Multicore Architecture Programming – SIMD II

Olivier Aumage

Inria & LaBRI lab.

olivier.aumage @ inria.fr

2024 – 2025

Practical Exercises

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy (x + y)**

- **axpy (α.x+y)**

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy** (x + y)
  - Load, store
  - Addition, element per element
- **axpy** (α.x+y)

```
void xpy (const float *x, const float *y, float *z,
          const int vector_size)
{
  /* Check that the vector size is a multiple of the number
     of elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_add_ps(reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File xpy.c — routine xpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy (x + y)**
  - Load, store
  - Addition, element per element
- **axpy (α.x+y)**

```
void xpy (const float *x, const float *y, float *z,
          const int vector_size)
{
  /* Check that the vector size is a multiple of the number
     of elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_add_ps(reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File xpy.c — routine xpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy** (x + y)
  - Load, store
  - Addition, element per element
- **axpy** (α.x+y)

```
void xpy (const float *x, const float *y, float *z,
          const int vector_size)
{
  /* Check that the vector size is a multiple of the number
     of elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_add_ps(reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File xpy.c — routine xpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy** (x + y)
  - Load, store
  - Addition, element per element
- **axpy** (α.x+y)

```
void xpy (const float *x, const float *y, float *z,
          const int vector_size)
{
  /* Check that the vector size is a multiple of the number
     of elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_add_ps(reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File xpy.c — routine xpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy** (x + y)
  - Load, store
  - Addition, element per element
- **axpy** (α.x+y)

```
void xpy (const float *x, const float *y, float *z,
          const int vector_size)
{
  /* Check that the vector size is a multiple of the number
     of elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_add_ps(reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File xpy.c — routine xpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy (x + y)**
  - Load, store
  - Addition, element per element

- **axpy (α.x+y)**
  - Initialization with a constant (set1)
  - Fused Multiply-Accumulate (FMA)

```
void axpy (const float alpha, const float *x, const float *y,
           float *z, const int vector_size)
{
  /* Check that the vector size is a multiple of the number of
elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 reg_a = _mm256_set1_ps(alpha);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_fmadd_ps(reg_a, reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File axpy.c — routine axpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy (x + y)**
  - Load, store
  - Addition, element per element

- **axpy (α.x+y)**
  - Initialization with a constant (set1)
  - Fused Multiply-Accumulate (FMA)

```
void axpy (const float alpha, const float *x, const float *y,
           float *z, const int vector_size)
{
  /* Check that the vector size is a multiple of the number of
elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 reg_a = _mm256_set1_ps(alpha);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_fmadd_ps(reg_a, reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File axpy.c — routine axpy()

x + y, α.x+y

Simple SIMD Arithmetics

- **xpy (x + y)**
  - Load, store
  - Addition, element per element

- **axpy (α.x+y)**
  - Initialization with a constant (set1)
  - Fused Multiply-Accumulate (FMA)

```
void axpy (const float alpha, const float *x, const float *y,
           float *z, const int vector_size)
{
  /* Check that the vector size is a multiple of the number of
elements in SIMD registers */
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 reg_a = _mm256_set1_ps(alpha);

  int i;
  for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
  {
    __m256 reg_x;
    __m256 reg_y;
    __m256 reg_z;

    /* Load A and B arrays in SIMD registers */
    reg_x = _mm256_load_ps(&x[i]);
    reg_y = _mm256_load_ps(&y[i]);

    /* Perform SIMD add operation */
    reg_z = _mm256_fmadd_ps(reg_a, reg_x, reg_y);

    /* Store SIMD register in C array */
    _mm256_store_ps(&z[i], reg_z);
  }
}
```

File axpy.c — routine axpy()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute

```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute

```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute

```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute


```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute

Elements (32-bit, single precision floating point)

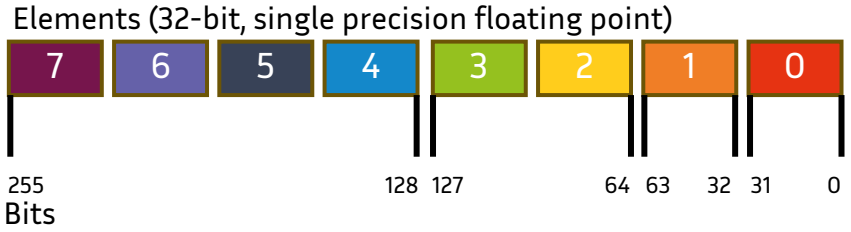

```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c **— routine** reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

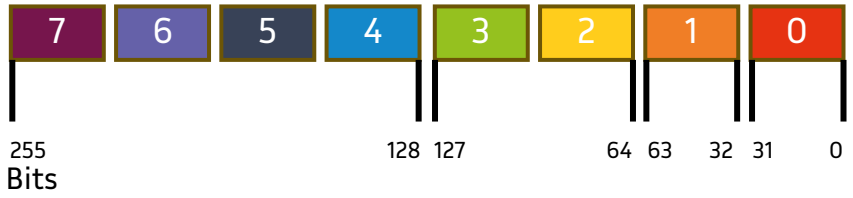
- **Reduction**
  - Permute

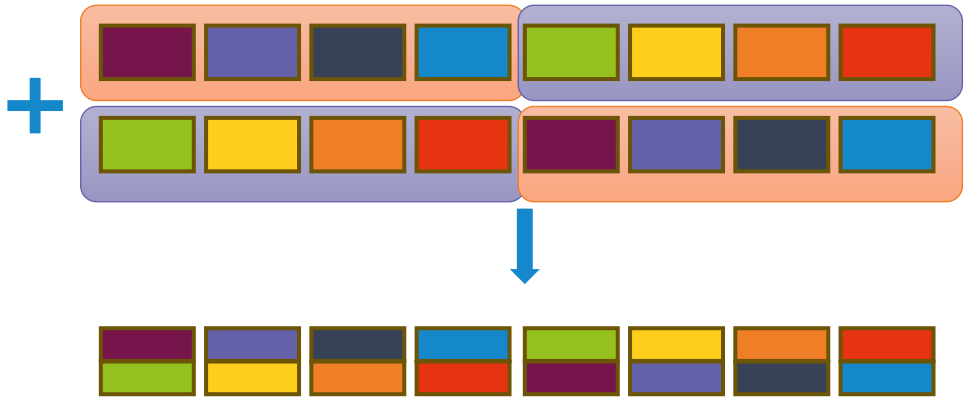



```c
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c **— routine** reduce_sum()

Reduction, Dot Product

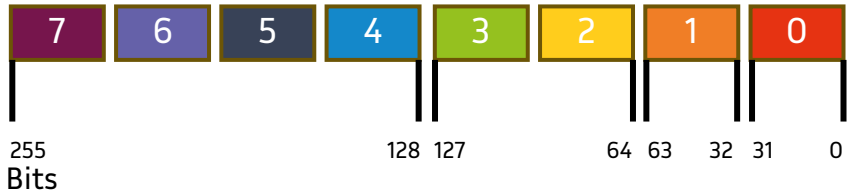


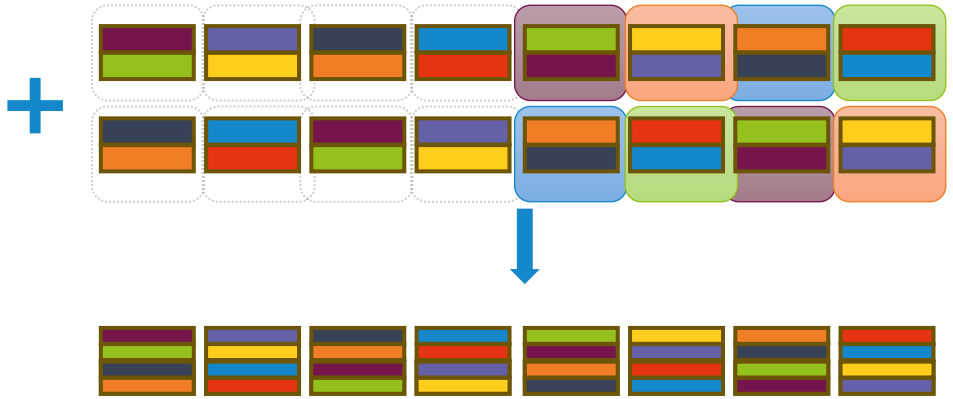


