2.	Fachgebiet Architektur eingebetteter Systeme **Advanced Computer Architectures**	SoSe2017

Assignment start: 16.05.2017 Submission deadline: 05.06.2017

Assignment 2 - SIMD 25 Points

In the second assignment you will use x86 SIMD extensions to accelerate the HEVC/H.265 16×16 inverse transform. Single Instruction Multiple Data instructions are useful in many applications to improve performance. On modern processors these instructions are always available in sometimes many extension sets. For x86 the MMX (64-bit), SSE, SSE2, SSSE3, SSE4.1, SSE4.2 (128-bit), AVX, AVX2 (256-bit) and FMA3–4 extensions are available. For conventional C statements, however, these instructions are typically left unused by the compiler. Sometimes, the compiler can auto-vectorize certain loops, but at the moment it rarely generates satisfactory code. To use these instruction programmers usually either write direct assembly (gcc inline asm, yasm, nasm) or use intrinsics.

In the lab you will use intrinsics, which provide a more convenient interface for using these SIMD instructions to programmers. The programmer still has to perform the so called vectorization, but is relieved from performing register allocation and spilling. An example of using SIMD instructions using intrinsics is provided in Listing 1, which performs a square and an add on each element of an array of 100 32-bit integers using SSE*.

Listing 1: Example of using x86 SSE intrinsics

```
0 int input[100]  __attribute__ ((aligned (16)));
1 int output[100] __attribute__ ((aligned (16)));
2
3 //scalar code
4 for (int i=0; i<100; i++){
5   output[i] = input[i]*input[i] + 3;
6 }
7
8 //intrinsics code
9 __m128i *in_vec = (__m128i *) input;          //cast to SIMD vector type
10 __m128i *out_vec = (__m128i *) output;
11 __m128i xmm0;
12 //set a vector to (0x00000003000000030000000300000003)
13 __m128i add3 = _mm_set1_epi32(3);
14 for (int i=0; i<25; i++){
15   xmm0 = _mm_load_si128 (&in_vec[i]);        //load 4 integers
16   xmm0 = _mm_mullo_epi32(xmm0, xmm0);        //square 4 integers
17   xmm0 = _mm_add_epi32  (xmm0, add3);        //add 4 integers
18   _mm_store_si128(&out_vec[i], xmm0);        //store back 4 integers
19 }
```

And another example in Listing 2 which transposes a 4x4 integer matrix.

The inverse discrete cosine transform (IDCT) used in this assignment is based on the fast 16×16 transform of the HEVC/H.265 video codec. This 2D IDCT can be decomposed in a 1D row and column IDCT, which is employed in the provided scalar implementation. The scalar code along with other documents can be downloaded from the ISIS page. Your assignment is to create the SIMD intrinsics version of the IDCT. The correctness and timing checking functionality is also available in

Listing 2: Transposing a 4x4 matrix with SSE

```
int matrix[4][4] __attribute__ ((aligned (16)));

//intrinsics code
__m128i *matrix_vec = (__m128i *) matrix;

__m128i I0 = _mm_load_si128 (&matrix_vec[0]);    //load 4x4 matrix
__m128i I1 = _mm_load_si128 (&matrix_vec[1]);
__m128i I2 = _mm_load_si128 (&matrix_vec[2]);
__m128i I3 = _mm_load_si128 (&matrix_vec[3]);

__m128i T0 = _mm_unpacklo_epi32(I0, I1);
__m128i T1 = _mm_unpacklo_epi32(I2, I3);
__m128i T2 = _mm_unpackhi_epi32(I0, I1);
__m128i T3 = _mm_unpackhi_epi32(I2, I3);

I0 = _mm_unpacklo_epi64(T0, T1);             //Assigning transposed values back
    into I[0-3]
I1 = _mm_unpackhi_epi64(T0, T1);
I2 = _mm_unpacklo_epi64(T2, T3);
I3 = _mm_unpackhi_epi64(T2, T3);

_mm_store_si128 (&matrix_vec[0], I0);            //store transposed 4x4 matrix
_mm_store_si128 (&matrix_vec[1], I1);
_mm_store_si128 (&matrix_vec[2], I2);
_mm_store_si128 (&matrix_vec[3], I3);
```

the source file.

A few tips and rules:

- Intel has a nice tool, the "Intel intrinsics guide". This tool contains compact and convenient documentation of all the available SIMD intrinsics for x86. A recent version is in the assignment sources on ISIS. The latest version can also be downloaded at `http://software.intel.com/en-us/avx/`.**Try this tool before doing anything else.**
- Use only intrinsics up to (and including) SSE4.2. AVX and 3-operand mode SSE are not supported on the lab machines.
- Generating the input in a transposed manner is allowed.
- Do not unroll the timing loop. Do not inline the idct functions.
- Variations in execution time are normal, since the thin clients in the lab room connect to virtual machines. When evaluating the performance, run the code multiple times and consider the fastest time as representative.
- You can use `objdump` to see the generated assembly code.
- If you are not satisfied with the code generation of intrinsics you are allowed to use GCC inline assembly instead.
- When working from another machine it is necessary to check the support for newer instructions.

Your code will be tested on the lab machines, using **gcc 4.8.4**. This version of GCC is installed on the Ubuntu 14.04 versions of the lab machines. The compiler version on Ubuntu 16.04 is different and will **not** provide the same timing results.

The deliverable of this assignment is the *hevc_idct16.c* source file.