

AlignTest: Technischen Details

Erste Vorbemerkung: Es geht hier *nicht* um einen Performance-Vergleich von Prozessoren untereinander. Sondern um die Frage, *welche Codevariante* auf dem jeweiligen Prozessor *relativ* schnell läuft und welche die jeweilige CPU hingegen *ausbremst*. Oder anders gesagt: *Was ist der beste Code für diesen und jenen Prozessor?* Nähere Informationen zur Aufgabenstellung des Programms sind in den folgenden Dateien enthalten:

- `ReadMe.pdf`
- `AlignTest_Beschreibung.pdf`

Zweite Vorbemerkung: Ich werde zunächst *Codevarianten* anhand der Ergebnisse des *AMD64 Athlon* und des *Intel Celeron* darstellen, dann zum Thema „*Alignment*“ übergehen. Um die Ergebnisse besser nachvollziehen zu können, habe ich folgende exemplarische Ergebnislisten beigefügt:

- `AlignTestResult_(AMD_Athlon(tm)_64_X2_Dual_Core_Processor_5000+).txt`
- `AlignTestResult_(Intel(R)_Celeron(R)_CPU_____900____220GHz).txt`

Diese Listen habe ich ferner in eine übersichtlichere Form gebracht und farblich markiert:

- `Uebersicht_AlignTest_(Intel_Celeron).pdf`
- `Uebersicht_AlignTest_(AMD64_Athlon).pdf`

1) Codierung

Betrachtet wird folgende Funktion:

```
function CharPos_OwnPascal_***Align(c : Char; const s : ShortString): SmallInt;
var
  LenS,
  CntS : Byte;
  PC    : PChar;
begin
  result := 0;
  LenS   := Length(s);
  CntS   := LenS;
  if CntS = 0 then exit;
  PC := @S[1];
  while PC^ <> c do
  begin
    dec(CntS);
    if CntS = 0 then exit;
    inc(PC);
  end;
  result := LenS - CntS + 1;
end;
```

Dies ist eine Abwandlung der originalen *System.Pos(Char; ShortString)*-Funktion, die ein wenig assemblerfreundlicher konstruiert ist und (allerdings abhängig von den Umständen) auch etwas schneller ist. Für die weitere Betrachtung ist eigentlich nur die innere (*while*-) Schleife interessant, alles davor wird nur einmal pro Aufruf durchlaufen, die Result-Zuweisung am Ende statistisch (im Test) sogar nur 0,3-mal pro Aufruf. Die Schleife selbst wird hingegen im Test durchschnittlich etwa 38-mal durchlaufen.

Warum ist dieses Schleifenkonstrukt „*assemblerfreundlicher*“ als das der originalen *System.Pos*? Nun, deren Schleife sieht so aus:

```
for i:=1 to length(s) do
begin
  if pc^=c then
  begin
    result:=i;
    exit;
  end;
  inc(pc);
end;
```

Der Unterschied ist, dass im ersten Fall der Schleifenzähler auf Null herunter dekrementiert wird, während im zweiten Fall ein Vergleich zwischen Ist- und Grenzgröße stattfinden muß. Die RTL-Funktion spart sich also die zusätzliche lokale Variable *CntS* und deren Wertzuweisung. Da der Prozessor aber beim Subtrahieren bzw. Dekrementieren immer auch das Zero-Flag setzt, kann der verlaufsentscheidende Zustand, ob *CntS* gleich oder ungleich Null ist, anschließend sofort für eine Sprungentscheidung ausgewertet werden, während im zweiten Fall erst noch ein zusätzlicher Registervergleich stattfinden muß, um das ZeroFlag zu setzen. (Möglicherweise wandelt der Compiler auf einer höheren Optimierungsstufe diese Logik entsprechend um, das habe ich bisher nicht überprüft).

