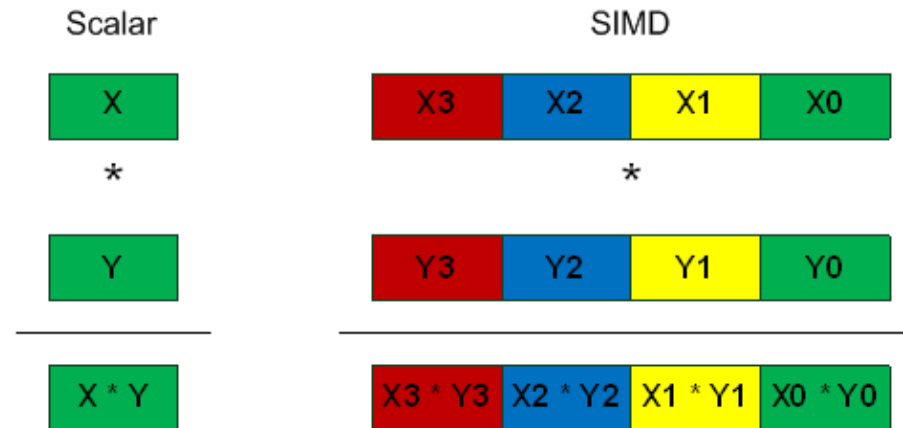


# VECTOR COMPUTING



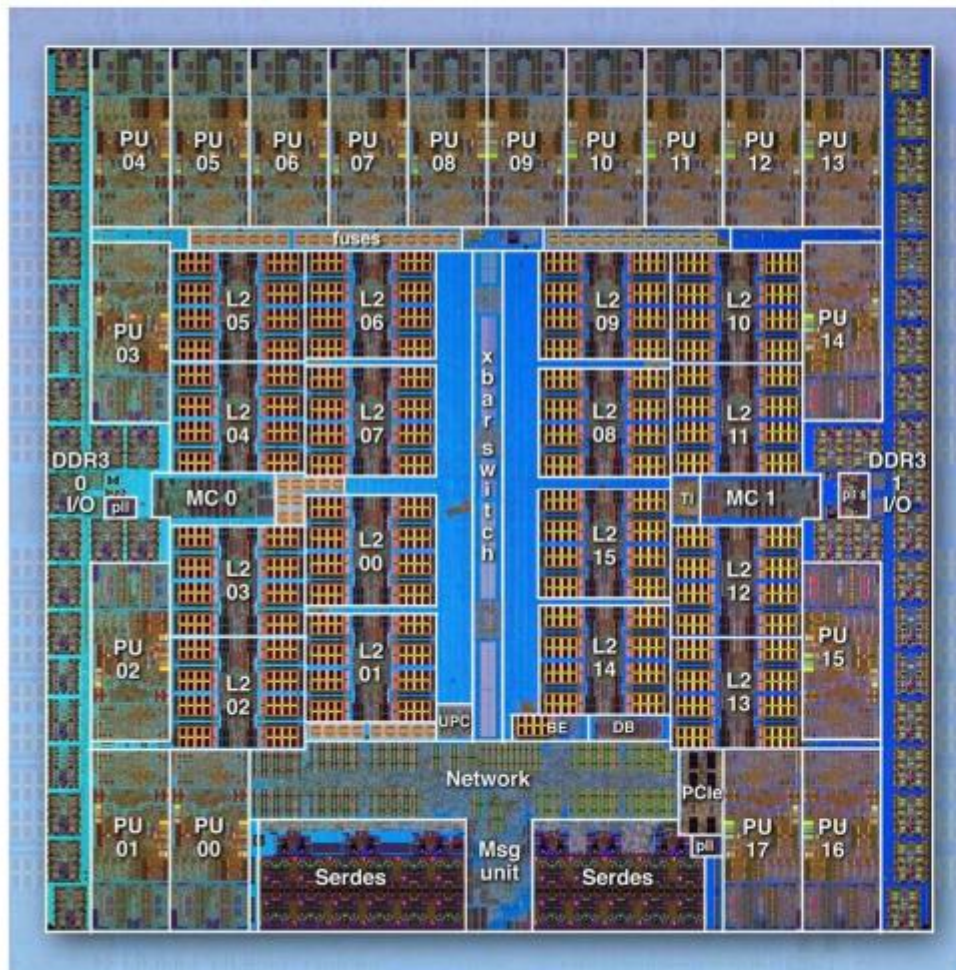
Claude TADONKI  
Mines ParisTech – Paris/France



