Introduction to the Intel Pentium Architecture
SIMD extensions SSE and MMX

LAB TWO

SMD077
lp2 – 2001
1 **Objectives** ........................................................................................................ 2
2 **Prelab** ............................................................................................................. 2
3 **SSE/MMX instructions** .................................................................................... 2
4 **Assignment** ..................................................................................................... 3
   4.1 Step one - SSE instructions ........................................................................... 4
   4.2 Step two - MMX instructions ......................................................................... 4
   4.3 Step three - questions ...................................................................................... 5
   4.4 Hints ................................................................................................................. 5
5 **Goal** .................................................................................................................. 6
6 **Submission** ........................................................................................................ 7
7 **References** ......................................................................................................... 8
1 Objectives

The objective of this lab is to get an introduction to the SSE and MMX extensions on the Intel Pentium III architecture. The particular example is the same as in lab one, i.e. to implement a Finite Impulse Response (FIR) filter, but now it will be implemented using SSE and MMX functionality.

2 Prelab

Before you begin with the lab it is important that you go through the below listed items.

- Run the different tutorials in the Intel Architecture Performance Training Center 4.0 package. The different tutorials are mmx_cbt that describes the MMX, Ppar_cbt that describes the Pentium Pro architecture (that Pentium III is developed from) and Sfp_cbt that describes the SIMD floating Point (SSE). These tutorials are available at Start->Programs->Intel Architecture Performance Training Center 4.0.
- Read through the documents that are listed in the reference chapter.
- Read through the documents that are listed at the smd077 course homepage.

Especially useful when implementing the MMX version of the FIR filter is document [9] in the reference chapter that describes an echo implementation using MMX. For the SSE implementation, document [3] is very useful.

3 SSE/MMX instructions

The Intel MMX and SSE extensions to the Intel Architecture has been designed to greatly enhance multimedia and communication applications. These extensions include new registers and instructions which are combined with a Single-Instruction, Multiple-Data (SIMD) execution model to improve performance. The SIMD technique speeds up performance by allowing the same operation to be executed on multiple data elements in parallel. The MMX extension is intended to be used on 8-bit, 16-bit and 32-bit integers and the SSE extension is intended for 32-bit and 64-bit floats.

SSE/MMX is not a technology limited to the Intel x86 architecture. Other vendors offer similar, and possible binary compatible technologies. For example, AMD offers SIMD instructions via its 3Dnow instruction set. SSE/MMX instructions can be utilized to speed-up processing of relatively small data elements (8-bit, 16-bit, 32-bit and 64-bit words) which require the same instruction to operate on multiple data. Rule of thumb, the smaller data elements the more you can do in parallel and thus greater speed-up.
4 Assignment

The assignment in this lab is to implement the same FIR filter equation that was described in lab one, but this time it will be implemented using the SSE and MMX extensions to the Intel Pentium Architecture.

For you to be able to implement the MMX version of the FIR filter without the need of converting between floats and integers you will have to download new lab files (the lab2_2001.zip file) from the smd077 course homepage [4]. The updates in the new files consists of the following.

- The first update is that a new command prompt option, -mmx_int, have been added. This option is used to execute your MMX implementation of the FIR filter without the conversions between floats and shorts added to the RDTSC count. This mode calls the fir_mmx_int() function directly with shorts and it will be the one that is used to finally report your timing information. When the -mmx option is entered at the command prompt instead of the -mmx_int option, the function fir_mmx() will be called with floats. This function will then convert from floats to shorts and call the fir_mmx_int() function. The output from the fir_mmx_int() function will then be converted back and written to file. In this case the entire process of conversions and filtering is added to the RDTSC count. The reason why this has been added is that Cubase only accepts float as input (and therefore will use the fir_mmx() function) in it's plug-ins and also so that you will discover how much the performance degrades when you need to do type-conversions. The fir_mmx() function has been given to you.

- The interface to the fir_setup() function in fir.c have been modified. In addition to the previous arguments one argument, run_mode, has been added. This sets the mode that the application will run in, i.e. C, ASM, SSE, MMX or MMX_INT. These definitions have been added to the fir.h file. Since the application uses shorts for the MMX and MMX_INT mode and floats for the other modes you might want to do different setups depending on what mode you are running in. Some setup entries for the float to short conversion of the taps in the MMX and MMX_INT modes have been given to you.

