CSCI 2500: Computer Organization

Lab 10 – Exercises The Processor : Pipelining and SIMD

1) In this lab you will do some basic with with the Intel SSE extensions. There are a very large number of available SIMD intrinsics for use in programs. A full listing of of all of them can be found on Intel's website:

   https://software.intel.com/sites/landingpage/IntrinsicsGuide/

   Open the above web page page and once there, click the checkboxes for everything that begins with "SSE" (SSE all the way to SSE4.2). Do your best to interpret the new syntax and terminology (refer to the lecture slides for how to decode the instruction abbreviations). Find the 128-bit intrinsics for the following SIMD operations (one for each):

   a) Four floating point divisions in single precision (i.e. float)

   b) Sixteen max operations over signed 8-bit integers (i.e. char)

   c) Arithmetic shift right of eight signed 16-bit integers (i.e. short)

2) Loop unrolling is a technique that compilers can use to optimize a program's execution speed at the expense of its overall executable (or binary) size. Loop unrolling increases a program's speed by mitigating (or removing) instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration. As we discussed in class loop unrolling reduces branch penalties by reducing the number of control hazards. To eliminate branching overhead, loops can be re-written as a repeated sequence of similar independent statements. In this lab you the programmer will be doing the transformation manually.

   a) Download lab10.c from LMS; Labs → Lab 10 (11/18/2015). The file lab10.c is a skeleton program that contains functions you must complete. The first function you must complete is unrolled_sum. The unrolled_sum function must unroll the for loop given in the function, basic_sum. Once you complete the unrolled_sum function, compile with program with the following command (no optimization by the compiler):

   ```
   gcc -o lab10 -O0 lab10.c
   ```

   Run the program. What is the the speed up factor of the unrolled sum function?

b) The second function to complete in lab10.c is the vectorized_sum function. In this function you vectorize the computation of the array sum using the Intel SSE intrinsics. The following functions are useful in the implementation of the vectorized sum:

Function Name	Description
`__m128i _mm_setzero_si128( )`	Returns 128-bit zero vector
`__m128i _mm_loadu_si128(__m128i *p)`	Returns 128-bit vector stored at pointer p
`__m128i _mm_add_epi32(__m128i a, __m128i b )`	Returns vector (`a0+b0, a1+b1,...,a3+b3`)
`void _mm_storeu_si128(__m128i *p, __m128i a)`	Stores a 128-bit vector at pointer p

Once you complete the vector_sum function, compile with program with the following command (no optimization by the compiler):

```
gcc -o lab10 -O0 lab10.c
```

Run the program. What is the the speed up factor of the vectorized sum function compared to the basic sum function? How does that speed-up factor compare to the unrolled sum function?

c) For the final part of this exercise, try experimenting with different compiler optimization levels (-O1 through -O3) and note the change in the timing results. Are the results what you expected. Comment on the size of the executable with each change in optimization level.