Seminar at **Universidad Santiago de Chile**  
**August 6, 2014** **SANTIAGO - CHILE**

# BlueGene/Q Compute chip

System-on-a-Chip design : integrates processors, memory and networking logic into a single chip

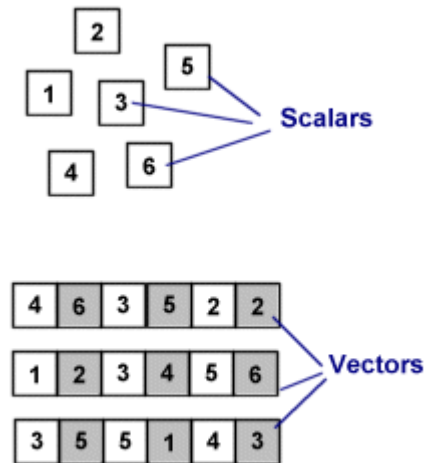
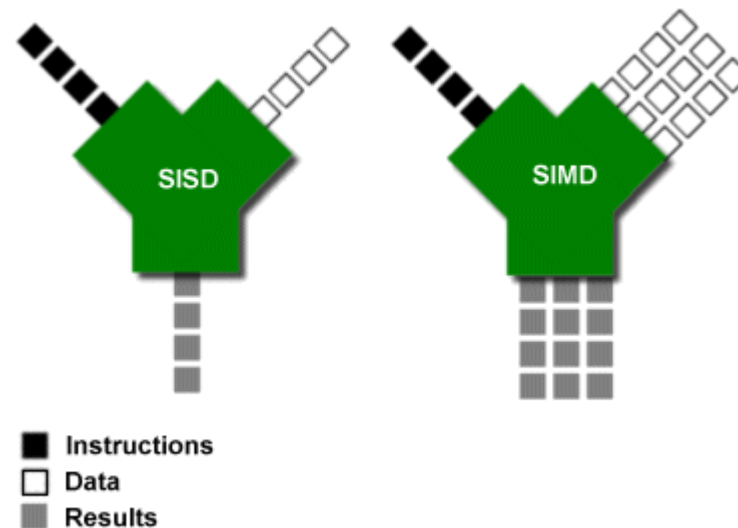


- 360 mm<sup>2</sup> Cu-45 technology (SOI)
- 16 user + 1 service PPC processors
  - plus 1 redundant processor
  - all processors are symmetric
  - 11 metal layer
  - each 4-way multi-threaded
  - 64 bits
  - 1.6 GHz
  - L1 I/D cache = 16kB/16kB
  - L1 prefetch engines
  - each processor has Quad FPU (4-wide double precision, SIMD)
  - peak performance 204.8 GFLOPS @ 55 W
- Central shared L2 cache: 32 MB
  - eDRAM
  - multiversioned cache – supports transactional memory, speculative execution.
  - supports scalable atomic operations
- Dual memory controller
  - 16 GB external DDR3 memory
  - 42.6 GB/s DDR3 bandwidth (1.333 GHz DDR3) (2 channels each with chip kill protection)
- Chip-to-chip networking
  - 5D Torus topology + external link
    - 5 x 2 + 1 high speed serial links
  - each 2 GB/s send + 2 GB/s receive
  - DMA, remote put/get, collective operations
- External (file) IO – when used as IO chip.
  - PCIe Gen2 x8 interface (4 GB/s Tx + 4 GB/s Rx)
  - re-uses 2 serial links
  - interface to Ethernet or Infiniband cards



Could you explain how to get the 204.8 GFLOPS for BlueGene/Q ?