- Similarly to the fir_setup() function you might want to do different cleanups in fir_cleanup() depending on what mode you are running in.

- As in the first assignment the code should support any blocksize (though we’re not testing for special case when blocksize is less then num_taps).

To make it easier for us to compare your code implemented in lab one to the SSE and MMX code that will be implemented in this lab we want you to do cut & paste between the fir.c from lab one and the new fir.c. Except for the above mentioned issues the prerequisites (wav files, filter files, etc.) for lab two is the same as in lab one.

To be able to use MMX efficiently all data need to be integers. The wav samples are easy to convert (they are actually just shorts typed to floats in wav.c). The problem is what to do with the taps, they can have values like 0.001. The answer is to use something called fixed point math. It means that you convert the floating point number to an integer with some bits representing the right half of the decimal point. The number of bits reserved to the decimal part is fixed, thereby fixed point math. A simple example is if you want to calculate 0.5*0.25 using 8 bit integers. If we decide that we should use four bits as the integer part and four bits as the decimal part it will result in the following calculations:

\[ A = 8 = 0.5*16 \ (\text{move the decimal left four steps}) \]
\[ B = 4 = 0.25*16 \]
\[ C = 32 = 8*4 \]
C is now in fixed point form so we have to convert it back. This is done by dividing it by 16*16 and gives 0.125 which is the same as 0.5*0.25.

You do not have to think about implementing this in the code since it has been done for you. It is though a very useful technique to know about if you need to get away from floating point arithmetic, so try to understand what we have done. Of course it will infer some rounding errors.

The SSE registers have a size of 128-bits and the MMX registers have a size of 64-bits. Since the incoming audio samples are 32-bit floats in the \texttt{fir\_sse()} function and 16-bit shorts in the \texttt{fir\_mmx\_int()} function it is possible to do computations of four samples in parallel with one SSE or MMX instruction. That is the basic idea behind SSE, MMX and comparable SIMD instructions - to enhance the computation speed of this algorithm when you have differing data values and want to perform the same independent operation on those data values.

Notice that when you with SSE/MMX are processing four samples in parallel, you need to have some special case for when some samples are samples from the previous call to \texttt{fir\_sse()} or \texttt{fir\_mmx\_int()} function (the history) and some samples are from the current input. You also need to handle the events when the number of taps and/or the number of samples are not divisible by four.

The bonus assignment for lab two will be presented on the lab web page in a later stage and it will as in lab one give a maximum of 5\% extra on the final exam.

4.1 Step one - SSE instructions

The first step is to implement the FIR filter algorithm using the SSE instructions. You should put your code in \texttt{fir\_sse()} in file \texttt{fir.c}. Use the documents in the reference chapter (especially document [3]) and go through the SSE tutorial in Intel Architecture Performance Training Center 4.0 to get good knowledge of how SSE works. How to reach the SSE tutorial is covered in the prelab chapter.

The SSE version is tested by running the program with the \texttt{-sse} option in the command prompt. Verify your implementation using \texttt{average\_diff.m} as described in lab one. When your are meeting the error requirements of less than 1*10^-8 for the test files proceed to step two.

4.2 Step two - MMX instructions

Now is the time to implement the FIR filter algorithm using the MMX instructions. You should put the code in the function \texttt{fir\_mmx\_int()} that also is in the file \texttt{fir.c}. Use the documents in the reference chapter (especially document [9]) and go through the MMX tutorial in Intel Architecture Performance Training Center 4.0 to get good knowledge of how MMX works. How to reach the MMX tutorial is covered in the prelab chapter.

The MMX version is tested by running the program with the \texttt{-mmx\_int} option in the command prompt. You can also use the \texttt{-mmx} option, but as described above it will include the type-conversions in the RDTSC count. It is very interesting to see the difference though to see how much conversions cost. And to see what you can gain if you do the conversions in a not time critical stage and uses faster integers in the time critical stage.

After a correct implementation using the MMX extension has been inserted and the verification with \texttt{average\_diff.m} shows that it is valid, it is possible to do a valid comparison of the speed-up obtained using these specialized instructions. This is done by comparing the RDTSC Instruction [2]. This should provide a fairly accurate estimate of the instruction count, and thus necessary time, to run the both versions of the algorithm. If done correctly, the instruction count for the SSE and MMX
versions should indicate a significant speed-up compared to the versions that was developed in lab one.

