**LABORATORY WORK 06** (due to end of the class)

**INTRODUCTION**

This assignment will teach you how to use the translate your C code into Assembly code.

To solve the problems, you must know:

* C programming language
* Memory image
* **GCC** compiler, preferably

**RULES**

* Remember that: You are not allowed to search for help online!
* Any time you receive help from your teachers, you should acknowledge that in your code with a comment starting with the string “assistance from”.
* You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.
* Reconsideration requests must be made within one week of our release of grades for the assignment.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why a few weeks later, if needed.

**INTRO**

**Computers** execute machine code, sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks. A compiler generates machine code through a series of stages, based on the rules of the programming language, the instruction set of the target machine, and the conventions followed by the operating system. **The GCC** **C compiler** generates its output in the form of **assembly code**, a **textual representation of the machine code** giving the individual instructions in the program. GCC then invokes both an assembler and a linker to generate the executable machine code from the assembly code. In this , we will take a close look at machine code and its human-readable representation as assembly code.

Suppose we write a C program as two files p1.c and p2.c. We can then compile this code using a Unix command line:

$gcc -Og -o p p1.c p2.c

The command-line option **-Og** instructs the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C code. Invoking higher levels of optimization can generate code that is so heavily transformed that the relationship between the generated machine code and the original source code is difficult to understand. We will therefore use -Og optimization as a learning tool and then see what happens as we increase the level of optimization.

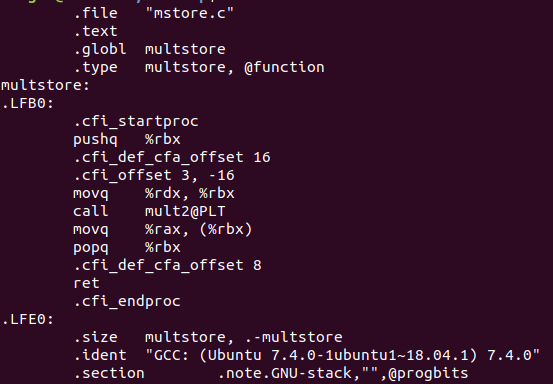
The GCC command invokes an entire sequence of programs to turn the source code into executable code.

* **First**, the C preprocessor expands the source code to include any files specified with *#include* commands and to expand any macros, specified with *#define* declarations.
* **Second**, the compiler generates assembly code versions of the two source files having names **p1.s** and **p2.s**.
* Next, the assembler converts the assembly code into binary object-code files **p1.o** and **p2.o**. Object code is one form of machine code—it contains binary representations of all of the instructions, but the addresses of global values are not yet filled in.
* Finally, the linker merges these two object-code files along with code implementing library functions (e.g., printf) and generates the final executable code file **p** (as specified by the command-line directive -o p). Executable code is the second form of machine code we will consider—it is the exact form of code that is executed by the processor.

To see the assembly code generated by the C compiler, we can use the -S option on the command line:

$ gcc -Og -S mstore.c

This will cause **GCC** to run the compiler, generating an assembly file mstore.s, and go no further.



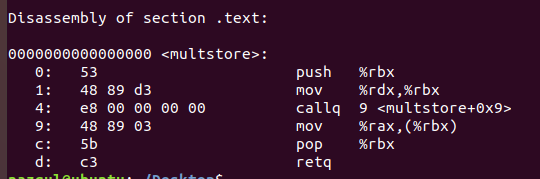
If we use the -c command-line option, **GCC** will both compile and assemble the code:

$ gcc -Og -c mstore.c

This will generate an object-code file mstore.o that is in binary format and hence cannot be viewed directly.

To inspect the contents of machine-code files, a class of programs known as ***disassemblers*** can be invaluable.

