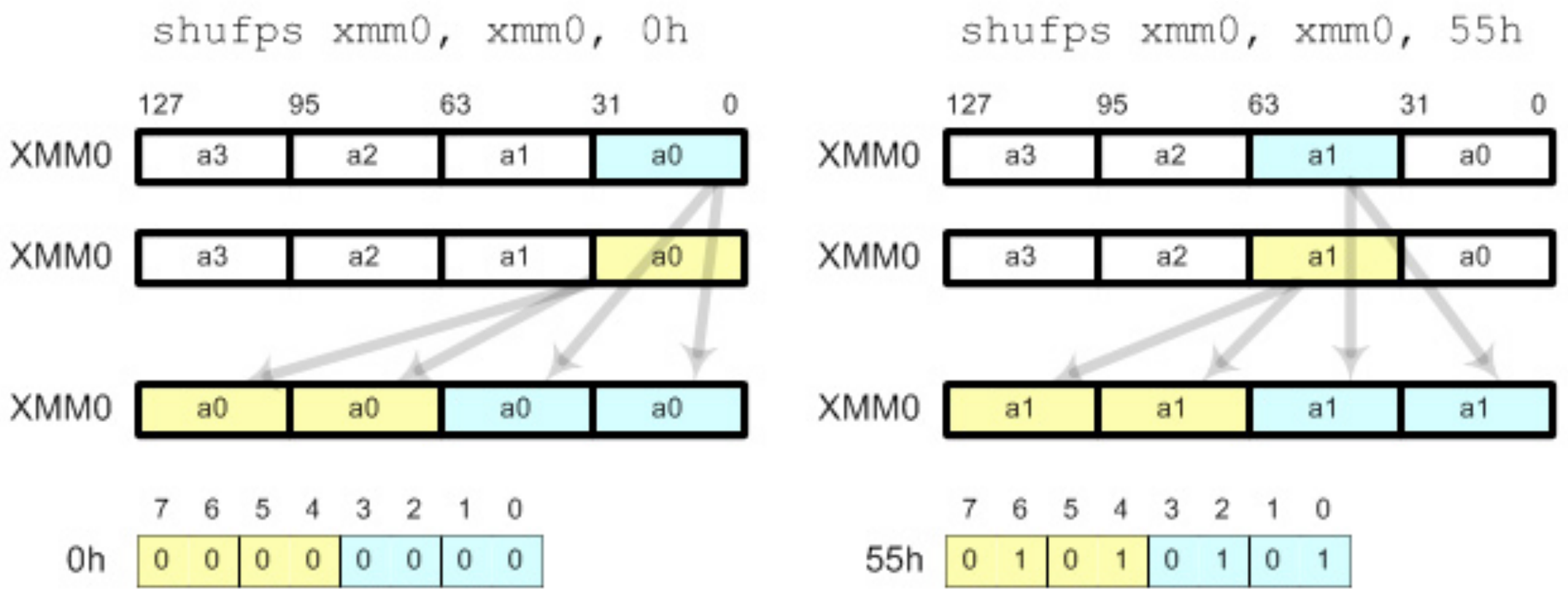


Hands-on SIMD

Eftychios Sifakis

CS758 Guest Lecture - 17 Oct 2012

Example 4-13. Simple Four-Iteration Loop

```
void add(float *a, float *b, float *c)
{
int i;
for (i = 0; i < 4; i++) {
    c[i] = a[i] + b[i];
  }
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

✓ Anything that ***can*** be done, can be coded up as inline assembly

✓ Maximum ***potential*** for performance accelerations

✓ Direct control over the code being generated

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

✓ Anything that ***can*** be done, can be coded up as inline assembly
✓ Maximum ***potential*** for performance accelerations
✓ Direct control over the code being generated

✗ Impractical for all but the smallest of kernels
✗ Not portable
✗ User needs to perform register allocation (and save old registers)
✗ User needs to (expertly) schedule instructions to hide latencies

Intrinsics

- A framework for generating assembly-level code without many of the drawbacks of inline assembly
  - Compiler (not programmer) takes care of register allocation
  - Compiler is able to schedule instructions to hide latencies
- Data types
  - Scalar : `float, double, unsigned int ...`
  - Vector : `__mm128, __m128d, __m256, __m256i ...`
- Intrinsic functions
  - Instruction wrappers : `_mm_add_pd, _mm256_mult_pd, _mm_xor_ps, _mm_sub_ss ...`
  - Macros : `_mm_set1_ps, _mm256_setzero_ps ...`
  - Math Wrappers : `_mm_log_ps, _mm256_pow_pd ...`

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-14. Streaming SIMD Extensions Using Inlined Assembly Encoding

