SEC204

Computer architectures and low level programming

Dr. Vasilios Kelefouras

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

Date

04/11/2019

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**
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++

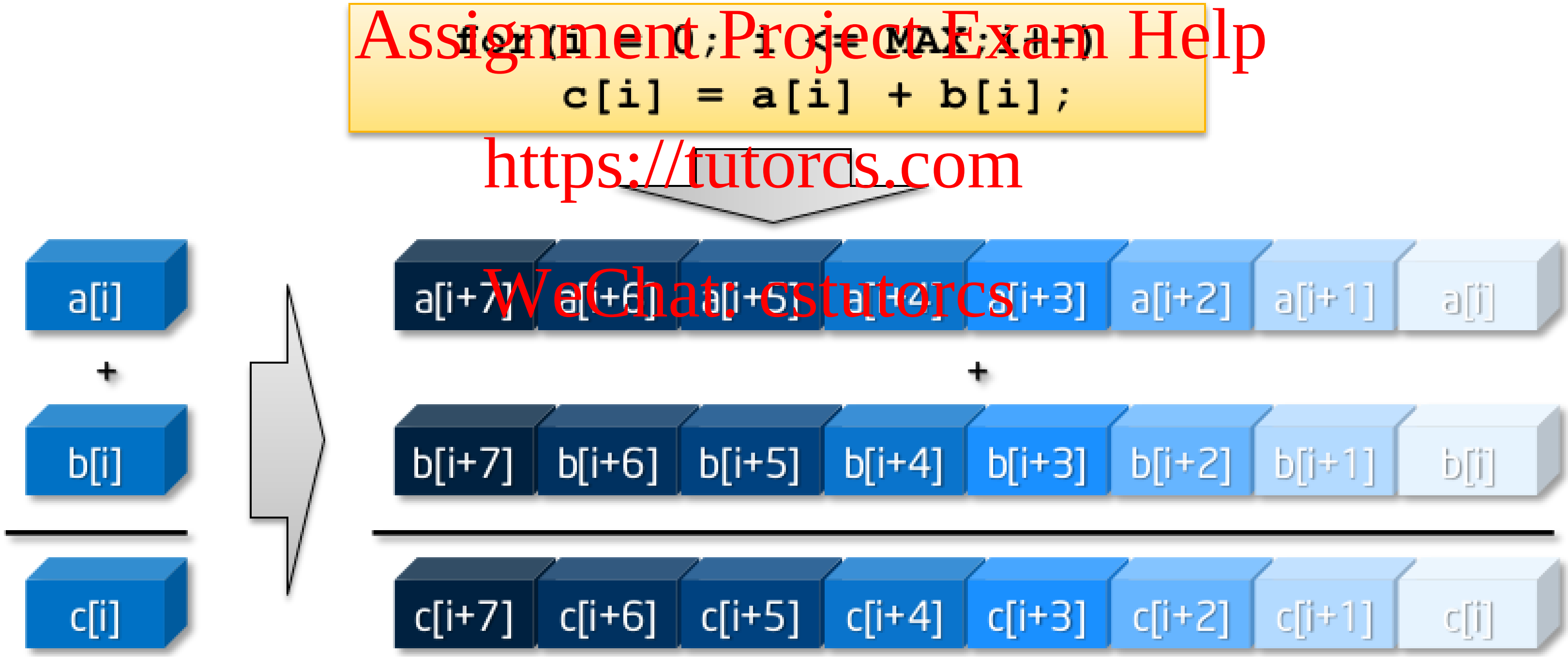
- **Main advantages**
  - Classes, if conditions, loops and functions are very easy to implement
  - Portability to almost all x86 architectures
  - Compatibility with different compilers
- **Main disadvantages**
  - Not all assembly instructions have intrinsic function equivalents
  - Unskilled use 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 manually using SIMD instrinsics can really boost performance

Single Instruction Multiple Data (SIMD) – Vectorization



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

a[i]
+
b[i]
c[i]
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
+
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]

