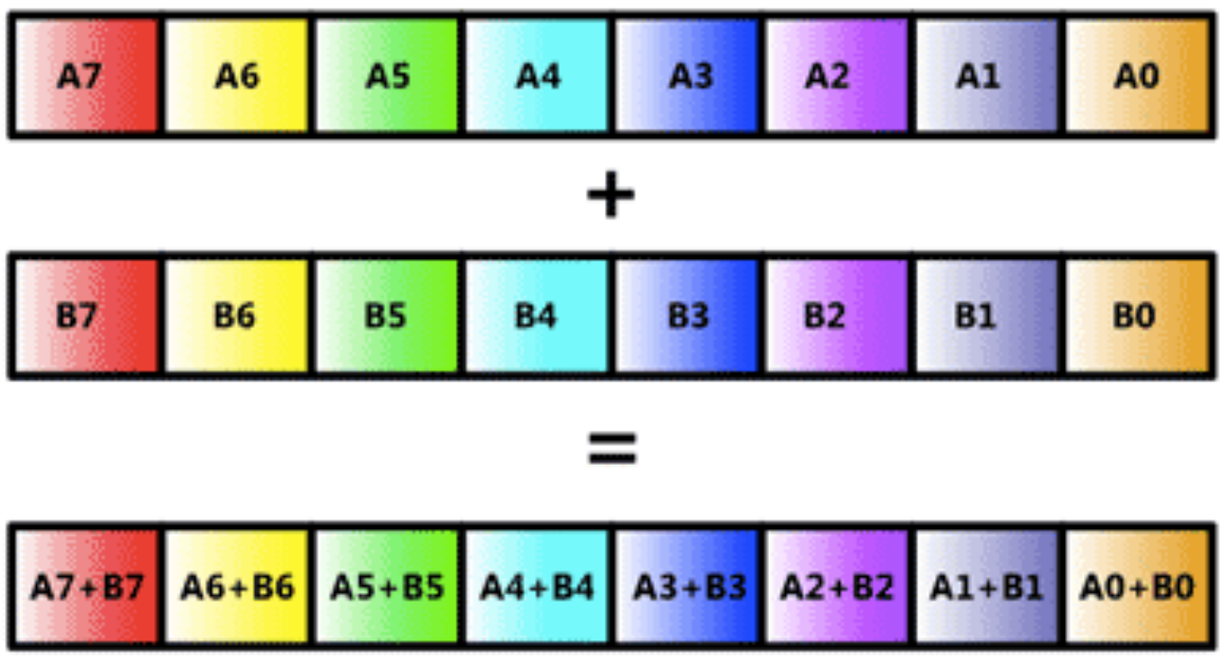
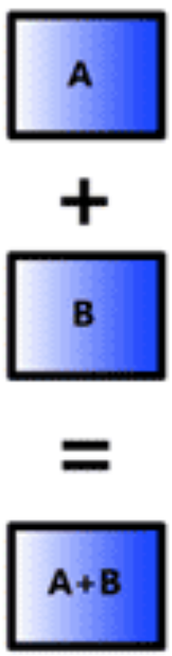
SIMD Parallelization