```
void add(float *a, float *b, float *c)
{
  __asm {
    mov     eax, a
    mov     edx, b
    mov     ecx, c
    movaps  xmm0, XMMWORD PTR [eax]
    addps   xmm0, XMMWORD PTR [edx]
    movaps  XMMWORD PTR [ecx], xmm0
  }
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly

✓ Somewhat portable

✓ Compiler takes care of register allocation (and spill, if needed)

✓ Compiler will shuffle & schedule instructions to best hide latencies

✓ Relatively easy migration from SSE -> AVX -> MIC ....

Example 4-15. Simple Four-Iteration Loop Coded with Intrinsics

```
#include <xmmintrin.h>
void add(float *a, float *b, float *c)
{
    __m128 t0, t1;
    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

✓ Almost as flexible as inline assembly
✓ Somewhat portable
✓ Compiler takes care of register allocation (and spill, if needed)
✓ Compiler will shuffle & schedule instructions to best hide latencies
✓ Relatively easy migration from SSE -> AVX -> MIC ....

✗ Coding large kernels is still challenging and bug-prone
✗ Un-natural notation (vs. C++ expressions and operators)
✗ SSE code is ***similar*** to AVX code, but different enough so that 2 distinct versions must be written
✗ Vector code looks very different than scalar code

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓Minimal effort required (assuming it works ...)

✓Development of SIMD code is no different than scalar code

✓Ability to use complex C++ expressions

✓Larger kernels are easier to tackle

Example 4-17. Automatic Vectorization for a Simple Loop

```
void add (float *restrict a,
          float *restrict b,
          float *restrict c)
{
    int i;
    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

✓Minimal effort required (assuming it works ...)
✓Development of SIMD code is no different than scalar code
✓Ability to use complex C++ expressions
✓Larger kernels are easier to tackle

✗ In practice it can be ***very*** challenging to achieve efficiency comparable to assembly/intrinsics
✗ Compilers are ***very*** conservative when vectorizing, for the risk of jeopardizing scalar equivalence
✗ The no-aliasing restriction might run contrary to the spirit of certain kernels

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

✓Fewer visual differences between vector and scalar code

✓Ability to use complex C++ expressions (assuming wrapper types have been overloaded)

✓Easy transition to different vector widths

Example 4-16. C++ Code Using the Vector Classes

```
#include <fvec.h>
void add(float *a, float *b, float *c)
{
        F32vec4 *av=(F32vec4 *) a;
        F32vec4 *bv=(F32vec4 *) b;
        F32vec4 *cv=(F32vec4 *) c;
                *cv=*av + *bv;
}
```

✓Fewer visual differences between vector and scalar code

✓Ability to use complex C++ expressions (assuming wrapper types have been overloaded)

✓Easy transition to different vector widths

✗ Heavy dependence on the compiler for eliminating temporaries (but it typically does a really good job at it)

✗ Limited to the semantics of the built-in vector wrapper classes (but we are free to extend those)

