CPU performance monitoring using the Time-Stamp Counter register

1 Purpose

This laboratory work introduces basic information on the Time-Stamp Counter CPU register, which is used for performance monitoring. The focus is on how to use this feature correctly and how to obtain accurate results when measuring the computing performance.

2 Time-Stamp Counter register

The Time-Stamp Counter (TSC) is a 64-bit model specific register (MSR) that accurately counts the cycles that occur on the processor. It is present on all x86 processors. The TSC is incremented every clock cycle and is set to zero every time the processor is reset. This register is accessible to the programmer since the Pentium processor.

To access the TSC, the programmer has to call the `RDTSC` (read time-stamp counter) instruction from assembly language. The `RDTSC` instruction loads the `EDX:EAX` with the content of the TSC register. The `EDX` will contain the high-order 32 bits and the `EAX` will contain the low-order 32 bits.

The TSC counts the CPU cycles, so the value returned by the `RDTSC` instruction will be the number of cycles counted from the last processor reset to the point `RDTSC` was called. To obtain time in seconds, the value provided by the TSC has to be divided with the processor frequency (in Hz) as shown below:

$$\#second = \frac{\#cycles}{frequency}$$

This method for performance monitoring is very useful for measuring the cycle count for small sections of code. For example when trying to compare the performance of sections of code that have the same result but use different instructions. Another case in which this method can be of use is to obtain the average execution time for a function or section of code.

3 How to use RDTSC instruction

To obtain accurate results when measuring the performance with `RDTSC` instruction, the programmer has to be aware of the main issues that affect the cycle count and how to work-around these issues. The main issues that affect cycle count are:

- **Out-of-order execution:** the order of instruction execution is not as in the source code, this may cause the `RDTSC` instruction to return a cycle count that is less or greater than the actual cycle count for the measured sequence of code.
- **Data cache and instruction cache:** if the code or data are not in the cache, the cycle count is much larger.
- **Context switches:** if they occur during measurement, the result will be biased.
- **Frequency changes:** results are not accurate if there are frequency changes during measurement.
- **Multi-core processors:** the cycle counters on the cores are not synchronized. If the process migrates during measurement, the result will be wrong.

A measurement methodology is proposed and the solutions for these issues are discussed as follows.

3.1 Basic measurement method from assembly language

To make a basic cycle measurement for an instruction, the programmer has to use the following code from assembly language:

```
rdtsc
mov time_high, edx
mov time_low, eax
add var, ecx
rdtsc
sub eax, time_low
sbb edx, time_high
mov time_high, edx
mov time_low, eax
```

The previous code section measures the cycles for the execution of an ADD instruction between a value found in the memory and the value inside ECX register. It is important to use the 64-bit value provided by the TSC. If we use only the low-order 32 bits, the counter overflows (turns zero) in about 11 seconds. Any code with execution time larger than 11 seconds will not be measured correctly. The ADD instruction can be replaced by any sequence of instructions or any function/procedure.

3.2 Solutions for issues affecting the cycle count

3.2.1 Out-of-order execution

In case of out-of-order execution the instructions do not execute necessarily in the order indicated by the source code, so the RDTSC instruction can be executed before or after the location indicated in the source code. This causes measurements that are not accurate. To prevent the RDTSC instruction from being executed out-of-order, a serializing instruction has to be executed before it. The CPUID instruction causes all the instructions preceding it to be executed, before its execution. If CPUID instruction is placed before, the RDTSC instruction will be executed in-order. To make an exact measurement, the overhead of CPUID can be measured and subtracted from the cycle count obtained for the measured code. The CPUID instruction takes longer to execute the first two times it is called. It is better to measure the execution of its third call, and use this value for all future measurements.

Measurement method in C/C++

There are some important considerations for measuring the cycles from code written in C/C++ language. Not all compilers recognize the RDTSC and CPUID instructions. The work-around for these cases is to use emit statements that replace the instructions with the corresponding opcodes in the "obj".

