CMPE-250
Assembly and Embedded Programming
Spring 2021
Laboratory Exercise Eight
Multiprecision Arithmetic

This exercise investigates input, storage, addition, and output of numbers larger than a KL05 word. The objective of this exercise is to write subroutines to perform multi-word number I/O and addition. An assembly language program is created to test these subroutines, and it is executed on the KL05Z board.

Application

This assignment tests addition of 96-bit numbers. The user enters the hexadecimal representation of two 96-bit numbers, they are added, and the sum is output in hexadecimal representation.

Part I. Multiprecision Addition (March 15–19)

Prelab Work I. (March 15–19)

Following the specifications below and the notes you have taken in lecture, write the following subroutine for multiprecision number addition. No registers other than APSR should have changed values after return from the subroutine.

- AddIntMultiU: Adds the *n*-word unsigned number in memory starting at the address in R2 to the *n*-word unsigned number in memory starting at the address in R1, and stores the result to memory starting at the address in R0, where the value in R3 is *n*. The subroutine uses ADCS to add word by word, starting from the least significant words of the augend and addend and working to the most significant words. If the result is a valid *n*-word unsigned number, it returns with the APSR C bit clear as the return code for success; otherwise it returns with the APSR C bit set as the return code for unsigned *n*-word overflow.

Program Specification I.

Write a properly formatted and properly commented assembly language program that prompts the user for two 96-bit unsigned numbers, uses a provided library subroutine to input the two numbers, uses the subouroutine from prelab work to compute their sum, and uses a provided library subroutine to output the result. The program must perform the following sequence of operations and must meet the requirements that follow.

I.1. For each of two 96-bit unsigned numbers needed from the user, the program must prompt the user for the number and then use GetHexIntMulti from the provided Exercise08_Lib library to get the number from the user, as shown below where user input is indicated in bold. To make input easier for the user, right-align the prompts so that the two numbers start in the same column on the terminal screen.

Note: GetHexIntMulti needs the address of the 96-bit number in R0 and 3 in R1.

Input parameter:

R1: *n*, the number of words in the input number (unsigned word value)

Output parameters:

R0: *n*-word number input by user (unsigned word address)

C: input number status: 0 valid; 1 invalid (APSR bit flag)

```
 Enter first 96-bit hex number:  0x00000000000000000000000FFFFFFFF
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
```

I.2. If the user input is not a valid hex representation of a 96-bit number, it must let the user know that the input was invalid and use GetHexIntMulti again to get user input. Step I.2 repeats until valid input is obtained, as shown below. Again, right-align the invalid message with the prompts so that the next number input starts in the same column as the previous number.

```
 Enter first 96-bit hex number:  0x2zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
     Invalid number--try again:  0xvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
     Invalid number--try again:  0x00000000000000000000000000000023
Enter 96-bit hex number to add:  0x2zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
     Invalid number--try again:  0xvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
     Invalid number--try again:  0x00000000000000000000000000000023
```

I.3. It must use AddIntMulitU from prelab work to add the two 96-bit unsigned numbers.

I.4. It must use PutHexIntMulti from the provided Exercise08_Lib library to show the result to the user, as shown below. To make it easier for the user to read, right-align the output so that the numbers start in the same column on the terminal screen. If the result is not valid, OVERFLOW should be output instead of a number.

Note: PutHexIntMulti needs the address of the 96-bit number in R0 and 3 in R1.

Input parameters:

R0: *n*-word number to output (unsigned word address)

R1: *n*, the number of words in the input number (unsigned word value)

Output parameter:

(none)

```
 Enter first 96-bit hex number:  0x00000000000000000000000FFFFFFFF
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0x00000000000000000000000100000000
 Enter first 96-bit hex number:  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0xOVERFLOW
```

I.5. After displaying the result, the program should repeat from step I.1.

In addition, the program must meet these requirements.

- It must use the subroutine from prelab work for 96-bit number addition. The following is a summary of the parameter specifications for the subroutine.
  - AddIntMultiU

    Input parameters:

    R1: *n*-word augend for addition (unsigned word address)
    R2: *n*-word addend for addition (unsigned word address)
    R3: *n*, the number of words in addition source and result operands (unsigned word value)

    Output parameters:

    R0: *n*-word sum from addition (unsigned word address)
    C: addition operation status: 0 success; 1 overflow (APSR bit flag)
- It must use subroutines from Lab Exercise Six for string I/O.
- It must use subroutines from Lab Exercise Five for character I/O.
- It must have properly commented subroutines, including a comment header block that includes these components.
  - Description of functionality
  - List of subroutines called
  - List of registers
    - Input parameters
    - Output parameters
    - Modified registers (after return, not preserved by subroutine)

Lab Procedure I.

I.1. Create a new directory and Keil MDK-ARM project for Part I of this exercise.

I.2. Copy the Exercise08_Lib.lib file (provided on myCourses for this exercise) to your project directory and add it to your project.)

I.3. Write a properly commented and properly formatted KL05 assembly language program to implement 96-bit unsigned integer input, addition, and output, according to the preceding specification.

