Detecting CPU Speed

From OSDev Wiki

Contents

- 1 What is CPU Speed
- 2 General Method
  - 2.1 Waiting for a given amount of time
  - 2.2 Knowing how many cycles your loop takes
  - 2.3 RDTSC Instruction Access
  - 2.4 Working Example Code
  - 2.5 Without Interrupts
- 3 Asking the SMBios for CPU speed
  - 3.1 SMBios Processor Information
  - 3.2 Getting the SMBIOS Structure
- 4 Links
  - 4.1 Related threads in the forum
  - 4.2 Other resources
    - 4.2.1 Regarding SMBIOS

What is CPU Speed

There are several different things that could be called "CPU speed":

1. how quickly it can execute code (e.g. instructions per second)
2. how fast its clock is running (e.g. cycles per second)

How quickly a CPU can execute code is important for determining the CPU's performance. How fast a CPU's clock is running is only useful for specific cases (e.g. calibrating the CPU's TSC to use for measuring time).

There are also several different measurements for these different "CPU speeds":

1. best case
2. nominal case
3. average case
4. current case
5. worst case

For example, if you look at a modern Core i7 CPU (with turbo-boost, power management and hyper-threading), the best case instructions per second would occur when there's no throttling/power saving at all, only one logical CPU is running (turbo-boost activated and hyper-threading not being used), you're executing simple instructions with no dependencies in a loop that fits in the CPU's "loop buffer", there are no branch mispredictions, and there are no accesses to memory (no data being transferred to/from caches or RAM). The worst case instructions per second would be the exact opposite; and may be several orders of magnitude worse (e.g. a best case of 4 billion instructions per second and a worst case of 100 million instructions per second). The nominal instructions per second is an estimation of "normal" - e.g. the normal average instructions per second you'd expect (note: "nominal cycles per second" is used more often). All of these things are fixed values - a specific CPU always has the same best case, worst case and nominal case, and these values don't change depending on CPU load, which instructions are/were executed, etc.

The current instructions per second is the instructions per second at a specific instant in time and must be somewhere between the best and worst cases. It can't actually be measured, but can be estimated by finding the average instructions per second for a very short period of time. The average case is something that has to be measured. Both the current instructions per second and the average instructions per second depend heavily on the code that was running. For example, the average instructions per second for a series of NOP instructions may be much higher that the average instructions per second for a series of DIV instructions.

General Method

In order to tell what's the CPU speed, we need two things:

1. being able to tell that a given (precise) amount of time has elapsed.
2. being able to know how much 'clock cycles' a portion of code took.

Once these two sub-problems are solved, one can easily tell the CPU speed using the following :

```
prepare_a_timer(X milliseconds ahead);
while (timer has not fired) {
   inc iterations_counter;
}
cpuspeed_mhz = (iteration_counter * clock_cycles_per_iteration)/1000;
```

Note that except for very special cases, using a busy-loop (even calibrated) to introduce delays is a bad idea and that it should be kept for very small delays (nano or micro seconds) that you must comply when programming hardware only.

Also note that PC emulators (like BOCHS, for instance) are rarely realtime and that you shouldn't be surprised if your clock appears to run faster than expected on those emulators.

Waiting for a given amount of time

There are two circuits in a PC that allows you to deal with time: the PIT (Programmable Interval Timer, 8253 iirc) and the RTC (Real Time Clock). The PIT is probably the better of the two for this task.

The PIT has two operating mode that can be useful for telling the cpu speed:

1. the *periodic interrupt* mode (0x36), in which a signal is emitted to the interrupt controller at a fixed frequency. This is especially interesting on PIT channel 0 which is bound to IRQ0 on a PC.
2. the *one shot* mode (0x34), in which the PIT will decrease a counter at its top speed (1.19318 MHz) until the counter reaches zero.

Whether or not an IRQ is fired by channel0 in 0x34 mode should be checked

Note that theoretically, _one shot_ mode could be used with a _polling_ approach, reading the current count on the channel's data port, but I/O bus cycles have unpredictable latency and one should make sure the timestamp counter is not affected by this approach.

Knowing how many cycles your loop takes

This step depends on your CPU. On 286, 386 and 486, each instruction took a well-known and deterministic amount of clock cycles to execute. This allowed the programmer to tell exactly how many cycles a loop iteration took by looking up the timing of each instruction and then sum them up.