$objdump -d mstore.o



And finally, we can generate executable program and disassemble the file prog:

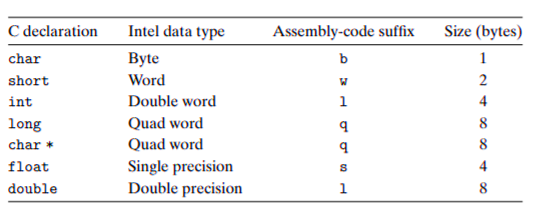
$ gcc -Og -o prog main.c mstore.c

$ objdump -d prog

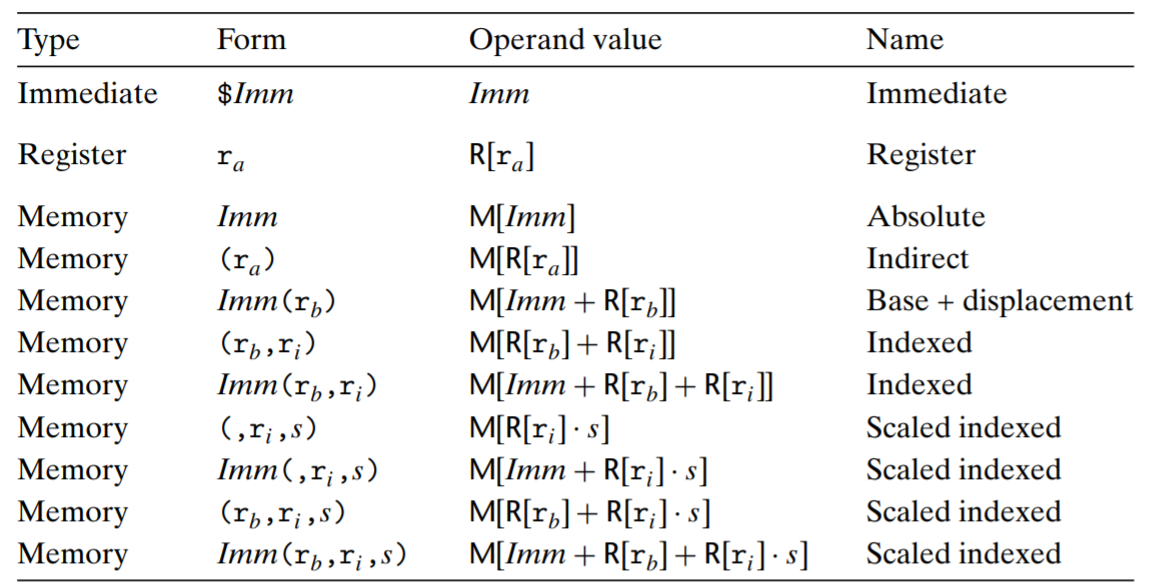
This code is almost identical to that generated by the disassembly of mstore.c. One important difference is that the **addresses listed** along the left are different— the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has filled in the address that the **callq** instruction should use in calling the function mult2 (line 4 of the disassembly).

**DATA FORMATS in AT&T**

Due to its origins as a **16-bit architecture** that expanded into a **32-bit one**, Intel uses the term “**word**” to refer to a **16-bit data type**. Based on this, they refer to **32-bit** quantities as “**double** **words**,” and **64-bit** quantities as “**quad** **words**.”



**ADDRESING MODES**

****

**INSTRUCTIONS**

1. Run Unix OS or similar (for tasks 4)
2. Compile the C source codes with -Og -S command
3. Examine your program output

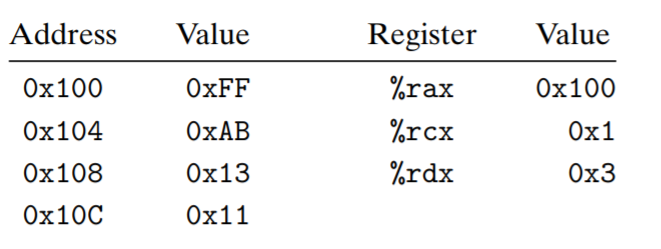
Your assignment is to complete each task, listed below:

**TASKS**

**All code for your tasks should be well documented (commented) and created as a function.**

**Absence of comments [-10%]**

**Task 1 [15%]:** Fill in the following table showing the values for the indicated operands:

****

|  |  |
| --- | --- |
| Operand | Value |
| %rax | 0x100 |
| %rcx | 0x1 |
| 0x100 | 0xFF |
| $0x108 | 0x108 |
| (%rax) | 0xFF |
| 4(%rax) | 0xAB |
| 9(%rax, %rdx) | 0x11 |
| 260(%rcx, %rdx) | 0x13 |
| 0xFC(, %rcx, 4) | 0xFF |
| (%rax, %rdx, 4) | 0x11 |

