Using the RDTSC Instruction for Performance Monitoring

CONTENTS:

1.0. Introduction

Programmers are often puzzled by the erratic performance numbers they see when using the RDTSC (read-time stamp counter) instruction to monitor performance. This erratic behavior is not an instruction flaw; the RDTSC instruction will always give back a proper cycle count. The variations appear because there are many things that happen inside the system, invisible to the application programmer, that can affect the cycle count. This document will go over all the system events that can affect the cycle count returned by the RDTSC instruction, and will show how to minimize the effects of these events.

Those who want to quickly use the RDTSC instruction to test their code and are not concerned about learning the underlying behavior need only read section 2.2 and all of section 4.

Although all of the concepts discussed in this paper are applicable to any compiler, the code sections were designed specifically to work with the Microsoft® Visual C++® Compiler Version 5.0, and may require changes to work with other compilers.

2.0. The Time-Stamp Counter

2.1. How the Time-Stamp Counter Works

Beginning with the Pentium® processor, Intel processors allow the programmer to access a time-stamp counter. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor. The Intel time-stamp counter is a 64-bit MSR (model specific register) that is incremented every clock cycle. On reset, the time-stamp counter is set to zero.

To access this counter, programmers can use the RDTSC (read time-stamp counter) instruction. This instruction loads the high-order 32 bits of the register into EDX, and the low-order 32 bits into EAX.

Remember that the time-stamp counter measures "cycles" and not "time". For example, two hundred million cycles on a 200 MHz processor is equivalent to one second of real time, while the same number of cycles on a 400 MHz processor is only one-half second of real time. Thus, comparing cycle counts only makes sense on processors of the same speed. To compare processors of different speeds, the cycle counts should be converted into time units, where:

seconds = # cycles / frequency

Note: frequency is given in Hz, where: 1,000,000 Hz = 1 MHz

2.2. When to Use the Time-Stamp Counter

The time-stamp counter should not be used to perform general profiling of code. Many profiling tools exist that are easy to use and provide much more information than the simple cycle counts that RDTSC provides. One example is VTune, a profiling tool created by Intel Corporation. Some other performance monitoring tools are described in the Survey of Pentium® Processor Performance Monitoring Capabilities & Tools, although at the time of writing this document some links in the survey are out of date.

That being said, there are a few cases where the time-stamp counter can be useful. One such case occurs when a very fine cycle measurement is needed for a section of code. For example, it can give a good idea of how many cycles a few instructions might take versus another set of instructions. Another use is to get an average time

estimate for a function or section of code. Both of these methods will be described in detail in section 4.

2.3. Compiler Issues with RDTSC

Some compilers, including the Microsoft® Visual C++® Compiler Version 5.0, do not recognize the RDTSC instruction (or the CPUID instruction discussed in section 3.1) in inline assembly code. One work-around for this problem is to use "emit" statements at the top of the file, as shown below:

```
#define rdtsc __asm __emit 0fh __asm __emit 031h
#define cpuid __asm __emit 0fh __asm __emit 0a2h
```

This will replace all occurrences of RDTSC and CPUID with their corresponding opcodes in the compiled ".obj" file. This allows the processor to understand the instruction without the compiler actually knowing what the instruction is. Using them as #define statements allows the programmer to use the instructions in inline assembly code.

3.0. Issues Affecting the Cycle Count

3.1. Out-of-Order-Execution

The Pentium® II Processor and Pentium® Pro Processor both support out-of-order execution, where instructions are not necessarily performed in the order they appear in the source code. This can be a serious issue when using the RDTSC instruction, because it could potentially be executed before or after its location in the source code, giving a misleading cycle count.

For example, look at the following code section, written to measure the time it takes to perform a floating-point division:

```
rdtsc                   ; read time stamp
mov     time, eax       ; move counter into variable
fdiv                    ; floating-point divide
rdtsc                   ; read time stamp
sub     eax, time       ; find the difference
```

In the above example, the floating-point divide will take a long time to compute. Potentially, the RDTSC instruction following the divide could actually execute *before* the divide. If this happened, the cycle count would not take the divide into account.

In order to keep the RDTSC instruction from being performed out-of-order, a *serializing instruction* is required. A serializing instruction will force every preceding instruction in the code to complete before allowing the program to continue. One such instruction is the CPUID instruction, which is normally used to identify the processor on which the program is being run. For the purposes of this paper, the CPUID instruction will only be used to force the in-order execution of the RDTSC instruction.