Vectorization on Arm Cortex series NEON technology

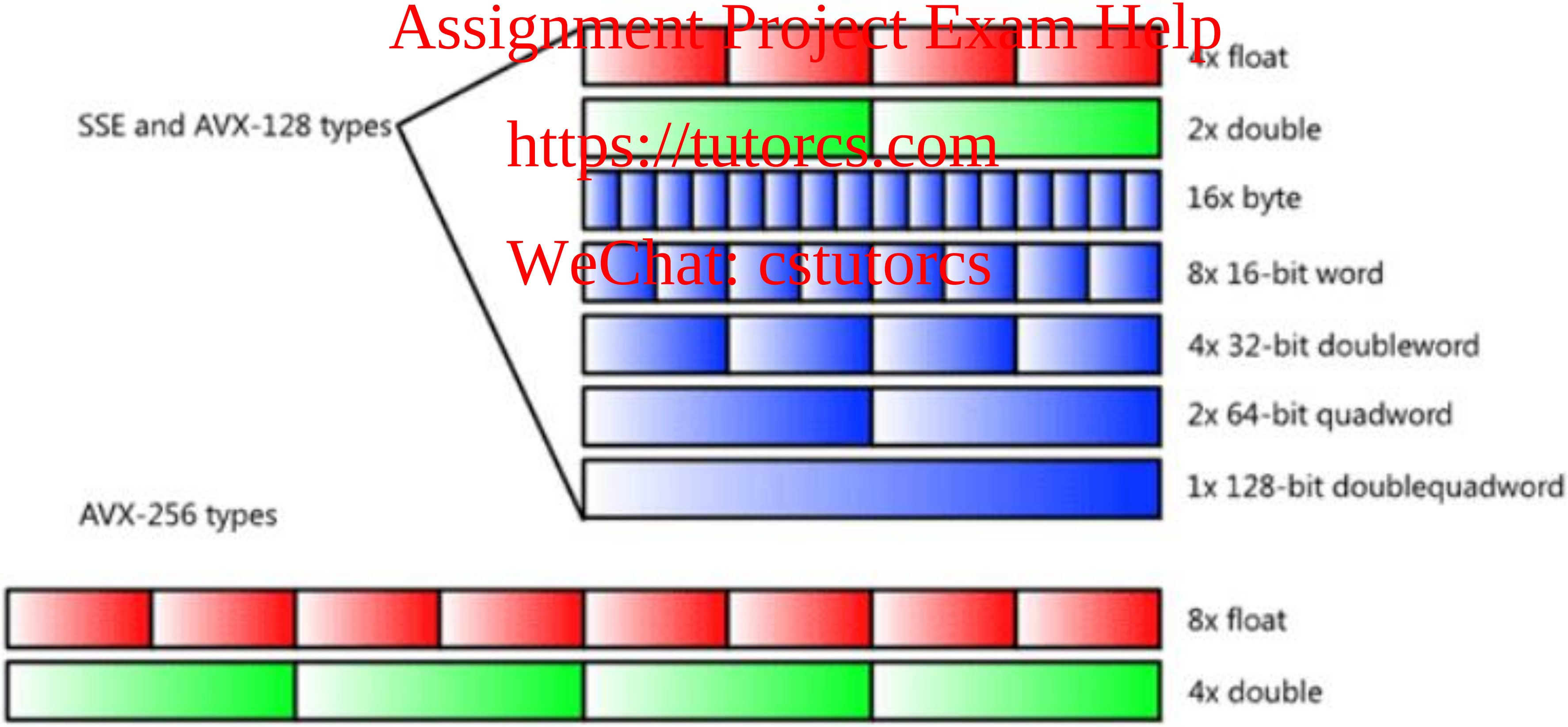
- Arm Neon technology is an advanced SIMD architecture extension for the Arm Cortex-A series and Cortex-R52 processors
  - 128-bit wide
  - They are widely used in embedded systems

- Neon instructions allow up to:
  - 16x8-bit, 8x16-bit, 4x32-bit, 2x64-bit integer operations
  - 8x16-bit, 4x32-bit, 2x64-bit floating-point operations

Vectorization on Intel 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
  - An entire L1 cache line is loaded to the RF in 1-3 cycles
- **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 1 128-bit operations
  - An entire L1 cache line is loaded to the RF in 1-3 cycles
- **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 1 256-bit operations
- **Intel AVX-512 technology**
  - 32 ZMM 512-bit registers

Vectorization on Intel Processors (2)



Vectorization on Intel Processors (3)

- The developer can use either SSE or AVX or both
  - AVX instructions improve throughput
  - SSE instructions are preferred for less data parallel algorithms
- Vector instructions work only for data that they are written in consecutive main memory addresses
- Aligned load/store instructions are faster than the no aligned ones.
- memory and arithmetical instructions are executed in parallel

- **All the Intel intrinsics can be found here :**

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

Basic SSE Instructions (1)

- __m128 **_mm_load_ps** (float * p) – Loads four SP FP values. The address must be 16-byte-aligned
- __m128 **_mm_loadu_ps** (float * p) - Loads four SP FP values. The address need not be 16-byte-aligned

L1

A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
….			