High Performance Scientific Computing 2018
Rio Yokota

SIMD

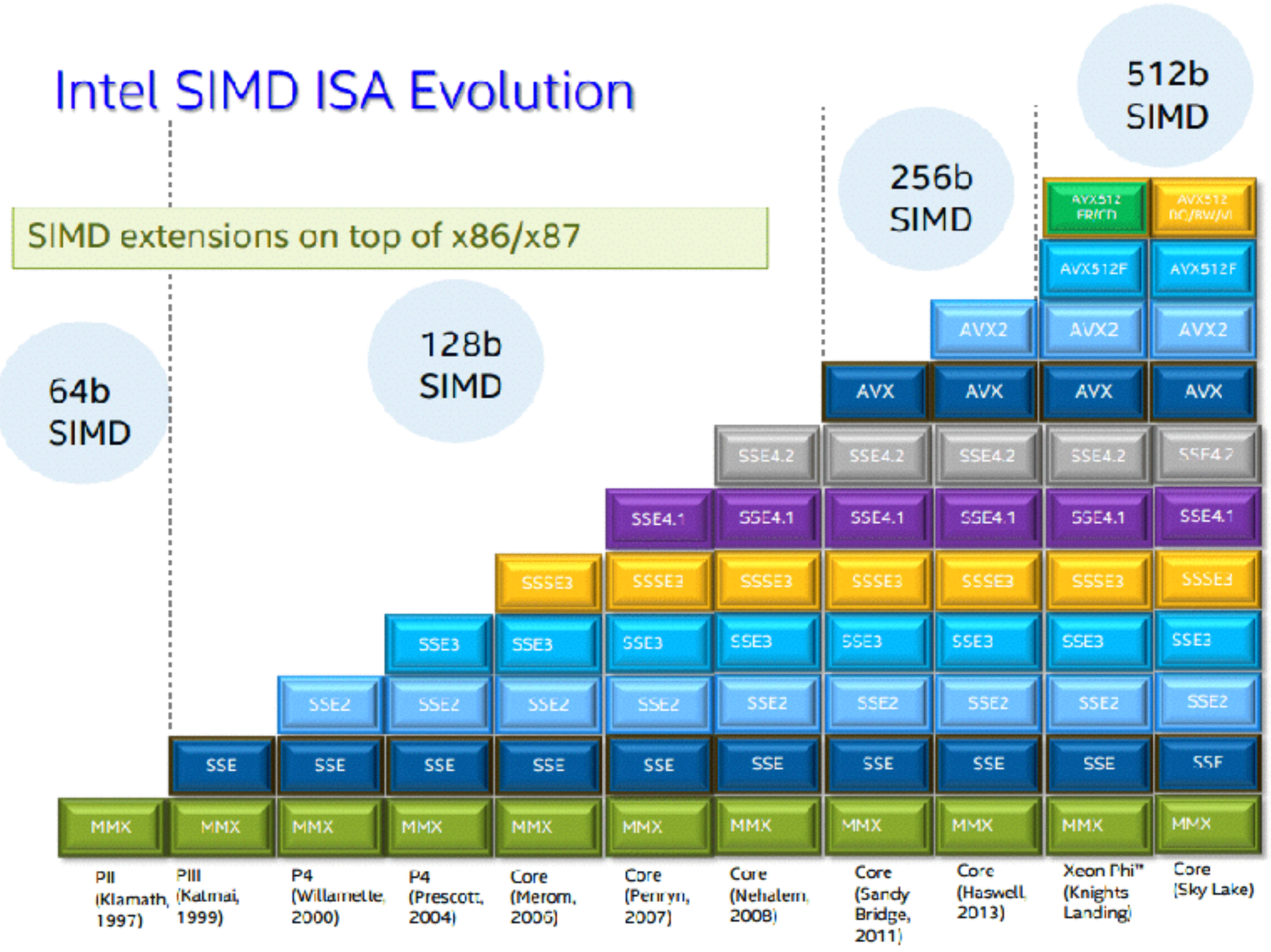





```
__mm256 x, y;
_mm256_add_ps(x, y);
```

```
float x, y;
x + y;
```

