Practical Work: Assembly

Master CSMI
Compilation & Performance
Bérenger Bramas

October 7, 2020

1 Summary

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

2 Ressources

- x86 Assembly LanguageReference 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`
- CPUID: `https://www.amd.com/system/files/TechDocs/25481.pdf`

3 Practical work organization (always the same)

In the practical work, you will obtain the code from my repository and push it to your repository. Therefore, you will have to clone one branch per session and push it to your own repository. It is mandatory that you **commit and push** frequently (after each question and at the end of the session, at least) such that I can easily look at what you have coded at the end of the session, how you did progress (and potentially compare it with the latest version you will have).

It is required that you filled the report.md file to let me know what you did.

In the rest of the document, we consider you have a repository named *cnp-tp-2020* on *git.unistra.fr* that is private but that I can access in read.

3.1 Get the practical work

Consider you are in your project directory do the following:

```
# Clone my repo
git clone https://git.unistra.fr/bbramas/csmi-tp-2020.git --branch=TP1 csmi-tp1
# If you use SSH replace [USER] and use:
# git clone git@git.unistra.fr:[USER]/csmi-tp-2020.git --branch=TP1 csmi-tp1
# Go in the newly created directory
cd csmi-tp1
```

3.2 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
git remote add origin https://git.unistra.fr/[YOU LOGIN HERE]/cnp-tp-2020.git
# If you use an SSH key:
```

```
# git remote add origin git@git.unistra.fr:[YOU LOGIN HERE]/cnp-tp-2020.git
# Push the current branch and active the tracking
git push -u origin TP1
```

3.3 During the session and while you work on the project

After each question or important modification push the current changes:

```
# No matter where you are in the project directory
git commit -a -m "I did something"
git push
```

3.4 When you are done

You have fully finished your work (at most D+14 H-2):

```
git commit -a -m "I did something"
git push
```

3.5 Important!

Do not forget that you have to fill the report.md file and commit it. Remember to commit regularly to keep track of your work and let me see a history of it if I need it. Do not share any code with someone else, as I am here to answer all questions and support all of you. Remember that you have questions to answer in the moodle before the end of the session and that you must push your branch at the end of the session too. Additional credits can be obtained if you make some modifications after the session to improve your solution. Changes can be made until two weeks after the session minus two hours, ie you must push before the beginning of the n+2 practical work. Do not remove code from the test functions as I use them to evaluate your code. Therefore, if you need you can add extra functions for your own testing/debugging. If some of them are showing interesting things about your code, simply leave a comment in the code and the report.md.

3.6 Compilation

To compile, we use CMake:

```
cd Code
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)
```

4 Reminder about registers

We have seen that modern CPUs have registers. If we leave aside the aspects related to perfomance, registers are used by the instructions as input/ouput (some instructions can also have main memory as input/output, but usually the output must be a register). In X86-64, the 64 bits regisers 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).

5 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 -alhnd 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.

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 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.

8 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
```

9 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".