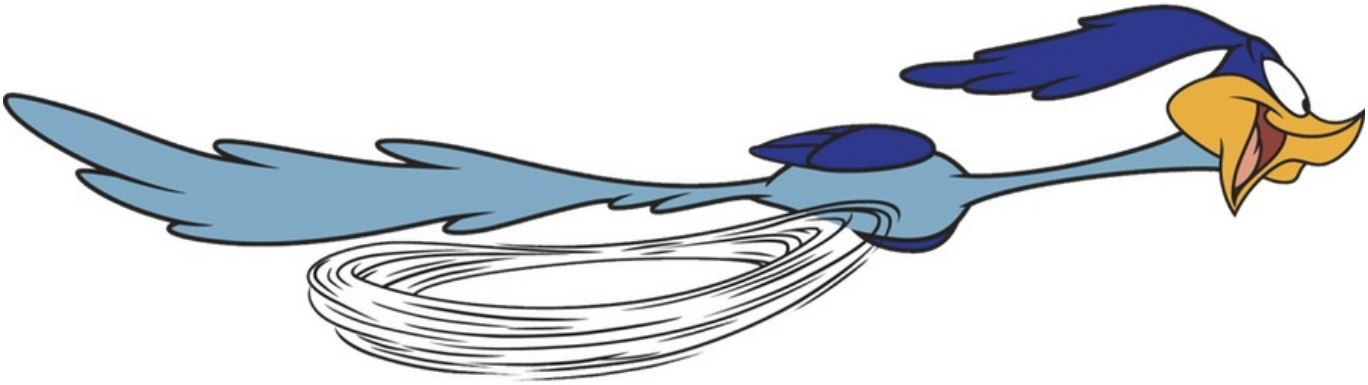
# MyIntrinsics++ (MIPP)

pipeline passed coverage 99.20%



## Purpose

MIPP is a portable and Open-source wrapper (MIT license) for vector intrinsic functions (SIMD) written in C++11. It works for SSE, AVX, AVX-512, ARM NEON and SVE (work in progress) instructions. MIPP wrapper supports simple/double precision floating-point numbers and also signed/unsigned integer arithmetic (64-bit, 32-bit, 16-bit and 8-bit).

With the MIPP wrapper you do not need to write a specific intrinsic code anymore. Just use provided functions and the wrapper will automatically generates the right intrisic calls for your specific architecture.

If you are interested by ARM SVE development status, please follow this link.

## Short Documentation

### Supported Compilers

At this time, MIPP has been tested on the following compilers:

- Intel: `icpc >= 16`,
- GNU: `g++ >= 4.8`,
- Clang: `clang++ >= 3.6`,
- Microsoft: `msvc >= 14`.

On `msvc 14.10` (Microsoft Visual Studio 2017), the performances are reduced compared to the other compilers, the compiler is not able to fully inline all the MIPP methods. This has been fixed on `msvc 14.21` (Microsoft Visual Studio 2019) and now you can expect high performances.

### Install and Configure your Code

You don't have to install MIPP because it is a simple C++ header file. The headers are located in the `include` folder (note that this location has changed since commit `6795891`, before they were located in the `src` folder).

Just include the header into your source files when the wrapper is needed.

```
#include "mipp.h"
```

mipp.h use a C++ `namespace`: `mipp`, if you do not want to prefix all the MIPP calls by `mipp::` you can do that:

```
#include "mipp.h"
using namespace mipp;
```

Before trying to compile, think to tell the compiler what kind of vector instructions you want to use. For instance, if you are using GNU compiler (`g++`) you simply have to add the `-march=native` option for SSE and AVX CPUs compatible. For ARMv7 CPUs with NEON instructions you have to add the `-mfpu=neon` option (since most of current NEONv1 instructions are not IEEE-754 compliant). However, this is no more the case on ARMv8 processors, so the `-march=native` option will work too. MIPP also uses some nice features provided by the C++11 and so we have to add the `-std=c++11` flag to compile the code. You are now ready to run your code with the MIPP wrapper.

In the case where MIPP is installed on the system it can be integrated into a cmake projet in a standard way. Example

```
# install MIPP
cd MIPP/
export MIPP_ROOT=$PWD/build/install
cmake -B build -DCMAKE_INSTALL_PREFIX=$MIPP_ROOT
cmake --build build -j5
cmake --install build
```

In your `CMakeLists.txt`:

```
# find the installation of MIPP on the system
find_package(MIPP REQUIRED)

# define your executable
add_executable(gemm gemm.cpp)

# link your executable to MIPP
target_link_libraries(gemm PRIVATE MIPP::mipp)
```

```
cd your_project/
# if MIPP is installed in a system standard path: MIPP will be found automatically
with cmake
cmake -B build
# if MIPP is installed in a non-standard path: use CMAKE_PREFIX_PATH
cmake -B build -DCMAKE_PREFIX_PATH=$MIPP_ROOT
```

**Generate Sources & Compile the Static Library**

MIPP is mainly a header only library. However, some macro operations require to compile a small library. This is particularly true for the `compress` operation that relies on generated LUTs stored in the static library.

