

## Lab “Yuv Viewer”

The goal of this lab is to draw a picture stored in YUV420 format in the process, learn new C++ language features such as the “*intrinsic*” library which facilitates the use of SIMD (Single Instruction Multiple Data) instructions.

### 1 - YUV Format

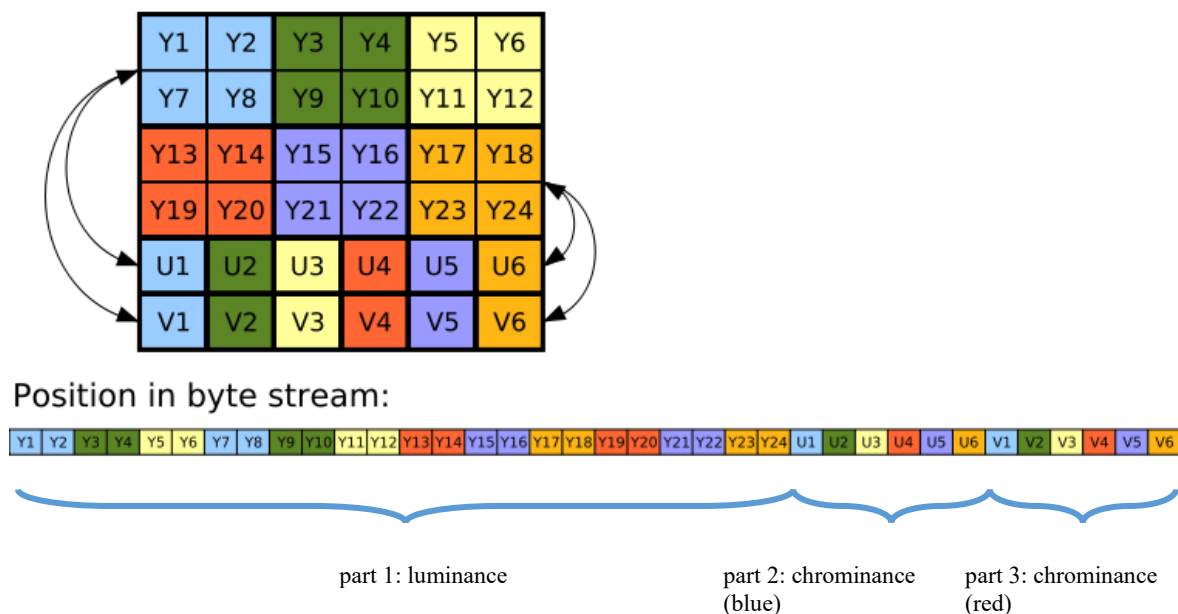
The YUV420 image format is mainly used in numerical TV broadcast as the first transformation step before image compression (H.264 or HEVC).

The format for the 8-bit YUV images is a very simple non-compressed format. For an image of size ( $width=w, height=h$ ), a byte stream of size  $3wh/2$  bytes can be decomposed in 3 parts:

- 1) A first block of size  $w \cdot h$ , storing the luminance, 1 byte per pixel of the image.
- 2) A second block of size  $(w \cdot h)/4$ , storing the U chrominance (blue), 1 byte per 4 pixels.
- 3) A third block of size  $(w \cdot h)/4$ , storing the Y chrominance (red), 1 byte per 4 pixels.

Note: for YUV image, the weight and height are always even numbers. The total number of bytes in a YUV is always a multiple of 6.

The following picture (from wikipedia) illustrates these 3 parts very well:





To be displayed on a monitor, the YUV image must be converted to RGB using the following matrix transformation

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M \begin{pmatrix} Y \\ U \\ V \\ 1 \end{pmatrix}$$

The matrix coefficients are given below:

$$M = \begin{matrix} & \begin{matrix} 1.1643836 & 0.0000000 & 1.8765140 & -268.2065015 \\ 1.1643836 & -0.2132486 & -0.5578116 & 82.8546329 \\ 1.1643836 & 2.1124018 & 0.0000000 & -289.0175656 \\ 0.0000000 & 0.0000000 & 0.0000000 & 1.0000000 \end{matrix} \end{matrix}$$

## 2 - Activity #1: Load the Calibration Image

Using the archive given in the email, extract and compile the project, verify that you can load the calibration image without problem (YUV image can be downloaded [here](#))

Resulting Image:



### 3 - Activity #2: Using Integer

Looking in the code, the function `convert_yuv_to_rgb()` uses floating point numbers. Find a way to convert this function to a similar one using integer only.