✗ Risk of more bloated executable code than by using intrinsics

```
template<class Tw,class T_DATA,int width>
void Matrix_Times_Transpose(const T_DATA (&A)[3][3], const T_DATA (&B)[3][3],
    T_DATA (&C)[3][3]){

    typedef Number<Tw> Tn;
    Tn rC11,rC21,rC31,rC12,rC22,rC32,rC13,rC23,rC33,rA1,rA2,rA3,rB;

    Vector3<Tn&> vA(rA1,rA2,rA3);
    Vector3<Tn&> vC1(rC11,rC21,rC31);
    Vector3<Tn&> vC2(rC12,rC22,rC32);
    Vector3<Tn&> vC3(rC13,rC23,rC33);

    rA1.Load(A[0][0]);
    rA2.Load(A[1][0]);
    rA3.Load(A[2][0]);

    rB.Load(B[0][0]);vC1=vA*rB;
    rB.Load(B[1][0]);vC2=vA*rB;
    rB.Load(B[2][0]);vC3=vA*rB;

    for(int k=1;k<3;k++){
        rA1.Load(A[0][k]);
        rA2.Load(A[1][k]);
        rA3.Load(A[2][k]);

        rB.Load(B[0][k]);vC1=vC1+vA*rB;
        rB.Load(B[1][k]);vC2=vC2+vA*rB;
        rB.Load(B[2][k]);vC3=vC3+vA*rB;
    }
    Store(C[0][0],rC11);Store(C[1][0],rC21);Store(C[2][0],rC31);
    Store(C[0][1],rC12);Store(C[1][1],rC22);Store(C[2][1],rC32);
    Store(C[0][2],rC13);Store(C[1][2],rC23);Store(C[2][2],rC33);
}
```

```
template void Matrix_Times_Transpose<float,float,1>(const float (&A)[3][3],
    const float (&B)[3][3], float (&C)[3][3]);

template void Matrix_Times_Transpose<float,float[8],1>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);

template void Matrix_Times_Transpose<__m128,float[8],4>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);

template void Matrix_Times_Transpose<__m256,float[8],8>(const float (&A)[3][3][8],
    const float (&B)[3][3][8], float (&C)[3][3][8]);
```

```
template<class Tw>
class Number
{
    Tw value;
public:
    Number();

    Number operator+(const Number& other) const;
    Number operator*(const Number& other) const;
    void Load(const float* data);
    void Load(const float& data);

    friend void Store<>(float* data,const Number& number);
    friend void Store<>(float& data,const Number& number);
};

template<>
Number<float>::Number()
{value=0.;}

template<>
Number<__m128>::Number()
{value=_mm_xor_ps(value,value);}

template<>
Number<__m256>::Number()
{value=_mm256_xor_ps(value,value);}

template<>
Number<float> Number<float>::operator+(const Number& other) const
{Number<float> result;result.value=value+other.value;return result;}

template<>
Number<__m128> Number<__m128>::operator+(const Number& other) const
```

```
template<>
Number<__m128> Number<__m128>::operator+(const Number& other) const
{Number<__m128> result;result.value=_mm_add_ps(value,other.value);return result;}

template<>
Number<__m256> Number<__m256>::operator+(const Number& other) const
{Number<__m256> result;result.value=_mm256_add_ps(value,other.value);return result;}

template<>
Number<float> Number<float>::operator*(const Number& other) const
{Number<float> result;result.value=value*other.value;return result;}

template<>
Number<__m128> Number<__m128>::operator*(const Number& other) const
{Number<__m128> result;result.value=_mm_mul_ps(value,other.value);return result;}

template<>
Number<__m256> Number<__m256>::operator*(const Number& other) const
{Number<__m256> result;result.value=_mm256_mul_ps(value,other.value);return result;}

template<>
void Number<float>::Load(const float* data)
{value=*data;}

template<>
void Number<__m128>::Load(const float* data)
{value=_mm_loadu_ps(data);}

template<>
void Number<__m256>::Load(const float* data)
{value=_mm256_loadu_ps(data);}

template<>
void Number<float>::Load(const float& data)
```

```
template<>
void Number<float>::Load(const float& data)
{value=data;}

template<>
void Number<__m128>::Load(const float& data)
{value=_mm_loadu_ps(&data);}

template<>
void Number<__m256>::Load(const float& data)
{value=_mm256_loadu_ps(&data);}

template<>
void Store(float* data,const Number<float>& number)
{*data=number.value;}

template<>
void Store(float* data,const Number<__m128>& number)
{_mm_storeu_ps(data,number.value);}

template<>
void Store(float* data,const Number<__m256>& number)
{_mm256_storeu_ps(data,number.value);}

template<>
void Store(float& data,const Number<float>& number)
{data=number.value;}

template<>
void Store(float& data,const Number<__m128>& number)
{_mm_storeu_ps(&data,number.value);}

template<>
void Store(float& data,const Number<__m256>& number)
```

```
template<>
void Store(float& data,const Number<float>& number)
{data=number.value;}

template<>
void Store(float& data,const Number<__m128>& number)
{_mm_storeu_ps(&data,number.value);}

template<>
void Store(float& data,const Number<__m256>& number)
{_mm256_storeu_ps(&data,number.value);}
```

```
template<class Tn> struct Vector3
{
    Tn x;
    Tn y;
    Tn z;

    {...}
};
```

```
..___tag_value__Z22Matrix_Times_TransposeI6__m256A8_fLi8EEvPA3_KT0_S5_PA3_S2_.7:
#10.1
        vmovups   (%rdi), %ymm1                                 #35.13
        vmovups   96(%rdi), %ymm0                               #36.13
        vmovups   192(%rdi), %ymm14                             #37.13