Aligned load

L1

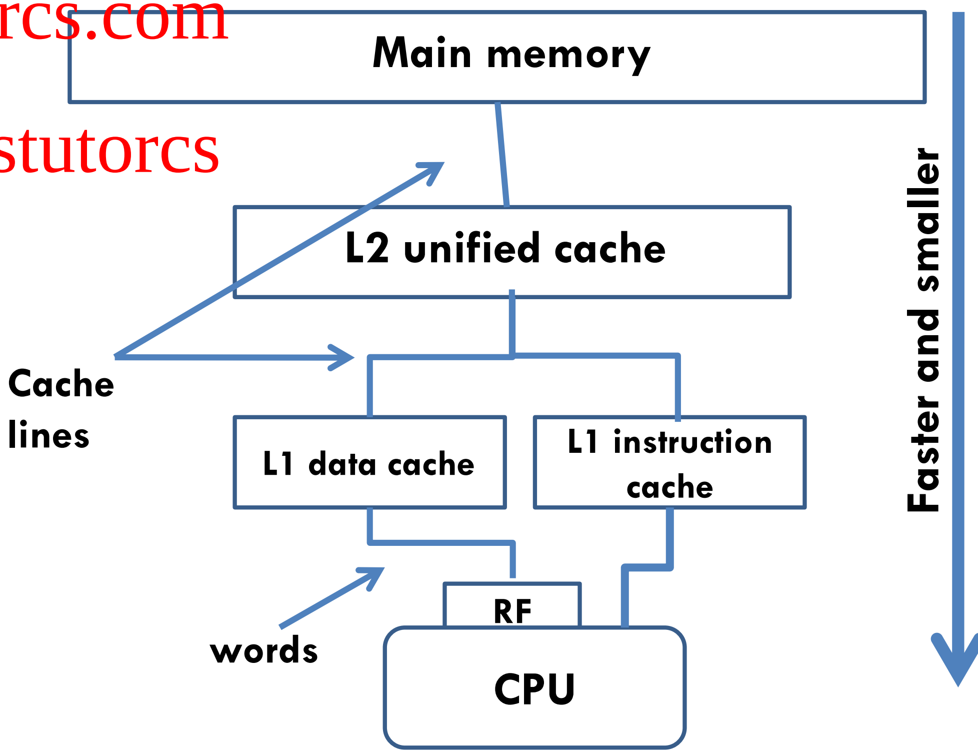
A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
….			

Misaligned load

L1

A[1]	A[2]	A[3]	A[4]
A[5]	A[6]	A[7]	A[8]
….			

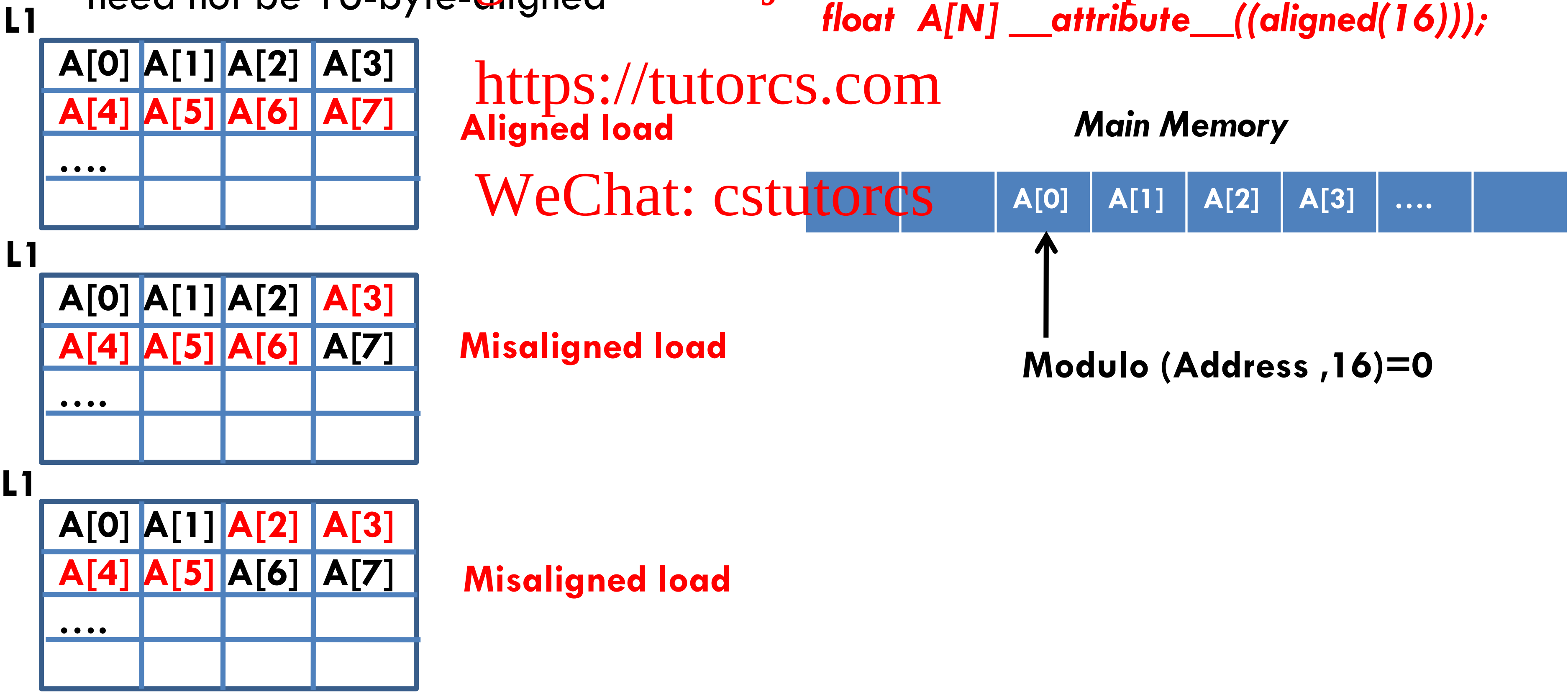
Misaligned load



Basic SSE Instructions (2)

