LOOP TIMINGS FOR THE 8088 CPU

UTTERCHAOS

1. Introduction

When doing cycle accurate work on the PC/XT it is frequently useful to have loops with a known number of cycles per iteration for adding delays to code.

The published timings for instructions on the 8088 CPU don't account for prefetch and various other delays introduced by the system and CPU, so it's often very hard to do this without significant trial and error.

Since the advent of Reenigne's XTCE which emulates a PC/XT in a cycle accurate way it is now possible to examine exactly how long every iteration of a loop is taking.

In this document, we give a number of example loops and loop contents to give precisely timed loops.

2. Limitations

All timings in this document assume interrupts do not occur (e.g. they are switched off), that DRAM refresh has been switched off and that no wait states will be incurred (e.g. by accessing VRAM).

The effect of DRAM refresh if it is switch on is hard to estimate. In general it is model specific, but on the original IBM PC/XT it ties up the bus for 4 cycles in a row out of every 76 cycles.

Without stopping and starting the Programmable Interval Timer that is used for DRAM refresh and starting one's code in sync with the timer it's hard to know when DRAM refresh interference is going to occur and even more difficult to assess its actual impact unless the loop is designed to be synced with the PIT timer 1.

Another limitation is that we typically don't know how long the first iteration of a loop will take as it depends on the state of the prefetch queue when the loop begins. One can do a jump to the start of the loop which will clear the prefetch queue before the loop starts, but one still has the problem of not knowing when the jump will occur due to needing information about the state of the prefetch queue when it was executed.

The general problem of adding delays between linear sequences of code is complex and is best evaluated using XTCE itself.

Fortunately there are many situations where only the number of cycles per iteration matters, e.g. when adjusting the relative phase of one's code with respect to some timer or other regular pulse, e.g. the start of a video frame.

Another problem is that single instruction loops that rely on long-running instructions can lead to the prefetch queue being in an inconsistent state depending on how many iterations were used. To mitigate this, one can always use a higher number of iterations so that the prefetch queue is always in a consistent state when it is done.

Some loops also don't work with an iteration count of zero. These corner cases can also be worked around by increasing the iteration count.

3. Basis loop costs per iteration

We will examine the timings in cycles per iteration of a number of basic loops. There are two types of loop we will look at: ones that terminate after a specified number of iterations and ones that terminate on some condition dependent on a value read from a port (e.g. the PIT channel 2 read back or the CRTC display enable or vertical retrace signals).

Let's begin with loops that terminate after a certain number of iterations.

The first loop we will look at is the following.

```
    mov cx, 5
    jmp loop_start

loop_start:
    loop loop_start
```

It has a cost of 18 cycles per iteration.

Here is a loop that doesn't use the loop instruction.

```
    mov bx, 5
    jmp loop_start

loop_start:
    dec bx
    jnz loop_start
```

This loop costs 22 cycles per iteration regardless of which 16 bit register is used for the count.

We can also use an 8 bit register.

```
    mov bl, 5
    jmp loop_start

loop_start:
    dec bl
    jnz loop_start
```

This now takes 26 cycles.

We can also use the sub instruction instead of dec.

```
    mov bx, 5
    jmp loop_start

loop_start:
    sub bx, 1
    jnz loop_start
```

This takes 30 cycles, independently of whether an 8 or 16 bit register is used, with the exception of al which will reduce the time back to 26 cycles.

We can add a jmp instruction to take longer. Here is the version using al.

```
    mov al, 5
    jmp loop_start

loop_start:
    sub al, 1
    jz loop_end
    jmp loop_start

loop_end:
```

This takes 34 cycles.

If we use a register other than al, the time goes up.

```
    mov bx, 5
    jmp loop_start

loop_start:
    sub bx, 1
    jz loop_end
    jmp loop_start

loop_end:
```

This takes 38 cycles.

We can also add a cmp instruction to take even longer.

```
    mov bx, 5
    jmp loop_start

loop_start:
    sub bx, 1
    cmp bx, 0
    jne loop_start
```

This now takes 42 cycles unless al is used in which case it drops back to 34 cycles.

As can be seen, all timings so far are an even number of cycles. We have iterations taking 18, 22, 26, 30, 34, 38 and 42 cycles.

In what follows we will focus on what needs to be added to the explicit examples above (following the label at the start of the loop) in order to reach other cycle counts.