For the example above, you could use the CPUID to insure proper time-stamp reading as follows:

```
cpuid                   ; force all previous instructions to comple
rdtsc                   ; read time stamp counter
mov     time, eax       ; move counter into variable
fdiv                    ; floating-point divide
cpuid                   ; wait for FDIV to complete before RDTSC
rdtsc                   ; read time stamp counter
sub     eax, time       ; find the difference
```

Now the RDTSC instructions will be guaranteed to complete at the desired time in the execution stream.

When using the CPUID instruction, however, the programmer must also take into account the cycles it takes for the instruction to complete, and subtract this from the recorded number of cycles. A strange quirk of the CPUID instruction is that it can take longer to complete the first couple of times it is called. Thus, the best policy is to call the instruction three times, measure the elapsed time on the third call, then subtract this measurement from all future measurements. A full example using these techniques is given in Appendix A.1.

Again, using a serializing instruction like CPUID is only necessary on an out-of-order execution processor, like the Pentium® II Processor and Pentium® Pro Processor. It is not necessary to use CPUID when running on the Pentium® Processor or Pentium® Processor with MMX™ Technology.

3.2. Data Cache and Instruction Cache

Using RDTSC on the same section of code can often produce very different results. This is often caused by cache effects. The first time a line of data or code is brought into cache, it takes a large number of cycles to transfer it into memory. Thus, to use RDTSC effectively these cache effects must be taken into account.

This section gives details on two different ways to handle cache effects. If a programmer is concerned with how many cycles a certain set of *instructions* takes to execute, and desires a repeatable measurement, he or she may wish to try to eliminate the cache effects. To find the average real-time execution time of a piece of code the cache effects should be left in.

Cache effects can only be removed entirely for small sections of code. The technique basically requires removing all effects of transactions between both memory to data cache and memory to instruction cache. To do this, both the instructions and data must be contained in the L1 cache, which is the cache closest to the processor. The technique of storing memory into a cache before it is actually used is known as "cache warming".

Warming the data cache is fairly straightforward. It simply requires "touching" the entire data set, so that it will be moved into the cache. The data set must be small, preferably below 1KB (ex: 128 doubles, 256 ints, and 1024 chars are each 1KB of data). Warming the instruction cache requires making a first pass through all instructions before the timing begins. This can only be accomplished by putting all instructions in a loop or a function call.

When leaving in cache effects, the misses to the data and instruction caches will occur differently in each loop iteration, producing a different cycle count. Thus, the cycle counts should be averaged over many loop iterations, to take into account these largely varying cycle counts.

The full implementation of both methods will be covered in section 4.

3.3. Register Overwriting

As discussed in section 2.3, some compilers do not implicitly recognize the RDTSC and CPUID function in inline assembly code. Compilers like Microsoft® Visual C++® 5.0 normally "guarantee" that any register affected by an inline assembly code section will not affect the C code around it. When overriding the compiler by using the emit statements, however, the compiler does not know those instructions are overwriting registers (RDTSC overwrites EAX and EDX, and CPUID overwrites EAX, EBX, ECX, and EDX). Thus, the compiler may not properly store away the affected registers, so this must be done manually by the programmer by pushing them onto the stack.

There are a few cases where this will not matter. If the code being time measured is a stand-alone section of code, completely surrounded by the calls to RDTSC, then the register overwriting cannot affect the code around it. If the measured code section is written in assembly, and the variables are actually used inside of this section, the compiler will handle the stack allocation itself. Finally, it will not matter if affecting the correctness of the code around the measured section is not an issue while cycle testing.

3.4. Counter Overflow

Especially on faster processors, the overflow of the time-stamp counter may come into play. As discussed above, the time-stamp counter is divided into a lower and upper counter. Thus, the time-stamp can be used with both 32 and 64 bits of precision. When using only the lowest 32 bits of precision, the counter can go up to 2^{32} cycles. On a 400 MHz processor, the total time between overflows is equal to:

$$2^{32} \text{ cycles} * (1 \text{ second}/400{,}000{,}000 \text{ cycles}) = 10.74 \text{ seconds}$$

When using *unsigned* numbers, a single overflow will not be a problem because the cycle count is computed as an absolute difference between two numbers, and even with the single overflow, the final result will still be correct.