To generate the source files containing these LUTs you need to install Python3 with the Jinja2 package:

```
sudo apt install python3 python3-pip
pip3 install --user -r codegen/requirements.txt
```

Then you can call the generator as follow:

```
python3 codegen/gen_compress.py
```

And, finally you can compile the MIPP static library:

```
cmake -B build -DMIPP_STATIC_LIB=ON
cmake --build build -j4
```

Note that **the compilation of the static library is optional**. You can choose to do not compile the static library then only some macro operations will be missing.

### Sequential Mode

By default, MIPP tries to recognize the instruction set from the preprocessor definitions. If MIPP can't match the instruction set (for instance when MIPP does not support the targeted instruction set), MIPP falls back on standard sequential instructions. In this mode, the vectorization is not guarantee anymore but the compiler can still perform auto-vectorization.

It is possible to force MIPP to use the sequential mode with the following compiler definition: `-DMIPP_NO_INTRINSICS`. Sometime it can be useful for debugging or to bench a code.

If you want to check the MIPP mode configuration, you can print the following global variable: `mipp::InstructionFullType` (`std::string`).

### Vector Register Declaration

Just use the `mipp::Reg<T>` type.

```
mipp::Reg<T> r1, r2, r3; // we have declared 3 vector registers
```

But we do not know the number of elements per register here. This number of elements can be obtained by calling the `mipp::N<T>()` function (`T` is a template parameter, it can be `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t` or `uint8_t` type).

```
for (int i = 0; i < n; i += mipp::N<float>()) {
    // ...
}
```

The register size directly depends on the precision of the data we are working on.

### Register `load` and `store` Instructions

Loading memory from a vector into a register:

```
int n = mipp::N<float>() * 10;
std::vector<float> myVector(n);
int i = 0;
mipp::Reg<float> r1;
r1.load(&myVector[i*mipp::N<float>()]);
```

The last two lines can be shorten as follow where the `load` call becomes implicit:

```
mipp::Reg<float> r1 = &myVector[i*mipp::N<float>()];
```

Store can be done with the `store(...)` method:

```
int n = mipp::N<float>() * 10;
std::vector<float> myVector(n);
int i = 0;
mipp::Reg<float> r1 = &myVector[i*mipp::N<float>()];

// do something with r1

r1.store(&myVector[(i+1)*mipp::N<float>()]);
```

By default the loads and stores work on **unaligned memory**. It is possible to control this behavior with the `-DMIPP_ALIGNED_LOADS` definition: when specified, the loads and stores work on **aligned memory** by default. In the **aligned memory** mode, it is still possible to perform unaligned memory operations with the `mipp::loadu` and `mipp::storeu` functions. However, it is not possible to perform aligned loads and stores in the **unaligned memory** mode.

To allocate aligned data you can use the MIPP aligned memory allocator wrapped into the `mipp::vector` class. `mipp::vector` is fully retro-compatible with the standard `std::vector` class and it can be use everywhere you can use `std::vector`.

```
mipp::vector<float> myVector(n);
```

## Register Initialization

You can initialize a vector register from a scalar value:

```
mipp::Reg<float> r1; // r1 = | unknown | unknown | unknown | unknown |
r1 = 1.0;            // r1 = |    +1.0 |    +1.0 |    +1.0 |    +1.0 |
```

Or from an initializer list (`std::initializer_list`):

```
mipp::Reg<float> r1;       // r1 = | unknown | unknown | unknown | unknown |
r1 = {1.0, 2.0, 3.0, 4.0}; // r1 = |    +1.0 |    +2.0 |    +3.0 |    +4.0 |
```

## Computational Instructions

**Add** two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 1.0;     // r1 = | +1.0 | +1.0 | +1.0 | +1.0 |
r2 = 2.0;     // r2 = | +2.0 | +2.0 | +2.0 | +2.0 |

r3 = r1 + r2; // r3 = | +3.0 | +3.0 | +3.0 | +3.0 |
```

**Subtract** two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 1.0;     // r1 = | +1.0 | +1.0 | +1.0 | +1.0 |
r2 = 2.0;     // r2 = | +2.0 | +2.0 | +2.0 | +2.0 |

r3 = r1 - r2; // r3 = | -1.0 | -1.0 | -1.0 | -1.0 |
```

**Multiply** two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 1.0;     // r1 = | +1.0 | +1.0 | +1.0 | +1.0 |
r2 = 2.0;     // r2 = | +2.0 | +2.0 | +2.0 | +2.0 |

r3 = r1 * r2; // r3 = | +2.0 | +2.0 | +2.0 | +2.0 |
```

**Divide** two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 1.0;     // r1 = | +1.0 | +1.0 | +1.0 | +1.0 |
r2 = 2.0;     // r2 = | +2.0 | +2.0 | +2.0 | +2.0 |

r3 = r1 / r2; // r3 = | +0.5 | +0.5 | +0.5 | +0.5 |
```

**Fused multiply and add** of three vector registers:

```
mipp::Reg<float> r1, r2, r3, r4;

r1 = 2.0;                    // r1 = | +2.0 | +2.0 | +2.0 | +2.0 |
r2 = 3.0;                    // r2 = | +3.0 | +3.0 | +3.0 | +3.0 |
r3 = 1.0;                    // r3 = | +1.0 | +1.0 | +1.0 | +1.0 |

// r4 = (r1 * r2) + r3
r4 = mipp::fmadd(r1, r2, r3); // r4 = | +7.0 | +7.0 | +7.0 | +7.0 |
```