SIMD Instruction Set Evolution

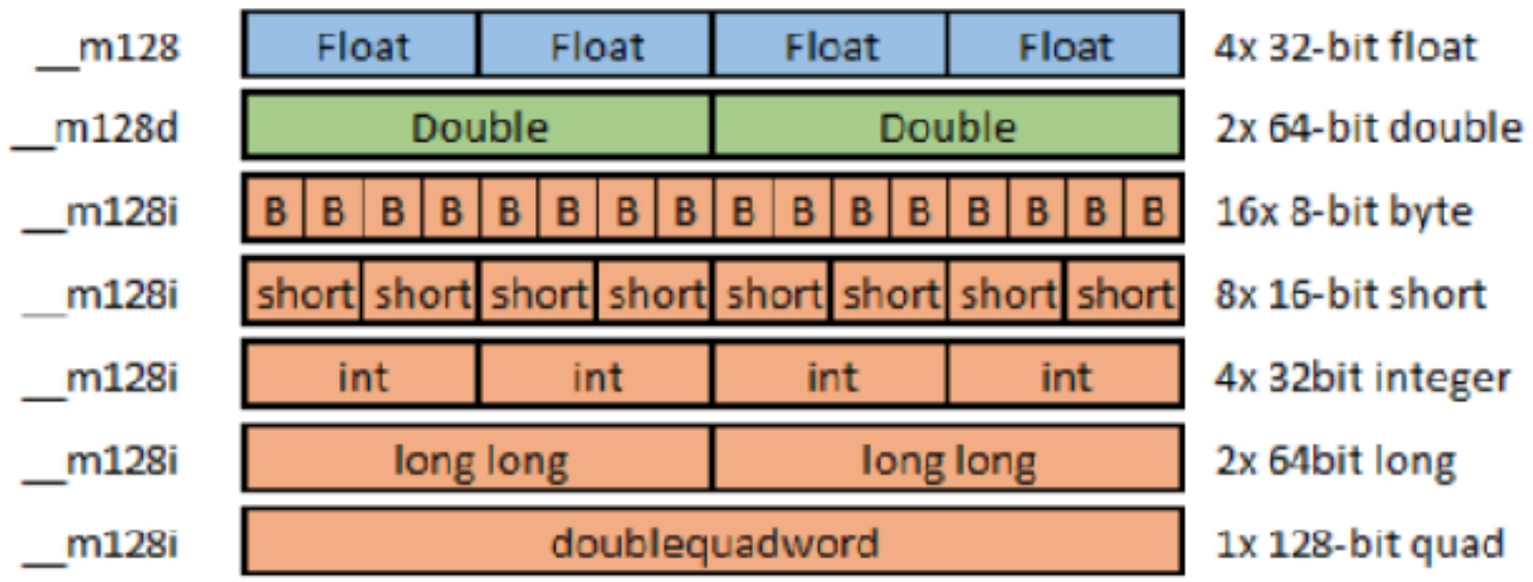


Header files

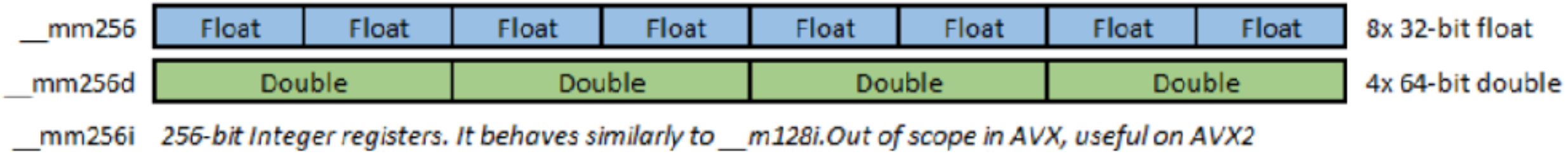
MMX	**<mmintrin.h>**
SSE	<xmmintrin.h>
SSE2	<emmintrin.h>
SSE3	<pmmintrin.h>
SSSE3	<tmmintrin.h>
SSE4.1	<smmintrin.h>
SSE4.2	<nmmintrin.h>
AES	<wmmintrin.h>
AVX, AVX2, FMA	<immintrin.h>
AVX-512	~~<zmmintrin.h>~~
ARM NEON	<arm_neon.h>

Data types

SSE Data Types (16 XMM Registers)



AVX Data Types (16 YMM Registers)

