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

v1.0.0

1 Introduction

This document explains information you might need during the SIMD labs of the Microprocessor Architectures practical sessions. Also, an example of inline assembly code for gcc(codeblocks) is provided.

2 Inline assembly for GCC

Here is an example of code using regular non-simd instructions (you can try it yourself and check what that code does) :

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    long size = 15;
    char * buffer = (char*)malloc(size*sizeof(char));

    // -- Print initial data:
    printf("%p, %d\n",buffer, size);
    for (int i=0;i<15;i++) printf("%d ",buffer[i]);
    printf("\n");

     __asm__(
        "mov %[out], %%rcx \n"
     "loop: \n"
        "mov %[size], (%%rcx) \n"
        "add $1, %%rcx \n"
        "sub $1, %[size] \n"
     "jnz loop \n"
     : [size] "+X" (size) //outputs
     : [out]"X"(buffer) //inputs
     : "rcx" //clobbers
     );

    // -- Print result:
    printf("%p, %d\n",buffer, size);
    for (int i=0;i<15;i++) printf("%d ",buffer[i]);
    printf("\n");

    return EXIT_SUCCESS;
}
```

Here is another example using SIMD instructions (you can try it yourself and check what that code does) :

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    long length = 16;
    char * inbuffer = (char*)malloc(length*sizeof(char));
    for (int i=0;i<16;i++) inbuffer[i] = 42;

    char * outbuffer = (char*)malloc(length*sizeof(char));

    // -- Print initial data:
    printf("%p, %d\n",outbuffer, length);
    for (int i=0;i<16;i++) printf("%d ",outbuffer[i]);
    printf("\n");

    __asm__(
        "mov %[in], %%rsi\n"
        "mov %[out], %%rax\n"
        "mov %[l], %%rcx;\n"
        "movdqu (%%rsi), %%xmm7\n"
        "movdqu %%xmm7, (%%rax)\n"
        "add $16, %%rsi\n"
        "sub $16, %%rcx\n"
        ://outputs
        :[in]"m" (inbuffer), [out]"m" (outbuffer), [l]"m" (length) //inputs
        : "rax", "rsi", "rcx", "xmm7" //clobbers
    );

    // -- Print result:
    printf("%p, %d\n",outbuffer, length);
    for (int i=0;i<16;i++) printf("%d ",outbuffer[i]);
    printf("\n");

    return EXIT_SUCCESS;
}
```

This example works 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 Windows computers running a 32bit compiler, 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/X86Assembly/X86_Architecture` for more 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.

2.1 Quick explanation about the syntax

The previous example uses the extended asm syntax of GCC: the **AT&T** syntax. It works with more constrains than Intel's to avoid any misinterpretation by the compiler.

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

The first thing to understand is that the variables you use will be copied from the context of your code to the context of the assembly function. So any constraints that you put affect the behavior of the copy between the C part of the code and the assembly part.

2.1.1 Outputs

Variables listed here are the ones that will be modified by the assembly code. Any modified variable, will not have its modification taken into account except if listed here. You can use numbering or `[labels]` for more readability.

Constraints is this part contains 2 parts, first the mode "=" or "+", and then the memory location:

- "=" means Write Only constraint. The variable is open for writing but not read at the beginning, which means that if you do not modify it in the asm but still return it's value, you could get random values in your returned variable;
- "+" means Read & Write constraint. The original variable is first copied to the memory location and then you can modify it. Which means any returned variable will have a "correct" value even if not modified in the asm code;

The memory location is the type of memory used to store the copy of your variable during the time of the execution of the asm code:

- "m" means in the data memory;
- "r" means in a register;
- "g" or "X" lets the compiler choose for you;

The type of memory location in important since it affect which kind of instruction you can use with a given variable. The memory location fixes where the variable is copied and thus which addressing mode is possible for this variable.

2.1.2 Inputs

Variables listed here are the ones that are going to be read in your asm code. If you have an output which is also an input, you can list it here too. You can use numbering or `[labels]` for more readability.

In this case you only have to choose the memory location of your variable, namely:

- "m" means in the data memory;
- "r" means in a register;
- "g" or "X" lets the compiler choose for you;

Same remarks apply concerning memory location as the outputs.

Remark (Array pointers)**.** *Note that the value of a pointer being an address, and since you only modify the content located at the address, it is not considered as output but as* ***input only****.*

2.1.3 Clobbers

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`

3 List of instructions

For **quick reference**: `https://www.felixcloutier.com/x86/`

Or in more details for Intel (Assuming your processor is in the list):
`https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`

Or AMD CPU:
`http://support.amd.com/TechDocs/26568.pdf`
`http://support.amd.com/TechDocs/24594.pdf`

4 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.

Example:

```
__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
```

5 Common errors

5.1 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: open “Project>Build options”, then in the tab "Compiler Flags", check the box corresponding to you architecture under "CPU architecture tuning". If you’re not sure which to choose, pick "Intel i7 CPU".

5.2 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