Once your code is ready, try it again on the calibration image, you should not see noticeable difference.

```
static
QRgb convert_yuv_to_rgb(uint8_t y, uint8_t u, uint8_t v) {
    auto yi = static_cast<double>(static_cast<uint>(y));
    auto ui = static_cast<double>(static_cast<uint>(u));
    auto vi = static_cast<double>(static_cast<uint>(v));

    double pyi = 1.1643836 * yi;

    double r = pyi + 1.8765140 * vi - 268.2065015;
    double g = pyi - 0.2132486 * ui - 0.5578116 * vi + 82.8546329;
    double b = pyi + 2.1124018 * ui - 289.0175656;

    Util::Clamp<double> clamp_to_rgb(0, 255);

    return QRgb( clamp_to_rgb(r), clamp_to_rgb(g), clamp_to_rgb(b));
}
```

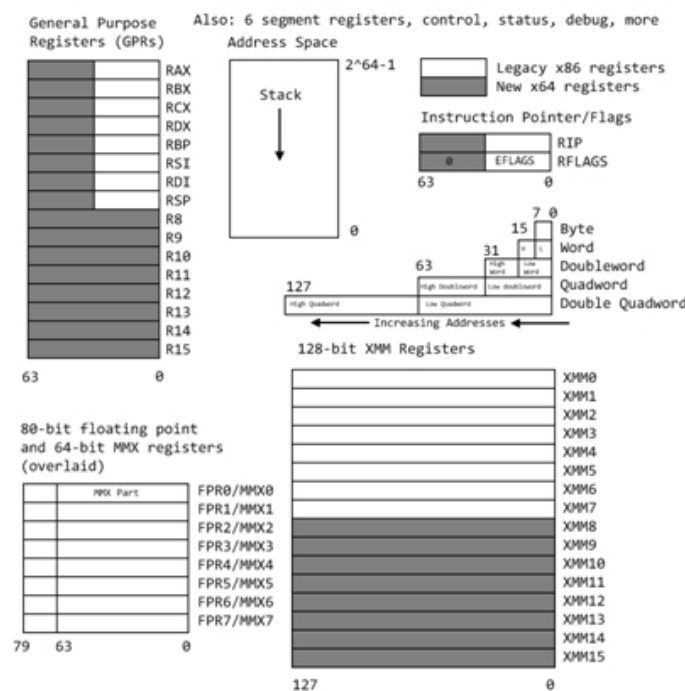
## 4 - Activity #3: Using SIMD

### 4.1 Introduction

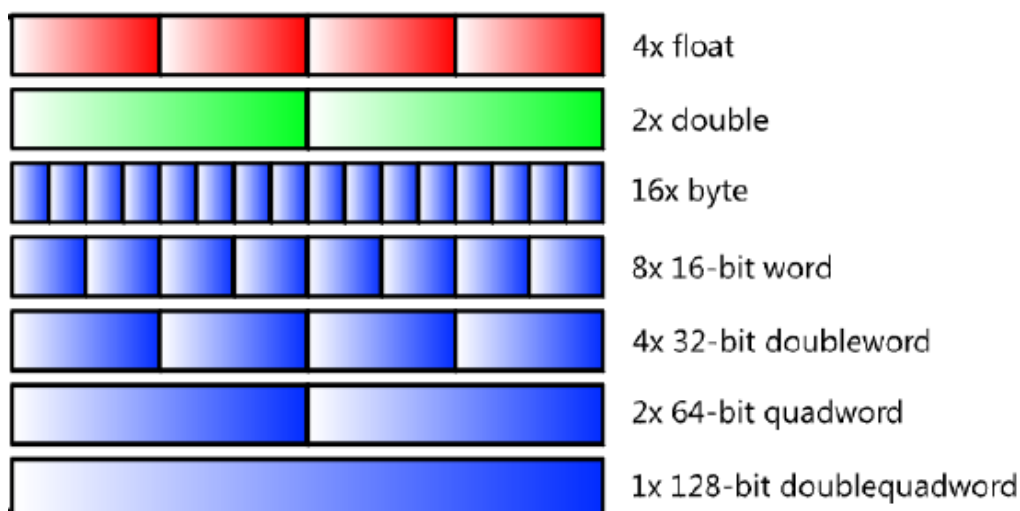
All modern CPUs have SIMD instructions ([AVX Intel](#), [Neon ARM](#)).

We'll cover the Intel SIMD.

Intel offers 16 or 32 registers of 128, 256 or 512 bits depending on the core architecture. The figure below shows a very common configuration with 16 registers of 128 bits.