Heruntergebrochen auf Assembler-Ebene sieht die Schleife (das Davor und Danach lasse ich weg) dann so aus: **Variante 1: *CharPos_Asm1_LoopStart_**** (für i386):

```
.LLoop:  cmpb    (%esi), %dl    // 2 Byte Code 3a16    (Vergleiche (@S[n] in ESI)^ mit c in DL)
        je     .LResult  // 2 Byte code 74xx    (Wenn gleich, springe zu Resultzuweisung)
        subw   $1, %ax    // 4 Byte code 662d0100 (Dekrementiere Schleifenzähler in AX)
        jz     .LExit     // 2 Byte code 74xx    (Wenn der Null ist, dann Exit(0))
        incl   %esi       // 1 Byte code 46      (Incrementiere Stringzeiger auf @S[n+1])
        jmp    .LLoop     // 2 Byte code ebxx   (Und springe zurück zum Loopanfang)
```

Die drei beteiligten CPU-Register sind in allen weiteren Beispielen gleich belegt:

- *ESI* (bzw. *RSI* in der 64-bit-Version) enthält den Zeiger auf die aktuelle Position im String *S*,
- *DL* enthält das Char *c* und in

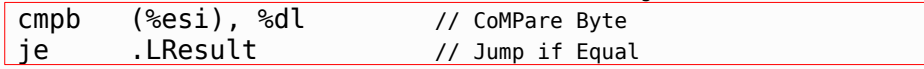
AX liegt der gegen Null laufende Schleifenzähler. (In *AX* deshalb, weil dort auch das Funktionsergebnis an die aufrufende Routine zurückgegeben wird, im Fall eines Mismatches (d.h. Abbruch bei Count = 0, was in 70 % aller Aufrufe passiert) braucht es

also keine weitere Ergebnisuweisung mehr.)

Was zum Teufel gibt es denn da noch zu optimieren? Aß sich dort nichts mehr *einsparen* lässt, ist ja offenkundig! Und doch läuft das verdammte Ding kaum schneller (bzw. auf dem Celeron sogar anderthalb mal *langsamer!*) als die doch schon erwiesenermaßen verbesserungsfähige RTL-Funktion! Woran liegt das?

In der **zweiten Variante** (und allen folgenden) habe ich die *Reihenfolge* der Anweisungen umgestellt: Anstatt als Erstes den eigentlichen Char-Vergleich durchzuführen, wird hier *erst der Schleifenzähler dekrementiert sowie der String-Zeiger erhöht und dann erst der Vergleich durchgeführt* (Voraussetzung ist, dass der Pointer in RSI/ESI anfangs auf S[0] statt auf S[1] zeigt und der Zähler um eins erhöht ist):

```
.LLoop:  subw    $1, %ax           // SUBtract Word
        jz     .LExit        // Jump if Zero
        incl   %esi          // INCrement Longword
        cmpb   (%esi), %dl    // CoMPare Byte
        je     .LResult      // Jump if Equal
        jmp    .LLoop        // JuMP
```



Das Ergebnis ist im Wesentlichen dasselbe wie bei Variante 1.

Bis hierher folgt die Codierung brav dem *etablierten Lehrbuchwissen* bzw. dem, was in unzähligen Variationen im Internet darüber zu lesen ist. Großes Gewicht hat dabei das von Werk von Agner: http://www.agner.org/optimize/optimizing_assembly.pdf, auf das immer wieder referenziert wird. Einige der Grundsätze sind folgende:

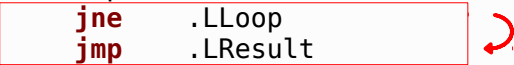
1. „Make conditional jumps most often not taken: The efficiency and throughput for not-taken branches is better than for taken branches on most processors. Therefore, it is good to place the most frequent branch first.“ (Agner)
2. Konditionale Jumps (Jcc) sind aufwendiger und unter Vorhersage-Gesichtspunkten unberechenbarer als unbedingte Sprunganweisungen (JMP).
3. „The INC and DEC instructions do not modify the carry flag but they do modify the other arithmetic flags. Writing to only part of the flags register costs an extra μop on P4 and P4E. (...) Use ADD and SUB when optimizing for speed. Use INC and DEC when optimizing for size or when no penalty is expected“. (Agner)

Aber wenn alles, was die Experten empfehlen, am Ende die Erwartungen doch nicht erfüllt, muß man auch mal das Gegenteil ausprobieren. Die *dritte Variante* schießt daher die Lehrmeinung in den Wind, *zieht den Rücksprung zum Loopanfang um eine Zeile vor* und überlässt diesen (weitaus häufigeren) Fall einer *bedingten - und zumeist befolgten - Sprunganweisung (Variante 3 a)*:

```

.LLoop:    subw    $1, %ax
           jz      .LExit
           incl    %esi
           cmpb    (%esi), %dl
           jne     .LLoop
           jmp     .LResult

```



Jetzt kommt die Überraschung: Diese Variante läuft auf dem AMD64 plötzlich ziemlich genau anderthalb *mal so schnell* wie die Variante 2! Beim Celeron hingegen ändert sich erstmal noch nichts.

