ELEC-H-473 Microprocessor Architectures
SIMD labs, support document

2016–2017

1 Objectives

This document explains what are the minimum requirements for the code you develop during the SIMD labs of the Microprocessor Architectures practical sessions.

Also, an example of inline assembly code for `gcc`(codeblocks) is provided.

The remaining parts of this document are a set of random pieces of information you might need during the labs.

2 Lab 1: threshold on image

This lab is about applying a threshold to a gray[1] image.

We expect you to demonstrate you are able to:

- open a file in C/C++ (and verify no error occurred)
- allocate dynamically memory for a buffer (and verify the memory space has been allocated)
- copy a (section of the) file to the buffer
- process it with pure C code (*i.e.* architecture independent code)
- process it with inline assembly code (gcc or visual studio style) demonstrating the use of SIMD instructions
- measure accurately the time spend for processing for both codes, conclude
- write the processed data back to a file
- display the result.

[1] byte coded 256 shades, because 50 is not enough

3 Lab 2: morphological image filtering

This lab is about morphological image filtering using a min/max application on a 3×3 neighbourhood.

We expect you to demonstrate (again) you are able to:

- open a file in C/C++ (and verify no error occurred)
- allocate dynamically memory for a buffer (and verify the memory space has been allocated)
- copy a (section of the) file to the buffer
- process it with pure C code (*i.e.* architecture independent code)
- process it with inline assembly code (gcc or visual studio style) demonstrating the use of SIMD instructions
- process the corner cases accordingly
- measure accurately the time spend for processing for both codes, conclude
- write the processed data back to a file
- display the result.

4 Lab 3: multi-threading

This lab is about using multi-threading to improve further the processing time for the exercises of the previous labs.

We expect you to demonstrate (again) you are able to:

- open a file in C/C++ (and verify no error occurred)
- allocate dynamically memory for a buffer (and verify the memory space has been allocated)
- copy a (section of the) file to the buffer
- create multiple threads
- divide wisely the data, the goal is to process them using the code from the previous labs
- process the corner cases accordingly
- measure accurately the time spend for processing for both codes, conclude
- write the processed data back to a file
- display the result and verify correctness

5 Inline assembly for GCC

This is an example of code:

```
1   __asm__(
2         "mov %2, %%rsi\n"
3         "mov %3, %%rcx;\n"
4    "l1:\n"
5         "movdqu (%%rsi),%%xmm7\n"
6         "add $16,%%rsi\n"
7         "sub $16,%%rcx\n"
8    "jnz l1\n"
9         :"=m"(input),"=m" (output) //outputs
10        :"m" (input), "m" (length),"m" (output): //inputs
11         "rsi", "rcx", "xmm7" //clobbers
12         );
```

This example works[2] in the following conditions:

- you compile it with options to produce 64bit code (`-m64` option of GCC)
- you compile it for a processor featuring SSE instructions (select yours in "Project>Build options>CPU architecture tuning"/ `-march=` option)

To get it working on the computers of the lab which are running a 32bit OS, you have to apply those changes:

- rename 64bit registers (name starting with "R") to 32 bits registers (name starting with "E"), read https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture for more precise explanations.
- replace 64bit instructions (name ending with "Q") with 32bit instructions (name ending with "L")
- add the `-m32` option to your build options to make sure 32bit code will be produced by the compiler, note that GCC can infer the suffix of the instruction ("B", "W", "L", "Q") from the name of the registers involved (`mov %2,%%rsi` will infer a `movq` instruction in assembly output). If you can't find the `-m32` option, you are running a 32bit GCC already.

[2] To avoid any misunderstanding: that code should compile and run without producing any segfault, it includes a SSE instruction to check if the correct options are set for your compiler. `input` should be a valid and accessible memory location, `length` is expected to be a multiple of 16

Thus a working 32bit code becomes:

```
__asm__(
        "mov %[in], %%esi\n"
        "mov %[l], %%ecx;\n"
   "l1:\n"
        "movdqu (%%esi),%%xmm7\n"
        "add $16,%%esi\n"
        "sub $16,%%ecx\n"
   "jnz l1\n"
        :"=m"(input),"=m" (output) //outputs
        :[in]"m" (input), [l]"m" (length),[out]"m" (output): //inputs
         "esi", "ecx", "xmm7" //clobbers
         );
```

5.1 Quick explanation about the syntax

The previous example uses the extended asm syntax of GCC. It uses the AT&T syntax.