**Fused negative multiply and add** of three vector registers:

```
mipp::Reg<float> r1, r2, r3, r4;

r1 = 2.0;                     // r1 = | +2.0 | +2.0 | +2.0 | +2.0 |
r2 = 3.0;                     // r2 = | +3.0 | +3.0 | +3.0 | +3.0 |
r3 = 1.0;                     // r3 = | +1.0 | +1.0 | +1.0 | +1.0 |

// r4 = -(r1 * r2) + r3
r4 = mipp::fnmadd(r1, r2, r3); // r4 = | -5.0 | -5.0 | -5.0 | -5.0 |
```

**Square root** of a vector register:

```
mipp::Reg<float> r1, r2;

r1 = 9.0;            // r1 = | +9.0 | +9.0 | +9.0 | +9.0 |

r2 = mipp::sqrt(r1);  // r2 = | +3.0 | +3.0 | +3.0 | +3.0 |
```

**Reciprocal square root** of a vector register (be careful: this intrinsic exists only for simple precision floating-point numbers):

```
mipp::Reg<float> r1, r2;

r1 = 9.0;            // r1 = | +9.0 | +9.0 | +9.0 | +9.0 |
```

```
r2 = mipp::rsqrt(r1); // r2 = | +0.3 | +0.3 | +0.3 | +0.3 |
```

### Selections

Select the **minimum** between two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 2.0;               // r1 = | +2.0 | +2.0 | +2.0 | +2.0 |
r2 = 3.0;               // r2 = | +3.0 | +3.0 | +3.0 | +3.0 |

r3 = mipp::min(r1, r2); // r3 = | +2.0 | +2.0 | +2.0 | +2.0 |
```

Select the **maximum** between two vector registers:

```
mipp::Reg<float> r1, r2, r3;

r1 = 2.0;               // r1 = | +2.0 | +2.0 | +2.0 | +2.0 |
r2 = 3.0;               // r2 = | +3.0 | +3.0 | +3.0 | +3.0 |

r3 = mipp::max(r1, r2); // r3 = | +3.0 | +3.0 | +3.0 | +3.0 |
```

### Permutations

The `rrot(...)` method allows you to perform a **right rotation** (a cyclic permutation) of the elements inside the register:

```
mipp::Reg<float> r1, r2;
r1 = {3.0, 2.0, 1.0, 0.0}  // r1 = | +3.0 | +2.0 | +1.0 | +0.0 |

r2 = mipp::rrot(r1);       // r2 = | +0.0 | +3.0 | +2.0 | +1.0 |
r1 = mipp::rrot(r2);       // r1 = | +1.0 | +0.0 | +3.0 | +2.0 |
r2 = mipp::rrot(r1);       // r2 = | +2.0 | +1.0 | +0.0 | +3.0 |
r1 = mipp::rrot(r2);       // r1 = | +3.0 | +2.0 | +1.0 | +0.0 |
```

Of course there are many more available instructions in the MIPP wrapper and you can find these instructions at the end of this page.

### Addition of Two Vectors

```
#include <cstdlib> // rand()
#include "mipp.h"
```

```
int main()
{
    // data allocation
    const int n = 32000; // size of the vA, vB, vC vectors
    mipp::vector<float> vA(n); // in
    mipp::vector<float> vB(n); // in
    mipp::vector<float> vC(n); // out

    // data initialization
    for (int i = 0; i < n; i++) vA[i] = rand() % 10;
    for (int i = 0; i < n; i++) vB[i] = rand() % 10;

    // declare 3 vector registers
    mipp::Reg<float> rA, rB, rC;

    // compute rC with the MIPP vectorized functions
    for (int i = 0; i < n; i += mipp::N<float>()) {
        rA.load(&vA[i]); // unaligned load by default (use the -
DMIPP_ALIGNED_LOADS
        rB.load(&vB[i]); // macro definition to force aligned loads and stores).
        rC = rA + rB;
        rC.store(&vC[i]);
    }

    return 0;
}
```

## Vectorizing an Existing Code

### Scalar Code

```
// ...
for (int i = 0; i < n; i++) {
    out[i] = 0.75f * in1[i] * std::exp(in2[i]);
}
// ...
```

### Vectorized Code

```
// ...
// compute the vectorized loop size which is a multiple of 'mipp::N<float>()'.
auto vecLoopSize = (n / mipp::N<float>()) * mipp::N<float>();
mipp::Reg<float> rout, rin1, rin2;
for (int i = 0; i < vecLoopSize; i += mipp::N<float>()) {
    rin1.load(&in1[i]); // unaligned load by default (use the -DMIPP_ALIGNED_LOADS
    rin2.load(&in2[i]); // macro definition to force aligned loads and stores).
    // the '0.75f' constant will be broadcast in a vector but it has to be at
    // the right of a 'mipp::Reg<T>', this is why it has been moved at the right
```

```
    // of the 'rin1' register. Notice that 'std::exp' has been replaced by
    // 'mipp::exp'.
    rout = rin1 * 0.75f * mipp::exp(rin2);
    rout.store(&out[i]);
}

// scalar tail loop: compute the remaining elements that can't be vectorized.
for (int i = vecLoopSize; i < n; i++) {
    out[i] = 0.75f * in1[i] * std::exp(in2[i]);
}
// ...
```

### Masked Instructions

MIPP comes with two generic and templatized masked functions (`mask` and `maskz`). Those functions allow you to benefit from the AVX-512 and SVE masked instructions. `mask` and `maskz` functions are retro compatible with older instruction sets.

