COMP1001

COMPUTER SYSTEMS

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk
Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

School of Computing
(University of Plymouth)

Outline

- Different ways of writing assembly code
- Using intrinsic functions in C/C++
- Writing C/C++ programs using Intel SSE intrinsics
- Writing C/C++ programs using Intel AVX intrinsics

Different ways of writing assembly

1. **Writing an entire function in assembly** (we did that in our previous examples)
2. **Using inline assembly in C/C++**
3. **Using intrinsic functions in C/C++**
   - highly recommended - much easier and safer
   - All the compilers support intrinsic functions
   - An intrinsic function is equivalent to an assembly instruction
   - **Mixes the good things of C++** (development time, portability, maintainability etc) **with the good things of assembly** (execution time)

- C and C++ are the most easily combined languages with assembly code

Different ways of writing assembly

Using intrinsic functions in C/C++

- **Main advantages**
  - Classes, if conditions, loops and functions are very easy to implement
  - Portability to almost all x86 architectures
  - Compatibility with different compilers
- **Main disadvantages**
  - Not all assembly instructions have intrinsic function equivalents
  - Unskilled use of intrinsic functions can make the code less efficient than simple C++ code

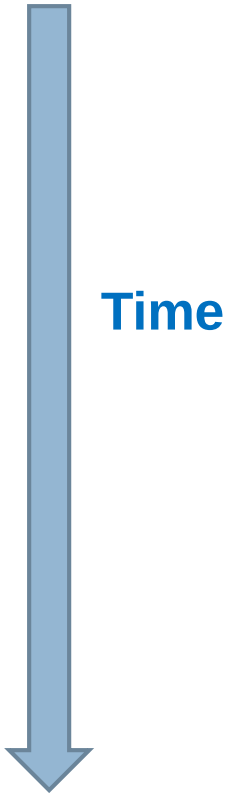
Using intrinsic functions in C/C++

- **For the rest of this lecture, you will be learning how to use intrinsic functions in C/C++**

- Normally, "90% of a program's execution time is spent in executing 10% of the code" - **loops**
  - What programmers normally do to improve performance is to analyze the code and find the computationally intensive functions
    - Then optimize those instead of the whole program
    - This safes time and money
  - **Rewriting loop kernels in C++ using SIMD intrinsics is an excellent choice**
    - Compilers vectorize the code (not always), but doing that manually, using SIMD instrinsics, can really boost performance