        vmovups   96(%rsi), %ymm11                              #42.12
        vmovups   32(%rsi), %ymm6                               #54.16
        vmovups   (%rsi), %ymm12                                #39.12
        vmovups   192(%rsi), %ymm4                              #45.12
        vmulps    %ymm11, %ymm1, %ymm8                          #43.16
        vmulps    %ymm11, %ymm0, %ymm9                          #43.16
        vmulps    %ymm11, %ymm14, %ymm10                        #43.16
        vmovups   128(%rdi), %ymm11                             #51.17
        vmulps    %ymm12, %ymm1, %ymm2                          #40.16
        vmulps    %ymm12, %ymm0, %ymm5                          #40.16
        vmulps    %ymm12, %ymm14, %ymm7                         #40.16
        vmulps    %ymm4, %ymm1, %ymm13                          #46.16
        vmulps    %ymm4, %ymm0, %ymm15                          #46.16
        vmulps    %ymm4, %ymm14, %ymm0                          #46.16
        vmovups   32(%rdi), %ymm14                              #50.17
        vmovups   224(%rdi), %ymm12                             #52.17
        vmulps    %ymm6, %ymm11, %ymm4                          #55.24
        vmulps    %ymm6, %ymm14, %ymm3                          #55.24
        vmulps    %ymm6, %ymm12, %ymm6                          #55.24
        vaddps    %ymm4, %ymm5, %ymm4                           #55.24
        vaddps    %ymm3, %ymm2, %ymm2                           #55.24
        vaddps    %ymm6, %ymm7, %ymm6                           #55.24
        vmovups   128(%rsi), %ymm5                              #57.16
        vmulps    %ymm5, %ymm14, %ymm7                          #58.24
        vmulps    %ymm5, %ymm11, %ymm1                          #58.24
        vmulps    %ymm5, %ymm12, %ymm3                          #58.24
        vaddps    %ymm7, %ymm8, %ymm7                           #58.24
```

```
vmulps    %ymm5, %ymm12, %ymm3                    #58.24
vaddps    %ymm7, %ymm8, %ymm7                     #58.24
vaddps    %ymm1, %ymm9, %ymm5                     #58.24
vaddps    %ymm3, %ymm10, %ymm3                    #58.24
vmovups   224(%rsi), %ymm8                        #60.16
vmulps    %ymm8, %ymm12, %ymm12                   #61.24
vmulps    %ymm8, %ymm14, %ymm9                    #61.24
vmulps    %ymm8, %ymm11, %ymm10                   #61.24
vmovups   64(%rsi), %ymm14                        #54.16
vaddps    %ymm12, %ymm0, %ymm12                   #61.24
vaddps    %ymm9, %ymm13, %ymm1                    #61.24
vaddps    %ymm10, %ymm15, %ymm11                  #61.24
vmovups   160(%rdi), %ymm0                        #51.17
vmovups   64(%rdi), %ymm15                        #50.17
vmovups   256(%rdi), %ymm13                       #52.17
vmulps    %ymm14, %ymm0, %ymm9                    #55.24
vmulps    %ymm14, %ymm15, %ymm8                   #55.24
vmulps    %ymm14, %ymm13, %ymm10                  #55.24
vaddps    %ymm9, %ymm4, %ymm4                     #55.24
vaddps    %ymm8, %ymm2, %ymm2                     #55.24
vaddps    %ymm10, %ymm6, %ymm6                    #55.24
vmovups   160(%rsi), %ymm9                        #57.16
vmulps    %ymm9, %ymm13, %ymm8                    #58.24
vmulps    %ymm9, %ymm15, %ymm10                   #58.24
vmulps    %ymm9, %ymm0, %ymm14                    #58.24
vaddps    %ymm8, %ymm3, %ymm3                     #58.24
vaddps    %ymm10, %ymm7, %ymm7                    #58.24
vaddps    %ymm14, %ymm5, %ymm5                    #58.24
vmovups   256(%rsi), %ymm8                        #60.16
vmovups   %ymm2, (%rdx)                           #65.9
vmovups   %ymm4, 96(%rdx)                         #66.9
vmovups   %ymm6, 192(%rdx)                        #67.9
vmovups   %ymm7, 32(%rdx)                         #69.9
```

```
        vmovups   %ymm6, 192(%rdx)                                  #67.9
        vmovups   %ymm7, 32(%rdx)                                   #69.9
        vmulps    %ymm8, %ymm15, %ymm15                             #61.24
        vmulps    %ymm8, %ymm0, %ymm0                               #61.24
        vmulps    %ymm8, %ymm13, %ymm13                             #61.24
        vmovups   %ymm5, 128(%rdx)                                  #70.9
        vmovups   %ymm3, 224(%rdx)                                  #71.9
        vaddps    %ymm15, %ymm1, %ymm1                              #61.24
        vaddps    %ymm0, %ymm11, %ymm0                              #61.24
        vaddps    %ymm13, %ymm12, %ymm8                             #61.24
        vmovups   %ymm1, 64(%rdx)                                   #73.9
        vmovups   %ymm0, 160(%rdx)                                  #74.9
        vmovups   %ymm8, 256(%rdx)                                  #75.9
        ret                                                         #79.1
        .align    16,0x90
..___tag_value__Z22Matrix_Times_TransposeI6__m256A8_fLi8EEvPA3_KT0_S5_PA3_S2_.9: #
```