4. Inserting single instructions in the loops

The instructions of most interest at this point are ones that could give us cycle counts that are multiples of 4 or odd.

As most 8088 instructions are prefetch bound we'll simply be adding a multiple of 4 cycles (the time it takes to prefetch a byte) unless we use instructions that take longer to execute than to prefetch.

Adding a NOP to a loop will simply increase the number of cycles by 4 (albeit without additional registers or flags being clobbered).

Unfortunately there are no other interesing single byte instructions with useful timings which are effectively a no operation.

Two byte instructions are another story. One can add 18 cycles by adding an unconditional (short) jump to a label immediately following the jump instruction.

This allows us to get all timings which are a multiple of 4 starting with 36 cycles per iteration.

If one is prepared to sacrifice a register then lodsb will add 14 cycles, although this will clobber the al register meaning that the loops using it cannot be used and one must ensure that the read is from system memory.

This allows us to get all timings which are a multiple of 4 starting with 32 cycles per iteration.

Of course the easiest way to get a multiple of 4 cycles per iteration is to not do a loop at all, but let the processor do it.

```
    mov cl, 5
    jmp loop_start

loop_start:
    shl al, cl
```

This will add 4 cycles for every iteration, starting at cl = 0.

One can also use the rep prefix to get some larger values.

```
    mov cx, 5
    jmp loop_start

loop_start:
    rep
    lodsb
```

This takes 13 cycles per iteration starting from cx = 0, though again one must ensure the read is from system memory.

Of course one can combine as many shl loops as desired with a rep lodsb loop to get every iteration cycle count in the sequence $13, 17, 21, 25, 29, 33, 37, 41 \ldots$.

One can replace lodsb with scasb to get 15 cycles per iteration from cx = 0, again ensuring the read is from system memory.

Combining with as many shl loops as desired one can get every iteration cycle count in the sequence $15, 19, 23, 27, 31, 35, \ldots$.

If one is prepared to write to a sequence of memory addresses, one can get 10 cycles per iterations using stosb and 14 with stosw.

5. Other iteration cycle counts

We so far have loops for every iteration cycle count except 1, 2, 3, 5, 6, 7, 9 and 11.

In order to get the remaining values we can use some self-modifying code with the following trick.

Suppose we need 11 cycles per iteration. If the number of iterations is even, say $2n$, then this is the same as n iterations of 22 cycles. For an odd number of iterations, $2n + 1$ we simply need an additional 11 cycles.

In order to manage the latter, we simply self-modify some existing code that is going to be run by replacing some instruction with one that will take 11 cycles longer. If this turns out to be difficult, we can do one iteration of shl and replace an instruction with one taking 7 cycles longer.

For example, the string instructions have the following cycle counts and opcodes:

- stosb = 12 cycles = 0xaa
- lodsb = 13 cycles = 0xac
- scasb = 16 cycles = 0xae
- lodsw = 17 cycles = 0xad
- scasw = 20 cycles = 0xaf

from which we see that replacing a lodsb with a scasw will add 7 cycles.

Here is a piece of code that does what we want:

```
    mov cx, 2

    jmp start
start:
    shr cl, 1
    db 0xd6
loop_start:
    nop
    loop loop_start

    mov cl, al
    neg cl
    and al, 3
    xor al, 0xac
    mov [patch], al

    add cl, 3
    shr ax, cl
patch:
    db 0
```

The instruction at the label patch is changed from 0xac (lodsb) to 0xaf (scasw), which is a difference of 7 cycles. The remaining 4 cycles is made up by the shr ax, cl instruction.

Note the add cl, 3 instruction to increase the count for shr so that the processor is in a consistent state afterwards.

The 0xdb instruction is salc, which loads al with all zeroes if the carry is not set and all ones if the carry is set.

The loop with 22 cycles per iteration uses the loop instruction. The extra nop pads this out from 18 cycles per iteration to 22.

This particular code only works for an iteration count of 2 or more, but it could be easily modified to increment the initial iteration count by a fixed amount so that it could accept a count of 0 and 1.

The only iteration cycle counts we can't get now are 2 and 6. Of course 6 can be obtained if we can get 2. For the latter we need a pair of instructions that differ in cycle count by 2. For this we can use stc (2 cycles, opcode 0xf9) and das (4 cycles, opcode 0x2f).