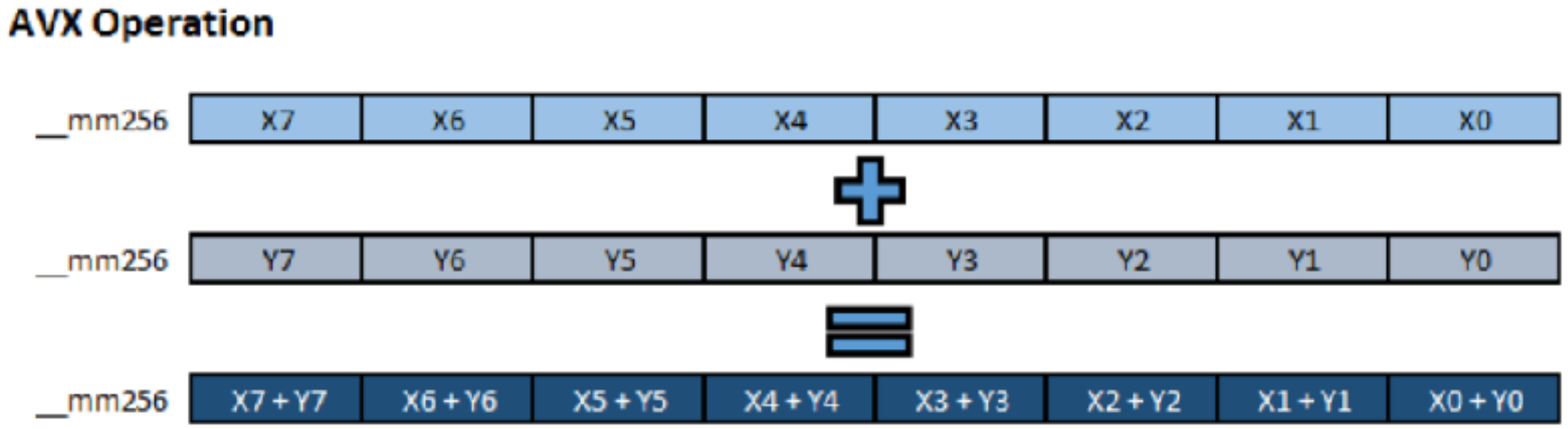


Functions

acos	ceil	fabs	round
acosh	Cos	floor	sin
asin	Cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	Erfc	log	tan
atan2	Erfinv	log10	tanh
atanh	Exp	log2	trunc
cbrt	exp2	pow	rsqrt

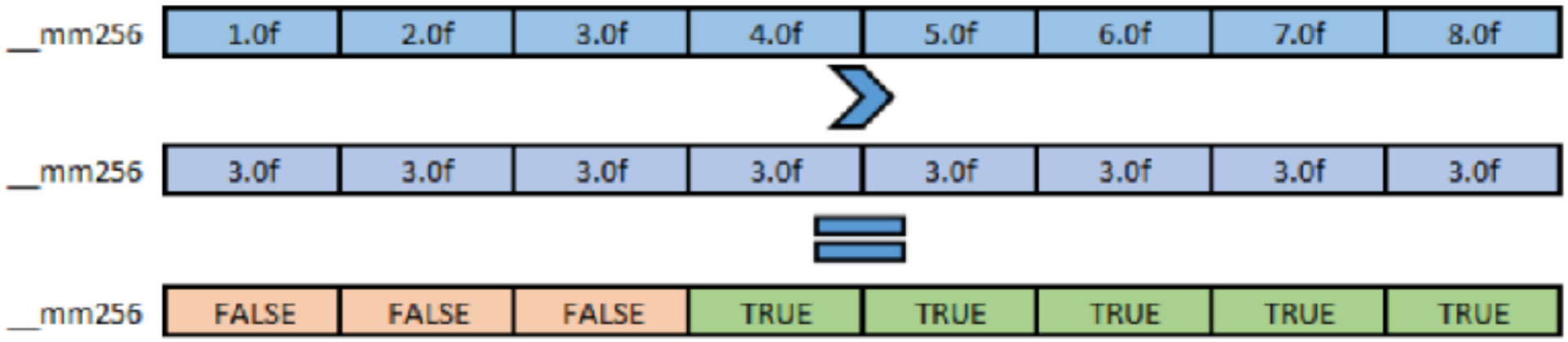
```
float
_mm_*_ps()
_mm256_*_ps()
_mm512_*_ps()

double
_mm_*_pd()
_mm256_*_pd()
_mm512_*_pd()
```