Because you don't want to imagine that your closed source compiler is doing around your code to outsmart it[3] because it's behaving strangely, GCC relies on the concept "The user knows what she/he does"[4].

It means, you have to constrain GCC so there is no misunderstand on the line.

An explanation about constraints: https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s6

Outputs Any modified variable[5] must be mentioned there. This ensure GCC does not uses an "unexpectedly" modified variable later in your C code.

"=m" means you modify some memory location. Other possibilities: "=r" for registers, "=g" equivalent to "=rim" for generic.

When in doubt, use "=g" to let the compiler use the best option depending on your code.

Inputs The syntax do NOT allow you to mention directly variables in your asm code[6], except when they are global[7]. If you have an output which is also an input, you can use a matching(digit) constraint.

You can use numbering (see 64bit example) or [labels] for more readability (see 32bit example).

[3] the compiler, your code is obviously perfect because you are an awesome programmer
[4] Remember that Clu fights for the user, especially Flynn
[5] any variable from your C code you modify in the asm code and which should be used later in the C code
[6] That's life, deal with it!
[7] The name will then be resolved during the link step

Clobbers are registers affected by your code. You tell your compiler that the content of these registers might have changed because of your code. The compiler will take measures ensuring the code produced saves and restores correctly those registers.

The syntax is explained in the official documentation of GCC:
https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

Also this page will help you: https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

6 List of instructions

Assuming your processor is in the list:
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
Oh, that's just a 2198page pdf, CTRL+F will be your friend.

Because one is not enough and you may have an AMD CPU:
http://support.amd.com/TechDocs/26568.pdf
http://support.amd.com/TechDocs/24594.pdf

You might like this one too http://support.amd.com/TechDocs/25112.PDF

Also, for quick reference: http://x86.renejeschke.de/

7 Different kinds of ASM directives with GCC

Read the[8] manual:
https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C

8 Intel vs AT&T syntax

Read https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3

Note that you can change from AT&T to intel representation using `.intel_syntax noprefix`. You have to revert the change at the end of your asm code using `.att_syntax noprefix`, else compilation will fail.

Anyway, you can use that syntax to write code similar to the code mentioned in the slides.

[8] you are imagining a word there, or maybe not

Example (compiling and running OK on an assistant's computer[9]):

```
__asm__(
     ".intel_syntax noprefix;\n" //switch to intel syntax
     "mov esi,%[in];\n" //numbering or labels are possible
     "mov ecx,%[l];\n"
"l11:\n"
     "movdqu xmm7,[esi]\n"
     "add esi,16\n"
     "sub ecx,16\n"
"jnz l11\n"
 ".att_syntax noprefix;\n" //revert to AT&T syntax
     :"=g"(input),"=g" (output) //outputs
     :[in]"g" (input), [l]"g" (length),[out]"g" (output): //inputs
      "esi", "ecx", "xmm7"); //clobbers
```

As you are switching representations, you might get `Error: junk '(%xxx)' after expression`. It's due to some additional code added by GCC to ensure everything is alright. If you impose a "m" or "r" constraint, GCC have to respect it and will add some code to respect your constraint. The specified representation (intel here) will be respected, which cannot be assembled later by GCC[10].

Note the "g" constraint will solve the issue 99%[11] of the time[12].

9 Common errors

9.1 Unknown register name 'xmm-'

`error: unknown register name 'xmm7' in 'asm'`

You have to tell your compiler to produce code for a processor having those registers. Else generic x86 code will be generated (for compatibility purposes) excluding those registers.

The key option in GCC to specify a target architecture is "`-march=`". Because you are writing inline assembly code for a more or less specific architecture, you have to specify which architecture you are using.

Solution: update "Project>Build options"

[9] For some reasons, they are haunted by ugly bugs and dangerous trolls. Some orange magic incantations were not necessary to get rid of them today for that snippet of code, so you know. The full Moon was last week by the way.

[10] Hence creating an wonderful error

[11] Citation needed. Honestly, this solved the only error I got because of an incorrectly specified constraint.

[12] To the 1%: it happens, even to the best

9.2 Segmentation faults

- pointer accessing unallocated memory, getting out of a page allocated by the OS, getting NULL value
- unaligned accesses when instructions requires aligned accesses (movdqa instruction for instance)
- reading/writing after the last element of an array[13]
- any segmentation fault

will cause a segmentation fault, which will lead to the MCP ans Sark derrrezing[14] your program without any afterthought. Be happy your operating system is only killing programs.