Since the multi-pipelined architecture of the Pentium, however, such numbers are no longer communicated (for a major part because the same instruction could have variable timings depending on its surrounding, which makes the timing almost useless).

It is possible to create code which is exceptionally pipeline hostile such as:

```
xor eax,edx
xor edx,eax
xor eax,edx
xor edx,eax
...
```

A simple xor instruction takes one cycle, and it's guaranteed that the processor cannot pipeline this code as the current instructions operands depend on the results from the last calculation. One can check that, for a small count (tested from 16 to 64), RDTSC will show the instruction count is almost exactly (sometimes off by one) the cycles count. Unfortunately, when making the chain longer you'll start experiencing code cache misses, which will ruin the whole process.

E.g. looping on a chain of 1550 XORs (http://www.sylvain-ulg.be.tf/resources/speed.c) may require a hundred of iterations before it stabilizes around 1575 clock cycles on a AMDx86-64, and I'm still waiting it to stabilize on my Pentium3

Despite this inaccuracy it gives relatively good results across the whole processor generation given a reasonably accurate timer but if very accurate measurements are needed the next method should prove more useful.

A Pentium developer has a much better tool to tell timings: the _Time Stamp Counter_: an internal counter that can be read using RDTSC special instruction

rdtscpm1.pdf (http://www.math.uwaterloo.ca/~~jamuir/rdtscpm1.pdf) explains how that feature can be used for performance monitoring and should provide the necessary information on how to access the TSC on a Pentium

RDTSC Instruction Access

The presence of the Time Stamp Counter (and thus the availability of RDTSC instruction) can be detected through the [CPUID] instruction. When calling CPUID with eax=1, you'll receive the features flags in edx. TSC is the bit #4 of that field.

Note that prior to use the CPUID instruction, you should also make sure the processor support it by testing the 'ID' bit in eflags (this is 0x200000 and is modifiable only when CPUID instruction is supported. For systems that doesn't support CPUID, writing a '1' at that place will have no effect)

In the case of a processor that does not support CPUID, you'll have to use more eflags-based tests to tell if you're running on a 486, 386, etc. and then pick up one of the 'calibrated loops' for that architecture (8086 through 80486 may have variable instruction timings).

Working Example Code

There is a Real Mode Intel-copyrighted example in the above-mentioned application note ... Here comes another code submitted by DennisCGC that will give the total measured frequency of a Pentium processor.

Some notes:

- *irq0_count* is a variable, which increases each time when the timer interrupt is called.
- in this code it's assumed that the [PIT] is programmed to 100 hz (of course, I give the formula about how to calculate it
- it's assumed that the command CPUID is supported.

```
;__get_speed__:
   ;first do a cpuid command, with eax=1
   mov  eax,1
   cpuid
   test edx,byte 0x10      ; test bit #4. Do we have TSC ?
   jz   detect_end         ; no ?, go to detect_end
   ;wait until the timer interrupt has been called.
   mov  ebx, ~[irq0_count]

;__wait_irq0__:

   cmp  ebx, ~[irq0_count]
   jz   wait_irq0
   rdtsc                   ; read time stamp counter
   mov  ~[tscLoDword], eax
   mov  ~[tscHiDword], edx
   add  ebx, 2             ; Set time delay value ticks.
   ; remember: so far ebx = ~[irq0]-1, so the next tick is
   ; two steps ahead of the current ebx ;)

;__wait_for_elapsed_ticks__:

   cmp  ebx, ~[irq0_count] ; Have we hit the delay?
   jnz  wait_for_elapsed_ticks
   rdtsc
   sub eax, ~[tscLoDword]  ; Calculate TSC
   sbb edx, ~[tscHiDword]
   ; f(total_ticks_per_Second) =  (1 / total_ticks_per_Second) * 1,000,0
```

```
    ; This adjusts for MHz.
    ; so for this: f(100) = (1/100) * 1,000,000 = 10000
    mov ebx, 10000
    div ebx
    ; ax contains measured speed in MHz
    mov ~[mhz], ax
```

See the intel manual (see links) for more information.

