)

Preparation activity

Become familiar with program development under Dynamic C by going through Chapter 3: Quick Tutorial in the Dynamic C user manual.

Chapters 2, 3 and 5 of the reference book "Embedded Systems – Where C and Assembly Meet" by Daniel Lewis, Prentice Hall, or chapters 2 and 3 of the reference book "Fundamentals of Embedded Software with the ARM Cortex-M3," by Daniel Lewis, Pearson, have the background material on number systems and algorithms for fixed point extended precision arithmetic. You do not need to use either the processor or the software associated with these reference books. We will use the Rabbit 4000 and Dynamic C for this exercise. Number systems and the algorithm are also discussed in the lectures and available on PowerPoint slides on Blackboard.

Understand 2s complement notation and its use in binary addition and subtraction.

Represent +1 in 2s complement representation in 8 bits.

-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	1

Now put a binary point in the middle. This is equivalent to dividing by 2 to the power of 4, which is 16. So this fixed point number should represent 1/16.

-2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	0	0	0	•	0	0	0	1

If we consider the first 4 bits to be the whole (integral) part and it is in signed 2s complement form, what is it? If we consider the last 4 bits to be the fraction part and in unsigned form what is it? What is the whole part + fraction part equal to?

Now represent -1 as a 2s complement number in 8 bits. Repeat the above steps.

1	1	1	1	•	1	1	1	1

What is the whole part now? What is the fractional part? And what is the sum of the two? Note that we do not have a whole part equal to zero or the fraction part equal to 1/16 in this case. But their sum is the number being represented, that is -1/16. Why do we add the whole part and the fractional part of the representation? In 2s complement representation, to find the number represented by a bit pattern we multiply each bit with the corresponding weights and add them up. Do this for each of the patterns above and evaluate to get the number represented.

Thus the integer +1 in 8 bits can be treated like the number +1/16 in 4.4 fixed point form, and the number -1 in 8 bits can be treated like the number -1/16 in 4.4 fixed point form provided we are using 2s complement notation for the integer. But in the fixed point form, the whole part remains in 2s complement notation because it retains the most significant (deciding the sign) bit while the fractional part should be considered as an unsigned 4 bit number. Integer arithmetic on the 8 bit integers will produce correct results for the 4.4 fixed point form except when there is overflow or loss of precision in the operations.

Now represent the number +1 in this 4.4 fixed point notation. Then represent -0.75 in this notation. Perform the operation 1 – 0.75 as a 2s complement addition. Verify that you get the correct answer.

If we want to assign a value such as -0.25 as an 8.8 fixed point number to a 16 bit integer operand that we have declared in C, it is convenient to visualize it as a bit pattern or binary number or as a hexadecimal number by combining groups of 4 bits. Since the number to be represented (-0.25) is actually -1 + 0.75, the whole part will be -1 in 2s complement form of FF and the fractional part will be 0.75 which is the bit pattern shown below or C0. The number -0.25 is therefore 0xFFC0.

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1	1	0	0	0	0	0	0

We can also represent these numbers in C as data structures with fields for the whole part and the fractional part. With object oriented languages that allow new data types and operations to be defined, we could do that instead but any new operations will need to be implemented. The implementation of new operations will not be automatically supported by a compiler. If there exists

a data type 'long' of 32 bits and we have to do fixed 16.16 bit operations, we can convert from whole and fraction representations in a C data structure to a 'long' representation and then use arithmetic for the 'long' data type supported by the compiler. This will work for addition and subtraction but not multiplication. Besides, if we want to do 32.32 fixed point arithmetic on this system, we will need a 64 bit integer data type (say 'long64 or longlong') and arithmetic operations on it. This will usually not be available. In these cases, we need to have algorithms to extend the arithmetic operations to longer data types. The algorithms can only be implemented by making use of library functions that work on shorter data types and ultimately arithmetic with the processor registers and instructions.

Find out the data types and operations supported by Dynamic C on the Rabbit 4000.