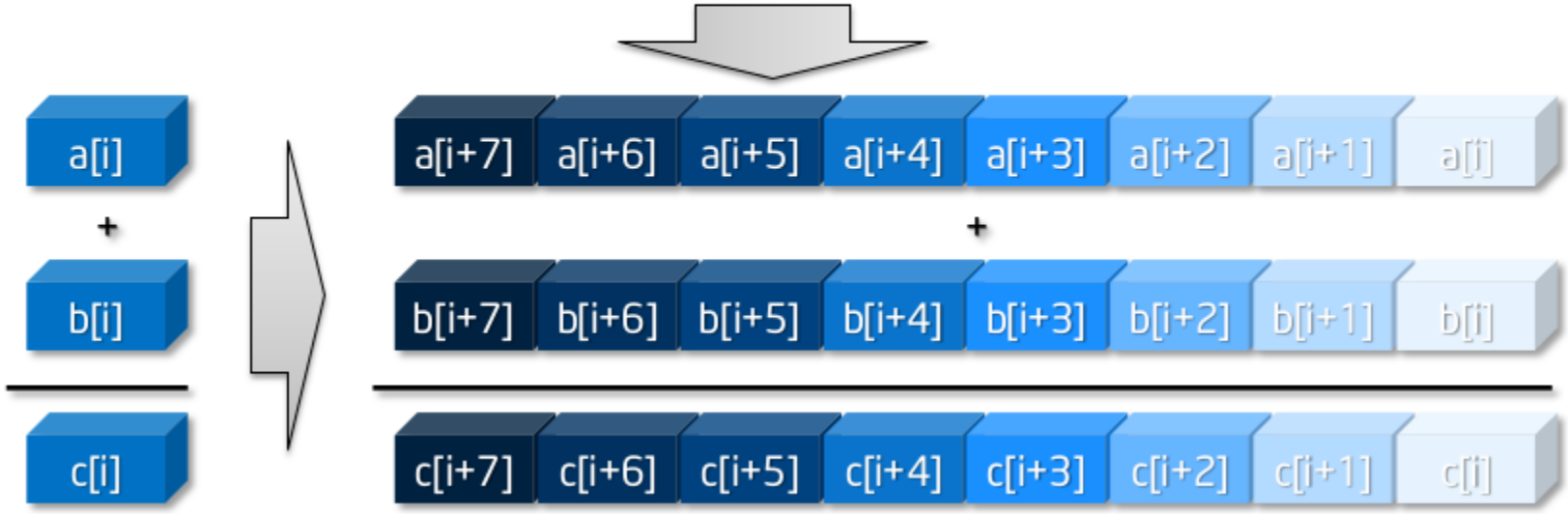
Computer Systems - Hardware Evolution

- Scalar Processors
- Pipelined Processors
- Superscalar and VLIW Processors
- Out of order Processors
- Processors support **Vectorization**
- Hyperthreading
- Multicore Processors
- Manycore Processors
- Heterogeneous systems



Single Instruction Multiple Data (SIMD) – Vectorization


```
for(i = 0; i <= MAX;i++)
    c[i] = a[i] + b[i];
```