```
mipp::Reg<        float    > ZMM1 = {   40,  -30,    60,    80};
mipp::Reg<        float    > ZMM2 = 0.1; // broadcast
mipp::Msk<mipp::N<float>()> k1   = {false, true, false, false};

// ZMM3 = k1 ? ZMM1 * ZMM2 : ZMM1;
auto ZMM3 = mipp::mask<float, mipp::mul>(k1, ZMM1, ZMM1, ZMM2);
std::cout << ZMM3 << std::endl; // output: "[40, -3, 60, 80]"

// ZMM4 = k1 ? ZMM1 * ZMM2 : 0;
auto ZMM4 = mipp::maskz<float, mipp::mul>(k1, ZMM1, ZMM2);
std::cout << ZMM4 << std::endl; // output: "[0, -3, 0, 0]"
```

## List of MIPP Functions

This section presents an exhaustive list of all the available functions in MIPP. Of course the MIPP wrapper does not cover all the possible intrinsics of each instruction set but it tries to give you the most important and useful ones.

In the following tables, `T`, `T1` and `T2` stand for data types (`double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t` or `uint8_t`). `N` stands for the number or elements in a mask or in a register. `N` is a strictly positive integer and can easily be deduced from the data type: `constexpr int N = mipp::N<T>()`. When `T` and `N` are mixed in a prototype, `N` has to satisfy the previous constraint (`N = mipp::N<T>()`).

In the documentation there are some terms that requires to be clarified:

- **register element**: a SIMD register is composed by multiple scalar elements, those elements are built-in data types (`double`, `float`, `int64_t`, ...),
- **register lane**: modern instruction sets can have multiple implicit sub parts in an entire SIMD register, those sub parts are called lanes (SSE has one lane of 128 bits, AVX has two lanes of 128 bits, AVX-512

has four lanes of 128 bits).

## Memory Operations

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `load` | `Reg <T> load (const T* mem)` | Loads aligned data from `mem` to a register. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `loadu` | `Reg <T> loadu (const T* mem)` | Loads unaligned data from `mem` to a register. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `store` | `void store (T* mem, const Reg<T> r)` | Stores the `r` register in the `mem` aligned data. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `storeu` | `void storeu (T* mem, const Reg<T> r)` | Stores the `r` register in the `mem` unaligned data. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `maskzld` | `Reg <T> maskzld (const Msk<N> m, const T* mem)` | Loads elements according to the mask `m` (puts zero when the mask value is false). | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `maskzlds` | `Reg <T> maskzlds (const Msk<N> m, const T* mem)` | Loads elements according to the mask `m` (puts zero when the mask value is false). Safe version, only reads masked elements in memory. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `maskst` | `void maskst (const Msk<N> m, T* mem, const Reg<T> r)` | Stores elements from the `r` register according to the mask `m` in the `mem` memory. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `masksts` | `void masksts (const Msk<N> m, T* mem, const Reg<T> r)` | Stores elements from the `r` register according to the mask `m` in the `mem` memory. Safe version, only writes masked elements in memory. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `gather` | `Reg <TD,TI> gather (const TD* mem, const Reg<TI> idx)` | Gathers elements from `mem` to a register. Selects elements according to the indices in `idx`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `scatter` | `void <TD,TI> scatter (TD* mem, const Reg<TI> idx, const Reg<TD> r)` | Scatters elements into `mem` from the `r` register. Writes elements at the `idx` indices in `mem`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `maskzgat` | `Reg <TD,TI> gather (const Msk<N> m, const TD* mem, const Reg<TI> idx)` | Gathers elements from `mem` to a register (according to the mask `m`). Selects elements according to the indices in `idx` (puts zero when the mask value is false). | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `masksca` | `void <TD,TI> scatter (const Msk<N> m, TD* mem, const Reg<TI> idx, const Reg<TD> r)` | Scatters elements into `mem` from the `r` register (according to the mask `m`). Writes elements at the `idx` indices in `mem`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| `set` | `Reg <T> set (const T[N] vals)` | Sets a register from the values in `vals`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `set` | `Msk <N> set (const bool[N] bits)` | Sets a mask from the bits in `bits`. | |
| `set1` | `Reg <T> set1 (const T val)` | Broadcasts `val` in a register. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `set1` | `Msk <N> set1 (const bool bit)` | Broadcasts `bit` in a mask. | |
| `set0` | `Reg <T> set0 ()` | Initializes a register to zero. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `set0` | `Msk <N> set0 ()` | Initializes a mask to false. | |
| `get` | `T get (const Reg<T> r, const size_t index)` | Gets a specific element from the register `r` at the `index` position. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `get` | `T get (const Reg_2<T> r, const size_t index)` | Gets a specific element from the register `r` at the `index` position. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `get` | `bool get (const Msk<N> m, const size_t index)` | Gets a specific element from the register `m` at the `index` position. | |