- □ __m128 **_mm_load_ps**(float * p) – Loads four SP FP values. The address must be 16-byte-aligned
- □ __m128 **_mm_loadu_ps**(float * p) - Loads four SP FP values. The address need not be 16-byte-aligned

float A[N] __attribute__((aligned(16)));



L1

A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
....			

Aligned load

L1

A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
....			

Misaligned load

L1

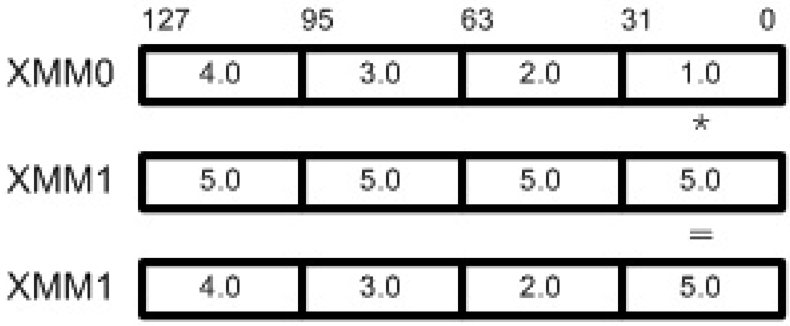
A[0]	A[1]	A[2]	A[3]
A[4]	A[5]	A[6]	A[7]
....			

Misaligned load

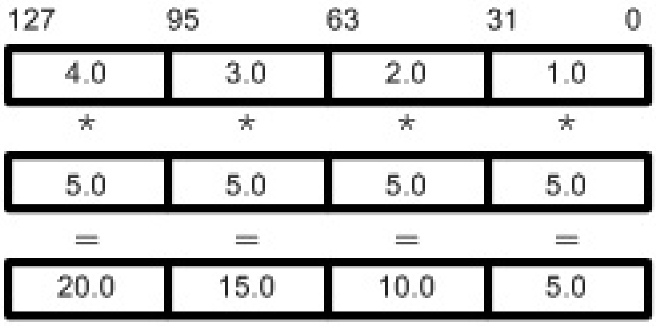
Basic SSE Instructions (3)

- __m128 **_mm_store_ps**(float * p) – Stores four SP FP values. The address must be 16-byte-aligned
- __m128 **_mm_storeu_ps**(float * p) – Stores four SP FP values. The address need not be 16-byte-aligned
- __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)



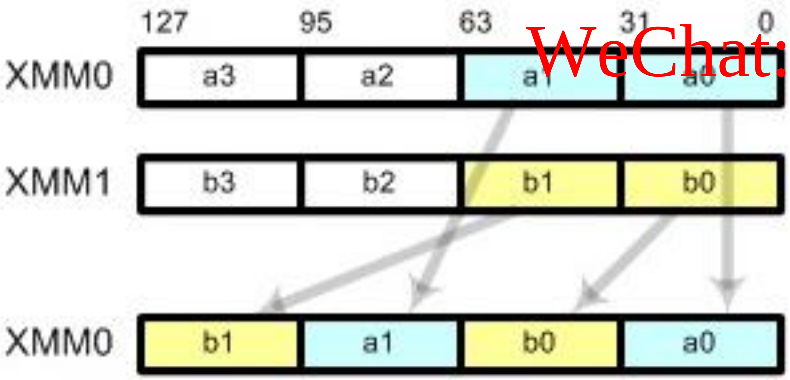
XMM1=_mm_mul_ps(XMM1,XMM0)



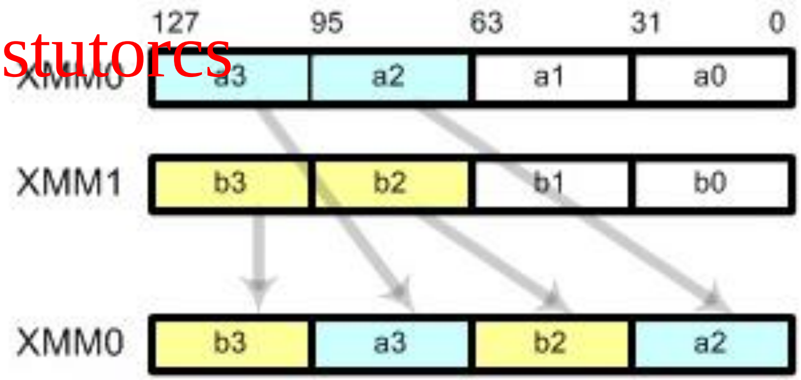
Basic SSE Instructions (4)