## Vector Computing – SIMD (Single Instruction Multiple Data)



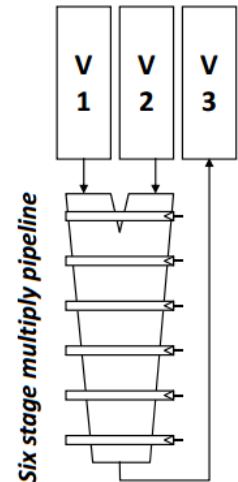
- ▶ A **SIMD** machine simultaneously operates on tuples of atomic data (*one instruction*).
- ▶ **SIMD** is opposed to **SCALAR** (the traditional mechanism).
- ▶ **SIMD** is about exploiting parallelism in the data stream (**DLP**), while superscalar **SISD** is about exploiting parallelism in the instruction stream (**ILP**).
- ▶ **SIMD** is usually referred as **VECTOR COMPUTING**, since its basic unit is the *vector*.
- ▶ Vectors are represented in what is called *packed data format* stored into *vector registers*.
- ▶ On a given machine, the length of the vector registers and their number are fixed and determine the hardware SIMD potential.
- ▶ SIMD can be implemented on using specific extensions **MMX**, **SSE**, **AVX**, ...

## Pipeline Floating Point Computation (multi-stage)

- ▶ Consider the 6 steps (stages) involved in a floating-point addition on a sequential machine with IEEE arithmetic hardware:
  - A. exponents are compared for the smallest magnitude.
  - B. exponents are equalized by shifting the significand smaller.
  - C. the significands are added.
  - D. the result of the addition is normalized.
  - E. checks are made for floating-point exceptions such as overflow.
  - F. rounding is performed.



Step	A	B	C	D	E	F
$x$	0.1234E4	0.12340E4				
$y$	-0.5678E3	-0.05678E4				
$s$			0.066620E4	0.66620E3	0.66620E3	0.6662E3



$$V3 \leftarrow v1 * v2$$

Image from Asanovic's PhD thesis.

Ucal Berkeley, 1998.

- ▶ A scalar implementation of adding two array of length  $n$  will require  $6n$  steps
- ▶ A pipeline implementation of adding two array of length  $n$  will require  $6 + (n-1)$  steps
- ▶ Some architectures provide a wider overlapping by chaining the pipelines.
- ▶ Roughly speaking, a  $p$ -length vector computation on a given  $n$ -array needs  $n/p$  steps.
- ▶ Depending on the architecture, pipeline processing applies to # operations (arith, logical).
- ▶ Pipeline feature is usually covered in the topic of **Instruction Level Parallelism(ILP)**

