Systems and Biomedical Engineering Dept,
Bioelectronic Systems (Winter 2018)
Instructors: Eman Ibrahim, MSc Student
Meena AbdelMaseeh, PhD

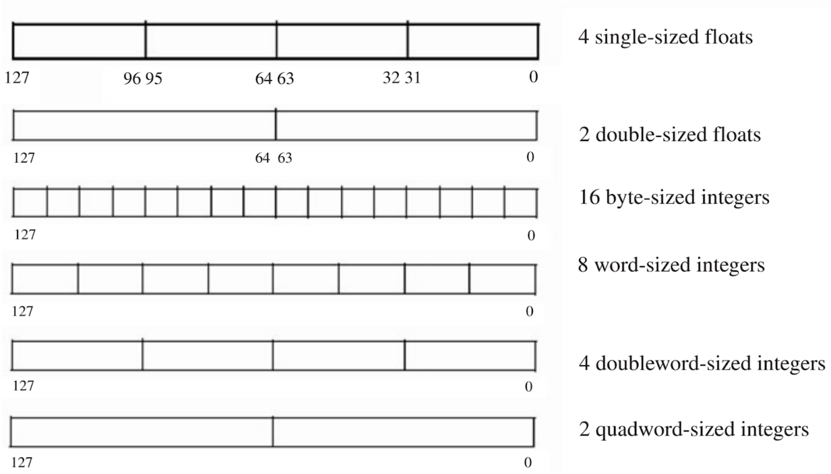
Course Project:
Milestone # 1

Vector Parallelism

Overview

Now more than ever, there is a need to analyze extremely large datasets, simulate very complicated systems, and run artificial intelligence algorithms in real time. Writing high performance software is therefore becoming a key skill for software engineer. This milestone introduces you to **vector parallelism**. A form of parallelism in which the same instruction is applied to chunks of data simultaneously.

SSE, Streaming Single Instructions Multiple Data (SIMD) Extension, was first introduced by Intel to extend the x86 ISA to process a small vector of operands in parallel. The extension was later adopted by AMD. SSE instructions operate on values stored in the XMM registers (XMM0 to XMM7). Each register is 128-bits wide.



The above diagram shows that an XMM register can hold four packed single-precision floating-point numbers (4x32 bits) or two packed double-precision floating-point numbers (2x64 bits). It can also work with integers and bytes, but in this milestone we are only interested in floating point arithmetics.

One of the easiest techniques to realize vector parallelism is through the use of **intrinsics**. They are functions, in a higher level language, say C++, that directly map to a sequence of assembly instructions, They allow the developer to easily and efficiently utilize processor-specific architectural enhancements.

Milestone Description

Linear algebra is widely abundant and heavily relied-on in graphics, scientific computations, and machine learning. One of the main uses of vector parallelism is to optimize linear algebra procedures, which typically involve one form or another of matrix multiplications. In this milestone, you are given an unoptimized C code for matrix multiplication in the code snippet below.

```
#include <xmmintrin.h> //Header for intrinsic functions
#include <stdio.h>
#include <time.h>
int main()
{
   // Variables definition and initialization
   int MAX_DIM = 100;
   float   a[MAX_DIM][MAX_DIM] __attribute__ ((aligned(16)));
   float   b[MAX_DIM][MAX_DIM] __attribute__ ((aligned(16)));
   float   c[MAX_DIM][MAX_DIM] __attribute__ ((aligned(16)));
   float   d[MAX_DIM][MAX_DIM] __attribute__ ((aligned(16)));
   for (int i = 0; i < MAX_DIM; ++i)
   {
       for (int j = 0; j < MAX_DIM; ++j)
       {
            a[i][j] = 1; // Arbitrary initializations - Replace to test your multiplication
            b[i][j] = 2; // Arbitrary initializations - Replace to test your multiplication
            c[i][j] = 0; // Necessary Initialization - Don't change
            d[i][j] = 0; // Necessary Initialization - Don't change
       }
   }
   // Unoptimized Matrix Multiplication
   clock_t Time1 = clock();
    for (int i = 0; i < MAX_DIM; ++i)
    {
         for (int j = 0; j < MAX_DIM; ++j)
         {
             for (int k = 0; k < MAX_DIM; k++)
             {
                      c[i][j] += a[i][k] * b[k][j];
             }
       }
   }
   clock_t Time2 = clock();

   clock_t Time3 = clock();
     // YOUR CODE HERE
   clock_t Time4 = clock();
   // Calculate and print execution times
   double TotalTimeLoop = ((double) Time2 - (double) Time1) / CLOCKS_PER_SEC;
   double TotalTimeSIMD = ((double) Time4 - (double) Time3) / CLOCKS_PER_SEC;
   printf(" Time taken by loop is %.7f \n", TotalTimeLoop);
   printf(" Time taken by SIMD optimized code is %.7f \n", TotalTimeSIMD);
   return 0;
}
```

Milestone Requirements

1. Copy the code snippet to a ".c" file.
2. You should be able to compile the code using **gcc -O0 Input_filename.c -o OutputFileName.o** and run it using **./OutputFileName.o**
3. Replace **"YOUR CODE HERE"** with a code of your own that implements matrix multiplication using SSE function intrinsics.
4. A two-page report that includes:
    a. the code you wrote,
    b. the performance gain you managed to achieve by vector parallelization, if any,
    c. the performance gain for different dimensions `int MAX_DIM = 100`,
    d. a comparison of the performance gain obtained for the same code by changing the type of operands to double precision and integers.
    e. a list of intrinsic functions you chose to use,
    f. and a list of problems that you think using vector parallelization can be useful for.