# Individual Project 1, Fall 2020
# CS5513, Computer Architecture
# x86 Assembly Language and Tools

## Purpose

This assignment will refresh your knowledge of x86 assembly language and reinforce your knowledge of the C calling convention, and familiarize you with a couple of key tools used in the analysis of programs in their binary form (e.g., gdb). The assignment will also refresh your memory regarding writing, compiling, and running C code on Linux.

## Due Date

This assignment is due on October 5th, 2020, Monday, at 11:59pm.

## Reference Systems

Our reference systems are CS department's fox servers (fox01.cs.utsa.edu to fox06.cs.utsa.edu). You should all have accounts on those servers. If you don't know your login information, try your abcid as user name and your banner id (probably with '@') as password. We will be grading on fox servers, so please make sure you code compiles on fox servers.

## x86 Assembly Language Programming

This assignment will be completed on your Ubuntu installation.
To exercise your x86 skills, do the following:

1. Consider the following C code that calls function dot_product() (this C code is available in the Blackboard):

```
#include <stdio.h>

#define LOOP_COUNT 10000000
#define VSIZE 200

float v1[VSIZE], v2[VSIZE];
float dot_product(float *, float *, int length);

int main(int argc, char *argv[]) {
 float result;
 int i;

 /* initialize the two vectors */
 for (i = 0; i < VSIZE; i++) {
     v1[i] = (float) 2;
     v2[i] = (float) 3;
 }

 /* call dot_product some number of times for timing purposes */
 for (i = 0; i < LOOP_COUNT; i++)
     result = dot_product(v1, v2, VSIZE);

 printf ("result is %f\n", result);
 return 0;
}
```

2. For this assignment you will create an assembly language file containing a function called

`dot_product()`. The function `dot_product()` computes the dot product of the two vectors (the first two parameters to function `dot_product()` ) and returns the result. Because we are writing assembly language, we can hand-tune our code to make it as efficient as possible. Consequently, we should use the MMX/SSE extensions which provide "vector" instructions. For example, there is an instruction (`mulps`) that will multiply two vectors consisting of four floating-point values. You will also need the horizontal add instruction, `haddps`.

3. To get started, you should write a C version of `dot_product()` and get the assembly code. The first step is to make sure our C code works properly. Here is a the compilation :

   ```
   gcc -m32 -march=corei7 -o a.out main.c dot_product.c
   ```

   Remember, the option -m32 tells gcc to generate code for the x86 (i.e., 32-bit code which we will always use in this class). The -march=corei7 option tells the compiler we are using a machine that has the SSE3 extensions (unless you still have a Pentium, you will have these extensions).

   Here is run of the program with the time command. You can check that the answer is correct.

   ```
   time ./a.out
   results is 1200.000000

   real    0m6.585s
   user    0m6.532s
   sys     0m0.012s
   ```

   Notice this code took about 6 seconds to run.

4. The next step is to get the assembly code so we can modify it by hand. The appropriate command is:

   ```
   gcc -m32 -c -S dot_product.c
   ```

   You can use the code that the C compiler generated (in dot_product.s ) as a starting point for your "optimized" code. That is, you should modify this assembly code to use the appropriate vector instructions. Here are some links to short tutorials about the vector instructions:

   http://sci.tuomastonteri.fi/programming/sse
   http://www.tommesani.com/index.php/simd/46-sse-arithmetic.html

   There is much information on the web about MMX and SSE. Remember: Google search is your friend (well, most of the time). You can also read "Intel 64 and IA-32 Architectures Software Developer's Manual" Volume 1 Chapter 10 and Volume 2 SSE instruction references, for more detailed explanation.

5. To run and compile your program using the assembly code file issue the following command:

   ```
   gcc -m32 -march=corei7 -o a.out main.c dot_product.s
   ```

6. Here is a sample run:

   ```
   time ./a.out
   result is 1200.000000

   0m1.269s
   0m1.248s
   0m0.008s
   ```

   Notice how much faster the "vector" version ran.

## Hints

1. You will need to use SSE instructions such as `movaps`, `mulps`, and `addps`. For better performance, you can also use `haddps`.

2. You can use AVX instructions instead of SSE instructions. However mixing SSE and AVX has a performance penalty. Therefore, if you choose to use AVX, you need to use only AVX instructions for computation.

3. GCC built-in SSE functions are probably the easiest way to generate SSE code. However it won't provide the best performance. If you have trouble manually coding in assembly, you can use these built-in SSE functions to generate blue-print assembly codes. Manually optimization is still required for better performance. You can learn more about the GCC built-in functions at,

   https://stackoverflow.blog/2020/07/08/improving-performance-with-simd-intrinsics-in-three-use-cases/
   https://gcc.gnu.org/onlinedocs/gcc-4.9.2/gcc/X86-Built-in-Functions.html

## Grading

This assignment will be graded based on the correctness and the performance of your code. You code should have a performance faster than the unoptimized version. No performance improvement will be considered as not finishing the assignment.

## Items to Submit

1. Submit your "optimized" assembly code dot_product.s.
2. It is **mandatory** that you use the file names specified to ease the task of grading. Throughout the semester, you will be given file names for assignments submitted using the Blackboard Website. **Failed to follow these instructions will result in 0 points.**
3. I will give bonus points to the three fastest submissions.