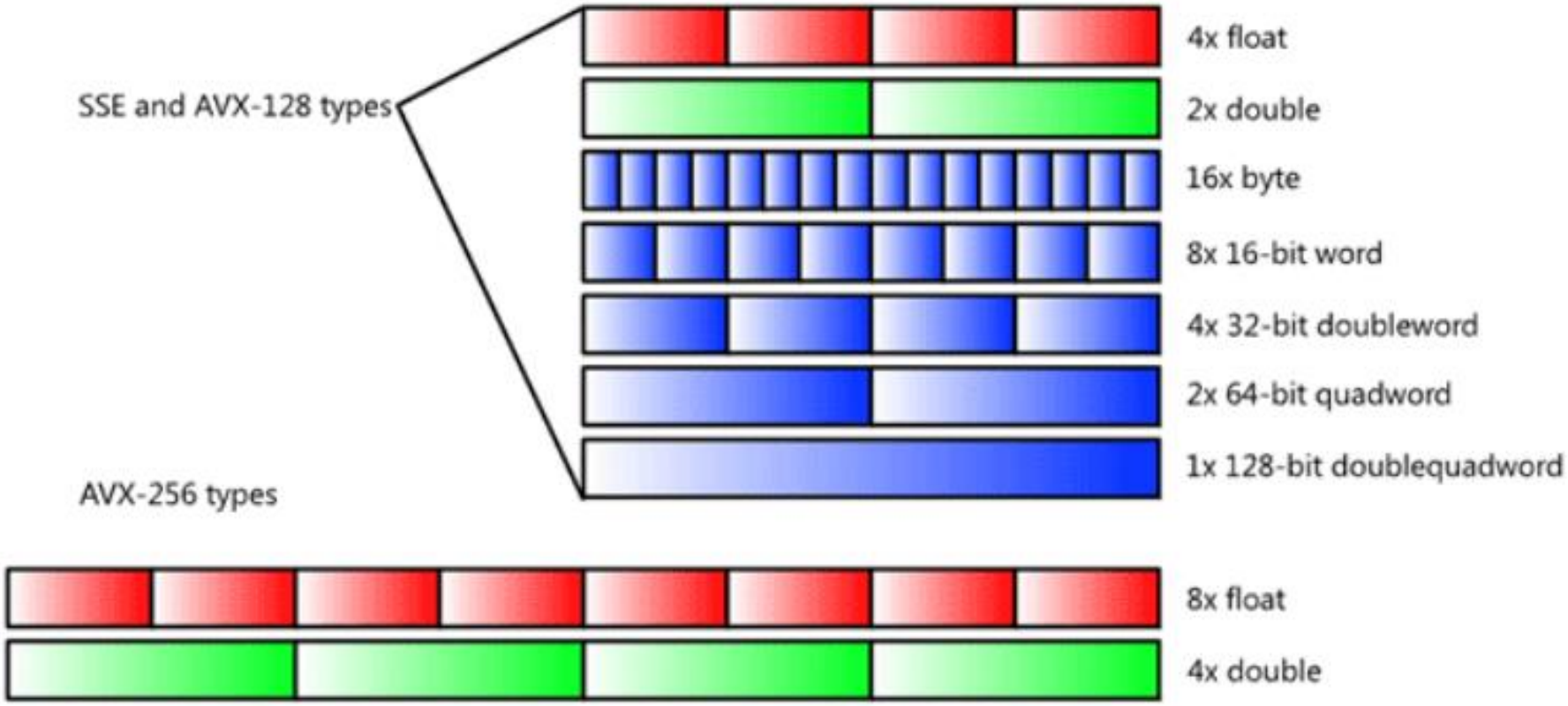


Vectorization on x86/x64 Processors

- **Intel MMX technology** (old – limited usage nowadays)
  - 8 mmx registers of 64 bit
  - extension of the floating point registers
  - can be handled as 8 8-bit, 4 16-bit, 2 32-bit and 1 64-bit, operations
- **Intel SSE technology**
  - 8/16 xmm registers of 128 bit (32-bit architectures support 8 registers only)
  - Can be handled from 16 8-bit to 2 64-bit operations
- **Intel AVX technology**
  - 8/16 ymm registers of 256 bit (32-bit architectures support 8 registers only)
  - Can be handled from 32 8-bit to 4 64-bit operations
- **Intel AVX-512 technology**
  - 32 ZMM 512-bit registers

Vectorization on x86/x64 Processors (2)

__m256 or __m128 (for single precision floats)
__m256d or __m128d (for double precision floats)
__m256i or __m128i (for integers, no matter the size)



Vectorization on x86/x64 Processors (3)

- The developer can use either SSE or AVX or both
  - AVX instructions provide higher throughput than SSE
  - SSE instructions are preferred for less data parallel algorithms
- Vector instructions work only for data that they are **written in consecutive main memory addresses**

- **All the Intel intrinsics are found here :**

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Basic SSE Instructions (1)

- __m128 **_mm_loadu_ps** (float * mem_addr) - Loads four floating point values (packed) from memory to var1

  e.g., ***var1 = _mm_loadu_ps (&A[3][4]);***

- void **_mm_storeu_ps**(float * mem_addr, __m128 a) – Stores four floating point values, from variable a to memory

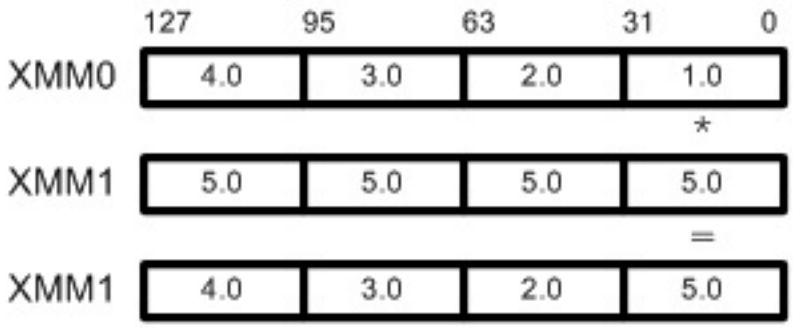
  e.g., ***_mm_storeu_ps (&array[4], num3);***

Basic SSE Instructions (2)