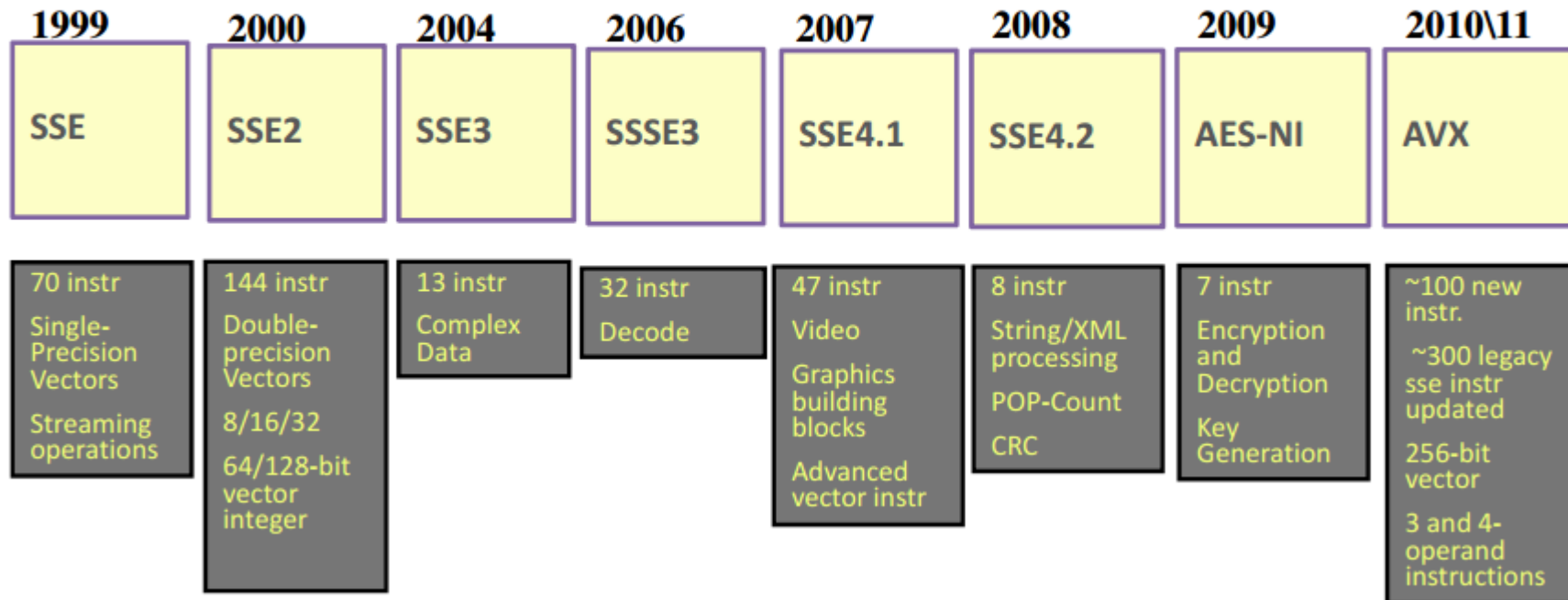


Could you identify and explain other type of pipeline in a standard computation scheme ?

FP comp // Integer comp // Load // store // ....

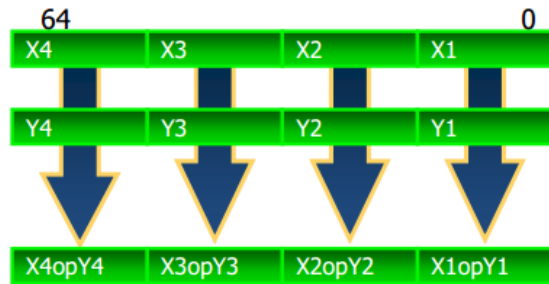
## SIMD Implementation

### SIMD: Continuous Evolution



Advanced Encryption Standard (AES)  
AES New Instructions (AES-NI)

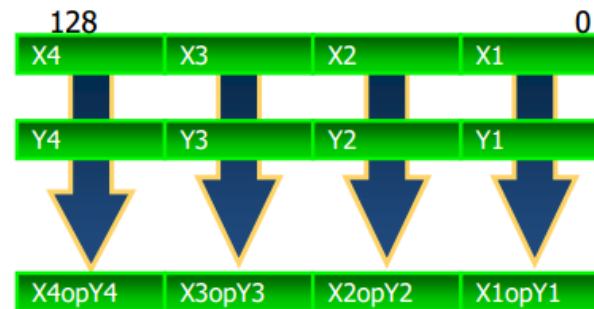
## SIMD Implementation



### MMX™

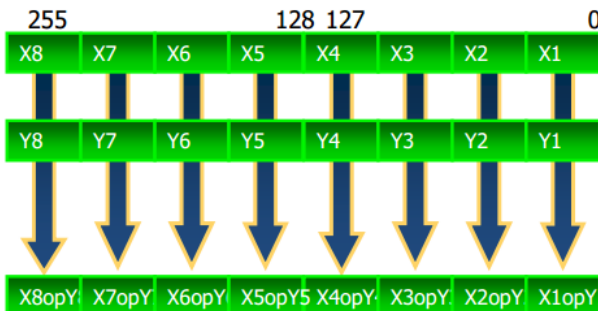
Vector size: 64bit  
 Data types: 8, 16 and 32 bit integers  
 VL: 2,4,8  
 For sample on the left:  $X_i, Y_i$  16 bit integers

**MMX** = MultiMedia eXtension  
**SSE** = Streaming SIMD Extension  
**AVX** = Advanced Vector Extensions  
**MIC** = Many Integrated Core