Addition



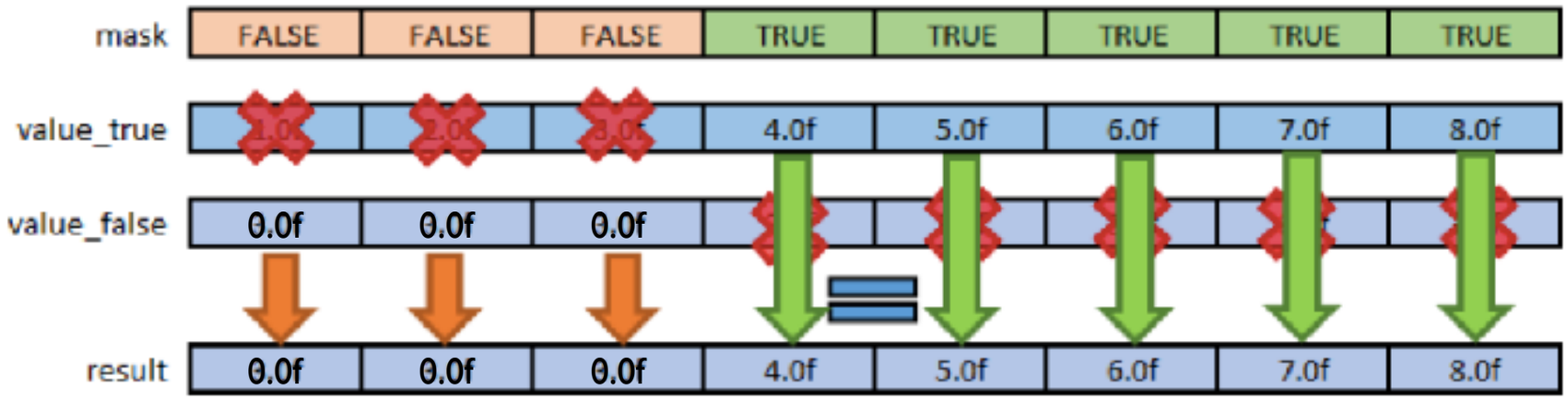

```
int main() {
  float a[8];
  for (int i=0; i<8; i++) a[i] = i + 1;
  __m256 b = _mm256_load_ps(a);
  b = _mm256_add_ps(b, b);
  _mm256_store_ps(a, b);
  for (int i=0; i<8; i++) printf("%f\n",a[i]);
}
```

Greater than




```
#include <cstdio>
#include <immintrin.h>
int main() {
  float a[8];
  for (int i=0; i<8; i++) a[i] = i + 1;
  __m256 b = _mm256_load_ps(a);
  __m256 three = _mm256_set1_ps(3);
  __m256 mask = _mm256_cmp_ps(b, three, _CMP_GT_OQ);
  int i = _mm256_movemask_ps(mask);
  printf("%d\n",i);
}
```

