2.	Fachgebiet Architektur eingebetteter Systeme **Advanced Computer Architectures**	SoSe2018

Assignment start: 15.05.2018 Submission deadline: 05.06.2018

Assignment 2 - SIMD 25 Points

In the second assignment you will use x86 SIMD extensions to accelerate a 2D Haar transform on a 16×16 byte matrix.

Single Instruction Multiple Data instructions (like SSE and AVX on x86) are useful in many applications to improve performance. While the compiler is capable of automatically vectorizing certain loops, it will often not produce satisfactory code for more complicated cases. In this case the programmer may perform manual vectorization, either directly using assembly, or through vector intrinsics.

In the lab you will use intrinsics, which provide a more convenient interface for using SIMD instructions. The programmer still has to perform the vectorization, but is relieved from register allocation and instruction scheduling decisions. An example of using SIMD 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

```
int input[100]  __attribute__ ((aligned (16)));
int output[100] __attribute__ ((aligned (16)));

// scalar code
for (int i=0; i<100; i++) {
  output[i] = input[i]*input[i] + 3;
}

// intrinsics code
__m128i *in_vec = (__m128i *) input;        // cast to SIMD vector type
__m128i *out_vec = (__m128i *) output;
__m128i xmm0;
// set a vector to (0x00000003000000030000000300000003)
__m128i add3 = _mm_set1_epi32(3);
for (int i=0; i<25; i++){
  xmm0 = _mm_load_si128 (&in_vec[i]);       // load 4 integers
  xmm0 = _mm_mullo_epi32(xmm0, xmm0);       // square 4 integers
  xmm0 = _mm_add_epi32  (xmm0, add3);       // add 4 integers
  _mm_store_si128(&out_vec[i], xmm0);       // store back 4 integers
}
```

The 2D Haar transform used in this assignment is a simple wavelet transform. It can be decomposed into two consecutive 1D transforms, the first one operating on rows of a matrix, the second one on the columns. The row transform operates as shown in Figure 1, by combining two adjacent elements a and b with the operations $(a+b)/2$ and $(a-b)/2$ respectively. The operation is then applied recursively on the left half.

A scalar implementation of the Haar transform can be downloaded from the ISIS page.[1] Your as-

[1]The transform as implemented is not strictly correct (in particular the use of unsigned arithmetric does not produce meaningful results), but has been chosen to make the vectorization work less involved, taking into account the specific intrinsics available on the x86 architecture.

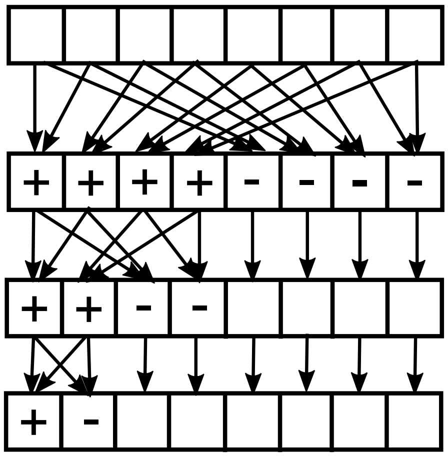

Figure 1: Schematic behavior of 1D Haar transform on a single 8-element row.

signment is to create a vectorized version of the transform with the goal of improving performance. Functionality to check correctness and benchmark your implementation is also available in the source file.

A few tips and rules:

- Intel provides an "Intel intrinsics guide", which contains compact and convenient documentation of all available SIMD intrinsics for x86. It is available only at `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. **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.
- The provided source code benchmarks both the full transform and the separate row and column transforms. However, if you think that you can achieve better performance by combining both steps, you can do so.
- In this specific case the column transform is much easier to vectorize than the row transform, so you should start there. For the row transform, you should familiarize yourself with different data movement intrinsics like shuffle, blend, unpack or alignr.
- 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.
- Your code will be tested on the lab machines, using **gcc 5.4.0**. This version of GCC is installed on the Ubuntu 16.04 versions of the lab machines. Using a different compiler version may produce different results. When working from another machine, make sure you don't use unsupported newer instructions.
- The deliverable of this assignment is the *haar.c* source file.