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

If the compiler does not recognize the RDTSC instruction, it will not be aware of it modifying the EDX and EAX registers. For this reason is better to save the general purpose registers before using this instruction. Use the RDTSC instruction from C/C++ as below:

```
unsigned cycles_high1=0, cycles_low1=0, cupid_time=0;
unsigned cycles_high2=0, cycles_low2=0;
unsigned __int64 temp_cycles1=0, temp_cycles2=0;
__int64 total_cycles=0;

// compute the CPUID overhead
__asm {
    pushad
    CPUID
    RDTSC
    mov cycles_high1, edx
    mov cycles_low1, eax
    popad
    pushad
    CPUID
```

```
    RDTSC
    popad
    pushad
    CPUID
    RDTSC
    mov cycles_high1, edx
    mov cycles_low1, eax
    popad
    pushad
    CPUID
    RDTSC
    popad
    pushad
    CPUID
    RDTSC
    mov cycles_high1, edx
    mov cycles_low1, eax
    popad
    pushad
    CPUID
    RDTSC
    sub eax, cycles_low1
    mov cupid_time, eax
    popad
}

cycles_high1=0;
cycles_low1=0;

// Measure the code sequence
__asm {
    pushad
    CPUID
    RDTSC
    mov cycles_high1, edx
    mov cycles_low1, eax
    popad
}

// Section of code to be measured

__asm {
    pushad
    CPUID
    RDTSC
    mov cycles_high2, edx
    mov cycles_low2, eax
    popad
}

temp_cycles1 = ((unsigned __int64)cycles_high1 << 32) | cycles_low1;
temp_cycles2 = ((unsigned __int64)cycles_high2 << 32) | cycles_low2;
total_cycles = temp_cycles2 - temp_cycles1 - cpuid_time;
```

3.2.2 Data cache and instruction cache

The repeated measurement of the same section of code can produce different results. This can be caused by the cache effects. If the instructions or data are not found in the L1 (instruction or data) cache the cycle count is larger because it takes longer to bring the instructions/data from the main memory. In some cases, the cache effects have to be taken into consideration, for example, when trying to obtain an average cycle count for some section of code. In other cases, the focus is to measure the exact cycle count for some instructions and to obtain a value which is repeatable. When leaving out the cache interference on cycle count, we have to be sure that the instructions of the sequence we want to measure, and the data we access in that sequence are in the L1 cache. This is done only for small sections of code and small data sets (less than 1KB). The access to the entire data set brings it to the cache (e.g. read every element from the data set). To bring the measured code in the cache, put the code in a function call the function twice and measure only the second call. To be sure that cache effects are taken out, repeat the measurements at least 5 times.

3.2.3 Context switches

Context switches can bias the measurements done with the RDTSC instruction (processes that interrupt during measurement and context switch time will be accounted for). The probability of the measured code to be interrupted by other processes increases with its length. In case of small sequences of code with short execution time, the probability is very small. For large sequences of code, set the process priority to the highest priority. In this way other processes will not interrupt during measurement.

3.2.4 Multi-core processors

The TSCs on the processor's cores are not synchronized. So it is not sure that if a process migrates during execution from one core to another, the measurement will not be affected. To avoid this problem, the measured process's affinity has to be set to just one core, to prevent process migration.

3.2.5 Frequency changes

If the processors support technologies that make possible to change the frequency while functioning (for power management), the execution time measurement is affected. The solution for this problem is to prevent frequency changes through the settings in the operating system. If the measurement is made in clock cycles, not in seconds, the measurement will be independent of the processor frequency.