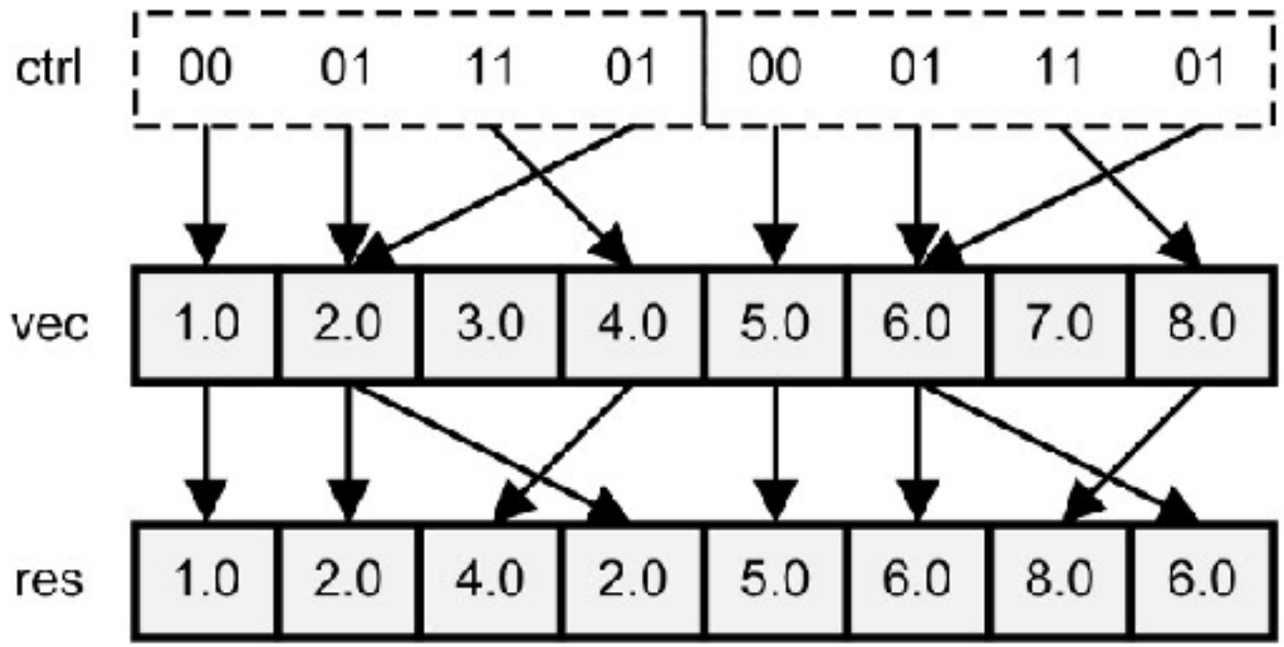
Conditional load




```
int main() {
  float a[8];
  for (int i=0; i<8; i++) a[i] = i + 1;
  __m256 b = _mm256_load_ps(a);
  __m256 three = _mm256_set1_ps(3);
  __m256 mask = _mm256_cmp_ps(b, three, _CMP_GT_OQ);
  __m256i maski = _mm256_castps_si256(mask);
  b = _mm256_maskload_ps(a, maski);
  _mm256_store_ps(a, b);
  for (int i=0; i<8; i++) printf("%f\n",a[i]);
}
```

Shuffle

```
res = _mm256_permute_ps(vec, 0b01110100)
```




```
int main() {
  float a[8];
  for (int i=0; i<8; i++) a[i] = i + 1;
  __m256 b = _mm256_load_ps(a);
  b = _mm256_permute_ps(b,0b01110100);
  _mm256_store_ps(a, b);
  for (int i=0; i<8; i++) printf("%f\n",a[i]);
}
```

