Calling and Returning from Procedures; Multiple-Precision Integer Arithmetic

Required Materials:
- Your textbook, *Assembly Language for Intel-Based Computers* (6th edition)
- Removable or network device (Flash drive, memory card, MyMocsNet account mapped to a drive letter, etc.) for storage of your programs
- These instructions
- Intel-compatible, Windows-based personal computer (like the ones in EMCS 306) with text editor, MASM 6.15, and Microsoft Visual Studio or CodeView (if needed for debugging)

Preparation for Laboratory:

Read sections 5.4 through 5.7, pages 157-178 of your textbook, on stack operations and calling and returning from procedures. Also read the following discussion of multiple-precision integer arithmetic operations.

Discussion:

The original Intel 8086 instruction set allowed programmers to perform arithmetic operations such as addition and subtraction on 8- or 16-bit integer operands. The 386 and higher processors (IA-32 architecture) extended this capability to 32-bit operands. In some cases, however, we may want to perform computations on numbers larger than the machine can handle ... at least in a single instruction. Fortunately, Intel built in special arithmetic instructions to help us handle these cases.

You should recall learning about the ADD and SUB instructions in Chapter 4. These commands allow us to add or subtract 8-, 16-, or 32-bit numbers directly, producing a result of corresponding size and either setting or clearing the C (carry) flag depending on whether or not a carry occurred. (For subtraction, the C flag actually indicates that a *borrow* from the next more significant position is needed; it is set when a larger number is subtracted from a smaller one.) There are a related pair of instructions ADC and SBB which can be used when adding or subtracting numbers larger than 32 bits in a "piecewise" fashion. ADC (Add with Carry) produces the sum of the two named operands plus the value of the Carry flag. If C = 0 the result is the same as for ADD; if C = 1 the result is one greater than that generated by ADD. SBB (Subtract with Borrow) works analogously; it subtracts the source operand from the destination operand and then subtracts the value of the C flag from the result.

These instructions can be cascaded together, either as in-line code or in a loop, in order to let us add or subtract integers of any arbitrary size. The process is very much like using pencil and paper to add or subtract large decimal numbers: we perform the computation on the least significant digits, note the result and the carry or borrow, if any, then proceed to the next more significant position, where we do the same thing (accounting for the carry from or borrow into the previous position), and so on, until we process the most significant digit position and complete the task. When processing large binary numbers on the computer, we can use ADD or SUB to process the least significant 8, 16, or 32 bits of the numbers (since there is no carry into, or borrow required from, the least significant position). Alternatively, we could use the CLC instruction to clear the carry flag before beginning (so the answer won't be affected by the results of any previous computation), followed by ADC or SBB. Either way, after we compute the least significant part of the result, we then use ADC or SBB to compute each more significant piece of the result in turn, until we are finished. By using ADC or SBB to include the carries/borrows between partial results, we ensure that the answer will be computed correctly no matter how many individual computations we have to perform to obtain it.

Indirect addressing and program loops, which were also introduced in Chapter 4, are useful programming tools for processing large numbers stored in memory in byte, word, or doubleword-sized pieces. We can set up pointers to the least significant part of each operand and the result, and then adjust the pointers as we process each step of the computation. **It is important to keep in mind that on Intel microprocessor-based systems, multi-byte numbers are stored in memory least significant byte "first" (at the lowest-numbered memory address), a.k.a. "little-endian" order.** To get the correct answer, we must start by adding or subtracting the least significant portion of the operands and work toward the most significant portion of the operands – not the other way around.

Instructions:

Write an assembly language program consisting of a **main program** and **a subprogram** (a second, separate procedure) to be called from the main program. The purpose of the subprogram is to perform arithmetic (either addition or subtraction, but not both in response to the same call) on **unsigned 96-bit integers** stored in memory. Communication between the main program and the subprogram is to be done via CPU registers (you will need to choose four).

One CPU register must be used to pass the **memory offset of the first operand** to the subprogram. A second register must be used to pass the **offset of the second operand** to the subprogram. A third register must be used to pass the **offset of the location where the result of the operation is to be stored**. *The subprogram should use **only** the information passed from the main program in these three registers to locate the source and destination operands in memory; **do not refer to these variables by name anywhere within the subprogram**.* A fourth register is to be used as a "**control flag**" to specify the desired operation; in other words, to tell the subprogram whether to compute (operand1 + operand2) or (operand1 - operand2). If this register contains a nonzero value, the 80-bit numbers in memory are to be added; if it contains zero, the numbers are to be subtracted.

Use assembler directives within your data segment to declare storage for and initialize the 96-bit test variables given below. **The main program should call the subprogram twice**: once to add the numbers 48C2E4C3552677F535D9607Bh plus 115724954F1792C6ED40C392h, and once to perform the subtraction 10792731164E95A224C2F0ABh minus 086234E52324642E3C7F1265h. Assemble and link your program and test it (using the debugger as necessary). At the conclusion of your program run, **observe** the *CPU register contents* and the values of the *variables of interest* (**the two source and one destination operands from *both* calculations - a total of *six* 96-bit values**) in memory. **Capture screen shots from within the debugger, or use Irvine's DumpRegs and DumpMem procedures, to show these final register and memory contents.**

To Hand In: *(due no later than 3:00 p.m. Tuesday, October 7)*

Turn in a printed copy of your program listing **(.LST file)** and the screen-captured *register values and all source and destination operands* showing the final results of your program. *Staple the program listings and results together neatly* and submit them by the date and time indicated above. ***Make sure your name, the date, and the course number and section are shown in comments at the top of your program listing,*** and of course make sure your listing is thoroughly documented to explain its operation as discussed in the *"Program Style and Grading Guidelines"* handout. (Pay particular attention to proper documentation of the called procedure!)