- __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)

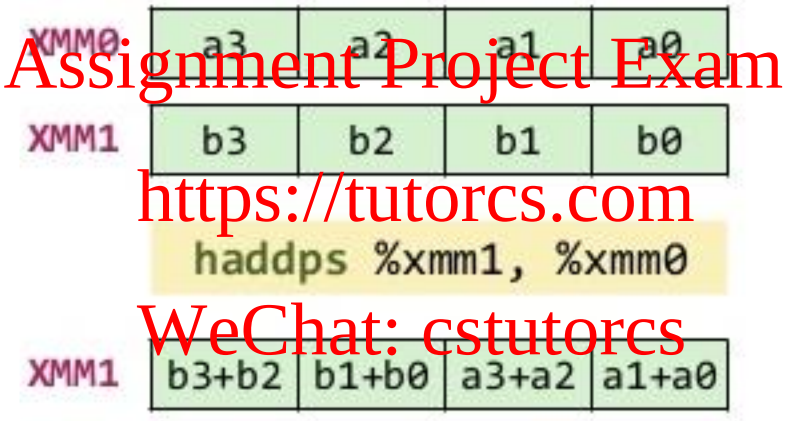


XMM0=_mm_unpackhi_ps (XMMO, XMM1)



Basic SSE Instructions (5)