In this exercise you will implement 16.16 fixed point arithmetic using the following C structure to represent such numbers.

```
typedef struct {
      int whole;
      unsigned int fraction;
} fp16;
```

We will need to write functions to create these structures, add, subtract and multiply fixed point numbers and also to display results in decimal human readable form. Assuming that we had these functions implemented the following test driver could be used to check them with different numbers entered conveniently as whole and fraction parts in hexadecimal.

```
main()
{
int fp16make(); // declaration for function to create fp16
int fp16sum();  // function to add
int fp16diff();  // function to subtract
int fp16prod();  // function to multiply
int fp16Todec(); // function to convert to decimal and display
// The above functions must also be defined and implemented with
// return types and arguments as explained in the task descriptions

// variables to be used for test operands, results
fp16 x,y,prod,*px, *py,*pprod, radd, *pradd, rdiff, *prdiff;
int a, c;    // whole parts of test operands
unsigned int b, d; // fraction parts of test operands

// hard coded test operand fields. Change, edit and recompile.
a = 0x0002;
b = (unsigned int) 0xC000; // a and b now represent 2.75
c = 0xFFFE;
d = (unsigned int) 0xC000; // c and d now represent -1.25

//pointers
px = &x;
py = &y;
pprod = &prod;
pradd = &radd;
prdiff = &rdiff;

// perform test operations to check
fp16make(a,b,px);
fp16make(c,d,py);
fp16prod(px,py,pprod);
fp16sum(px,py,pradd);
fp16diff(px,py,prdiff);
```

```
// print out the operands and results as hex numbers
printf("x = %4x.%4x \n",px->whole,px->fraction);
printf("y = %4x.%4x \n",py->whole,py->fraction);
printf("product = %4x.%4x \n",prod.whole,prod.fraction);

// print out the operands and results as decimal numbers
printf("IN DECIMAL FORM  \n");
printf("x = "); fp16Todecimal(px); printf("\n");
printf("y = "); fp16Todecimal(py); printf("\n");
printf("product = "); fp16Todecimal(pprod); printf("\n");
printf("sum = ");fp16Todecimal(pradd);  printf("\n");
printf("diff = ");fp16Todecimal(prdiff);  printf("\n");

exit(0);

}
```

Conversion to decimal

The conversion of fixed point numbers to decimal digits for displaying is a bit tricky.

We saw earlier that **whole.frac** when represented as a long in 2s complement form is actually whole + frac whether it is positive or negative, where **whole** is a signed 2s complement number and **frac** is unsigned. Thus, 2.75 has **whole** = 2 and **frac** = 0.75 (this is 0x0002.C000) and -1.25 has **whole** = -2 and **frac** = 0.75 (this is 0xFFFE.C000).

Separate the whole and fractional parts using bit masking, clearing and bit shifting.

We can first check if the number is positive or negative using the most significant bit of the whole part. Note the sign and change the whole part to its absolute value.

If the number is positive we can convert each part independently into decimal digits and then display them together with a decimal point in between.

If the number is negative we need to check if the faction part is 0 and if it is, then the whole part remains the same. If it is not, then **frac** = 1 – **frac** and the asbsolute value of whole should be decremented by 1. We can then convert each independently to decimal digits and display with a decimal point in between.

Displaying the whole part is easy using formatted write with printf and “%d” or “%ld” in the format string. This assumes support of a library function for the conversion. In general decimal digits can be pulled out one at a time from the bits representing the whole part to the bits representing the fraction part by dividing by the number 10.

The fractional part is an unsigned integer with an assumed scale factor. This can be displayed in hexadecimal form or as a bit pattern using formatted write. To display the decimal equivalent we need to obtain the decimal digits. This is achieved using repeated multiplication by 10. At each such step, a decimal digit from the fraction part will get transferred to the bits representing the whole part. For example, 0.25 (represented in binary as 0x00004000) when multiplied by 10 yields 2.5 (represented in binary as 0x00028000) and the integer part is 2 which is the first digit required. This portion can be bit shifted into an int and displayed as a decimal integer using formatted write as before. Repeated application of this will give 5 next and then 0. The process can be terminated after the number that remains in the fraction part becomes 0.