| `getfirst` | `T getfirst (const Reg<T> r)` | Gets the first element from the register `r`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `getfirst` | `T getfirst (const Reg_2<T> r)` | Gets the first element from the register `r`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `getfirst` | `bool getfirst (const Msk<N> m)` | Gets the first element from the register `m`. | |
| `low` | `Reg_2<T> low (const Reg<T> r)` | Gets the low part of the `r` register. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `high` | `Reg_2<T> high (const Reg<T> r)` | Gets the high part of the `r` register. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `combine` | `Reg <T> combine (const Reg_2<T> r1, const Reg_2<T> r2)` | Combine two half registers in a full register, `r1` will be the low part and `r2` the high part. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `combine` | `Reg <S,T> combine (const Reg<T> r1, const Reg<T> r2)` | `S` elements of `r1` are shifted to the left, `(S - N) + N` elements of `r2` are shifted to the right. Shifted `r1` and `r2` are combined to give the result. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `compress` | `Reg <T> compress (const Reg<T> r1, const Msk<N> m)` | Pack the elements of `r1` at the beginning of the register according to the bitmask `m` (if the bit is 1 then element is picked, otherwise it is not). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmask` | `Reg <T> cmask (const uint32_t[N] ids)` | Creates a cmask from an indexes list (indexes have to be between 0 and N-1). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmask2` | `Reg <T> cmask2 (const uint32_t[N/2] ids)` | Creates a cmask2 from an indexes list (indexes have to be between 0 and (N/2)-1). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmask4` | `Reg <T> cmask4 (const uint32_t[N/4] ids)` | Creates a cmask4 from an indexes list (indexes have to be between 0 and (N/4)-1). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `shuff` | `Reg <T> shuff (const Reg<T> r, const Reg<T> cm)` | Shuffles the elements of `r` according to the cmask `cm`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `shuff2` | `Reg <T> shuff2 (const Reg<T> r, const Reg<T> cm2)` | Shuffles the elements of `r` according to the cmask2 `cm2` (same shuffle is applied in both lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `shuff4` | `Reg <T> shuff4 (const Reg<T> r, const Reg<T> cm4)` | Shuffles the elements of `r` according to the cmask4 `cm4` (same shuffle is applied in the four lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleave` | `Regx2<T> interleave (const Reg<T> r1, const Reg<T> r2)` | Interleaves `r1` and `r2` : `[r1_1, r2_1, r1_2, r2_2, ..., r1_n, r2_n]`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `deinterleave` | `Regx2<T> deinterleave (const Reg<T> r1, const Reg<T> r2)` | Reverts the previous defined interleave operation. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleave2` | `Regx2<T> interleave2 (const Reg<T> r1, const Reg<T> r2)` | Interleaves `r1` and `r2` considering two lanes. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleave4` | `Regx2<T> interleave4 (const Reg<T> r1, const Reg<T> r2)` | Interleaves `r1` and `r2` considering four lanes. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleavelo` | `Reg <T> interleavelo (const Reg<T> r1, const Reg<T> r2)` | Interleaves the low part of `r1` with the low part of `r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleavelo2` | `Reg <T> interleavelo2 (const Reg<T> r1, const Reg<T> r2)` | Interleaves the low part of `r1` with the low part of `r2` (considering two lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleavelo4` | `Reg <T> interleavelo4 (const Reg<T> r1, const Reg<T> r2)` | Interleaves the low part of `r1` with the low part of `r2` (considering four lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleavehi` | `Reg <T> interleavehi (const Reg<T> r1, const Reg<T> r2)` | Interleaves the high part of `r1` with the high part of `r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `interleavehi2` | `Reg <T> interleavehi2 (const Reg<T> r1, const Reg<T> r2)` | Interleaves the high part of `r1` with the high part of `r2` (considering two lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `interleavehi4` | `Reg <T> interleavehi4 (const Reg<T> r1, const Reg<T> r2)` | Interleaves the high part of `r1` with the high part of `r2` (considering four lanes). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `lrot` | `Reg <T> lrot (const Reg<T> r)` | Rotates the `r` register from the left (cyclic permutation). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `rrot` | `Reg <T> rrot (const Reg<T> r)` | Rotates the `r` register from the right (cyclic permutation). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `blend` | `Reg <T> blend (const Reg<T> r1, const Reg<T> r2, const Msk<N> m)` | Combines `r1` and `r2` register following the `m` mask values (`m_i ? r1_i : r2_i`). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `select` | `Reg <T> select (const Msk<N> m, const Reg<T> r1, const Reg<T> r2)` | Alias for the previous `blend` function. Parameters order is a little bit different. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