I.4. Add the following directives needed for the Exercise08_Lib library in the MyCode AREA of your program after the EXPORT Reset_Handler directive already in the program template. (The IMPORT directive for the Exercise08_Lib subroutines makes them available for your program to use, and the EXPORT directive for your subroutines make them available for the library to use.)

```
IMPORT GetHexIntMulti
IMPORT PutHexIntMulti
EXPORT GetStringSB
EXPORT PutNumHex
```

I.5. Build the program with Keil MDK-ARM to create a listing file and a linker map file.

I.6. Test your program. Some example input and output is shown below.

```
 Enter first 96-bit hex number:  0x00000000000000000000000000000000
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0x00000000000000000000000000000001
 Enter first 96-bit hex number:  0x00000000000000000000000FFFFFFFE
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0x00000000000000000000000FFFFFFFF
 Enter first 96-bit hex number:  0x00000000000000000000000FFFFFFFE
Enter 96-bit hex number to add:  0x00000000000000000000000000000002
                            Sum:  0x00000000000000000100000000000000
 Enter first 96-bit hex number:  0x0000000000000000A000000000000000
Enter 96-bit hex number to add:  0x00000000000000006000000000000000
                            Sum:  0x00000000000000010000000000000000
 Enter first 96-bit hex number:  0x00000000B0000000000000000000000000
Enter 96-bit hex number to add:  0x00000000500000000000000000000000
                            Sum:  0x00000001000000000000000000000000
 Enter first 96-bit hex number:  0xC0000000000000000000000000000000
Enter 96-bit hex number to add:  0x40000000000000000000000000000000
                            Sum:  0xOVERFLOW
 Enter first 96-bit hex number:  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0xOVERFLOW
 Enter first 96-bit hex number:  0x00000000000000000000000000000002z
     Invalid number--try again:  0xv0000000000000000000000000000000
     Invalid number--try again:  0x00000000000000000000000000000023
Enter 96-bit hex number to add:  0x0000000000000000000000000000000z
     Invalid number--try again:  0x3v0000000000000000000000000000000
     Invalid number--try again:  0x00000000000000000000000000000023
                            Sum:  0x00000000000000000000000000000046
 Enter first 96-bit hex number:  0x
```

I.7. Demonstrate step I.6 for your lab instructor.

I.8. Submit your source file, listing file, and map file to the myCourses assignments for this exercise.

Part II. Multiprecision Number I/O (March 22–26)

Prelab Work II. (March 22–26)

Following the specifications below and the notes you have taken in lecture, write the following subroutine for multiprecision number I/O. No registers other than APSR should have changed values after return from the subroutines.

- GetHexIntMulti: Gets an *n*-word unsigned number from the user typed in text hexadecimal representation, and stores it in binary in memory starting at the address in R0, where the value in R1 is *n*. The subroutine reads characters typed by the user until the enter key is pressed by calling the subroutine GetStringSB. It then converts the ASCII hexadecimal representation input by the user to binary, and it stores the binary value to memory at the address specified in R0. If the result is a valid *n*-word unsigned number, it returns with the APSR C bit clear; otherwise, it returns with the APSR C bit set. The table below provides an example for $n = 2$.
  Note 1: A string buffer is required for GetStringSB. Although allocating the string buffer on the stack is the preferred implementation, a global string buffer may be used.
  Note 2: Basic implementation requires $8n$ uppercase hex digits from the user.
  Hint: Use $8n + 1$ as the GetStringSB string buffer capacity.
  (See Extra Credit Opportunities section for extra credit implementation.)

User Input	*n*-Word Unsigned Number (*n* = 2)
1234567890ABCDEF	0x1234567890ABCEDF

- PutHexIntMulti: Outputs an *n*-word unsigned number, from memory starting at the address in R0, to the terminal in text hexadecimal representation using $8n$ hex digits, where the value in R1 is *n*. The table below provides an example for $n = 2$. (Hint: Use PutNumHex from Lab Exercise Seven.)

n-Word Unsigned Number (*n* = 2)	Terminal Output
0x1234567890ABCEDF	1234567890ABCDEF

Program Specification II.

Other than using prelab subroutines GetHexIntMulti and PutHexIntMulti instead of the Exercise08_Lib library subroutines, the program specification for Part II is the same as for Part I. The program must meet the requirements that follow.

- It must use subroutines from prelab work for 96-bit number I/O. The following is a summary of the parameter specifications for each of these subroutines.
  - GetHexIntMulti
    Input parameter:
    R1: *n*, the number of words in the input number (unsigned word value)
    Output parameters:
    R0: *n*-word number input by user (unsigned word address)
    C: input number status: 0 valid; 1 invalid (APSR bit flag)
  - PutHexIntMulti
    Input parameters:
    R0: *n*-word number to output (unsigned word address)
    R1: *n*, the number of words in the input number (unsigned word value)
    Output parameter:
    (none)
- It must use AddIntMultiU from Part I for 96-bit number addition.
- It must use subroutines from Lab Exercise Six for character I/O.
- It must use subroutines from Lab Exercise Five for string I/O.
- It must have properly commented subroutines, including a comment header block that includes these components.
  - Description of functionality
  - List of subroutines called
  - List of register parameters: input, output, and modified

Lab Procedure II.