```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

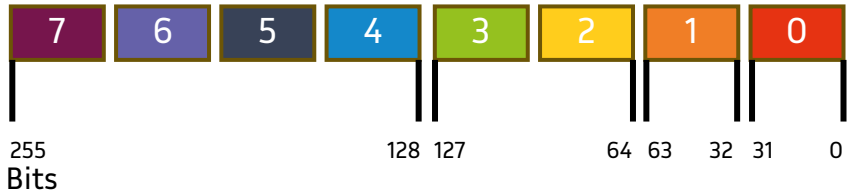


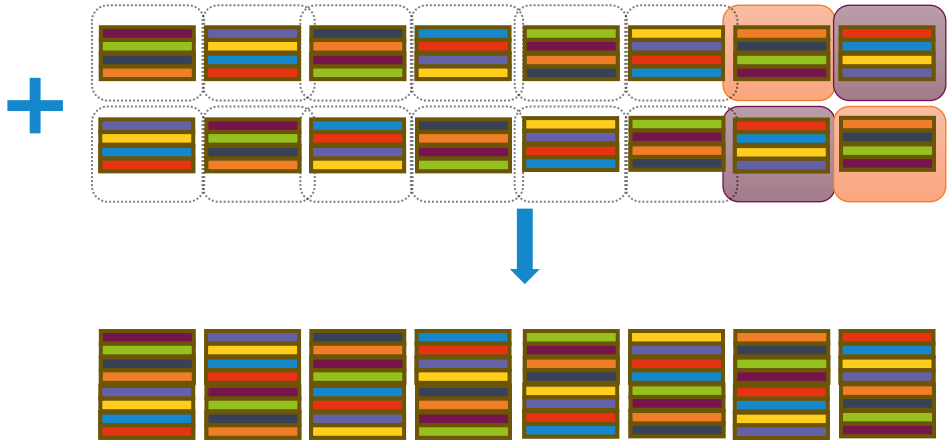


```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c **— routine** reduce_sum()

Reduction, Dot Product





```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c **— routine** reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD Arithmetics

- **Reduction**
  - Permute
- **Dot product**

```
void reduce_sum (const float *x, const int vector_size, __m256 *r)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);

  __m256 a;
  if (vector_size > 0)
  {
    a = _mm256_load_ps(&x[0]);

    int i;
    for (i=REG_NB_ELEMENTS; i<vector_size; i+=REG_NB_ELEMENTS)
    {
        __m256 v = _mm256_load_ps(&x[i]);
        a = _mm256_add_ps(a, v);
    }

    /* Add elements in the lower 128-bit and the elts in the higher 128-bit
     * This sums element i with element i + 4 */
    a = _mm256_add_ps(a, _mm256_permute2f128_ps(a, a, _MM_SHUFFLE(0,0,0,1)));

    /* This sums element i with element i + 2 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(1,0,3,2)));

    /* This sums element i with element i + 1 */
    a = _mm256_add_ps(a, _mm256_permute_ps(a, _MM_SHUFFLE(2,3,0,1)));
  }
  else
  {
    a = _mm256_setzero_ps();
  }
  *r = a;
}
```

File reduce_sum.c — routine reduce_sum()

Reduction, Dot Product

Slightly Less Simple SIMD A

- **Reduction**
  - Permute
- **Dot product**
  - FMA
  - Reduction

```c
float dot_product (const float *x, const float *y, const int vector_size)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);
  float result = 0;

  if (vector_size > 0)
  {
    __m256 reg_acc = _mm256_setzero_ps();

    int i;
    for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
    {
      __m256 reg_x;
      __m256 reg_y;

      /* Load A and B arrays in SIMD registers */
      reg_x = _mm256_load_ps(&x[i]);
      reg_y = _mm256_load_ps(&y[i]);

      /* Perform SIMD add operation */
      reg_acc = _mm256_fmadd_ps(reg_x, reg_y, reg_acc);
    }

    /* Perform the sum reduction */
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute2f128_ps(reg_acc, reg_acc, _MM_SHUFFLE(0,0,0,1)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(1,0,3,2)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(2,3,0,1)));

    result = reg_acc[0];
  }
  return result;
}
```