### Bitwise Operations

The `pipe` keyword stands for the "|" binary operator.

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| `andb` | `&` and `&=` | `Reg<T> andb (const Reg<T> r1, const Reg<T> r2)` | Computes the bitwise AND: `r1 & r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `andb` | `&` and `&=` | `Msk<N> andb (const Msk<N> m1, const Msk<N> m2)` | Computes the bitwise AND: `m1 & m2`. | |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| andnb | | Reg<T> andnb (const Reg<T> r1, const Reg<T> r1) | Computes the bitwise AND NOT: (~r1) & r2. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| andnb | | Msk<N> andnb (const Msk<N> m1, const Msk<N> m2) | Computes the bitwise AND NOT: (~m1) & m2. | |
| orb | pipe and pipe= | Reg<T> orb (const Reg<T> r1, const Reg<T> r2) | Computes the bitwise OR: r1 pipe r2. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| orb | pipe and pipe= | Msk<N> orb (const Msk<N> m1, const Msk<N> m2) | Computes the bitwise OR: m1 pipe m2. | |
| xorb | ^ and ^= | Reg<T> xorb (const Reg<T> r1, const Reg<T> r2) | Computes the bitwise XOR: r1 ^ r2. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| xorb | ^ and ^= | Msk<N> xorb (const Msk<N> m1, const Msk<N> m2) | Computes the bitwise XOR: m1 ^ m2. | |
| lshift | << and <<= | Reg<T> lshift (const Reg<T> r, const uint32_t n) | Computes the bitwise LEFT SHIFT: r << n. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| lshiftr | << and <<= | Reg<T> lshiftr (const Reg<T> r1, const Reg<T> r2) | Computes the bitwise LEFT SHIFT: r1 << r2. | int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| lshift | << and <<= | Msk<N> lshift (const Msk<N> m, const uint32_t n) | Computes the bitwise LEFT SHIFT: m << n. | |
| rshift | >> and >>= | Reg<T> rshift (const Reg<T> r, const uint32_t n) | Computes the bitwise RIGHT SHIFT: r >> n. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| rshiftr | >> and >>= | Reg<T> rshiftr (const Reg<T> r1, const Reg<T> r2) | Computes the bitwise RIGHT SHIFT: r1 >> r2. | int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| `rshift` | `>>` and `>>=` | `Msk<N> rshift (const Msk<N> m, const uint32_t n)` | Computes the bitwise RIGHT SHIFT: `m >> n`. | |
| `notb` | `~` | `Reg<T> notb (const Reg<T> r)` | Computes the bitwise NOT: `~r`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `notb` | `~` | `Msk<N> notb (const Msk<N> m)` | Computes the bitwise NOT: `~m`. | |

### Logical Comparisons

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| `cmpeq` | `==` | `Msk<N> cmpeq (const Reg<T> r1, const Reg<T> r2)` | Compares if equal to: `r1 == r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmpneq` | `!=` | `Msk<N> cmpneq (const Reg<T> r1, const Reg<T> r2)` | Compares if not equal to: `r1 != r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmpge` | `>=` | `Msk<N> cmpge (const Reg<T> r1, const Reg<T> r2)` | Compares if greater or equal to: `r1 >= r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmpgt` | `>` | `Msk<N> cmpgt (const Reg<T> r1, const Reg<T> r2)` | Compares if strictly greater than: `r1 > r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmple` | `<=` | `Msk<N> cmple (const Reg<T> r1, const Reg<T> r2)` | Compares if lower or equal to: `r1 <= r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cmplt` | `<` | `Msk<N> cmplt (const Reg<T> r1, const Reg<T> r2)` | Compares if strictly lower than: `r1 < r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

### Conversions and Packing

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `toReg` | `Reg<T> toReg (const Msk<N> m)` | Converts the mask `m` into a register of type `T`, the number of elements `N` has to be the same for the mask and the register. If the mask is `false` then all the bits of the corresponding element are set to 0, otherwise if the mask is `true` then all the bits are set to 1 (be careful, for float datatypes `true` is interpreted as NaN!). | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `cvt` | `Reg<T2> cvt (const Reg<T1> r)` | Converts the elements of `r` into an other representation (the new representation and the original one have to have the same size). | `float -> int32_t`, `float -> uint32_t`, `int32_t -> float`, `uint32_t -> float`, `double -> int64_t`, `double -> uint64_t`, `int64_t -> double`, `uint64_t -> double` |
| `cvt` | `Reg<T2> cvt (const Reg_2<T1> r)` | Converts elements of `r` into bigger elements (in bits). | `int8_t -> int16_t`, `uint8_t -> uint16_t`, `int16_t -> int32_t`, `uint16_t -> uint32_t`, `int32_t -> int64_t`, `uint32_t -> uint64_t` |
| `pack` | `Reg<T2> pack (const Reg<T1> r1, const Reg<T1> r2)` | Packs elements of `r1` and `r2` into smaller elements (some information can be lost in the conversion). | `int32_t -> int16_t`, `uint32_t -> uint16_t`, `int16_t -> int8_t`, `uint16_t -> uint8_t` |