Der wiederum kommt bei der nächsten kleinen Änderung auf Hochtouren: In **Variante 3b** habe ich lediglich die *Subtraktion* (um den Wert 1) durch einen *Dekrementier*--Befehl ersetzt:

```

.LLoop:    decw    %ax
           jz      .LExit
           incl    %esi
           cmpb    (%esi), %dl
           jne     .LLoop
           jmp     .LResult

```

Während diese Änderung den AMD64 überhaupt nicht beeindruckt, reagiert der Celeron darauf mit einer exorbitanten *Beschleunigung um den Faktor 1,8!*

Wobei anzumerken ist, dass der Assembler diese Anweisung „*decw %ax*“ bei 64-bit-Systemen anders übersetzt (als 3-Byte-Code 66ffc8) als im 32-Bit-Modus (dort wird dieselbe Anweisung als 2-Byte 6648 codiert).

Diese Variante 3b ist damit diejenige, die bei beiden Prozessoren die besten Ergebnisse liefert.

2) Alignment

Der Begriff „*Alignment*“ bezeichnet in unserem Zusammenhang die *Anordnung* der Prozessorbefehle *innerhalb von 16-, 32- oder 64-Byte-Speicherblöcken*. Die Frage ist dabei, ob ein Maschinenbefehl in hexadezimaler Schreibweise z.B. auf einer Null, einer 7 oder etwa auf C liegt. Das ist deshalb von Bedeutung, weil der Prozessor seine Befehlsfolgen in ebensolchen Blöcken aus dem Speicher in seinen Cache kopiert. Im Prozessor-internen Speicher wiederum haben diese Blöcke eine überragende Bedeutung, weil sich innerhalb dieser Blöcke z.B. die Ablaufvorhersage organisiert. Um das anhand der vorigen Variante 3b zu verdeutlichen:

```

.balign 16
.LLoop:  decw    %ax                // 2 Byte code 6648    - beginnt auf Adresse xyz0 h
          jz     .LExit            // 2 Byte code 74xx    - beginnt auf Adresse xyz2 h
          incl   %esi              // 1 Byte code 46       - liegt auf Adresse xyz4 h
          cmpb   (%esi), %dl       // 2 Byte code 3a16  - beginnt auf Adresse xyz5 h
          jne    .LLoop            // 2 Byte code 75xx    - beginnt auf Adresse xyz7 h
          jmp    .LResult          // 2 Byte code ebxx  - beginnt auf Adresse xyz9 h
.LResult: subw   %ax, %cx          //                - beginnt auf Adresse xyzB h

```

Die Anweisung „*.balign 16*“ sorgt dafür, dass der Assembler genau so viele *NOPs* (No-Operation-Anweisungen, die nichts tun, außer Platz und minimal Rechnerzeit zu verbrauchen) einfügt, dass die nachfolgende Anweisung auf einer hexadezimalen Null, d.h. auf einem 16-Byte-Blockanfang zu liegen kommt. Mit dieser Anweisung ist also sichergestellt, dass der Loopstart immer an einer 16-Byte-Blockgrenze ausgerichtet ist. Dasselbe lässt sich auch im Pascal-Quellcode erzwingen: Der Free-Pascal-Compiler stellt hierfür die Compileranweisung `{ $CODEALIGN LOOP=xx }` zur Verfügung.

Nun passt dieser gesamte Loop bequem in einen einzelnen 16-Byte-Block – das sollte doch für hohe Geschwindigkeiten optimal sein – oder etwa nicht? Könnte also das Alignment so eine Art „*Zauberstab*“ sein? Nächste Überraschung: Nein, das *glatte Gegenteil* scheint – bei manchen Prozessoren jedenfalls - der Fall zu sein!