If the test takes a time greater than the time between overflows, than the absolute difference will produce an incorrect result, and it will be necessary to use the full 64 bits of precision. On a 400 MHz processor, this would give a time between overflows of:

$$2^{64} \text{ cycles} * (1 \text{ second}/400{,}000{,}000 \text{ cycles}) = 1462.36 \text{ years}$$

As an example, look at the code below, which returns the run time of the Sleep() function to the nearest second (truncated) on a 150 MHz system:

```
unsigned time;

__asm   rdtsc                           // Read time stamp to EAX
__asm   mov             time, eax

Sleep (35000);                          // Sleep for n/1000 seconds

__asm   rdtsc
__asm   sub             eax, time       // Find the difference
__asm   mov             time, eax

time = time/150000000;                  // divide by 150 MHz
printf("Seconds:  %u\n", time);
```

When run on a 150 MHz processor, the low 32 bit counter flips about every 28.6 seconds. Thus, when the sleep time is 15000 (15 seconds), the correct result will be printed, but when 35000 (35 seconds) is input, the result claims 6 seconds, since the counter flips between 28 and 29 seconds. The proper way to write this code is to use the full 64 bits of the time-stamp counter as follows:

```
unsigned time, time_low, time_high;
unsigned mhz = 150000000;       // 150 MHz processor

__asm   rdtsc                           // Read time stamp to EAX
__asm   mov             time_low, eax
__asm   mov             time_high, edx

Sleep (35000);                          // Sleep for 2 seconds

__asm   rdtsc
__asm   sub             eax, time_low   // Find the difference
__asm   sub             edx, time_high
__asm   div             mhz             // Unsigned divide EDX:EAX by mhz
__asm   mov             time, eax

printf("Seconds:  %u\n", time);
```

This case would produce the correct result on a 150 MHz system for almost 3,900 years.

4.0. Using RDTSC Properly

4.1 Pentium® Pro Processors and Pentium® II Processors

This section discusses code samples to quickly use RDTSC to test your own code for two different scenarios.

The first scenario is measuring the time taken in a recurring function or other piece of code. In this case, the cache effects are left in, but the time is averaged, since each run through the code section will produce different results due to memory access times. The code in Appendix A.2. contains six sections of code that may be cut out and placed into an existing program, then used immediately without modification. All modified registers are saved so that the program will not affect the surrounding code. The sample also uses the full 64-bit counter, so that code of any length can be used.

The second scenario is for a small section of code, where repeatable results are desired. This scenario removes all cache effects, from both the data and instruction caches. This case will measure the time it takes to execute the instructions and all associated penalties, but does not take into account memory transfer time, which can radically alter the cycle count between runs. The code for this scenario is included in Appendix A.3. To use this code, simply replace the marked variables and code sections with your own program. This sample code should only be used for small sections of code (about 100 total lines of C code with minimal branching and a data set smaller than 1KB), since a larger code size could reintroduce cache effects and overrun the 32-bit counter used in the sample. It should also be used as a stand-alone section of code, since no register saving is done around the user code.

4.2 Pentium® Processors and Pentium® with MMX™ Technology Processors

All of the rules and code samples described in section 4.1 also apply to the Pentium® Processor and Pentium® with MMX™ Technology Processor. The only difference is, the CPUID instruction is not necessary for serialization. Thus, *all* calls to CPUID can be omitted from the code samples. Leaving the calls to CPUID in will not make the code incorrect, only slower.

5.0 Conclusion

Although the use of the time-stamp counter can be complicated by the many issues discussed in this paper, it is possible to isolate these issues on an independent basis and create a useful timing mechanism with the RDTSC instruction. Even if the code sections contained in this paper are not directly useful for a particular application, the concepts described are applicable whenever RDTSC is used.

A.0. Appendix: Code Samples

A.1. Using the CPUID Instruction for Serialization

```
#define cpuid __asm __emit 0fh __asm __emit 0a2h
#define rdtsc __asm __emit 0fh __asm __emit 031h

#include <stdio.h>

void main () {
        int time, subtime;
        float x = 5.0f;

        __asm {

                //  Make three warm-up passes through the timing routine to ma
                //  sure that the CPUID and RDTSC instruction are ready

                cpuid
                rdtsc
                mov     subtime, eax
                cpuid
                rdtsc
                sub     eax, subtime
                mov     subtime, eax

                cpuid
                rdtsc
                mov     subtime, eax
                cpuid
                rdtsc
                sub     eax, subtime
                mov     subtime, eax

                cpuid
                rdtsc
                mov     subtime, eax
                cpuid
                rdtsc
                sub     eax, subtime
                mov     subtime, eax    // Only the last value of subtime is k
                // subtime should now represent the overhead cost of the
                // MOV and CPUID instructions

                fld     x
                fld     x
                cpuid                   // Serialize execution
                rdtsc                   // Read time stamp to EAX
                mov     time, eax
                fdiv                    // Perform division
                cpuid                   // Serialize again for time-stamp read
                rdtsc
                sub     eax, time       // Find the difference
                mov     time, eax
        }

        time = time - subtime;  // Subtract the overhead

        printf ("%d\n", time);  // Print total time of divide to screen
}
```