Having achieved the desired results in steps 1 & 2, the mandatory programming portion of this lab is now complete. There are some additional questions in section 4.3 which you will be required to answer along with your lab submissions. Please feel free to experiment further with the SSE and MMX extensions.

4.3 Step three - questions

Answer these questions the best you can.

Question 1) – As you have experienced, you must end the MMX section by the \texttt{emms} instruction
   \begin{enumerate}
   \item[1a)] Why is this needed for MMX
   \item[1b)] Why is this NOT needed for SSE
   \end{enumerate}

Question 2) – You should observe that the conversion between floating point and integer is quite time consuming. Explain why?

Question 3) – Will your implementation work on asymmetric filters i.e. \texttt{taps[i]} \neq \texttt{taps[numtaps - 1 - i]}? Explain why, or why not (you may actually go ahead and try an asymmetric filter)

See Goal and Submission where to write the answers and submit the lab

4.4 Hints

- You might find it valuable to develop a second version of the C-code which makes the outer and inner loops behave a bit more like the SSE or MMX-code will do. That is, a C-code which unrolls the outer and inner loops to fit the “loop-pattern” (number of times you do the outer loop) when using SSE or MMX instructions to do the multiplication and adding in the filter. Make sure that this function produce the correct output before transforming it to inline SSE/MMX assembler. You should only have to do the outer and inner loop \(\frac{1}{4}\) of the number of times you do it in your first C-code version (if you process one sample for each outer loop in the first version). Look at the speed-up if you did not do any loop unrolling in lab1, loops are costly.
- Use comments when coding assembler, especially when using SSE and MMX instructions since the code tends to become extensive.
5 Goal

In order to get this lab accepted you must

1. Develop a code that implements the filter algorithm using the SSE extension. This code should be put into \texttt{fir\_sse()} in file \texttt{fir.c}. This implementation has to be functionally correct and the error requirements have to be met.

2. Develop a second function that uses the MMX extension. This code should be put into \texttt{fir\_mmx\_int()} in file \texttt{fir.c}. This also has to be functionally correct and meet the error requirements.

3. Write a text-file, called \texttt{lab2.txt} which has your names (see below). Then it should contain timing information of your assembler SSE and MMX functions. You just have to cut & paste from the output of the program when running with the -sse or -mmx\_int option. This text file should also contain a description of what you have done (e.g. optimization techniques) and why.

4. Answers to the questions in section 4.3 (step three). Put these answers in \texttt{lab2.txt} (see below).

\texttt{lab2.txt} format

-------------------------------------------------------------
<table>
<thead>
<tr>
<th>Firstname1 Lastname1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firstname2 Lastname2</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
</tbody>
</table>
-------------------------------------------------------------

Timing information of \texttt{fir\_sse()} and \texttt{fir\_mmx\_int()}

-------------------------------------------------------------

Answer to questions

1a) 
1b) 
2) 
3) 

-------------------------------------------------------------

Optimization description

-------------------------------------------------------------
6 Submission

For submission of the lab, write an email with the format shown below. Attach the files `fir.c` and `lab2.txt`. Send the email to: smd077@sm.luth.se Note! For due date refer to the lab page [4]. Make sure to submit the lab before this deadline for full credit.

Email format

Subject: lab2 smd077
Attachments: fir.c, lab2.txt
Body of mail:
Firstname1 Lastname1 email address
Firstname2 Lastname2 email address

This is how we will test and verify your lab
We will compile your `fir.c` together with the C-program which first uses a "correct" implementation of the algorithm (at least with the requirements stated in the lab) on a wave file and then run the same file on your `fir_sse()` and `fir_mmx_int()` and compare the output. If the output is the same, within a marginal, as our "correct" function - you have passed part one and two. Also we will time the functions just to verify that your timings are within reasonable limits (they will not be the same of course) and look at your answers. If everything is ok - you have passed lab two and will receive reply by mail. If not passed you will have one week from the date you receive a reply (by mail) to correct and re-submit your lab.
7 References

   Instruction set summary for the Intel Architecture

   Application note describing how to access time stamp counter in Pentium processors
   (used in rtdsc_timing.h)

   Application note describing how they implement this filter algorithm using 16 filter coefficients

   Computer Architecture course info

   Additional information which may be helpful

   Additional information which may be helpful

   Reference guides for the Intel Architecture

   Application note containing further optimization of the filter algorithm

   Application note describing the implementation of an echo with the MMX extension