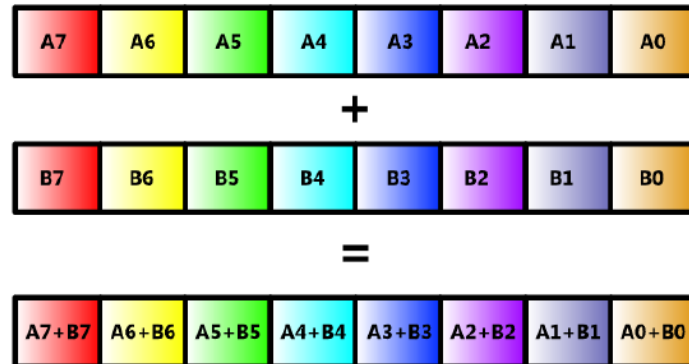


A single 128 bit register  $XMM_i$  can be viewed in multiple different ways:



Intel has developed a library compatible with C++ which facilitates the use of SIMD: the intrinsic library.

For example to perform 8 16-bit additions in a single operation, shown below:



You write:

```
#include "emmintrin.h"
#include "immintrin.h"

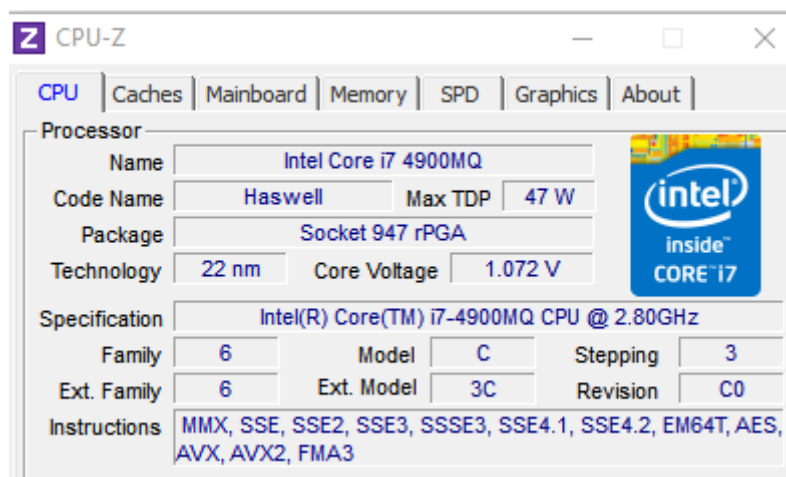
...

__m128i r128_a = ...; // see next page on how to initialize registers
__m128i r128_b = ...; // same

__m128i r128_r = _mm_add_epi16(r128_a, r128_b);
```

Note the introduction of a new type: `__m128i` defined by Intel, additional type `__m256i` and `__m512i` are available depending on your architecture.

Many utilities are available to provide details on the architecture: look for SSE (128 bit registers) AVX (256 bit registers) or AVX2 (512 bit registers).

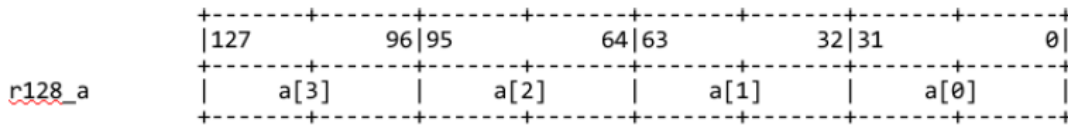


In this lab, we shall use only the 128 bit instructions which are available on all laptops.

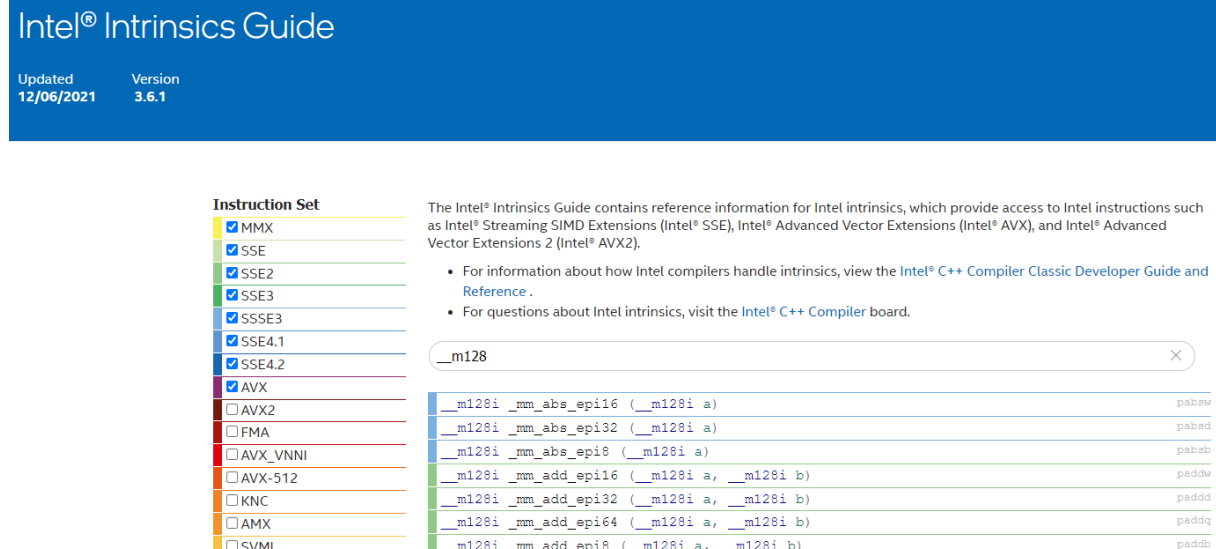
## 4.2 To Do

You have to update again the function `convert_yuv_to_rgb()` to use the SIMD 128 bits instructions.