A.2. Taking a Time Average for a Repeated Code Section

```
// This code will find an average number of cycles taken to go
// through a loop.  There is no cache warming, so all cache effects
// are included in the cycle count.
```

```
// To use this in your own code, simply paste in the six marked
// sections into the designated locations in your code.

#include <stdio.h>
#include <windows.h>

//  *** BEGIN OF INCLUDE SECTION 1
//  *** INCLUDE THE FOLLOWING DEFINE STATEMENTS FOR MSVC++ 5.0
#define CPUID __asm __emit 0fh __asm __emit 0a2h
#define RDTSC __asm __emit 0fh __asm __emit 031h
//  *** END OF INCLUDE SECTION 1

#define SIZE 5

//  *** BEGIN OF INCLUDE SECTION 2
//  *** INCLUDE THE FOLLOWING FUNCTION DECLARATION AND CORRESPONDING
//  *** FUNCTION (GIVEN BELOW)
unsigned FindBase();
//  *** END OF INCLUDE SECTION 2

void main () {

        int i;

        //  *** BEGIN OF INCLUDE SECTION 3
        //  *** INCLUDE THE FOLLOWING DECLARATIONS IN YOUR CODE
        //      *** IMMEDIATELY AFTER YOUR DECLARATION SECTION.
        unsigned base=0, iterations=0, sum=0;
        unsigned cycles_high1=0, cycles_low1=0;
        unsigned cycles_high2=0, cycles_low2=0;
        unsigned __int64 temp_cycles1=0, temp_cycles2=0;
        __int64 total_cycles=0;  // Stored signed so it can be converted
                                 // to a double for viewing
        double   seconds=0.0L;
        unsigned mhz=150;       // If you want a seconds count instead
                                // of just cycles, enter the MHz of your
                                // machine in this variable.
        base=FindBase();
        //      *** END OF INCLUDE SECTION 3

        for (i=0; i<SIZE; i++) {

                //  *** BEGIN OF INCLUDE SECTION 4
                //  *** INCLUDE THE FOLLOWING CODE IMMEDIATELY BEFORE SECTION
                //  *** TO BE MEASURED
                __asm {
                        pushad
                        CPUID
                        RDTSC
                        mov     cycles_high1, edx
                        mov     cycles_low1, eax
                        popad
                }
                //  *** END OF INCLUDE SECTION 4

                //  User code to be measured is in this section.
                Sleep(3000);

                //  *** BEGIN OF INCLUDE SECTION 5
                //  *** INCLUDE THE FOLLOWING CODE IMMEDIATELY AFTER SECTION
                //  *** TO BE MEASURED
                __asm {
                        pushad
```

```
                        CPUID
                        RDTSC
                        mov     cycles_high2, edx
                        mov     cycles_low2, eax
                        popad
                }
                //  Move the cycle counts into 64-bit integers
                temp_cycles1 = ((unsigned __int64)cycles_high1 << 32) | cycles_low1;
                temp_cycles2 = ((unsigned __int64)cycles_high2 << 32) | cycles_low2

                //  Add to total cycle count
                total_cycles += temp_cycles2 - temp_cycles1 - base;
                iterations++;
                //  *** END OF INCLUDE SECTION 5

        }

        // Now the total cycle count and iterations are available to
        // be used as desired.
        // Example:
                seconds = double(total_cycles)/double(mhz*1000000);

                printf("Average cycles per loop:  %f\n",
                        double(total_cycles/iterations));
                printf("Average seconds per loop:  %f\n", seconds/iterations);
}


//  *** BEGIN OF INCLUDE SECTION 6
//  *** INCLUDE THE FOLLOWING FUNCTION WITH YOUR CODE TO MEASURE
//  *** THE BASE TIME
unsigned FindBase() {
        unsigned base,base_extra=0;
        unsigned cycles_low, cycles_high;

        //  The following tests run the basic cycle counter to find
        //  the overhead associated with each cycle measurement.
        //  It is run multiple times simply because the first call
        //  to CPUID normally takes longer than subsequent calls.
        //  Typically after the second run the results are
        //  consistent.  It is run three times just to make sure.

        __asm {
                pushad
                CPUID
                RDTSC
                mov     cycles_high, edx
                mov     cycles_low, eax
                popad
                pushad
                CPUID
                RDTSC
                popad

                pushad
                CPUID
                RDTSC
                mov     cycles_high, edx
                mov     cycles_low, eax
                popad
                pushad
                CPUID
                RDTSC
                popad
```

```
                pushad
                CPUID
                RDTSC
                mov     cycles_high, edx
                mov     cycles_low, eax
                popad
                pushad
                CPUID
                RDTSC
                sub     eax, cycles_low
                mov     base_extra, eax
                popad

                pushad
                CPUID
                RDTSC
                mov     cycles_high, edx
                mov     cycles_low, eax
                popad
                pushad
                CPUID
                RDTSC
                sub     eax, cycles_low
                mov     base, eax
                popad

        } // End inline assembly

        // The following provides insurance for the above code,
        // in the instance the final test causes a miss to the
        // instruction cache.
        if (base_extra < base)
                base = base_extra;

        return base;
}
//  *** END OF INCLUDE SECTION 6
```