Laboratory Tasks: (Marks allocation A: 1.5 B: 0.75 C: 0.75 D: 0.75 E: 1.25)

Task A. (due in 2 weeks)

Write the following functions in Dynamic C (a) to create 16.16 signed numbers (in 2s complement form) given the whole part and the fraction part, and (b) to convert them to decimal form and display them on the STDIO window.

```
int fp16make(int a, unsigned int b, fp16 *px)

int fp16Todec(fp16 *px)
```

Test the functions and fill out the table below. You can write a main program that will call these functions and enter the numbers as statements or interactively.

Whole part	Fraction part	Number represented	Number displayed
2	0.5		
-2	0.5		
-3	0.125		

Task B. (due in 2 weeks)

Write the following function in Dynamic C to perform addition of 16.16 fixed point real numbers.

```
int fp16sum(fp16 *px, fp16 *py, fp16 *psum)
```

Hint:
Note that addition of long (32 bits) is supported by Dynamic C. You can copy from the structure to an equivalent long variable and use these variables to perform the addition, then put the result back into the structure.

Test the function and fill out columns 3 and 4 (expected and observed A+B) in the table below.

A	B	A+B expected	A+B observed	A-B expected	A-B observed
2.5	0.5				
-2.5	0.5				
$2^{15}-1$	2^{15}-1				
$2^{15}-1$	-2^{15}				

Task C. (due in 2 weeks)

Write the following function in Dynamic C to perform subtraction of 16.16 fixed point real numbers.

```
int fp16diff(fp16 *px, fp16 *py, fp16 *pdiff)
```

Hint:
Note that subtraction of long (32 bits) is supported by Dynamic C. You can copy from the structure to an equivalent long variable and use these variables to perform the addition, then put the result back into the structure.

Test the function and fill out columns 5 and 6 (expected and observed A-B) in the table above .

Task D. (due in 3 weeks)

Write the function below for multiplication of two 16 bit unsigned integers using Rabbit 4000 assembly language.

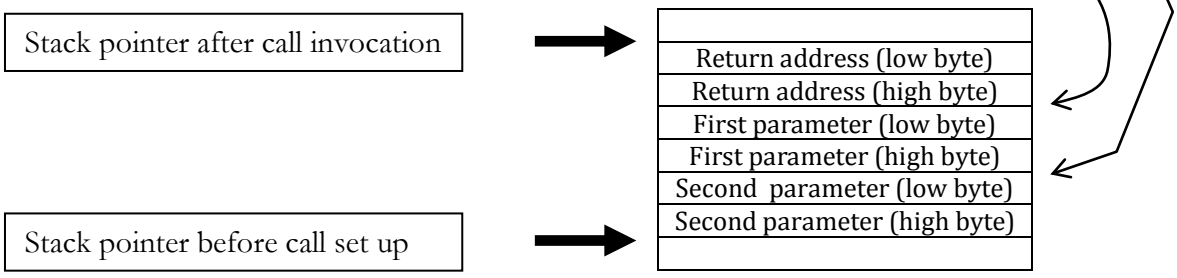
```
unsigned long prodUnsigned(unsigned int x, unsigned int y)
```

Hints:
Note that you will need to know how parameters are passed when C functions are called and the order in which they are stored in the stack. You can use the MUL instruction. Parameters will need to be in the appropriate registers and the result should be in the appropriate register(s) to be returned by the function.