Circumventing branches

```
for(int i=0;i<4;i++){
   if(a[i]<b[i])
      c=a[i]+b[i];
   else
      c=a[i]-b[i];
}
```

Circumventing branches

```
for(int i=0;i<4;i++){
   if(a[i]<b[i])
      c=a[i]+b[i];
   else
      c=a[i]-b[i];
}
```

```
bool cond[4];
float c1[4],c2[4];

for(int i=0;i<4;i++){
   cond[i]=(a[i]<b[i]);
   c1[i]=a[i]+b[i];
   c2[i]=a[i]-b[i];
   c[i]=cond[i]?c1[i]:c2[i];
}
```

Circumventing branches

```
bool cond[4];
float c1[4],c2[4];

for(int i=0;i<4;i++){
  cond[i]=(a[i]<b[i]);
  c1[i]=a[i]+b[i];
  c2[i]=a[i]-b[i];
  c[i]=cond[i]?c1[i]:c2[i];
}
```

SSE3 or later :

```
__m128 a,b,c,c1,c2,cond;

for(int i=0;i<4;i++){
  cond=_mm_cmple_ps(a,b);
  c1=_mm_add_ps(a,b);
  c2=_mm_sub_ps(a,b);
  c=_mm_blendv_ps(cond,c1,c2);
}
```