- □ **__m128 _mm_hadd_ps** (__m128 a, __m128 b) - Adds adjacent vector elements



XMM0:	a3	a2	a1	a0
XMM1	b3	b2	b1	b0

```
haddps %xmm1, %xmm0
```

XMM1	b3+b2	b1+b0	a3+a2	a1+a0

- □ void **_mm_store_ss** (float * p, __m128 a) - Stores the lower SP FP value

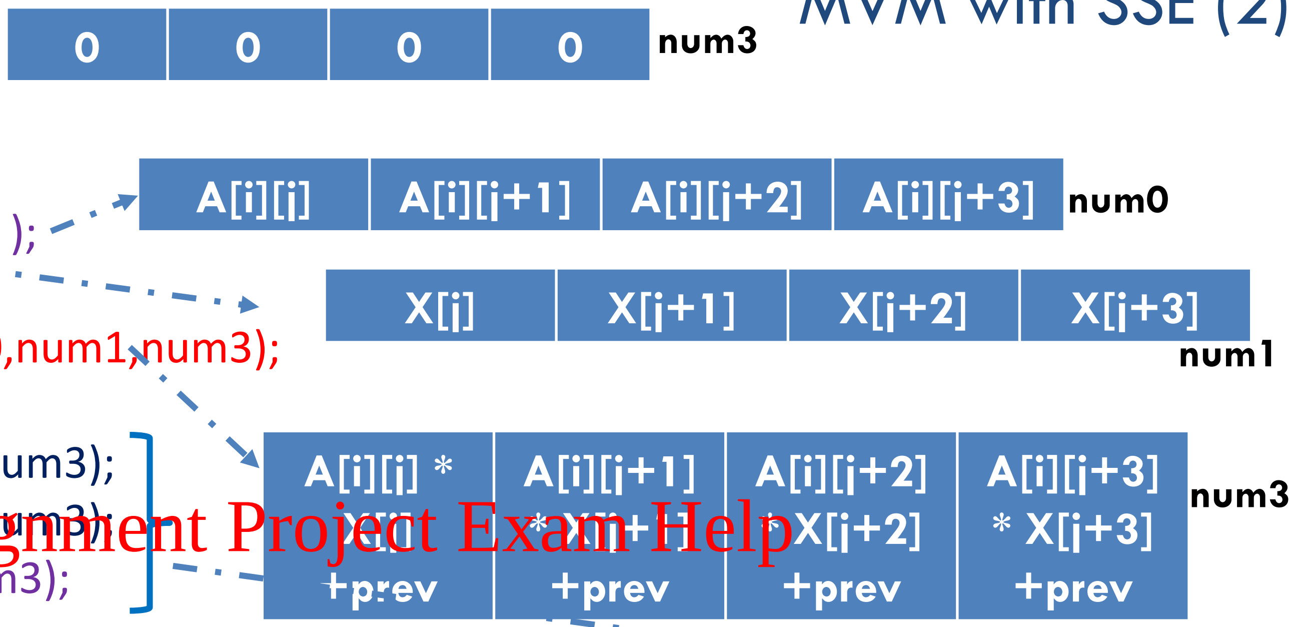
Case Study
MVM using SSE technology

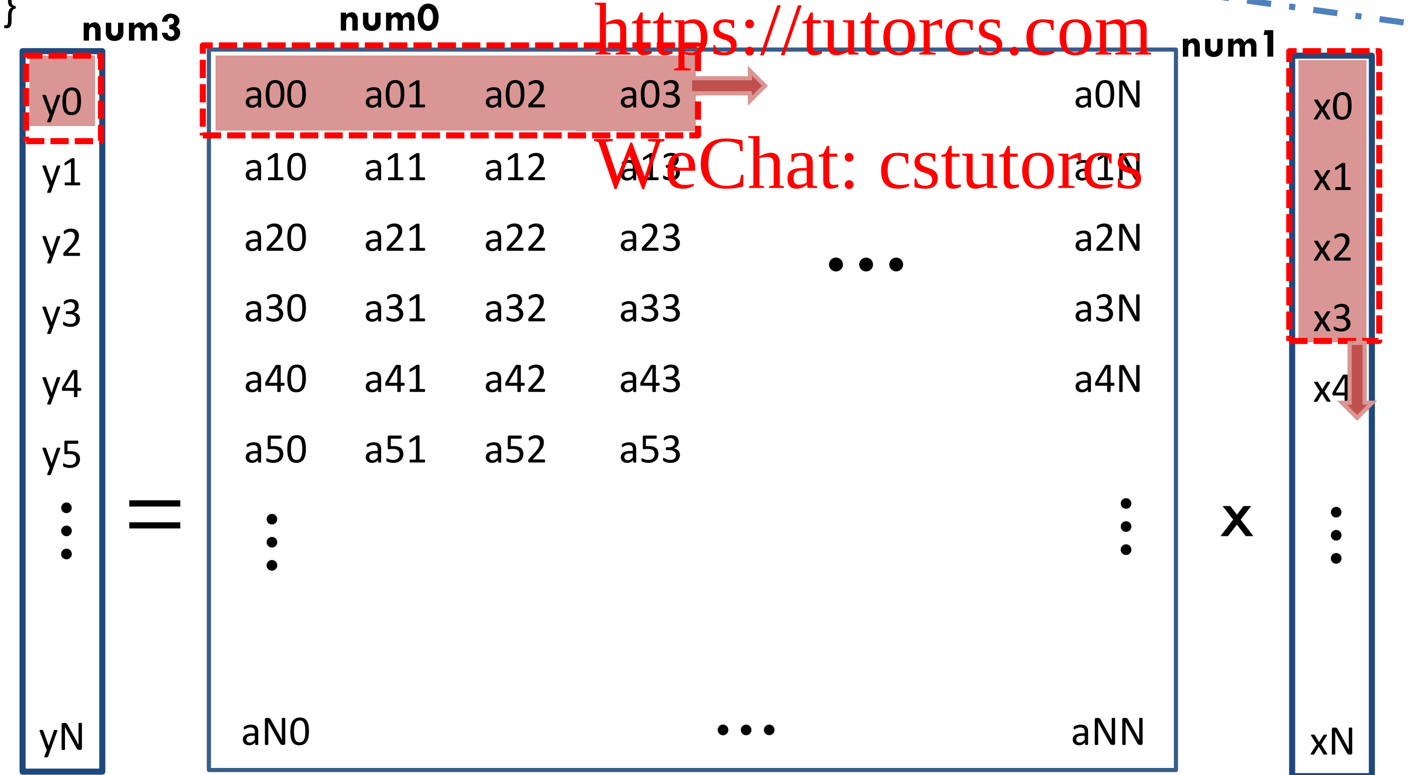
```
float A[N][N];
float X[N], Y[N];
int i,j;

for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    Y[i] += A[i][j] * X[j];
```

```
float A[N][N];
float X[N], Y[N];
int i,j;

for (i=0; i<N;  i++){

num3= _mm_setzero_ps();

 for (j=0; j<N; j+=4){
   num0=_mm_load_ps( &A[i][j] );
   num1=_mm_load_ps(X + j );
   num3=_mm_fmadd_ps(num0,num1,num3);
 }
num4=_mm_hadd_ps(num3, num3);
num4=_mm_hadd_ps(num4, num4);
_mm_store_ss((float *)Y+i, num4);
}
```

MVM with SSE (2)

```
for (i=0; i!=N;  i++){
num3= _mm_setzero_ps();
 for (j=0; j!=N; j+=4){
  num0=_mm_load_ps( &A[i][j] );
  num1=_mm_load_ps(X + j );
  num3=_mm_fmadd_ps(num0,num1,num3);
 }
num3=_mm_hadd_ps(num3, num3);
num3=_mm_hadd_ps(num3, num3);
_mm_store_ss((float *)Y+i, num3);
}
```