File dotp.c — routine dot_product()

Reduction, Dot Product

Slightly Less Simple SIMD A

- **Reduction**
  - Permute
- **Dot product**
  - FMA
  - Reduction

```
float dot_product (const float *x, const float *y, const int vector_size)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);
  float result = 0;

  if (vector_size > 0)
  {
    __m256 reg_acc = _mm256_setzero_ps();

    int i;
    for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
    {
      __m256 reg_x;
      __m256 reg_y;

      /* Load A and B arrays in SIMD registers */
      reg_x = _mm256_load_ps(&x[i]);
      reg_y = _mm256_load_ps(&y[i]);

      /* Perform SIMD add operation */
      reg_acc = _mm256_fmadd_ps(reg_x, reg_y, reg_acc);
    }

    /* Perform the sum reduction */
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute2f128_ps(reg_acc, reg_acc, _MM_SHUFFLE(0,0,0,1)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(1,0,3,2)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(2,3,0,1)));

    result = reg_acc[0];
  }
  return result;
}
```

File dotp.c — routine dot_product()

Reduction, Dot Product

Slightly Less Simple SIMD A

- **Reduction**
  - Permute
- **Dot product**
  - FMA
  - Reduction

```
float dot_product (const float *x, const float *y, const int vector_size)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);
  float result = 0;

  if (vector_size > 0)
  {
    __m256 reg_acc = _mm256_setzero_ps();

    int i;
    for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
    {
      __m256 reg_x;
      __m256 reg_y;

      /* Load A and B arrays in SIMD registers */
      reg_x = _mm256_load_ps(&x[i]);
      reg_y = _mm256_load_ps(&y[i]);

      /* Perform SIMD add operation */
      reg_acc = _mm256_fmadd_ps(reg_x, reg_y, reg_acc);
    }

    /* Perform the sum reduction */
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute2f128_ps(reg_acc, reg_acc, _MM_SHUFFLE(0,0,0,1)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(1,0,3,2)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(2,3,0,1)));

    result = reg_acc[0];
  }
  return result;
}
```

File dotp.c — routine dot_product()

Reduction, Dot Product

Slightly Less Simple SIMD A

- **Reduction**
  - Permute

- **Dot product**
  - FMA
  - Reduction

```
float dot_product (const float *x, const float *y, const int vector_size)
{
  assert(vector_size % REG_NB_ELEMENTS == 0);
  float result = 0;

  if (vector_size > 0)
  {
    __m256 reg_acc = _mm256_setzero_ps();

    int i;
    for (i=0; i<vector_size; i+=REG_NB_ELEMENTS)
    {
      __m256 reg_x;
      __m256 reg_y;

      /* Load A and B arrays in SIMD registers */
      reg_x = _mm256_load_ps(&x[i]);
      reg_y = _mm256_load_ps(&y[i]);

      /* Perform SIMD add operation */
      reg_acc = _mm256_fmadd_ps(reg_x, reg_y, reg_acc);
    }

    /* Perform the sum reduction */
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute2f128_ps(reg_acc, reg_acc, _MM_SHUFFLE(0,0,0,1)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(1,0,3,2)));
    reg_acc = _mm256_add_ps(reg_acc, _mm256_permute_ps(reg_acc, _MM_SHUFFLE(2,3,0,1)));

    result = reg_acc[0];
  }
  return result;
}
```

File dotp.c — routine dot_product()

Minmax

Comparisons and Masks

- **minmax version 1**

Minmax

~~Comparisons and Masks~~

- **minmax version 1**
  - Dedicated routines
  - Always check whether a function already exists! 😉

```c
void minmax(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  *zmax = _mm256_max_epi32(x, y);
  *zmin = _mm256_min_epi32(x, y);
}
```