Um die Auswirkungen des Alignments zu untersuchen, habe ich nun jede der oben beschriebenen Code-Varianten ver-18-facht: Vor dem Loopbeginn wird zunächst per *.balign 16* sichergestellt, dass die nächste Anweisung auf einer hexadezimalen Null beginnt. Zwischen diesem *.balign* und dem tatsächlichen Loopstart habe ich dann weitere *NOPs* eingefügt, von 1 – 15 Byte Länge, also z.B. so:

```

.balign 16                                // 0 + 12 byte NOP (never reached)
.byte 0x66, 0x66, 0x0F, 0x1F, 0x84, 0x00, 0x00, 0x00, 0x00, 0x00 // 10 x NOP
.byte 0x66, 0x90                          // 2 x NOP
.LLoop:  decw    %ax                        // beginnt auf Adresse xyzC h

```

Da der Loopbeginn von oben her *angesprungen* (statt angelaufen) wird, werden die *NOPs* selbst, und zwar sowohl die von *.balign* automatisch eingefügten wie meine zusätzlichen, niemals selbst durchlaufen. Sie nehmen einfach nur Platz weg, kosten aber keine Rechnerzeit.

Auf diese Weise gibt es nun jeweils 16 Funktionen, die sich nur darin unterscheiden, dass der *Loopanfang* auf allen denkbaren Positionen innerhalb des 16-Byte-Blocks zu liegen kommt. Zusätzlich habe ich noch zwei weitere Subvarianten mit *.balign 32* und *.balign 64* hinzugefügt, insgesamt also pro Code-Variante je 18 Alignment-Subvarianten.

Das Ergebnis ist hochinteressant – und verwirrend: Die beiden untersuchten Prozessoren reagieren weitgehend genau *gegenläufig* auf dieses Alignment.

Der Celeron verhält sich so, wie man es erwarten würde: Er mag es nicht, wenn der Loop auf zwei Blöcke aufgeteilt wird, und solange diese Bedingung erfüllt ist, ist er nicht besonders wählerisch. Aber allzu groß ist der Unterschied dann auch nicht (5-6 %).

Frappierend ist hingegen das Verhalten des AMD64: Der reagiert mit einer deutlichen *Leistungssteigerung* (um 15 %), *sobald der unbedingte Sprung der untersten Zeile in den nächsten Block abgeschoben wird*. Und das, obwohl diese Sprunganweisung in nicht einmal 3 % aller Loopedurchläufe überhaupt erreicht wird! Was bedeutet, dass nicht die *eigentliche Ausführung* dieses Sprungs Zeit kostet, sondern dass vielmehr *die Vorhersage-Struktur des CPU-Cache-Blocks durch die Anwesenheit dieser Sprungmöglichkeit empfindlich gestört wird!*

Was die Variante 3b zur Favoritin beider Welten macht, ist also die Kombination dreier verschiedener Maßnahmen:

- Ersetzen der mehrheitlich durchlaufenen unbedingten Sprunganweisung zurück zum Loopanfang durch einen bedingten Sprung (gut für den AMD64),
- Ersetzen der Subtraktion durch ein Dekrementieren (gut für den Celeron)
- Setzen der Startposition des Loops auf einen Wert zwischen hexadezimal 4 und A (gut für den AMD64)

Die weiteren Varianten sind schnell erklärt: **Variante 3c** ist identisch mit der Variante 3a, mit dem Unterschied, dass die NOPs statt am Loopstart innerhalb des Loops selbst liegen, hier werden sie also tatsächlich durchlaufen.

```
.balign 16
.LLoop:  subw    $1, %ax
        jz     .LExit
        .byte  0x66, 0x66, 0x0F, 0x1F, 0x84, 0x00, 0x00, 0x00, 0x00, 0x00 // 10 x NOP
        .byte  0x66, 0x90 // 2 x NOP
        incl   %esi
        cmpb   (%esi), %dl
        jne    .LLoop
        jmp     .LResult
```

Auch hier reagiert der AMD64 empfindlicher als der Celeron (und unterm Strich sogar etwas besser), aber die Effekte sind, wie nicht anders erwartet, gering.

Anders als die bisher beschriebenen Funktionen versucht der vierte Block nicht, die Funktionalität der System.Pos(Char, ShortString) zu realisieren. Diesen Block habe ich in den Test aufgenommen, nachdem ich über eine besondere Merkwürdigkeit im Verhalten des AMD64 gestolpert bin.