A.3. Testing a Small Code Section for Repeatable Results

```
// Code for testing a small, stand-alone section of code
// for repeatable results.
// This code will work as is in Microsoft(R) Visual C++(R) 5.0 compiler.
// Simply add in your own code in the marked sections and compile
// it.

#include <stdio.h>

#define CPUID __asm __emit 0fh __asm __emit 0a2h
#define RDTSC __asm __emit 0fh __asm __emit 031h

unsigned TestFunc();

void main () {

        unsigned base=0;
        unsigned cyc, cycles1, cycles2, cycles3, cycles4, cycles5;

        //  The following tests run the basic cycle counter to find
        //  the overhead associated with each cycle measurement.
        //  It is run multiple times simply because the first call
        //  to CPUID normally takes longer than subsequent calls.
        //  Typically after the second run the results are
```

```
        //  consistent.  It is run three times just to make sure.

        __asm {
                CPUID
                RDTSC
                mov     cyc, eax
                CPUID
                RDTSC
                sub     eax, cyc
                mov     base, eax

                CPUID
                RDTSC
                mov     cyc, eax
                CPUID
                RDTSC
                sub     eax, cyc
                mov     base, eax

                CPUID
                RDTSC
                mov     cyc, eax
                CPUID
                RDTSC
                sub     eax, cyc
                mov     base, eax

        } // End inline assembly

        // This section calls the function that contains the user's
        // code.  It is called multiple times to eliminate instruction
        // cache effects and get repeatable results.

        cycles1 = TestFunc();
        cycles2 = TestFunc();
        cycles3 = TestFunc();
        cycles4 = TestFunc();
        cycles5 = TestFunc();

        // By the second or third run, both data and instruction
        // cache effects should have been eliminated, and results
        // will be consistent.
        printf("Base :  %d\n", base);
        printf("Cycle counts:\n");
        printf("%d\n", cycles1-base);
        printf("%d\n", cycles2-base);
        printf("%d\n", cycles3-base);
        printf("%d\n", cycles4-base);
        printf("%d\n", cycles5-base);
}


unsigned TestFunc() {

//  ***  REPLACE THIS SECTION WITH YOUR OWN VARIABLES
        float z,q,x,y;
        float result;
        unsigned cycles;

        //  Load the values here, not at creation, to make sure
        //  the data is moved into the cache.
        cycles=0;
        result=0.0f;
        x=2.0f;
```

```
        y=100.0f;
        z=12.0f;
        q=5.0f;
//  ***  END OF USER SECTION

        __asm {
                CPUID
                RDTSC
                mov     cycles, eax
                }

//  ***  REPLACE THIS SECTION WITH THE CODE TO BE TESTED
        z += y;
        q *= x;
        result = z/q;
//  ***  END OF USER SECTION

        __asm {
                CPUID
                RDTSC
                sub     eax, cycles
                mov     cycles, eax
                }
        return  cycles;
}
```