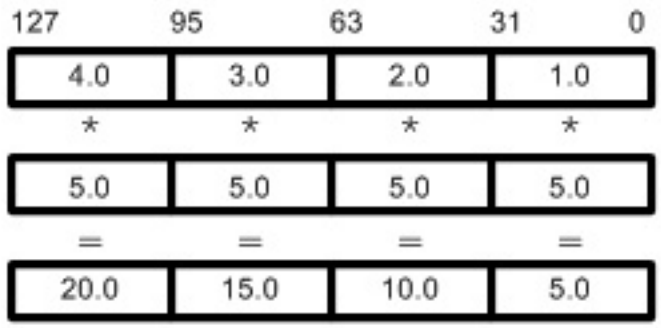
- __m128 **_mm_mul_ps**(__m128 a, __m128 b) - Multiplies the four SP FP values of a and b

- __m128 **_mm_mul_ss**(__m128 a, __m128 b) - Multiplies the lower SP FP values of a and b; the upper 3 SP FP values are passed through from a.

XMM1=_mm_mul_ss(XMM1, XMM0)



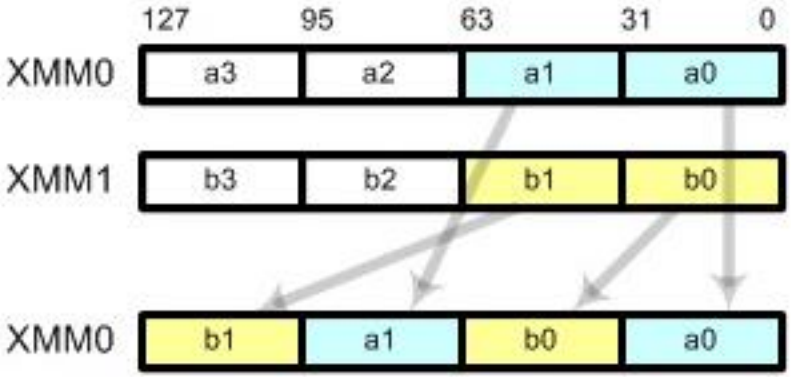
XMM1=_mm_mul_ps(XMM1,XMM0)



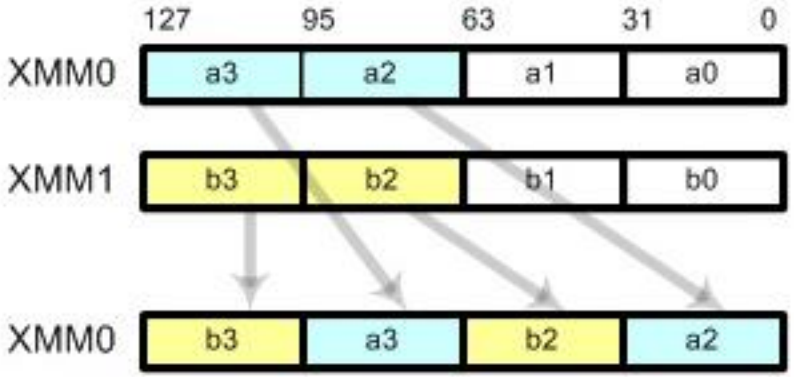
Basic SSE Instructions (3)

- __m128 **_mm_unpackhi_ps** (__m128 a, __m128 b) - Selects and interleaves the upper two SP FP values from a and b.
- __m128 **_mm_unpacklo_ps** (__m128 a, __m128 b) - Selects and interleaves the lower two SP FP values from a and b.

XMM0=_mm_unpacklo_ps (XMMO, XMM1)



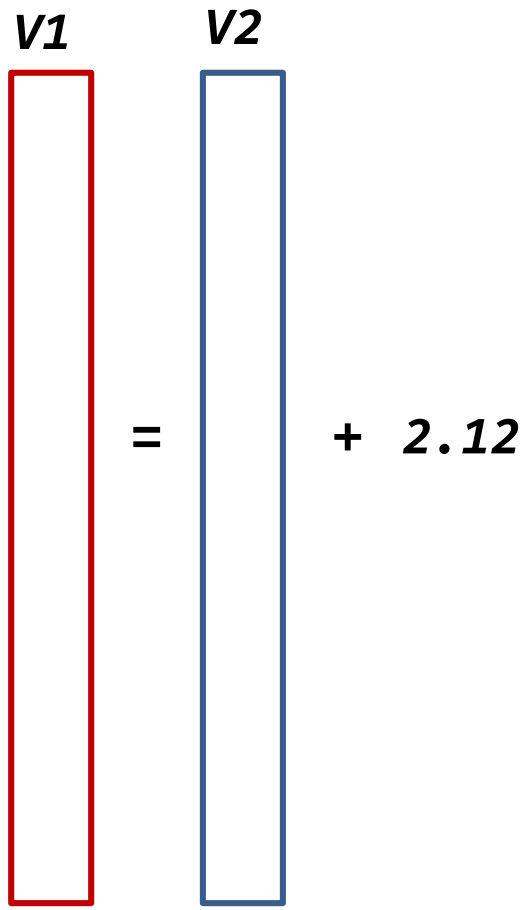
XMM0=_mm_unpackhi_ps (XMMO, XMM1)