Die **Variante 4a** enthält keinerlei Operationen und auch keinen Loop - sie macht nichts anderes als einen sofortigen „*Rücksturz zur Erde*“ anzutreten:

```
asm
    jmp    .LExit
    .balign 16
    .byte  0x0F, 0x1F, 0x44, 0x00, 0x00           // 5 x NOP
.LExit:
end;
```

Der Compiler setzt an die Stelle des Pascal-„*end*;“ den Prozessorbefehl „*RET*“ (sonst ist nichts zu tun, die Funktion baut keinen Stackframe auf). Nun zeigt sich, dass beim AMD64 die Speicherposition dieser Return-Anweisung erstaunliche Oszillationen im Verhältnis von knapp 1:3 bewirkt:

| | | | |
|--|----------|---|--------------|
| Function CharPos_Asm4a_NoLoop_ExitStart_0 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_1 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_2 | 301.3 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_3 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_4 | 301.5 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_5 | 113.3 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_6 | 301.4 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_7 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_8 | 301.4 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_9 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_A | 301.4 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_B | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_C | 301.5 ms | = | 14 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_D | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_E | 327.3 ms | = | 15 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_F | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_32 | 112.9 ms | = | 5 ns / Call |
| Function CharPos_Asm4a_NoLoop_ExitStart_64 | 360.8 ms | = | 16 ns / Call |

Zusätzlich erstaunt, dass der AMD64 hier offenbar *ungerade* Speicherstellen (sowie die hexadezimale Null) bevorzugt. Der Celeron wiederum zeigt bei diesem Test keinerlei Auffälligkeit. Und sobald man eine weitere Anweisung hinzufügt (**Variante 4b**):

```
asm
    pushq  %rsi
    jmp    .LExit
    .balign 16
    .byte  0x0F, 0x1F, 0x44, 0x00, 0x00           // 5 x NOP
.LExit:    popq  %rsi
end;
```

... ist der Spuk auch beim AMD64 sogleich vorüber.

3) Ausblick

Ich bin offen gestanden weit davon entfernt, diese Ergebnisse *interpretieren* oder gar *verstehen* zu können. Die Messungen zeigen allerdings, dass die Effekte in einer Größenordnung liegen, die alles andere als marginal ist. Es *könnte* sich also zumindest lohnen, der Sache tiefer auf den Grund zu gehen:

- Gibt es genügend *Gemeinsamkeiten* hinsichtlich des Micro-Laufzeitverhaltens heute gebräuchlicher Prozessoren oder sind deren *Unterschiede* zu groß? Lassen sich wenigstens ein paar *verallgemeinerbare Erkenntnisse* aus solchen Querschnitts-Tests gewinnen?
- Kann man auf niedrigster (Assembler-) Ebene überhaupt Codefolgen finden, die auf (nahezu) allen Rechnern gut bis perfekt laufen, oder sieht man da kein Land und *muß sich damit abfinden*, dass alles, was auch immer man macht, auf dem einen Rechner hervorragend läuft und auf dem nächsten ggfls. quälend langsam? Die CharPos-Funktionen, die im Test verwendet werden, sind mit einiger Wahrscheinlichkeit noch *nicht* das Ende der Fahnenstange. *Aber gibt es denn soetwas wie ein verbindliches "Ende der Fahnenstange" überhaupt?* Oder rennt man da hinter einer Fata Morgana her?
- Würde es womöglich Sinn machen, die Option *Hersteller- oder gar CPU-Familien-spezifischer Optimierungsalgorithmen* ins Auge zu fassen?

Anders gefragt: Mit welcher Unschärfe müssen wir leben? Wenn uns dieser Test dabei hilft, der Antwort auf diese Fragen *näherzukommen*, hat er seinen Zweck erfüllt.

Mir ist klar, dass dieses Testprogramm bestenfalls an der Kruste des Problems zu kratzen vermag. Aber vielleicht legen die weiteren Ergebnisse nahe, dass es durchaus lohnend wäre, das Thema in größerem Umfang systematisch anzugehen – und dann sicher auch mit mehr Sachverstand, als ich ihn (alleine) aufzubringen vermag.

Danke für Eure Geduld!

März 2015
Rüdiger Walter