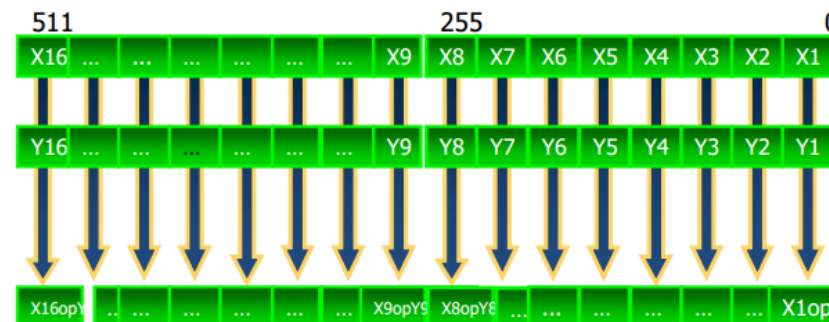
### Intel® SSE

Vector size: 128bit  
 Data types:  
 8,16,32,64 bit integers  
 32 and 64bit floats  
 VL: 2,4,8,16  
 Sample:  $X_i, Y_i$  bit 32 int / float



### Intel® AVX

Vector size: 256bit  
 Data types: 32 and 64 bit floats  
 VL: 4, 8, 16  
 Sample:  $X_i, Y_i$  32 bit int or float



### Intel® MIC

Vector size: 512bit  
 Data types:  
 32 and 64 bit integers  
 32 and 64bit floats  
 (some support for 16 bits floats)  
 VL: 8,16  
 Sample: 32 bit float

## SSE (Overview)

- ▶ **SSE = Streaming SIMD Extensions**
- ▶ SSE programming can be done either through **(inline) assembly** or from a high-level language (C and C++) using **intrinsics**.
- ▶ The **{x,e,p}mmintrin.h** header file contains the declarations for the SSEx instructions intrinsics.
  - xmmintrin.h** -> SSE
  - emmintrin.h** -> SSE2
  - pmmmintrin.h** -> SSE3
- ▶ SSE instruction sets can be enabled or disabled. If disabled, SSE instructions will not be possible. It is recommended to leave this BIOS feature enabled by default. In any case MMX (MultiMedia eXtensions) will still be available.
- ▶ compile your SSE code with "**gcc -o vector vector.c -msse -msse2 -msse3**"
- ▶ SSE intrinsics use types **\_\_m128** (*float*), **\_\_m128i** (*int, short, char*), and **\_\_m128d** (*double*)
- ▶ Variable of type **\_\_m128**, **\_\_m128i**, and **\_\_m128d** (exclusive use) maps to the XMM[0-7] registers (**128 bits**), and automatically aligned on 16-byte boundaries.
- ▶ Vector registers are **xmm0**, **xmm1**, ..., **xmm7**. Initially, they could only be used for single precision computation. Since SSE2, they can be used for any primitive data type.

## SSE (Connecting vectors to scalar data)

- ▶ Vector variables can be connected to scalar variables (arrays) using one of the following ways

```
float a[N] __attribute__((aligned(16)));  
__m128 *ptr = (__m128*)a;
```

`ptr[i]` or `*(ptr+i)` represents the vector  
`{a[4i], a[4i+1], a[4i+2], a[4i+3]}`

```
float a[N] __attribute__((aligned(16)));  
__m128 mm_a;  
mm_a = _mm_load_pd(&a[i]); // here we explicitly load data into the vector
```

`mm_a` represents the vector  
`{a[4i], a[4i+1], a[4i+2], a[4i+3]}`

- ▶ Using the above connection, we can now use SSE instruction to process our data.

This can be done through

- ★ (inline) assembly

- ★ intrinsics (interface to keep using high-level instructions to perform vector operations)



Pros and cons of using (inline)assembly versus intrinsics.



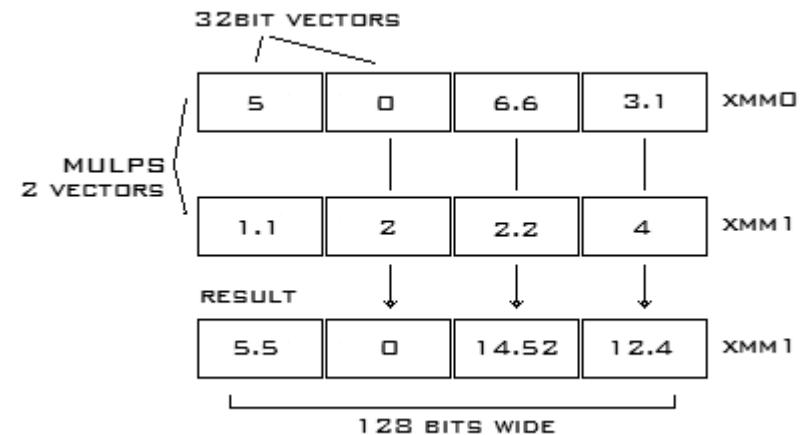
## SSE (basic assembly instructions)