File minmax.c — routine minmax()

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(_mm256_and_si256 (c,x), _mm256_andnot_si256(c,y));
  *zmin = _mm256_or_si256(_mm256_andnot_si256(c,x), _mm256_and_si256 (c,y));
}
```

File minmax.c — routine minmax()

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(
                  _mm256_and_si256   (c,x),
                  _mm256_andnot_si256(c,y));

  *zmin = _mm256_or_si256(
                   _mm256_andnot_si256(c,x),
                   _mm256_and_si256   (c,y));

}
```

File minmax.c — routine minmax()

x	-3	-1	4	-12	7	-7	15	2
y	-7	18	3	4	-1	-19	11	-10

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(
                  _mm256_and_si256   (c,x),
                  _mm256_andnot_si256(c,y));

  *zmin = _mm256_or_si256(
                   _mm256_andnot_si256(c,x),
                   _mm256_and_si256   (c,y));

}
```

File minmax.c — routine minmax()

	x	-3	-1	4	-12	7	-7	15	2
	y	-7	18	3	4	-1	-19	11	-10
x > y	c	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(
                  _mm256_and_si256   (c,x),
                  _mm256_andnot_si256(c,y));

  *zmin = _mm256_or_si256(
                   _mm256_andnot_si256(c,x),
                   _mm256_and_si256   (c,y));

}
```

File minmax.c — routine minmax()

	x	-3	-1	4	-12	7	-7	15	2
	c	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF
c *and* x	*Result 1*	-3	0	4	0	7	-7	15	2

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(
                  _mm256_and_si256   (c,x),
                  _mm256_andnot_si256(c,y));

  *zmin = _mm256_or_si256(
                  _mm256_andnot_si256(c,x),
                  _mm256_and_si256   (c,y));

}
```

File minmax.c — routine minmax()

	y	-7	18	3	4	-1	-19	11	-10
	c	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0x00000000	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF
c *andnot* y	*Result 2*	0	18	0	4	0	0	0	0

Minmax

Comparisons and Masks

- **minmax version 2**
  - *For illustration purpose*
  - Comparisons
  - Masks & Boolean operations

```
void minmax2(__m256i x, __m256i y, __m256i *zmin, __m256i *zmax)
{
  __m256i c = _mm256_cmpgt_epi32(x, y);

  *zmax = _mm256_or_si256(
                  _mm256_and_si256   (c,x),
                  _mm256_andnot_si256(c,y));

  *zmin = _mm256_or_si256(
                   _mm256_andnot_si256(c,x),
                   _mm256_and_si256   (c,y));

}
```

File minmax.c — routine minmax()

	Result 1	-3	0	4	0	7	-7	15	2
	Result 2	0	18	0	4	0	0	0	0
zmax: R1 *or* R2	Result	**-3**	**18**	**4**	**4**	**7**	**-7**	**15**	**2**

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
    - (a+ib) + (c+id) == (a+c) + i(b+d)
  - product
    - (a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

```
void cmplx_sum(__m256i x, __m256i y, __m256i *z)
{
  *z = _mm256_add_epi32(x, y);
}

void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_sum() & cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

```
void cmplx_sum(__m256i x, __m256i y, __m256i *z)
{
  *z = _mm256_add_epi32(x, y);
}

void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

(a+ib) + (c+id) == (a+c) + i(b+d)

File complex.c — routines cmplx_sum() & cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

```
void cmplx_sum(__m256i x, __m256i y, __m256i *z)
{
  *z = _mm256_add_epi32(x, y);
}

void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_sum() & cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

```
(a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)
```

```
void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

$$(a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)$$

```
void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

$$(a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)$$

```
void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

$$(a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)$$

```
void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_product()