Horizontal add

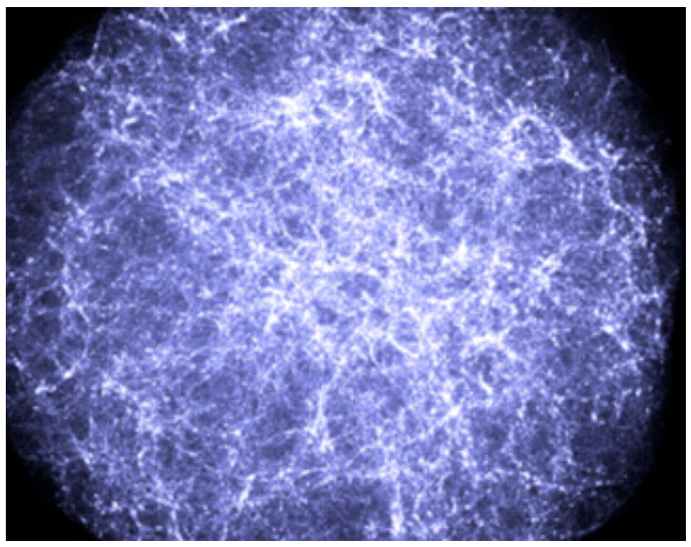
1	2	3	4	5	6	7	8

1+2+3+4+5+6+7+8=36

36	36	36	36	36	36	36	36

```
int main() {
  float a[8];
  for (int i=0; i<8; i++) a[i] = i + 1;
  __m256 b = _mm256_load_ps(a);
  __m256 c = _mm256_permute2f128_ps(b,b,1);
  c = _mm256_add_ps(c,b);
  c = _mm256_hadd_ps(c,c);
  c = _mm256_hadd_ps(c,c);
  _mm256_store_ps(a, c);
  for (int i=0; i<8; i++) printf("%f\n",a[i]);
}
```

10_nbody.cpp



```
#pragma omp parallel for
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      float dx = x[i] - x[j];
      float dy = y[i] - y[j];
      float dz = z[i] - z[j];
      float r2 = dx * dx + dy * dy + dz * dz;
      float invR = 1.0f / sqrtf(r2+1e-6);
      p[i] += m[j] * invR;
    }
  }
```