**Task 2 [15%]:** Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line:

|  |  |
| --- | --- |
| Operand | Comment |
| movb $0xF, (%ebx) | %ebx is not a base register |
| movl %rax, (%rsp) | Can’t take 4 bytes from %rax |
| movw (%rax),4(%rsp) | Can’t transfer from mem to mem |
| movb %al,%sl | %sl doesn’t exist |
| movq %rax,$0x123 | $0x123 is a number, we can’t transfer to it |
| movl %eax,%dx | Can’t transfer from bigger to smaller |
| movb %si, 8(%rbp) | Can’t take 1 byte from %si |

**Task 3 [20%]:** Suppose register **%rbx** holds value **p** and **%rdx** holds value **q**. Fill in the table below with formulas indicating the value that will be stored in register %rax for each of the given assembly-code instructions:

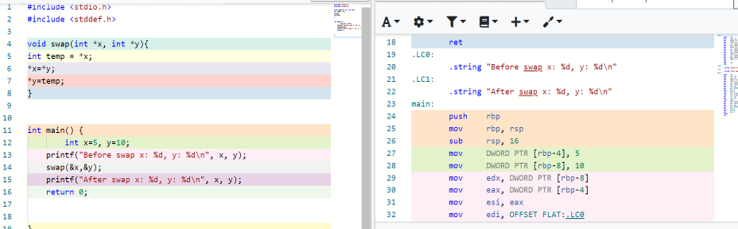
***%rbx = p, %rdx=q***

|  |  |
| --- | --- |
| Instruction | Result |
| leaq 9(%rdx), %rax | 9+q |
| leaq (%rdx,%rbx), %rax | 1+q+p |
| leaq (%rdx,%rbx,3), %rax | 1+q+p\*3 |
| leaq 2(%rbx,%rbx,7), %rax | 2+p+p\*7 |
| leaq 0xE(,%rdx,3), %rax | 0xE+q\*3 |
| leaq 6(%rbx,%rdx,7), %rax | 6+p+q\*7 |

**Task 4[50%]:** Unzip the archive lab6\_example.c and disassemble it.

$ gcc -Og -S lab6\_example.c

Put the output here (screen shot ):



Now, find the **main:** section in assembly code and line by line explain what’s happening. ( Ignore codes like ***.cfi\_startproc*** )

Example,

**subq $24, %rsp** - subtracted 24 from current stack pointer (allocated the memory for the stack)

main:

        push    rbp - pushes its argument onto the stack

        mov     rbp, rsp – transfers value from rsp to rbp

        sub     rsp, 16 – subtracts from the top of the stack rsp - 16

        mov     DWORD PTR [rbp-4], 5 – transfers 5 to dword ptr

        mov     DWORD PTR [rbp-8], 10 – transfers 10 to dword ptr

        mov     edx, DWORD PTR [rbp-8] – transfers value 10 from dword ptr to edx

        mov     eax, DWORD PTR [rbp-4] – transfers value 5 from dword ptr to eax

        mov     esi, eax – transfers value 5 from eax to esi

        mov     edi, OFFSET FLAT:.LC0

        mov     eax, 0 – transfer value 0 to eax

        call    printf

        lea     rdx, [rbp-8] – transfers value 10 from ptr to rdx

        lea     rax, [rbp-4] – transfers value 5 from ptr to rax

        mov     rsi, rdx – transfers value from rdx to rsi

        mov     rdi, rax – transfers value from rax to rdi

        call    swap

        mov     edx, DWORD PTR [rbp-8] – transfers value from dword ptr to edx

        mov     eax, DWORD PTR [rbp-4] – transfers value from dword ptr to eax

        mov     esi, eax – transfers value from eax to esi

        mov     edi, OFFSET FLAT:.LC1

        mov     eax, 0 – transfers value 0 to eax

        call    printf

        mov     eax, 0 //return 0 = quit the program

        leave

        ret