Complex

Masks, Boolean ops, Integer Multiply

- **complex**
  - sum
  - product

$$(a+ib) * (c+id) == (a*c - b*d) + i(a*d + b*c)$$

```
void cmplx_prod(__m256i x, __m256i y, __m256i *z)
{
  __m256i factor = _mm256_set_epi32(-1, 1, -1, 1, -1, 1, -1, 1);
  __m256i real_mask = _mm256_set_epi32(0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff, 0, 0xffffffff);

  __m256i real = _mm256_mullo_epi32(x, y);
  real = _mm256_mullo_epi32(factor, real);
  __m256i real_shuf = _mm256_shuffle_epi32(real, _MM_SHUFFLE(2, 3, 0, 1));
  real = _mm256_add_epi32(real, real_shuf);
  real = _mm256_and_si256(real_mask, real);

  __m256i y_shuf = _mm256_shuffle_epi32(y, _MM_SHUFFLE(2, 3, 0, 1));
  __m256i imag = _mm256_mullo_epi32(x, y_shuf);
  __m256i imag_shuf = _mm256_shuffle_epi32(imag, _MM_SHUFFLE(2, 3, 0, 1));
  imag = _mm256_add_epi32(imag, imag_shuf);
  imag = _mm256_andnot_si256(real_mask, imag);

  *z = _mm256_or_si256(real, imag);
}
```

File complex.c — routines cmplx_product()

Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

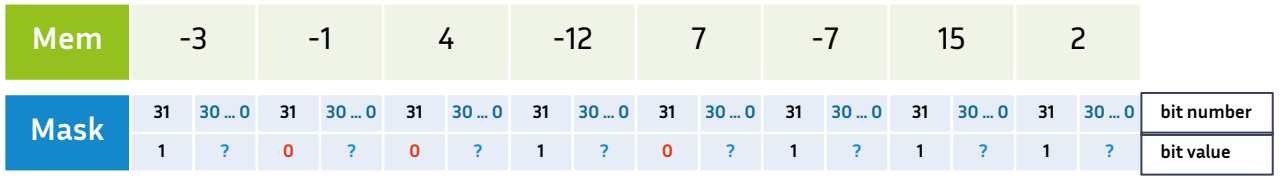
- **Masked Load**
  - Load data from memory into a SIMD register
    - `_mm256_maskload_*`

- **Masked Store**
  - Store data from a SIMD register into memory
    - `_mm256_maskstore_*`

Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*

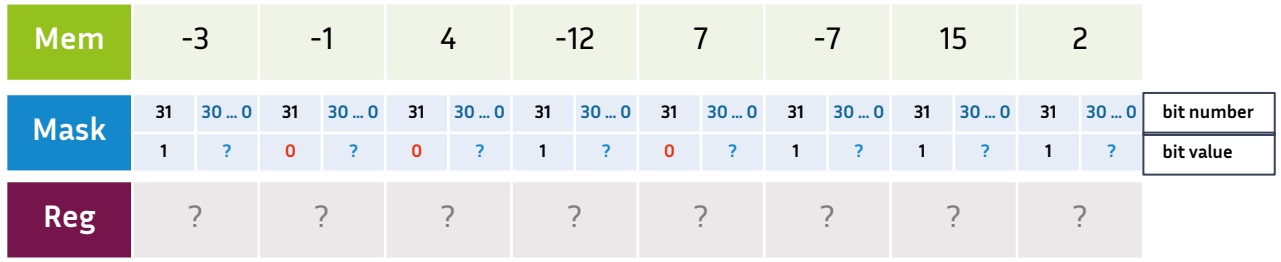


- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*

Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*

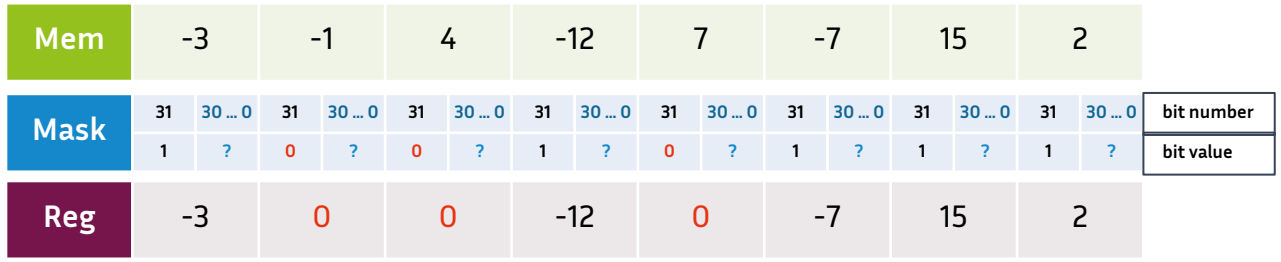


- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*

Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*



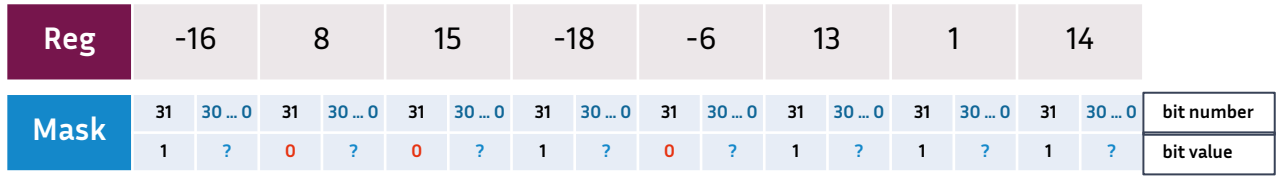
- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*

Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*

- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*

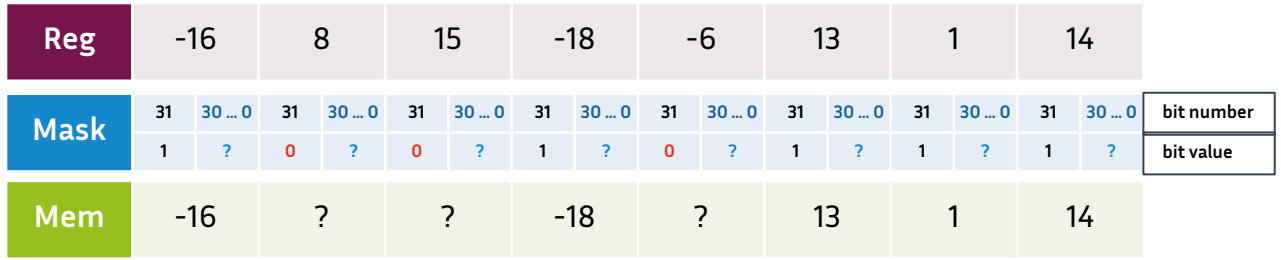


Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*

- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*

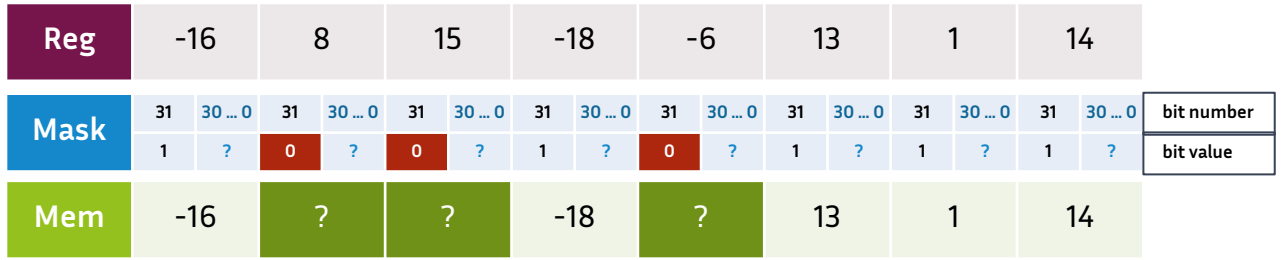


Examples of Common SIMD Instructions – AVX2

Masked Load / Masked Store

- **Masked Load**
  - Load data from memory into a SIMD register
    - _mm256_maskload_*

- **Masked Store**
  - Store data from a SIMD register into memory
    - _mm256_maskstore_*



TD

Introduction to using *intrinsics* routines for the Intel AVX2 instruction set, part II

- **Assignment, source codes, additional slides**
  - Moodle ENSEIRB, IT390 course