### Data Movement Instructions

- MOVUPS** - Move 128bits of data to an SIMD register from memory or SIMD register. Unaligned.
- MOVAPS** - Move 128bits of data to an SIMD register from memory or SIMD register. Aligned.
- MOVHPS** - Move 64bits to upper bits of an SIMD register (high).
- MOVLPS** - Move 64bits to lower bits of an SIMD register (low).
- MOVHLPS** - Move upper 64bits of source register to the lower 64bits of destination register.
- MOVLHPS** - Move lower 64bits of source register to the upper 64bits of destination register.
- MOVMSKPS** - Move sign bits of each of the 4 packed scalars to an x86 integer register.
- MOVSS** - Move 32bits to an SIMD register from memory or SIMD register.

### Arithmetic Instructions

- |                 |   |  |
|-----------------|---|--|
| <u>Parallel</u> | <u>Scalar</u> ( <i>will perform the operation on the first elements only.</i> ) |  |
| <b>ADDPS</b>    | <b>ADDSS</b>  | - Adds operands                        |
| <b>SUBPS</b>    | <b>SUBSS</b>  | - Subtracts operands                   |
| <b>MULPS</b>    | <b>MULSS</b>  | - Multiplies operands                  |
| <b>DIVPS</b>    | <b>DIVSS</b>  | - Divides operands                     |
| <b>SQRTPS</b>   | <b>SQRTSS</b>   | - Square root of operand               |
| <b>MAXPS</b>    | <b>MAXSS</b>  | - Maximum of operands                  |
| <b>MINPS</b>    | <b>MINSS</b>  | - Minimum of operands                  |
| <b>RCPSPS</b>   | <b>RCPSS</b>  | - Reciprocal of operand                |
| <b>RSQRTPS</b>  | <b>RSQRTSS</b>  | - Reciprocal of square root of operand |



Truth Table:

Destination	Source	ANDPS	ANDNPS	ORPS	XORPS
1	1	1	0	1	0
1	0	0	0	1	1
0	1	0	1	1	1
0	0	0	0	0	0

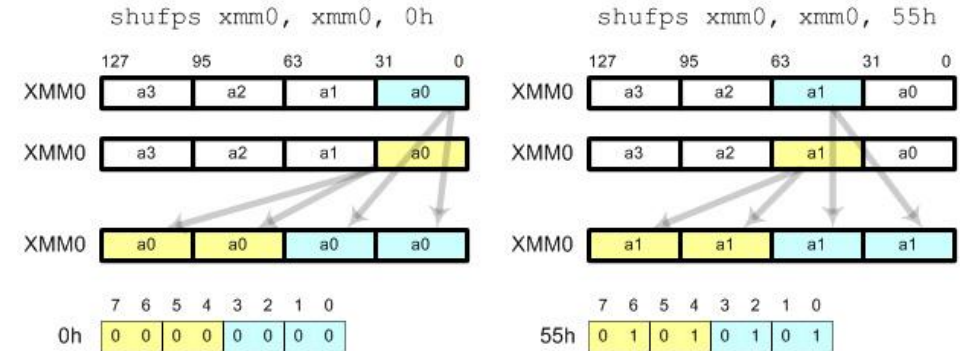
### Logical Instructions

- ANDPS** - Bitwise AND of operands
- ANDNPS** - Bitwise AND NOT of operands
- ORPS** - Bitwise OR of operands
- XORPS** - Bitwise XOR of operands

## SSE (basic assembly instructions)

Shuffling offers a way to

- change the order of the elements within a single vector or
- combine the elements of two separate registers.