Think of a 128 bit registers as 4 independent word of 32 bits:



Searching the web side: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/> you can search the instructions which could help you:



**Intel® Intrinsics Guide**

Updated 12/06/2021 Version 3.6.1

**Instruction Set**

- ☒ MMX
- ☒ SSE
- ☒ SSE2
- ☒ SSE3
- ☒ SSE3
- ☒ SSE4.1
- ☒ SSE4.2
- ☒ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX\_VNNI
- ☐ AVX-512
- ☐ KNC
- ☐ AMX
- ☐ SVM

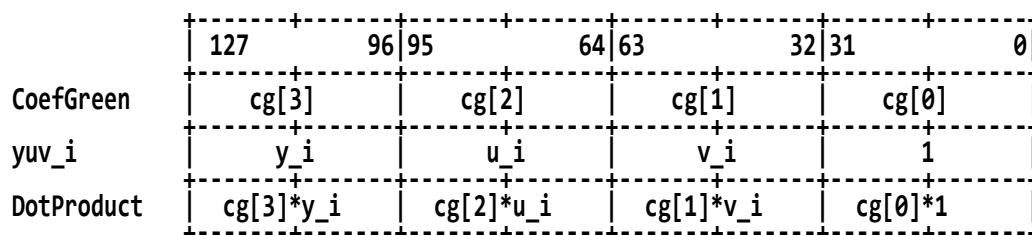
The Intel® Intrinsics Guide contains reference information for Intel intrinsics, which provide access to Intel instructions such as Intel® Streaming SIMD Extensions (Intel® SSE), Intel® Advanced Vector Extensions (Intel® AVX), and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- For information about how Intel compilers handle intrinsics, view the [Intel® C++ Compiler Classic Developer Guide and Reference](#).
- For questions about Intel intrinsics, visit the [Intel® C++ Compiler board](#).

Search:

<code>_mm128i_mm_abs_epi16</code>	<code>(__m128i a)</code>	psbaw
<code>_mm128i_mm_abs_epi32</code>	<code>(__m128i a)</code>	psabd
<code>_mm128i_mm_abs_epi8</code>	<code>(__m128i a)</code>	psabw
<code>_mm128i_mm_add_epi16</code>	<code>(__m128i a, __m128i b)</code>	psaddw
<code>_mm128i_mm_add_epi32</code>	<code>(__m128i a, __m128i b)</code>	psaddq
<code>_mm128i_mm_add_epi64</code>	<code>(__m128i a, __m128i b)</code>	psaddq
<code>_mm128i_mm_add_epi8</code>	<code>(__m128i a, __m128i b)</code>	psaddb

For example, the instruction: `_mm_mullo_epi32` can do 4 multiplications of 32 bits numbers in a single instruction as shown below:



The dot product is a good start for matrix multiplication, you will have to search for the other instructions.

### 4.3 Loading of 128-bit Registers

2 instructions `_mm_load_si128(__m128i const* mem_addr)` and `_mm_loadu_si128(__m128i const* mem_addr)` can be used (differences on alignment constraints).

The first instruction requires that the input data be aligned to a multiple of 16 bytes which allows better optimization of cache lines.

You can use the following piece of code to guarantee proper alignment.

```
#define ALIGN32(X) X __attribute__((aligned(32)))

static int32_t ALIGN32(coef_red[]) = { 298, 0, /*hidden to student */ };
static int32_t ALIGN32(coef_green[]) = { 298, /* hidden to student */ };
static int32_t ALIGN32(coef_blue[]) = { 298, /* hidden to student */ };
```

### 4.4 Unloading of 128-bit Registers.

The instruction `_mm_extract_epi32` will allow you to store the register content to a C++ variable:

```
int sum = _mm_extract_epi32(r128_tmp3, 0);
```

Now you have all the piece of the puzzle to complete the task.

Good luck



## 5 - Appendix: Dividing by a power of 2.

To perform a integer division by a power of 2, you can use a right shift without forgetting the rounding management (round to nearest).

The formula is:

$$\frac{n}{2^q} = (n + 2^{q-1}) \gg q$$