4 Exercises

1. **Using the RDTSC instruction, measure the average execution time of the `CPUID` instruction and the execution time of assembly language instructions. The code template is provided in `SCS_lab02_ex1_win.cpp` (for Windows) and `SCS_lab02_ex1_linux.c` (for Linux).**

Note: Linux users have to install `gcc-multilib` to be able to compile 32bit applications on 64bit machines. The installation and compilation steps are given below:

i. Install appropriate packages to be able to compile 32bit applications on 64bit machines.
   - For Ubuntu: `sudo apt install gcc-multilib`
   - For Arch: `sudo pacman -S lib32-glibc lib32-gcc-libs`
   - For other distributions consult the web for the appropriate package names.

ii. Use `gcc` to compile the source into 32bit executable:

```
gcc -m32 <file_name.c> -o <exec_name>
```

To check that the executable is indeed 32bit you can use:

```
file <exec>
```

Output example:

```
<exec>: ELF 32-bit LSB pie executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2,
BuildID[sha1]=8e5c037c8f02c94d0ee03123297c8f48cce1c333,
for GNU/Linux 4.4.0, not stripped
```

iii. Run the application (from the directory where the executable file was created):

```
./<exec_name>
```

Step 1. Compute the average overhead of the `CPUID` instruction. For this, perform at least 10 execution time measurements of the `CPUID` instruction as described above. **Ignore the negative values of the execution time.** Record the measurements into Table 1. Based on the recorded measurements, compute the average execution time of `CPUID`.

Measurement number	CPUID execution time (clock cycles)
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
Average time	

Table 1: CPUID execution time measurements

Step 2. Measure the execution time of the following instructions.

a. ADD (both values in local registers)

b. ADD (value in variable)

c. MUL

d. FDIV

e. FSUB

Perform at least 5 measurements for each instruction from the previous step and record the values into Table 2. Use the previously computed average CPUID execution time (hardcode the average value instead of recomputing the overhead at each step). Compute the average execution time for each instruction. Explain the results.

Measurement number	ADD (reg)	ADD (var)	MUL	FDIV	FSUB
1					
2					
3					
4					
5					
Average time					

Table 2: Arithmetic instructions execution time measurements

2. Write a function that sorts an array of unsigned integer values. The array is declared first static, then dynamic (two versions). The code template is provided in `SCS_lab02_ex2_win.cpp` (for Windows) and `SCS_lab02_ex2_linux.c` (for Linux).

Step 1. Measure the average execution time in cycles and seconds (for static/dynamic array) using both RDTSC instruction and `clock()` for the sort function using an array of 1000 elements and record the execution time in Table 3. Perform at least 5 measurements. Compute the average execution time in each case. Briefly explain the results.

Step 2. Which part/sequence of instructions of the sort function takes longer to execute? Try to optimize your sort function or find a more efficient sort (e.g. Randomized Quick Sort) function and perform execution time measurements using RDTSC for various array lengths to fill Table 4.

Step 3. Based on Table 4 plot a line graph showing the variation of the execution time (y-axis) with the length of the array (x-axis) in each case (static/dynamic array). Each column in Table 4 should be represented as an individual line in the graph. Explain and discuss on the results.

Measurement number	`rdtsc` time (clock cycles)		`clock()` time (clock cycles)	
	static array	dynamic array	static array	dynamic array
1				
2				
3				
4				
5				
Average time				

Table 3: Sort function execution time measurements

Array length	Initial sort time (clock cycles)		Optimized sort time (clock cycles)	
	static array	dynamic array	static array	dynamic array
100				
500				
1000				
5000				
10000				

Table 4: Initial and optimized sort function execution time measurements

References

[1] Intel, *Using the RDTSC Instruction for Performance Monitoring* [accessed Sept. 2024], https://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf

[2] Peter Kankowski, *Performance measurements with RDTSC* [accessed Sept. 2024], https://www.strchr.com/performance_measurements_with_rdtsc

[3] John Lyon-Smith, *Getting accurate per thread timing on Windows* [accessed Sept. 2024], https://web.archive.org/web/20090510073435/http://lyon-smith.org/blogs/code-o-rama/archive/2007/07/17/timing-code-on-windows-with-the-rdtsc-instruction.aspx