## Arithmetic Operations

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| `add` | `+` and `+=` | `Reg<T> add (const Reg<T> r1, const Reg<T> r2)` | Performs the arithmetic addition: `r1 + r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| `sub` | `-` and `-=` | `Reg<T> sub (const Reg<T> r1, const Reg<T> r2)` | Performs the arithmetic subtraction: `r1 - r2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `mul` | `*` and `*=` | `Reg<T> mul (const Reg<T> r1, const Reg<T> r2)` | Performs the arithmetic multiplication: `r1 * r2`. | `double`, `float`, `int32_t`, `int16_t`, `int8_t` |
| `div` | `/` and `/=` | `Reg<T> div (const Reg<T> r1, const Reg<T> r2)` | Performs the arithmetic division: `r1 / r2`. | `double`, `float` |
| `fmadd` | | `Reg<T> fmadd (const Reg<T> r1, const Reg<T> r2, const Reg<T> r3)` | Performs the fused multiplication and addition: `r1 * r2 + r3`. | `double`, `float` |
| `fnmadd` | | `Reg<T> fnmadd (const Reg<T> r1, const Reg<T> r2, const Reg<T> r3)` | Performs the negative fused multiplication and addition: `-(r1 * r2) + r3`. | `double`, `float` |
| `fmsub` | | `Reg<T> fmsub (const Reg<T> r1, const Reg<T> r2, const Reg<T> r3)` | Performs the fused multiplication and subtraction: `r1 * r2 - r3`. | `double`, `float` |
| `fnmsub` | | `Reg<T> fnmsub (const Reg<T> r1, const Reg<T> r2, const Reg<T> r3)` | Performs the negative fused multiplication and subtraction: `-(r1 * r2) - r3`. | `double`, `float` |
| `min` | | `Reg<T> min (const Reg<T> r1, const Reg<T> r2)` | Selects the minimum: `r1_i < r2_i ? r1_i : r2_i`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| max | | `Reg<T> max (const Reg<T> r1, const Reg<T> r2)` | Selects the maximum: `r1_i > r2_i ? r1_i : r2_i`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| div2 | | `Reg<T> div2 (const Reg<T> r)` | Performs the arithmetic division by two: `r / 2`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| div4 | | `Reg<T> div4 (const Reg<T> r)` | Performs the arithmetic division by four: `r / 4`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| abs | | `Reg<T> abs (const Reg<T> r)` | Computes the absolute value of `r`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| sqrt | | `Reg<T> sqrt (const Reg<T> r)` | Computes the square root of `r`. | `double`, `float` |
| rsqrt | | `Reg<T> rsqrt (const Reg<T> r)` | Computes the reciprocal square root of `r`: `1 / sqrt(r)`. | `double`, `float` |
| sat | | `Reg<T> sat (const Reg<T> r, const T minv, const T maxv)` | Saturates the register values: `max(min(r, minv), maxv)`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| neg | | `Reg<T> neg (const Reg<T> r, const Msk<N> m)` | Negates the register elements following the mask values: `m_i ? -r_i : r_i`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| neg | | `Reg<T> neg (const Reg<T> r1, const Reg<T> r2)` | Negates the register elements following the last register values: `r2_i < 0 ? -r1_i : r1_i`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |
| sign | | `Msk<N> sign (const Reg<T> r)` | Returns the sign: `r < 0`. | `double`, `float`, `int64_t`, `int32_t`, `int16_t`, `int8_t` |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| round | | Reg<T> round (const Reg<T> r) | Rounds the register values: fractional_part(r) >= 0.5 ? integral_part(r) + 1 : integral_part(r). | double, float |
| trunc | | Reg<T> trunc (const Reg<T> r) | Truncates the register values: integral_part(r) . | double, float |

### Arithmetic Operations on Complex Numbers

The complex operations are exclusively performed on Regx2<T> objects (one Regx2<T> object contains two Reg<T> hardware registers). Each Regx2<T> object contains mipp::N<T>() complex number. If we declare a Regx2<T> cmplx object, the cmplx[0] register will contain the real part of the complex numbers and cmplx[1] will contain the imaginary part. Depending on how you stored your complex numbers in memory you can need to use reordering before calling a complex operation. For instance, if you choose to store the complex numbers in a mixed format like this: r0, i0, r1, i1, r2, i2, ..., rn, in you will need to call the mipp::deinterleave operation before and the mipp::interleave operation after the complex operation.

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| cadd | + and += | Regx2<T> cadd (const Regx2<T> r1, const Regx2<T> r2) | Performs the complex addition: r1 + r2. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| csub | - and -= | Regx2<T> csub (const Regx2<T> r1, const Regx2<T> r2) | Performs the complex subtraction: r1 - r2. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| cmul | * and *= | Regx2<T> cmul (const Regx2<T> r1, const Regx2<T> r2) | Performs the complex multiplication: r1 * r2. | double, float, int32_t, int16_t, int8_t |
| cdiv | / and /= | Regx2<T> cdiv (const Regx2<T> r1, const Regx2<T> r2) | Performs the complex division: r1 / r2. | double, float |
| cmulconj | | Regx2<T> cmulconj (const Regx2<T> r1, const Regx2<T> r2) | Performs the complex multiplication with conjugate: r1 * conj(r2). | double, float, int32_t, int16_t, int8_t |

| Short name | Operator | Prototype | Documentation | Supported types |
|---|---|---|---|---|