II.1. Create a new directory and Keil MDK-ARM project for Part II of this exercise.

II.2. Generate a properly commented and properly formatted KL05 assembly language program to implement the 96-bit unsigned integer input, addition, and output, according to the preceding specification by doing the following.
- Copy the source code from the Part I program (but *do not add* the Exercise08_Lib library to your project).
- Delete the IMPORT statements for GetHexIntMulti and PutHexIntMulti.
- Add subrotines GetHexIntMulti and PutHexIntMulti from prelab work Part II.

II.3. Build the program with Keil MDK-ARM to create a listing file and a linker map file.

II.4. Test your program as in Part I. (See sample input and output in step I.6 on p. 4.)

II.5. Examine the KL05 memory map, and note the starting address in memory and size in bytes for these components to include in your documentation.
a. AddIntMultiU
b. GetHexIntMulti
c. main (which shows as Reset_Handler in the map)
d. PutHexIntMulti

II.6. Demonstrate steps II.4 and II.5 for your lab instructor. Following demonstration, capture the terminal screen output to include in your documentation.

II.7. Submit your source file, listing file, map file, and documentation to the myCourses assignments for this exercise.
Note: The source, listing, and map files are submitted to the same myCourses assignments as in Part I since there is only one Lab Exercise Eight myCourses assignment for each type of file.

Extra Credit Opportunities II. (20 points)

1. (15 points) Instead of requiring users to enter all 32 hex digits ($8n$) of the value of a 96-bit unsigned number ($n = 4$) using GetHexIntMulti, implement GetHexIntMulti to allow users to enter only the significant digits [1, $8n$] of the number, as shown in the program output below. (Note: as shown below, it is not necessary to align the least significant digits of the numbers for this input mode.)
2. (5 points) Allow the user to enter hex digits in either uppercase or lowercase, as shown below.

```
 Enter first 96-bit hex number:  0x0
Enter 96-bit hex number to add:  0x1
                            Sum:  0x00000000000000000000000000000001
 Enter first 96-bit hex number:  0xFFffFFfe
Enter 96-bit hex number to add:  0x1
                            Sum:  0x00000000000000000000000FFFFFFFF
 Enter first 96-bit hex number:  0xffFFffFE
Enter 96-bit hex number to add:  0x2
                            Sum:  0x00000000000000000000000100000000
 Enter first 96-bit hex number:  0xA000000000000000
Enter 96-bit hex number to add:  0x6000000000000000
                            Sum:  0x00000000000000010000000000000000
 Enter first 96-bit hex number:  0xb00000000000000000000000
Enter 96-bit hex number to add:  0x50000000000000000000000
                            Sum:  0x00000010000000000000000000000000
 Enter first 96-bit hex number:  0xC00000000000000000000000000000000
Enter 96-bit hex number to add:  0x40000000000000000000000000000000
                            Sum:  0xOVERFLOW
 Enter first 96-bit hex number:  0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Enter 96-bit hex number to add:  0x00000000000000000000000000000001
                            Sum:  0xOVERFLOW
 Enter first 96-bit hex number:  0x2z
      Invalid number--try again:  0xv00000000000000000000000000000000
      Invalid number--try again:  0x23
Enter 96-bit hex number to add:  0xz
      Invalid number--try again:  0x3v00000000000000000000000000000000
      Invalid number--try again:  0x23
                            Sum:  0x00000000000000000000000000000046
 Enter first 96-bit hex number:  0x
```

Baseline Metrics

The metrics in the table below can be used to compare your solution to a baseline solution. They are provided solely for your information and personal curiosity about the efficiency of your solution. There are no grading criteria associated with how your code compares with them.

Language	Routine	Instructions/LOC	Code Size (bytes)
Assembly	main program*	40	104
C	main	16	132
Assembly	AddIntMultiU	14	42
C	AddIntMultiU	13	88
Assembly	GetHexIntMulti	46	120
C	GetHexIntMulti	24	178
Assembly	GetHexIntMulti (Extra Credit)	88	190
C	GetHexIntMulti (Extra Credit)	36	242
Assembly	PutHexIntMulti	11	24
C	PutHexIntMulti	3	28

*Not including instructions from provided program template.

Documentation

Prepare a document (in Microsoft Word or PDF format) containing the following.

- The screen capture from procedure step II.6 (or step I.7 if only Part I was completed) along with an explanation of how the output shown in the screen capture verifies that the multiprecision input, addition, and output are correct.
- The starting addresses and size (in bytes) of the subroutines listed in procedure step II.5. (If only Part I was completed, answer based on the Part I map file.)

Submission

myCourses Assignments
Separate assignment for each item below • Documentation (including cover sheet) • Source code (.s) (both Parts I and II) • Listing (.lst) (both Parts I and II) • Map (.map) (both Parts I and II)

As specified in "Laboratory and Report Guidelines," late submissions are penalized per day late, where "day late" is calculated by rounding submitted time to the nearest greater or equal integer number of days late, (i.e., ceiling function).

Note: To receive grading credit for this lab exercise you must earn points for *both* the demonstration *and* the documentation.