Circumventing branches

SSE3 or later :

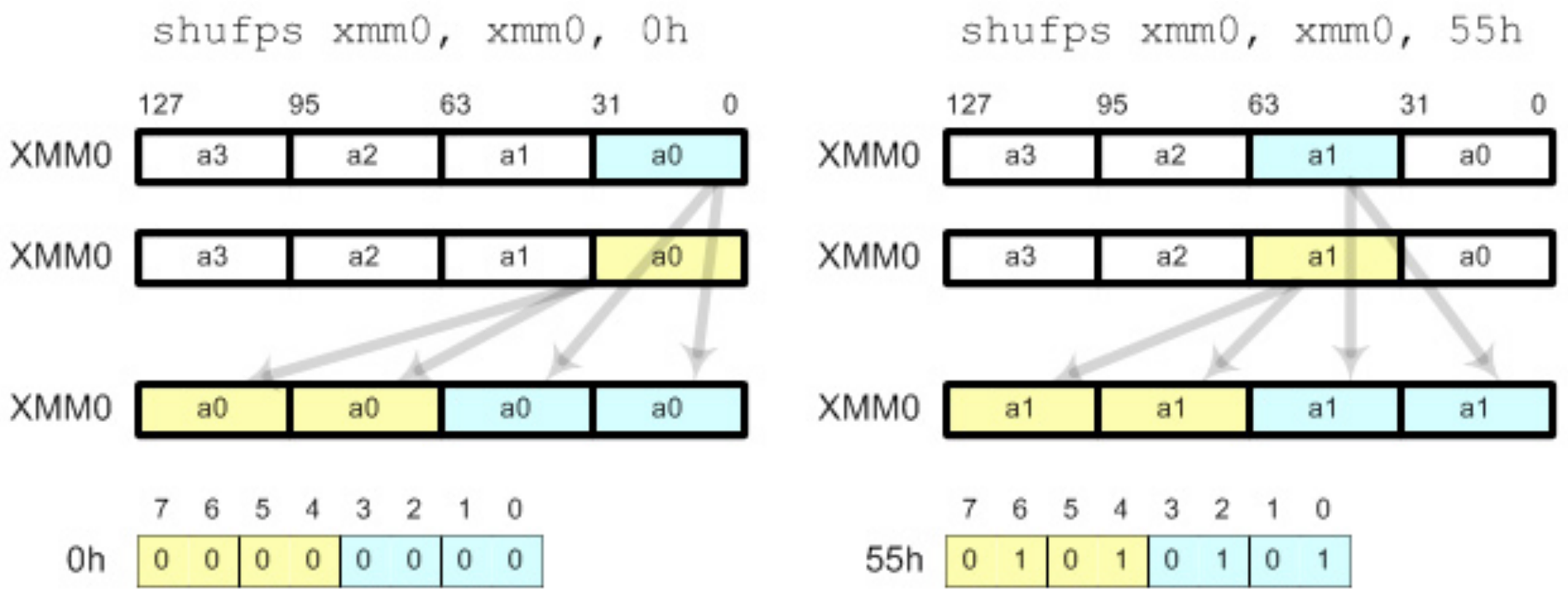
```
__m128 a,b,c,c1,c2,cond;

for(int i=0;i<4;i++){
  cond=_mm_cmple_ps(a,b);
  c1=_mm_add_ps(a,b);
  c2=_mm_sub_ps(a,b);
  c=_mm_blendv_ps(cond,c1,c2);
}
```

SSE2 or earlier :

```
__m128 a,b,c,c1,c2,cond;

for(int i=0;i<4;i++){
  cond=_mm_cmple_ps(a,b);
  c1=_mm_add_ps(a,b);
  c2=_mm_sub_ps(a,b);
  c1=_mm_and_ps(cond,c1);
  c2=_mm_andnot_ps(cond,c2);
  c=_mm_or_ps(c1,c2);
}
```



Hands-on SIMD

Eftychios Sifakis

CS758 Guest Lecture - 17 Oct 2012