Vectorization 1st example (1)

- Compiler vectorise such simple loop kernels, automatically (auto-vectorization)
- **Without** vectorization
  - Load V2[0]
  - V2[0]+2.12
  - Store the result into V1[0]
- **With** vectorization
  - Load V2[0:3] (load 4 values together)
  - V2[0:3] + 2.12 (apply 4 additions together)
  - Store the result into V1[0:3] (store 4 values together)

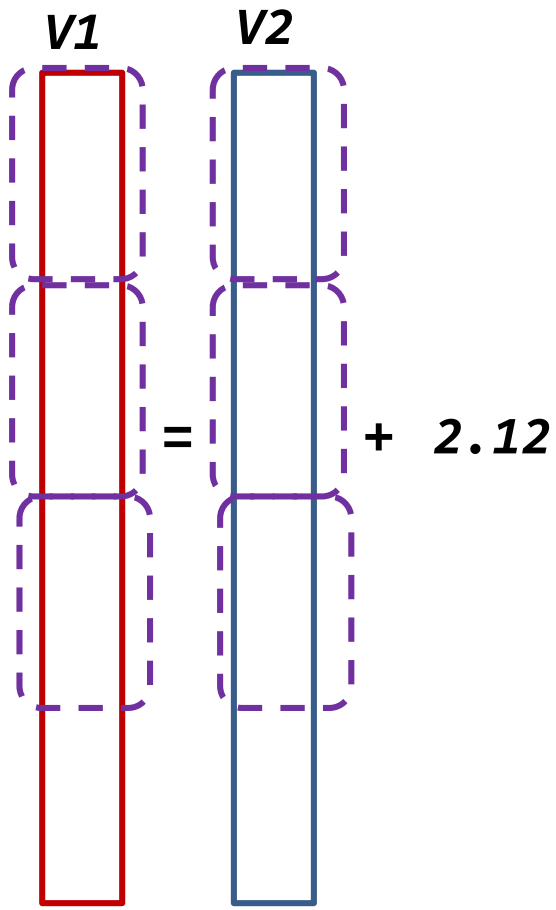
```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```



Vectorization 1st example (2)

- 4 elements are processed together using vector instructions
- Performance is improved by x4
- Low-level C code is easier than assembly, isn't it?

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```




```
__m128 num1, num2, num3;