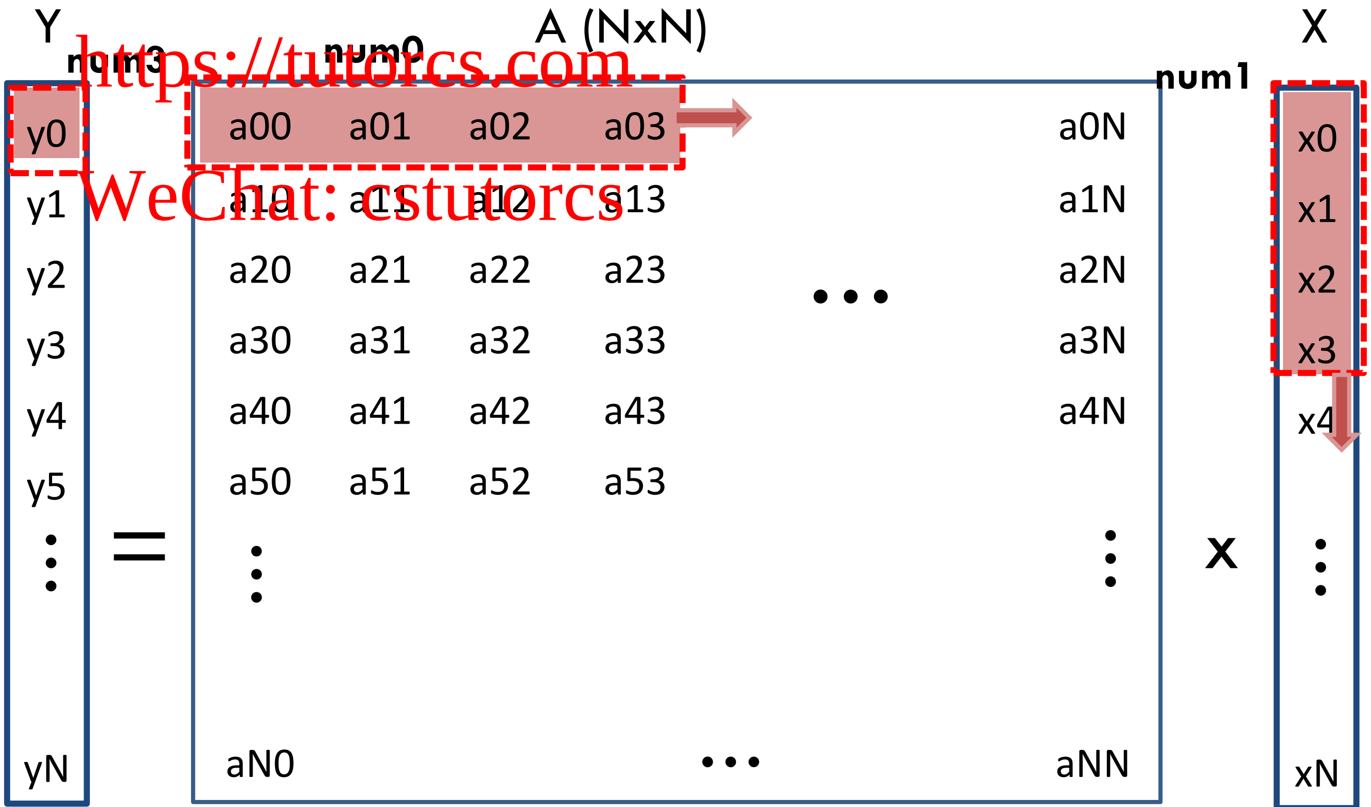


This part of code adds the four values of num3 and stores the result into Y[i]

MVM with SSE (3)

```
....
}
num3=_mm_hadd_ps(num3, num3);
num3=_mm_hadd_ps(num3, num3);
_mm_store_ss((float *)Y+i, num3);
}
```

- After j loop finishes its execution, num3 contains the output data of Y[i]
- **num3=[ya, yb, yc, yd] where Y[i]=ya+yb+yc+yd**
- after the **1st hadd -> num3=[ya+yb, yc+yd, ya+yb, yc+yd]**
- after the **2nd hadd -> num3=[ya+yb+yc+yd, ya+yb+yc+yd, ya+yb+yc+yd, ya+yb+yc+yd]**