> - bugs report are welcome. IM to DennisCGC (http://www.mega-tokyo.com/forum/index.php?action=viewprofile;user=DennisCGc)

Without Interrupts

I'd be tempted to say 'yes', though I haven't gave it a test nor heard of it elsewhere so far. Here is the trick:

```
disable()     // disable interrupts (if still not done)
outb(0x43,0x34);   // set PIT channel 0 to single-shot mode
outb(0x40,0);
outb(0x40,0);      // program the counter will be 0x10000 - n after n tic
long stsc=CPU::readTimeStamp();
for (int i=0x1000;i>0;i--);
long etsc=CPU::readTimeStamp();
outb(0x43,0x04);   // read PIT counter command ??
byte lo=inb(0x40);
byte hi=inb(0x40);
```

Now, we know that

- ticks=(0x10000 - (hi*256+lo)) periods of 1/1193180 seconds have elapsed at least and no more than ticks+1.
- etsc-stsc clock cycles have elapsed during the same time.

Thus (etsc-stsc)*1193180 / ticks should be your CPU speed in Hz ...

As far as i can say, 0x1000 iterations lead to 10 PIT ticks on a 1GHz CPU and a bit less than 0x8000 ticks on the same CPU running BOCHS. This certainly means that on very high speed systems, the discovered speed may not be accurate at all, or worse, less than 1 tick could occur ...

This technique is currently under evaluation in [the forum|Forum:5849]

> - hope you like my technique /PypeClicker

Asking the SMBios for CPU speed

The SMBios (System Management BIOS) Specification addresses how motherboard and system vendors present management information about their products in a standard format by extending the BIOS interface on Intel architecture systems. The information is intended to allow generic instrumentation to deliver this information to management applications that use DMI, CIM or direct access, eliminating the need for error prone operations like probing system hardware for presence detection.

SMBios Processor Information

A Processor information (type 4) structure describes features of the CPU as detected by the SMBios. The exact structure is depicted in section 3.3.5 (p 39) of the standard (http://www.dmtf.org/standards/documents/SMBIOS/DSP0134.pdf) . Within that information you will find the processor type, family, manufacturer etc. and also:

- the External Clock (bus) frequency, which is a word at offset 0x12,
- the Maximum CPU speed in MHz, which is a word at offset 0x14 (e.g. 0xe9 is a 233MHz processor),
- the Current CPU speed in MHz, (word at offset 0x16).

Getting the SMBIOS Structure

SMBios provide a _Get SMBIOS Information_ function that tells you how many structures exists. You can then use _Get SMBIOS Structure_ function to read processor information.

As an alternative, you can locate the _SMBIOS Entry Point_ and then traverse manually the SMBIOS structure table, looking for type 4.

All this is depicted in 'Accessing SMBIOS Information' structure of the standard (p 11).

The SMBIOS Entry Point structure, described below, can be located by application software by searching for the anchor-string on paragraph (16-byte) boundaries within the physical memory address range 000F0000h to 000FFFFFh. This entry point encapsulates an intermediate anchor string that is used by some existing DMI browsers.

00-03	Anchor String (_SM_ or 5f 53 4d 5f)
04	Checksum
05	Length
06	major version
07	minor version
08-09	max structure size
0A	entry point revision
0B-0F	formatted area
10-14	_DMI_ signature
15	intermediate checksum
16-17	structure table length
18-1B	structure table (physical) address
1C-1D	number of SMBIOS structures

1E	SMBIOS revision (BCD)

I don't feel like re-explaining the PnP calling convention etc. as chances are it will be useless in Protected Mode ...

Links

Related threads in the forum

- Forum:5849
- Forum:767
- Forum:922
- Forum:8949 featuring info on bogomips, how linux does it and durand's code.

Other resources

- http://cs.usfca.edu/~cruse/cs630s04/lesson23.ppt, a crash course on PIT, and how to use it to compute CPU speed.
- http://www.sandpile.org/post/msgs/20004561.htm
- http://www.midnightbeach.com/jon/pubs/rdtsc.htm
- ftp://download.intel.com/support/processors/procid/

  especially section 12: "Operating Frequency" on page 29 of 24161815.pdf (ftp://download.intel.com/support/processors/procid/24161815.pdf)

Regarding SMBIOS

- http://www.dmtf.org/standards/smbios
- http://www.dmtf.org/standards/documents/SMBIOS/DSP0134.pdf
- http://www.pcpitstop.com/faq/smbios.asp

Retrieved from "http://wiki.osdev.org/index.php?title=Detecting_CPU_Speed&oldid=15576"
Category: X86 CPU