SIMD N-body

```
#pragma omp parallel for
  for (int i=0; i<N; i+=4) {
    __m128 pi = _mm_setzero_ps();
    __m128 axi = _mm_setzero_ps();
    __m128 ayi = _mm_setzero_ps();
    __m128 azi = _mm_setzero_ps();
    __m128 xi = _mm_load_ps(x+i);
    __m128 yi = _mm_load_ps(y+i);
    __m128 zi = _mm_load_ps(z+i);
    for (int j=0; j<N; j++) {
      __m128 R2 = _mm_set1_ps(EPS2);
      __m128 x2 = _mm_set1_ps(x[j]);
      x2 = _mm_sub_ps(x2, xi);
      __m128 y2 = _mm_set1_ps(y[j]);
      y2 = _mm_sub_ps(y2, yi);
      __m128 z2 = _mm_set1_ps(z[j]);
      z2 = _mm_sub_ps(z2, zi);
      __m128 xj = x2;
      x2 = _mm_mul_ps(x2, x2);
      R2 = _mm_add_ps(R2, x2);
      __m128 yj = y2;
      y2 = _mm_mul_ps(y2, y2);
      R2 = _mm_add_ps(R2, y2);
      __m128 zj = z2;
      z2 = _mm_mul_ps(z2, z2);
      R2 = _mm_add_ps(R2, z2);
      __m128 mj = _mm_set1_ps(m[j]);
      __m128 invR = _mm_rsqrt_ps(R2);
      mj = _mm_mul_ps(mj, invR);
      pi = _mm_add_ps(pi, mj);
      invR = _mm_mul_ps(invR, invR);
      invR = _mm_mul_ps(invR, mj);
      xj = _mm_mul_ps(xj, invR);
      axi = _mm_add_ps(axi, xj);
      yj = _mm_mul_ps(yj, invR);
      ayi = _mm_add_ps(ayi, yj);
      zj = _mm_mul_ps(zj, invR);
      azi = _mm_add_ps(azi, zj);
    }
    _mm_store_ps(p+i, pi);
    _mm_store_ps(ax+i, axi);
    _mm_store_ps(ay+i, ayi);
    _mm_store_ps(az+i, azi);
  }
```

```
#pragma omp parallel for
  for (int i=0; i<N; i+=8) {
    __m256 pi = _mm256_setzero_ps();
    __m256 axi = _mm256_setzero_ps();
    __m256 ayi = _mm256_setzero_ps();
    __m256 azi = _mm256_setzero_ps();
    __m256 xi = _mm256_load_ps(x+i);
    __m256 yi = _mm256_load_ps(y+i);
    __m256 zi = _mm256_load_ps(z+i);
    for (int j=0; j<N; j++) {
      __m256 R2 = _mm256_set1_ps(1e-6);
      __m256 x2 = _mm256_set1_ps(x[j]);
      x2 = _mm256_sub_ps(x2, xi);
      __m256 y2 = _mm256_set1_ps(y[j]);
      y2 = _mm256_sub_ps(y2, yi);
      __m256 z2 = _mm256_set1_ps(z[j]);
      z2 = _mm256_sub_ps(z2, zi);
      __m256 xj = x2;
      x2 = _mm256_mul_ps(x2, x2);
      R2 = _mm256_add_ps(R2, x2);
      __m256 yj = y2;
      y2 = _mm256_mul_ps(y2, y2);
      R2 = _mm256_add_ps(R2, y2);
      __m256 zj = z2;
      z2 = _mm256_mul_ps(z2, z2);
      R2 = _mm256_add_ps(R2, z2);
      __m256 mj = _mm256_set1_ps(m[j]);
      __m256 invR = _mm256_rsqrt_ps(R2);
      mj = _mm256_mul_ps(mj, invR);
      pi = _mm256_add_ps(pi, mj);
      invR = _mm256_mul_ps(invR, invR);
      invR = _mm256_mul_ps(invR, mj);
      xj = _mm256_mul_ps(xj, invR);
      axi = _mm256_add_ps(axi, xj);
      yj = _mm256_mul_ps(yj, invR);
      ayi = _mm256_add_ps(ayi, yj);
      zj = _mm256_mul_ps(zj, invR);
      azi = _mm256_add_ps(azi, zj);
    }
    _mm256_store_ps(p+i, pi);
    _mm256_store_ps(ax+i, axi);
    _mm256_store_ps(ay+i, ayi);
    _mm256_store_ps(az+i, azi);
  }
```

```
#pragma omp parallel for
  for (int i=0; i<N; i+=16) {
    __m512 pi = _mm512_setzero_ps();
    __m512 axi = _mm512_setzero_ps();
    __m512 ayi = _mm512_setzero_ps();
    __m512 azi = _mm512_setzero_ps();
    __m512 xi = _mm512_load_ps(x+i);
    __m512 yi = _mm512_load_ps(y+i);
    __m512 zi = _mm512_load_ps(z+i);
    for (int j=0; j<N; j++) {
      __m512 xj = _mm512_set1_ps(x[j]);
      xj = _mm512_sub_ps(xj, xi);
      __m512 yj = _mm512_set1_ps(y[j]);
      yj = _mm512_sub_ps(yj, yi);
      __m512 zj = _mm512_set1_ps(z[j]);
      zj = _mm512_sub_ps(zj, zi);
      __m512 R2 = _mm512_set1_ps(EPS2);
      R2 = _mm512_fmadd_ps(xj, xj, R2);
      R2 = _mm512_fmadd_ps(yj, yj, R2);
      R2 = _mm512_fmadd_ps(zj, zj, R2);
      __m512 mj = _mm512_set1_ps(m[j]);
      __m512 invR = _mm512_rsqrt14_ps(R2);
      mj = _mm512_mul_ps(mj, invR);
      pi = _mm512_add_ps(pi, mj);
      invR = _mm512_mul_ps(invR, invR);
      invR = _mm512_mul_ps(invR, mj);
      axi = _mm512_fmadd_ps(xj, invR, axi);
      ayi = _mm512_fmadd_ps(yj, invR, ayi);
      azi = _mm512_fmadd_ps(zj, invR, azi);
    }
    _mm512_store_ps(p+i, pi);
    _mm512_store_ps(ax+i, axi);
    _mm512_store_ps(ay+i, ayi);
    _mm512_store_ps(az+i, azi);
  }
```

Operator overloading

```
template<int N, typename T>
class vec {};

template<>
class vec<8,float> {
 __m256 data;
const vec &operator+=(const vec & v) {
  data = _mm256_add_ps(data,v.data);
  return *this;
};
```

Agner Fog
http://www.agner.org/optimize/