LAB2, Mixing C and Assembly. Performance Issues

Goal

Learn to use C and assembly in the same program. Become aware of performance issues.

Given files

Assuming you copied the lab files according to the lab setup instruction you can find the files you need in `~/TDDI11/lab2`. Your modifications goes to `llmultiply.asm` and `main.c`.

Assignment 1, Assembly implementation

Processors used a small (embedded) part in a mass produced product have to be very cheap. Therefore, they are quite rudimentary. For example, they seldom have floating point arithmetic capability, or they may be limited to 8-bit integer operations. Software must compensate for the limited hardware capability. We will look at an example in which we want to multiply two numbers larger than the hardware supports natively. On an 8-bit processor this could involve multiplication of two 16-bit integers. As out target already can do that we will look at multiplying two 64-bit operands on out target machine that natively supports only 32-bit operands.

Our compiler supports 64-bit integers with a data type called `long long int`. Since the registers of the Intel processor are only 32-bits wide, how does the compiler generate code to implement `a*b` when `a` and `b` are `long long int`'s?

Write a function in *assembly* that has the following C signature:

```
void llmultiply(unsigned long long int a,
                unsigned long long int b,
                unsigned char *result);
```

The function multiplies the two 64-bit parameters `a` and `b`. The result of the multiplication is a 128-bit number. It has to be copied to the array of 16 bytes that is pointed to by result. Note that the x86 machines are little-endian, meaning that the least significant byte of a multi-byte number is placed at the lower address, and the most significant byte of a multi-byte number is placed at the higher address.

Test your function with at least the given test cases. The given test cases are tailored to generate carries in all possible steps of the calculation.

Assignment 2, C implementation

Implement the same function, but this time in C. Here you must make sure to use appropriate data type for the multiplication and addition in order to be able to store the entire result in a sufficiently large type.

Compile it **without** any optimization and verify that it gives correct results.
Compile it **with** optimization turned on and verify that it gives correct results.

Assignment 3, Optimization (voluntary due to testing problems)

Call `llmultiply` from a C program in which you test the function. Put the invocation of the function in a loop in order to invoke it many times. Obtain the contents of the CPU cycle register before entering and after exiting the loop. Print the difference of the two values of the CPU cycle register. Make sure to read the section “Obtaining CPU Clock Cycles correctly” below.

Test both your C-version (optimized and non-optimized) and your assembler version in the same way (the same number of iterations). Which version is most effective? How big improvement does compiler optimization give?

Assignment 4, Comparison of the solutions

By looking at the generated assembly code it is possible to get an estimation of how efficient the code is. Compare your assembly solution to your optimized and non-optimized C code (so you have three solutions to compare). Are you able to beat the compiler (less instructions)? How much improvement does compiler optimization give?

To see the assembler code generated by the assembler you can utilize the debugger. The debugger can disassemble a memory region or function. The disassembly can be saved to file in the following way. Start by creating a file with the debugger command to disassemble:

```
echo disass llmultiply > disassemble.cmd
```

The above command creates the file `disassemble.cmd` with one line of text, indicating to the debugger that the function `llmultiply` should be disassembled. You may need to substitute the function name for the function name you’ve chosen. The next step is to start the debugger, tell it to load the compiled code, and to look in `disassemble.cmd` for commands to execute:

```
i386-elf-gdb main.o -x disassemble.cmd -batch > asm.out
```

In this case the output is redirected to the file asm.out, but you can use any file name (you will have to do this for both the optimized and normal C solution). Then you can open `asm.out` in `emacs` to look at it or use the command line again to count the number of lines in the file:

```
wc -l asm.out
```

Finally, you can of course start the debugger with only `main.o` as argument, and then run the disassemble command interactively if you want, but only the above procedure save the output to a file for later analysis.