If you try to write the multiplication routine in C rather than assembly and rely on the Dynamic C compiler to generate code for unsigned long (or int) multiplied with unsigned long (or int) to produce a long result correctly, it will not work. This is because the compiler does not generate the right code.
Example C code for an addition function with two int parameters and returning an int data type is given below. The function is declared in C but implemented in assembly - note the #asm and #endasm and the label result:: which is the same name as the function and is of global scope. Parameter x is stored at (sp+2) and parameter y at (sp+4) where sp is the stack poiner. When an int is returned, it should be placed in register pair HL. Where does the output of the add operation go? What is the convention when a long is returned? You will need to work with an unsigned multiplication instruction and look up the manual to find out which registers it uses.

```
int result(int x, int y)
#asm root
result::
        ld hl,(sp+4)
        ex de,hl
        ld hl,(sp+2)
        add hl,de
ret
#end asm
```

A visual representation of the stack and parameters for this example is given below:



Addresses decrease towards the top in the above chart. The address at the top arrow is sp and its contents are (sp) after the call invocation.

Task E. (due in 3 weeks)

Write the function below in Dynamic C to perform multiplication of 16.16 fixed point real numbers (signed numbers in 2s complement notation).

```
int fp16prod(fp16 *px, fp16 *py, fp16 *pprod)
```

Hints: This function will make use of the function from Task D and the algorithm for extended precision multiplication discussed in class (and in the reference books).

Multiplication of 32 bit integers

$$\boldsymbol{A} = \boldsymbol{A}_{31}\boldsymbol{A}_{30}.....\boldsymbol{A}_0 = 2^{16}\boldsymbol{A}_{\boldsymbol{H}} + \boldsymbol{A}_{\boldsymbol{L}}$$

$$\boldsymbol{B} = \boldsymbol{B}_{31}\boldsymbol{B}_{30}.....\boldsymbol{B}_0 = 2^{16}\boldsymbol{B}_{\boldsymbol{H}} + \boldsymbol{B}_{\boldsymbol{L}}$$

$$\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{A}_{\boldsymbol{H}}\boldsymbol{B}_{\boldsymbol{H}} + 2^{16}(\boldsymbol{A}_{\boldsymbol{H}}\boldsymbol{B}_{\boldsymbol{L}} + \boldsymbol{B}_{\boldsymbol{H}}\boldsymbol{A}_{\boldsymbol{L}}) + \boldsymbol{A}_{\boldsymbol{L}}\boldsymbol{B}_{\boldsymbol{L}}$$

If the 32 bit integer numbers are actually representing fixed point 16.16, the most significant 16 bits and least significant 16 bits may be discarded from the product to keep the product also in 16.16 format after proper scaling. The discarded bits are the overflow and the loss of precision – when keeping the length of representation unchanged. Thus only the middle 32 bits are retained. These are obtained from the terms $\boldsymbol{A}_{\boldsymbol{H}}\boldsymbol{B}_{\boldsymbol{L}}$ and $\boldsymbol{B}_{\boldsymbol{H}}\boldsymbol{A}_{\boldsymbol{L}}$.

For multiplication of two unsigned 32 bit numbers the above procedure can be used along with a routine to do 16 bit by 16 bit unsigned multiplication and return a 32 bit product – which is available as the routine written for Task D.

For multiplication of two signed 32 bit numbers returning a signed product, see the rest of the algorithm explained in section 3.3.3 of the reference book "Fundamentals of Embedded Software with the ARM Cortex-M3" by Daniel W. Lewis (Pearson). There is an example in section 3.3.4. Essentially, if the subscripts s and u stand for 'signed' and 'unsigned', respectively,

$$\boldsymbol{A}_{\boldsymbol{s}} \times \boldsymbol{B}_{\boldsymbol{s}} = \boldsymbol{A}_{\boldsymbol{u}}\boldsymbol{B}_{\boldsymbol{u}} - 2^{32}\boldsymbol{A}_{31}\boldsymbol{B}_{30..0} - 2^{32}\boldsymbol{B}_{31}\boldsymbol{A}_{30..0}$$

This is equivalent to subtracting $B_{30..0}$ if A is negative and $A_{30..0}$ if B is negative from the unsigned product to get the signed product.

After you have implemented the required function check it using test cases and debug if necessary.

Test the function and fill out the table below.

A	B	A*B expected	A*B observed
2.5	0.5		
-2.5	0.5		
-3.5	-2.5		
-1.5	0.0		