### Shuffle Instructions

**SHUFPS**

- Shuffle numbers from one operand to another or itself.

**UNPCKHPS**

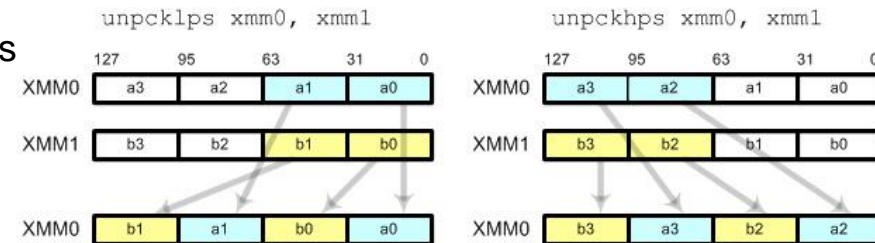
- Unpack high order numbers to an SIMD register.

**UNPCKLPS**

- Unpack low order numbers to a SIMD register.

The **SHUFPS** instruction takes two SSE registers and an 8 bit hex value. (elements are numbered from right to left !!!)

- The first two elements of the destination operand are overwritten by any two elements of the destination register.
- The third and fourth elements of the destination register are overwritten by any two elements from the source register.
- The hex string is used to tell the instruction which elements to shuffle.
- 00, 01, 10, and 11 are used to access elements within the registers



### Examples

**SHUFPS** XMM0, XMM0, 0x1B // 0x1B = 00 01 10 11 and reverses the order of the elements

**SHUFPS** XMM0, XMM0, 0xAA // 0xAA = 10 10 10 10 and sets all elements to the 3rd element



Write the shuffling instruction to obtain (a2, a3, a0, a1) from (a3, a2, a1, a0) in XMM0  
 What is XMM0 after SHUFPS XMM0, XMM0, 93h ?  
 What is XMM0 after SHUFPS XMM0, XMM0, 39h ?

## SSE (assembly examples)

```
// Use sse to multiply vector elements by a real number a * b
vector4 sse_vector4_multiply(const vector4 &op_a, const float &op_b)
{
    vector4 ret_vector;
    __m128 f = _mm_set1_ps(op_b);    // Set all 4 elements to op_b
    __asm
    {
        MOV        EAX, op_a          // Load pointer into CPU reg
        MOVUPS     XMM0, [EAX]        // Move the vectors to SSE regs
        MULPS      XMM0, f            // Multiply elements
        MOVUPS     [ret_vector], XMM0 // Save the return vector
    }
    return ret_vector;
}
```

```
struct vector4
{
    float x, y, z, w;
};
```

```
// Use sse to add the elements of two vectors a + b
vector4 sse_vector4_add(const vector4 &op_a, const vector4 &op_b)
{
    vector4 ret_vector;
    __asm
    {
        MOV        EAX, op_a          // Load pointers into CPU regs
        MOV        EBX, op_b
        MOVUPS     XMM0, [EAX]        // Move the vectors to SSE regs
        MOVUPS     XMM1, [EBX]
        ADDPS      XMM0, XMM1        // Add elements
        MOVUPS     [ret_vector], XMM0 // Save the return vector
    }
    return ret_vector;
}
```

## SSE (assembly examples)

We need to write a SSE code to calculate the cross product

$$R.x = A.y * B.z - A.z * B.y$$

$$R.y = A.z * B.x - A.x * B.z$$

$$R.z = A.x * B.y - A.y * B.x$$



Complete the following code

```
// Use sse to add the elements of two vectors a + b
vector4 sse_vector4_cross_product(const vector4 &op_a, const vector4 &op_b){
    vector4 ret_vector;
    __asm
    {
        MOV        EAX, op_a           // Load pointers into CPU regs
        MOV        EBX, op_b
        MOVUPS     XMM0, [EAX]        // Move the vectors to SSE regs
        MOVUPS     XMM1, [EBX]

        MOVUPS     [ret_vector], XMM0 // Save the return vector
    }
    return ret_vector;
}
```