| conj | | Regx2<T> cmulconj (const Regx2<T> r) | Computes the conjugate: conj(r). | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| norm | | Reg <T> norm (const Regx2<T> r) | Computes the squared magnitude: norm(r). | double, float, int32_t, int16_t, int8_t |

Reductions (Horizontal Functions)

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| hadd or sum | T hadd (const Reg<T> r) | Sums all the elements in the register r: r_1 + r_2 + ... + r_n. | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| hmul | T hmul (const Reg<T> r) | Multiplies all the elements in the register r : r_1 * r_2 * ... * r_n. | double, float, int64_t, int32_t, int16_t, int8_t |
| hmin | T hmin (const Reg<T> r) | Selects the minimum element in the register r : min(min(min(..., r_1), r_2), r_n). | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| hmax | T hmax (const Reg<T> r) | Selects the maximum element in the register r : max(max(max(..., r_1), r_2), r_n). | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| testz | bool testz (const Reg<T> r1, const Reg<T> r2) | Mainly tests if all the elements of the registers are zeros: r = (r1 & r2); !(r_1 OR r_2 OR ... OR r_n). | double, float, int64_t, uint64_t, int32_t, uint32_t, int16_t, uint16_t, int8_t, uint8_t |
| testz | bool testz (const Msk<N> m1, const Msk<N> m2) | Mainly tests if all the elements of the masks are zeros: m = (m1 & m2); ! (m_1 OR m_2 OR ... OR m_n). | |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `testz` | `bool testz (const Reg<T> r)` | Tests if all the elements of the register are zeros: `!(r_1 OR r_2 OR ... OR r_n)`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |
| `testz` | `bool testz (const Msk<N> m)` | Tests if all the elements of the mask are zeros: `!(m_1 OR m_2 OR ... OR m_n)`. | |
| `Reduction<T,OP>` | `T Reduction<T,OP>::sapply (const Reg<T> r)` | Generic reduction operation, can take a user defined operator `OP` and will performs the reduction with it on `r`. | `double`, `float`, `int64_t`, `uint64_t`, `int32_t`, `uint32_t`, `int16_t`, `uint16_t`, `int8_t`, `uint8_t` |

Math Functions

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `exp` | `Reg<T> exp (const Reg<T> r)` | Computes the exponential of `r`. | `double` (only on `icpc`), `float` |
| `log` | `Reg<T> log (const Reg<T> r)` | Computes the logarithm of `r`. | `double` (only on `icpc`), `float` |
| `sin` | `Reg<T> sin (const Reg<T> r)` | Computes the sines of `r`. | `double` (only on `icpc`), `float` |
| `cos` | `Reg<T> cos (const Reg<T> r)` | Computes the cosines of `r`. | `double` (only on `icpc`), `float` |
| `tan` | `Reg<T> tan (const Reg<T> r)` | Computes the tangent of `r`. | `double` (only on `icpc`), `float` |
| `sincos` | `void sincos (const Reg<T> r, Reg<T>& s, Reg<T>& c)` | Computes at once the sines (in `s`) and the cosines (in `c`) of `r`. | `double` (only on `icpc`), `float` |
| `sincos` | `Regx2<T> sincos (const Reg<T> r)` | Computes and returns at once the sines and the cosines of `r`. | `double` (only on `icpc`), `float` |
| `cossin` | `Regx2<T> cossin (const Reg<T> r)` | Computes and returns at once the cosines and the sines of `r`. | `double` (only on `icpc`), `float` |
| `sinh` | `Reg<T> sinh (const Reg<T> r)` | Computes the hyperbolic sines of `r`. | `double` (only on `icpc`), `float` |

| Short name | Prototype | Documentation | Supported types |
|---|---|---|---|
| `cosh` | `Reg<T> cosh (const Reg<T> r)` | Computes the hyperbolic cosines of `r`. | `double` (only on `icpc`), `float` |
| `tanh` | `Reg<T> tanh (const Reg<T> r)` | Computes the hyperbolic tangent of `r`. | `double` (only on `icpc`), `float` |
| `asinh` | `Reg<T> asinh (const Reg<T> r)` | Computes the inverse hyperbolic sines of `r`. | `double` (only on `icpc`), `float` |
| `acosh` | `Reg<T> acosh (const Reg<T> r)` | Computes the inverse hyperbolic cosines of `r`. | `double` (only on `icpc`), `float` |
| `atanh` | `Reg<T> atanh (const Reg<T> r)` | Computes the inverse hyperbolic tangent of `r`. | `double` (only on `icpc`), `float` |

# ARM SVE

## SVE Length Specific

An ARM SVE version is under construction. This version uses *SVE length specific* which is more appropriated to the MIPP architecture. This way, the size of the *MIPP registers* is defined at the compilation time. As a reminder, the vector length can vary from a minimum of 128 bits up to a maximum of 2048 bits, at 128-bit increments. On GNU and Clang compilers, it is specified at the compilation time with the `-msve-vector-bits=<size>` flag.

## Supported MIPP Operations

- **Memory operations:** `load`, `store`, `blend`, `set`, `set1`, `gather`, `scatter`, `maskzld`, `maskst`, `maskzgat`, `masksca`
- **Logical comparisons:** `cmpeq`, `cmneq`
- **Bitwise operations:** `andb`, `notb` (msk)
- **Arithmetic operations:** `fmadd`, `add`, `sub`, `mul`, `div`
- **Reductions:** `testz` (msk), `Reduce<T, add>`

*Byte* and *word* operations are not yet implemented.

# How to cite MIPP

We recommend you to cite the following article:

- Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux and Christophe Jégo, **MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard**, *The 5th International Workshop on Programming Models for SIMD/Vector Processing (WPMVP 2018), February 2018.*