num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

for (int i = 0; i < M; i += 4) {
  num2 = _mm_load_ps (&V2[i]);
  num3 = _mm_add_ps (num1, num2);
  _mm_store_ps( &V1[i], num3);
}
```

Vectorization 1st example (3)

Define three 128bit variables of type float, thus each 128bit variable contains 4 FP values

Initialize the 128bit variable

Load 128bit of data starting from the memory address &V2[j]

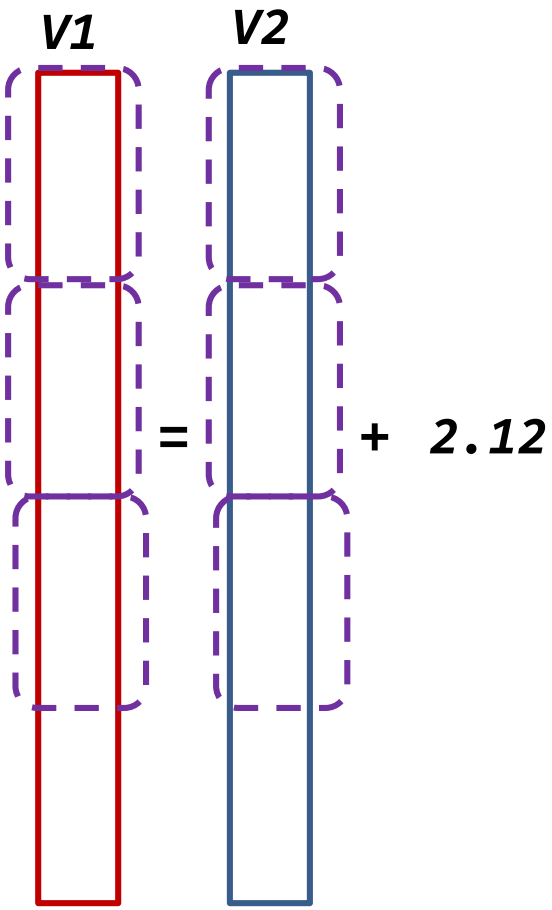
4 iterations are processed in every iteration

```
__m128 num1, num2, num3;

num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

for (int i = 0; i < M; i += 4) {
   num2 = _mm_load_ps (&V2[i]);
   num3 = _mm_add_ps (num1, num2);
   _mm_store_ps( &V1[i], num3);
}
```

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```



Vectorization 1st example (4)

- **What if M==10?**
  - Or not a multiple of 4

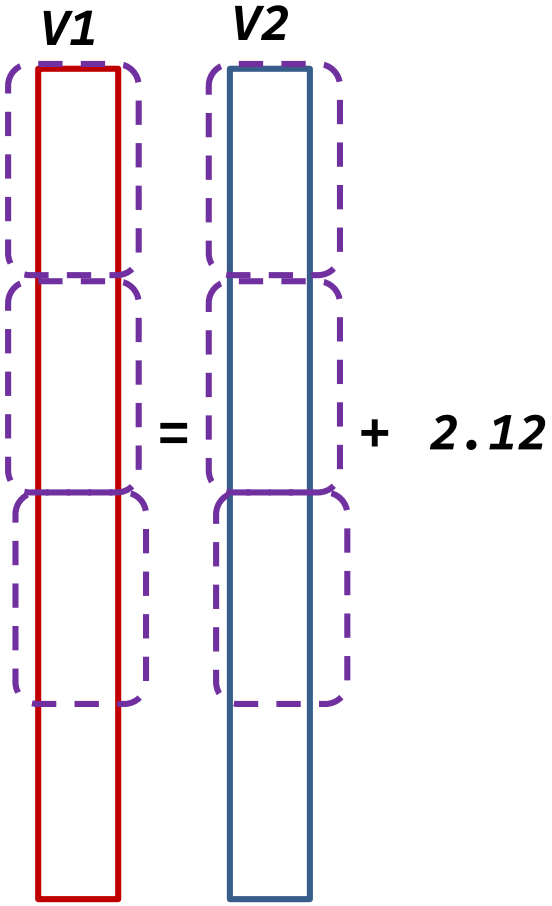
```
__m128 num1, num2, num3;

num1 = _mm_set_ps (2.12, 2.12, 2.12, 2.12);

for (int i = 0; i < 8; i += 4) {
   num2 = _mm_load_ps (&V2[i]);
   num3 = _mm_add_ps (num1, num2);
   _mm_store_ps( &V1[i], num3);
}
```

```
for (j = 8; j < 10; j++) {
V1[j] = V2[j] + 2.1234;
}
```

```
for (j = 0; j < M; j++) {
V1[j] = V2[j] + 2.1234;
}
```



Further Reading

- Chapter 12 and 16 in 'Computer Organization and architecture' available at http://home.ustc.edu.cn/~leedsong/reference_books_tools/Computer%20Organization%20and%20Architecture%2010th%20-%20William%20Stallings.pdf

Any Questions?