Multiplication theory

Consider that each 64-bit operand can be split in two 32-bit parts, one contain the high order bits and the second contain the low order bits.

$$a = a_h * 2^{32} + a_l$$
$$b = b_h * 2^{32} + b_l$$

If we expand the operands in the parts as above an perform the multiplication with the expanded expressions according to normal mathematical rules we get:

$$a * b = (a_h * 2^{32} + a_l) * (b_h * 2^{32} + b_l)$$

$$\begin{aligned} &= a_h * b_h * 2^{64} + \\ &\quad (a_h * b_l + a_l * b_h) * 2^{32} + \\ &\quad a_l * b_l \end{aligned}$$

All multiplications that appear in the expression are now 32-bit multiplications and therefore they can be implemented on the Intel x86 processor. Your only concern is to handle the additions and carries that may appear after the additions. Note that each 32-bit multiplication yields a 64-bit result, but you can only add 32-bits in each addition. A graphical view of the procedure (think carefully of where carries can occur):

	$(a_h * b_h)_h$	$(a_h * b_h)_l$		
		$(a_h * b_l)_h$	$(a_h * b_l)_l$	
		$(a_l * b_h)_h$	$(a_l * b_h)_l$	
+			$(a_l * b_l)_h$	$(a_l * b_l)_l$
	result 15.. 12	result 11.. 8	result 7 .. 4	result 3 .. 0

How many carries can you add to the high order result of a multiplication without risk of generating a new carry? Can the high order doubleword of the result overflow when adding carries from previous steps?

Test cases

We provide the following test cases. You should of course add your own. All digits are hexadecimal.

```
    111122223333 *     555566667777 =         5B061D958BF0ECA7C0481B5
3456FEDCAAAA1000 * EDBA00112233FF01 = 309A912AF7188C57E62072DD409A1000
FFFFEEEEDDDDCCCC * BBBBAAAA99998888 = BBBB9E2692C5DDDCC28F7531048D2C60
FFFFFFFFFFFFFFFF * FFFFFFFFFFFFFFFF = FFFFFFFFFFFFFFFE0000000000000001
00000001FFFFFFFF * 00000001FFFFFFFF = 0000000000000003FFFFFFFC00000001
FFFEFFFFFFFFFFFF * FFFF0001FFFFFFFF = FFFE0002FFFDFFFE0001FFFE00000001
```

C function call interface

When writing the assembly code that cooperate with C-code you have to follow the compilers idea of how to pass parameters to a function. The parameters are passed on the stack. The stack grows from large addresses to small ones. The last parameter to a function is pushed first on the stack. Hence, the last parameter will be at a larger address than the first parameter. The stack pointer points to the top of the stack and not to the first free location! That means that pushing a value on the stack first decrements the stack pointer and then writes the value.

To understand better, look at the following snapshot of a stack frame just after entering the function:

```
byte 0 of return address                  | 0x3fffffe8 <-- stack top (esp)
byte 1 of return address                  | 0x3fffffe9
byte 2 of return address                  | 0x3fffffea
byte 3 of return address                  | 0x3fffffeb
;; The first parameter (a) start here.
;;  Notice the 32-bit little endianess:
;;  The least significant byte come first (byte 0)
;;  The most significant byte come last (byte 3)
byte 0 of a, byte 0 of al                 | 0x3fffffec
byte 1 of a, byte 1 of al                 | 0x3fffffed
byte 2 of a, byte 2 of al                 | 0x3fffffee
byte 3 of a, byte 3 of al                 | 0x3fffffef
byte 4 of a, byte 0 of ah                 | 0x3ffffff0
byte 5 of a, byte 1 of ah                 | 0x3ffffff1
byte 6 of a, byte 2 of ah                 | 0x3ffffff2
byte 7 of a, byte 3 of ah                 | 0x3ffffff3
;; The second parameter (b) start here.
byte 0 of b, byte 0 of bl                 | 0x3ffffff4
byte 1 of b, byte 1 of bl                 | 0x3ffffff5
byte 2 of b, byte 2 of bl                 | 0x3ffffff6
byte 3 of b, byte 3 of bl                 | 0x3ffffff7
byte 4 of b, byte 0 of bh                 | 0x3ffffff8
byte 5 of b, byte 1 of bh                 | 0x3ffffff9
byte 6 of b, byte 2 of bh                 | 0x3ffffffa
byte 7 of b, byte 3 of bh                 | 0x3ffffffb
;; The third parameter (c) start here.
;;  Notice that only the address to the array is passed.
byte 0 of result array address            | 0x3ffffffc
byte 1 of result array address            | 0x3ffffffd
byte 2 of result array address            | 0x3ffffffe
byte 3 of result array address            | 0x3fffffff <-- stack bottom
```