9.3 Error: junk '(%xxx)' after expression

```
Error:  junk '(%xxx)' after expression
```

It's due to some additional code added by GCC to ensure everything is alright. If you impose a "m" or "r" constraint, GCC have to respect it and will add some code to respect your constraint. The specified asm representation (intel if you switch to intel instead of AT&T) will be respected, which leads to the error mentioned.

10 Assembly code corresponding to some C code

If you need to get the assembly code corresponding to some C code, you can get it using the disassembly window of your IDE, the tools of gcc (objdump...) or this website: https://gcc.godbolt.org/

11 Expected results for Lab 1

Because I just want to go back home on this Friday evening and I'm tired, some references to films just showed up in the document. There is no bonus point for identifying all of them.

12 Installing GCC for codeblocks

If you are using any recent flavour of windows operating system[15], you can use this installer to install gcc/g++ on your computer:

https://sourceforge.net/projects/mingw/files/Installer/

[13] This leads to buffer overflow exploits.
[14] I mean, your OS
[15] Don't be ashamed, we all make bad choices at some point in life.

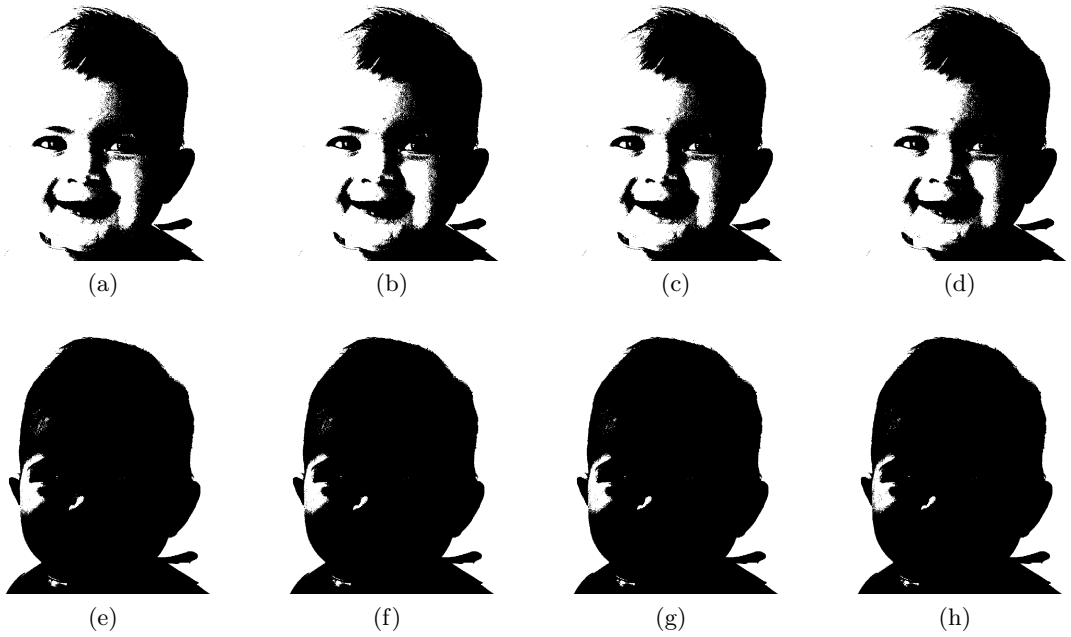

Figure 1: (a)< 63; (b)< 64; (c)< 65; (d)< 66; (e)< 127; (f)< 128; (g)< 129; (h)< 130

13 Random thoughts

There exist an `aligned` attribute which can be used to align variables in memory. Then aligned accesses can be performed with a higher bandwidth than unaligned accesses.

The proper use of intrinsics can lead to more portable code and very good performances.

When processing arrays of elements using batches. Make sure the last batch only process elements of the array and leaves the variables stored in memory after the "official" end of the array unaffected.

14 Additional section for superstitious reasons

PS: it's way past midnight on a Friday night and I'm not very proud of that document written too quickly for any use in a teaching situation. It's mostly undocumented and poorly written.

I really want you to learn meaningful things in these practical sessions so I'm putting elements together in this document for you during my free time. It will really help you if you make the effort to understand what happens in your program. The best piece of advice I can give you to succeed is to read the build log of your project.

Hopefully it's the last time I have to do that for this course[16].
Happy programming!

No computers nor keyboards were harmed to produce this document during the 60th hour.

`END OF LINE`

[16] And because there are not enough footnotes in this document yet, I use this one to apologise for all the hairy trolls I included in this document and the unfortunate English mistakes that remains.