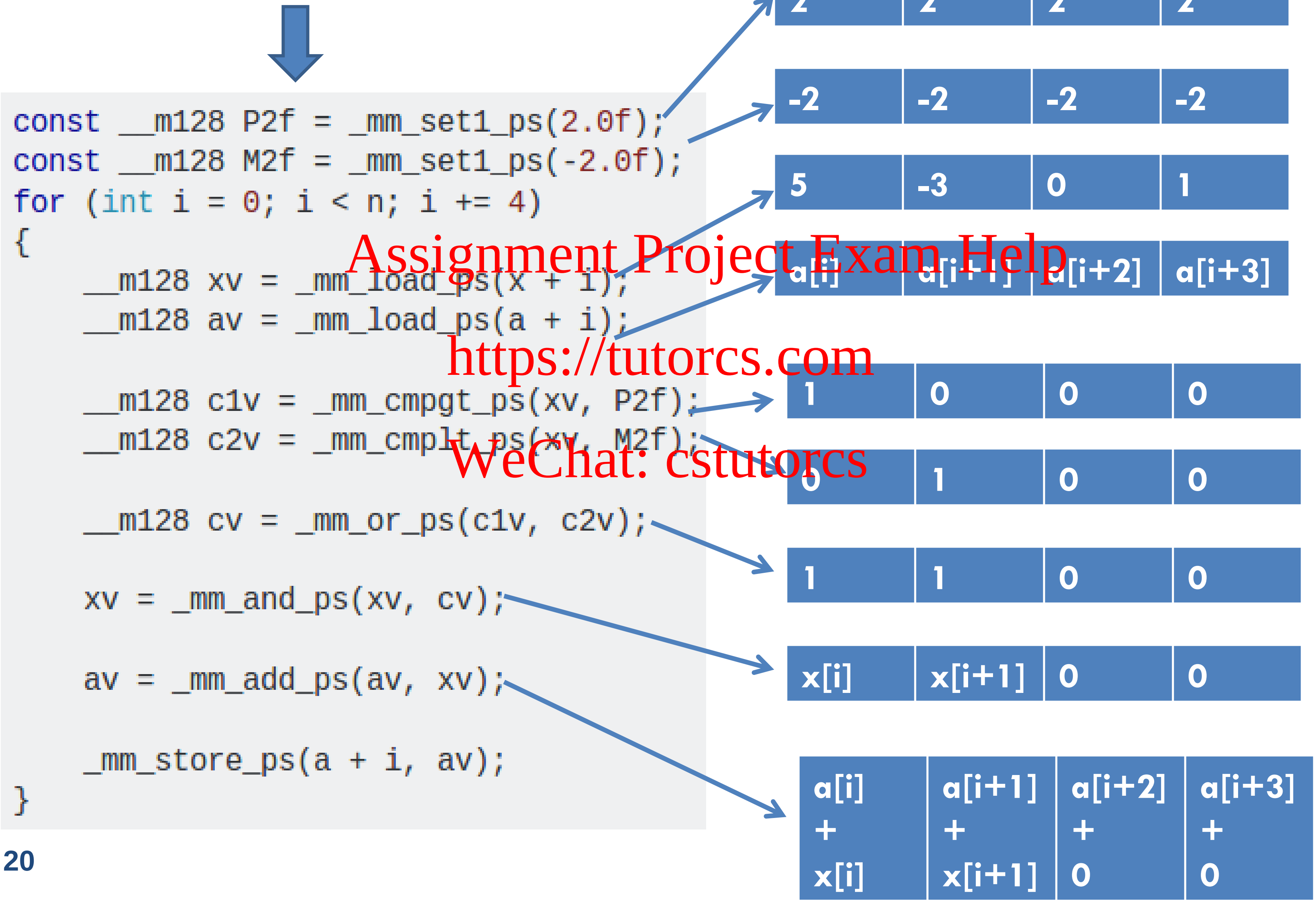
Case Study
MVM using AVX technology

```
float A[N][N];
float X[N], Y[N];
int i,j;

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  Y[i] += A[i][j] * X[j];
```

```
for (i=0;i!=N;i++)
 for (j=0;j!=N;j++){
  ymm0=_mm256_setzero_ps();
  for (k=0;k!=N;k+=8){
  ymm1=_mm256_load_ps( A + N*i + k);
  ymm2=_mm256_load_ps( Btrans + N*j + k);
  ymm0=_mm256_fmadd_ps(ymm1,ymm2,ymm0);
}

ymm2 = _mm256_permute2f128_ps(ymm0,ymm0,1);
ymm0 = _mm256_add_ps(ymm0, ymm2);
ymm0 = _mm256_hadd_ps(ymm0, ymm0);
ymm0 = _mm256_hadd_ps(ymm0, ymm0);
_mm_store_ss((float *) C + N*i + j,
       _mm256_extractf128_ps(ymm0,0));
}
}
```

What about if-conditions on SSE ?

```
for (i=0; i < n; i++) {
   if ( x[i] > 2 || x[i] < -2 )
      a[i]+=x[i];  }
```



```
const __m128 P2f = _mm_set1_ps(2.0f);
const __m128 M2f = _mm_set1_ps(-2.0f);
for (int i = 0; i < n; i += 4)
{
    __m128 xv = _mm_load_ps(x + i);
    __m128 av = _mm_load_ps(a + i);

    __m128 c1v = _mm_cmpgt_ps(xv, P2f);
    __m128 c2v = _mm_cmplt_ps(xv, M2f);

    __m128 cv = _mm_or_ps(c1v, c2v);

    xv = _mm_and_ps(xv, cv);

    av = _mm_add_ps(av, xv);

    _mm_store_ps(a + i, av);
}
```

2	2	2	2
-2	-2	-2	-2
5	-3	0	1
a[i]	a[i+1]	a[i+2]	a[i+3]
1	0	0	0
0	1	0	0
1	1	0	0
x[i]	x[i+1]	0	0
a[i] + x[i]	a[i+1] + x[i+1]	a[i+2] + 0	a[i+3] + 0