Typically a function has the following prologue:

```
push ebp      ;; save the value of ebp register on the stack
mov  ebp, esp ;; save the address of the stack frame
              ;; (the value of esp after entering the function)
```

The reason for the prologue is to get a fix base pointer for convenient access to the function parameters (the stack pointer `esp` may be changed in the function to store local variables). It also make printing of a stack trace easy, which is a important debug feature. After the prologue `ebp` will contain the value `0x3ffffe4`, i.e. the value of the stack pointer before pushing `ebp`, that is `0x3ffffe8`, minus the four locations occupied by `ebp`. The stack will now look like this:

```
byte 0 of previous stack frame (ebp)      | 0x3fffffe4 <-- top (ebp, esp)
byte 1 of previous stack frame (ebp)      | 0x3fffffe5
byte 2 of previous stack frame (ebp)      | 0x3fffffe6
byte 3 of previous stack frame (ebp)      | 0x3fffffe7
byte 0 of return address                  | 0x3fffffe8 <-- previous top
byte 1 of return address                  | 0x3fffffe9
byte 2 of return address                  | 0x3fffffea
byte 3 of return address                  | 0x3fffffeb
byte 0 of a, byte 0 of a_l                | 0x3fffffec <-- parameter 1
...
...
```

To illustrate how you can get that said convenient access to the parameters, let's add the values of b_h and a_l as an example (this addition is of course irrelevant for the assignment). You will find b_h 20 bytes from the address in `ebp` (count in the picture). To get the value of b_h to register `eax` we write:

```
mov eax, [ebp + 20]
```

Then we fetch a_l, but as `eax` now is occupies we have to use another destination register:

```
mov ebx, [ebp + 8]
```

And to add the two (with the result in `eax`) we do:

```
add eax, ebx
```

To make your code more readable it is good to use symbolic names for the offsets:

```
mov eax, [ebp + BH_OFFSET]
mov ebx, [ebp + AL_OFFSET]
add eax, ebx
```

As an exercise, think of how to load the address of the first byte of the result array in `ebx`.

Since we push `ebp` in the prologue inside the function we should restore the previous value before we return. We also have to make sure the stack pointer point to the same address it had when we entered the function, in order to use the correct return address. This is the function epilogue, that is to pop `ebp` from the stack before leaving the function.

```
mov esp, ebp   ;; perhaps needed...
pop ebp
ret
```

Obtaining CPU Clock Cycles correctly

Unfortunately `CPU_Clock_Cycles` does not seem to work correctly unless QEMU runs on a real Intel CPU. Since solaris run on Sparc and our available Linux server runs on AMD you must find your own solution as to how to test your solution.

Compile time optimization

In order to compile *without any* optimization, pass the `-O0` switch (minus capital `O` -- as in "Origami" -- followed by zero) to the gcc compiler.

In order to compile with *full optimization,* pass `-O3` to the compiler.

You can do it by modifying the `CFLAGS` in your `Makefile`.

Deliverables

The assembly language and C language implementations of your `llmultiply` function. Demo the application for the lab assistant. Present your conclusions with respect to the three run times.

Reference information and documentation

NASM referencehttp://www.nasm.us/doc/

NASM tutorialhttp://www.grack.com/downloads/djgpp/nasm/djgppnasm.txt

Instruction sethttp://courses.ece.uiuc.edu/ece390/books/labmanual/inst-ref-general.html

x86 Assemblyhttp://www.arl.wustl.edu/~lockwood/class/cs306/books/artofasm/toc.html