## SSE (assembly examples)

We need to write a SSE code to calculate the cross product

$$R.x = A.y * B.z - A.z * B.y$$

$$R.y = A.z * B.x - A.x * B.z$$

$$R.z = A.x * B.y - A.y * B.x$$




Complete the following code

```
// Use sse to add the elements of two vectors a + b
vector4 sse_vector4_cross_product(const vector4 &op_a, const vector4 &op_b){
    vector4 ret_vector;
    __asm
    {
        MOV        EAX, op_a           // Load pointers into CPU regs
        MOV        EBX, op_b
        MOVUPS     XMM0, [EAX]         // Move the vectors to SSE regs
        MOVUPS     XMM1, [EBX]
        MOVAPS     XMM2, XMM0
        MOVAPS     XMM3, XMM1
        SHUFPS     XMM0, XMM0, 0xD8
        SHUFPS     XMM1, XMM1, 0xE1
        MULPS      XMM0, XMM1
        SHUFPS     XMM2, XMM2, 0xE1
        SHUFPS     XMM3, XMM3, 0xD8
        MULPS      XMM2, XMM3
        SUBPS      XMM0, XMM2
        MOVUPS     [ret_vector], XMM0 // Save the return vector
    }
    return ret_vector;
}
```


## SSE (common intrinsics)

```
__mm_add_ps(__m128 a, __m128 b)  
__mm_sub_ps(__m128 a, __m128 b)  
__mm_mul_ps(__m128 a, __m128 b)  
__mm_div_ps(__m128 a, __m128 b)  
__mm_sqrt_ps(__m128 a, __m128 b)  
__mm_min_ps(__m128 a, __m128 b)  
__mm_max_ps(__m128 a, __m128 b)  
  
__mm_cmpeq_ps(__m128 a, __m128 b)  
__mm_cmlt_ps(__m128 a, __m128 b)  
__mm_cmpgt_ps(__m128 a, __m128 b)  
  
__mm_and_ps(__m128 a, __m128 b)  
__mm_prefetch(__m128 a, _MM_HINT_T0)
```

; m1 = 

; m2 = 

m3 = `_mm_shuffle_ps(m1, m2, _MM_SHUFFLE(1,0,3,2))`

; m3 = 



Pros and cons of the prefetch.

## SSE (illustrations)

```
void scalar_sqrt(float *a){
    int i;
    for(i = 0; i < N; i++)
        a[i] = sqrt(a[i]);
}
```

scalar

```
void sse_sqrt(float *a){
    // We assume N % 4 == 0.
    int nb_iters = N / 4;
    __m128 *ptr = (__m128*)a;
    int i;
    for(i = 0; i < nb_iters; i++, ptr++, a += 4)
        _mm_store_ps(a, _mm_sqrt_ps(*ptr));
}
```

SSE

```
Tadonki@TADONKI-PC ~/vector
$ ./test
Running time of the scalar code: 0.286017
Running time of the SSE code: 0.031001
```

10 times faster !!!!!!!

## SSE (exercices)

? Write the SSE loop equivalent to the following scalar loop (use vectors mm\_d, mm\_a, mm\_b, mm\_c).

```
for(i = 0; i < N; i++)  
    d[i] = (a[i] - b[i])*c[i];
```

```
for( i = 0; i <N; i+= 4){  
    mm_a = _mm_load_ps(&a[i]);  
    mm_b = _mm_load_ps(&b[i]);  
    mm_c = _mm_load_ps(&c[i]);  
    mm_r = _mm_add_ps( mm_a, mm_b );  
    mm_a = _mm_mul_ps( mm_r , mm_c );  
    _mm_store_ps( &r[i], mm_a );  
}
```

? Write the SSE loop equivalent to the following scalar loop (use vectors mm\_c, mm\_a, mm\_b).

```
for(i = 0; i < N; i+= 2){  
    c[2*i] = (a[2*i] - b[2*i+1]);  
    c[2*i+1] = (a[2*i+1] - b[2*i]);  
}
```

? Write the SSE loop equivalent to the following scalar loop (typedef struct {float re; float im} complex;).

```
for(i = 0; i < N; i++) c[i] = multiply(a[i], b[i]);
```

? Write the SSE loop equivalent to the following scalar loop

```
for(i = 0; i < N; i++) b[i] = 2*a[i] + 1;
```

? Back to the cross product