

# Practical Work: Assembly

Master CSMI  
Compilation & Performance  
Bérenger Bramas

October 2, 2024

## 1 Summary

In this work, you will program in assembly language (ASM).

## 2 Ressources

- x86 Assembly Language Reference Manual: <https://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf>
- x86 and amd64 instruction reference: <https://www.felixcloutier.com/x86/>
- Registers: [https://en.wikibooks.org/wiki/X86\\_Assembly/X86\\_Architecture](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)
- CPUID: <https://www.amd.com/system/files/TechDocs/25481.pdf>

### 2.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
# If you use SSH, use:
# git clone git@git.unistra.fr:bbramas/csmi-tp-2024.git --branch=TP1 csmi-tp1
# With https
git clone https://git.unistra.fr/bbramas/csmi-tp-2024.git --branch=TP1 csmi-tp1
# Go in the newly created directory
cd csmi-tp1
```

### 2.2 Add your repository as remote

You will push on your own repository:

```
# Rename my remote
git remote rename origin old-origin
# Add your own remote
# If you use an SSH key:
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2024.git
# If you use https:
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2024.git
# Push the current branch and active the tracking
git push -u origin TP1
```

### 2.3 Compilation

To compile, we use CMake:

```
cd TP1
mkdir build
```

```

cd build
cmake ..
make # Will make all
make something # Will build only something
VERBOSE=1 make # Will show the commands used to compile (including the flags)

```

By default CMake will not use any optimization flag, and looking at the output of `VERBOSE=1 make` will show you that there is `-Ox`. To enable optimization you have to specify to `cmake` to generate a make file with the correct flags. To so, you can use `ccmake`. (in the build directory, note that `ccmake` should be installed) and edit `DCMAKE_BUILD_TYPE`, or directly set the variable with `cmake.. -DCMAKE_BUILD_TYPE = Release` in the build directory.

### 3 Reminder about registers

We have seen that modern CPUs have registers. If we leave aside the aspects related to performance, registers are used by the instructions as input/output (some instructions can also have main memory as input/output, but usually the output must be a register). In X86-64, the 64 bits registers are: `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rbp`, `%rsp`, and `%r8-r15` (from 8 to 15). These registers are also used to pass parameters when calling a function (if the data to pass are less or equal than 64 bits). However, a convention is used to know which registers must be saved by the caller or the callee. In fact, a function might put values in all the registers, then call another function, and thus the question is asked to know which registers can be safely overwritten by the called function, and which registers must be saved and restored. Registers `%rax`, `%rcx`, `%rdx`, `%rdi`, `%rsi`, `%rsp`, and `%r8-r11` must be saved by the caller. Therefore, the callee can erase their content and use these registers. Registers `%rbx`, `%rbp`, and `%r12-r15` must be saved by the callee before being used. The register `%rsp` is used as stack pointer (to know where is the top of the stack) and thus should not be modified.

The six first parameters of a function call are passed using `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` (additional parameters or those which excess 64 bits are passed using the stack).

### 4 Passing a real number as parameter

Generate the assembly of the code in `code_real.cpp` and find-out how a real number is passed by parameter to the function `just_add` and how the number is returned. To do so, try the three methods:

- GCC and AS:

```

g++ -S -fverbose-asm -g -O2 code_real.cpp -o code_real.s
as -alhd code_real.s > code_real.lst

```

- GCC only

```

g++ -g -O -Wa,-aslh code_real.cpp > code_real.txt

```

- Online tool <https://godbolt.org/>, remember to disable Intel and enable demangle.

### 5 Debug an assembly code

In the `debugme.cpp` file, you will find a kernel that has a bug. As shown in the main, the objective of this kernel is to do:  $param0 + param1 * param2 - param3$ . But in the code there are two bugs. I recall in the code that the parameters are in the following registers: (val1 rdi) (val2 rsi) (val3 rdx) (val4 rcx). The results should be in rax. As a reminder, `leaq (a,b,c)` does  $a + b * c$ . `c` default value is 1, and can only be 1, 2, 4 or 8. If needed, use godbolt to get a possible solution, in this case you could need to play with the optimization flags.

## 6 Update an existing function: return the power of the number given in parameter

In the *power.cpp* file, you will find two kernels, both add 1 to a number and return it, but one version is for integers and the other for real numbers. Update these functions to return the power of the number given in parameter. If needed, use godbolt to get a possible solution, in this case you could need to play with the optimization flags.

## 7 Update an existing function: return the power5 of the number given in parameter

In the *power5.cpp* file, you will find one kernel that adds 1 to a number and return it. Update this function to return the power 5 of the number given in parameter. We want to use only two registers (rdi and rax).

## 8 Sum of all the parameters

In *sum.cpp* add your own assembly code to sum two parameters together (*sum2\_asm*) and seven parameters together (*sum7\_asm*). Remember that the first six parameters are passed in registers and the next ones in the stack.

## 9 Dot product of integers

In *dot.cpp*, add your own code to create a vector product of long int. To compare two registers, you can use the *cmp* instruction follow by *je* or *jne* for "jump if equal" and "jump if not equal", respectively. A register can easily be set to 0 using a *xor* between a register and itself:

```
xor %rax, %rax; // set rax = 0
```

You will need to jump to different execution paths:

```
i_am_a_label:
cmp %rax, %r8; // I compare %rax %r8
je i_am_a_label; // If they are equal, then jump to i_am_a_label
```

You can use godbolt to help you, but I am expecting a hand written code.

## 10 Know more about your CPU (CPUID)

CPUs support special instructions to provide information about themselves or the current hardware. For instance, such instructions can be used to know if a CPU supports a given features or instruction extension, or even to know the size of the caches, etc. To do so, specific registers have to be used to store the information query. Then, the CPUID instruction can be called and will fill other registers with the answer.

```
"movl    $X, %eax;\n" // Query part 1
"movl    $Y, %ecx;\n" // Query part 2
"cpuid;\n" // Ask the CPU
// Answers are now in %eax, %ebx, %ecx, %edx
// where a bit set to 1 at a given position will mean "yes"
```

Implement such a function to get the information from the CPUID instruction. As you will see, 0x00000001 is passed to EAX before calling CPUID. Then we use the value from EDX at bits/features: HTT/28, MMX/23, SSE/25 and SSE2/26. Remark: you can notice that we now work with *int* instead of *long int*, and thus the registers have now different prefix, and the instructions can be post-fixed with "l".