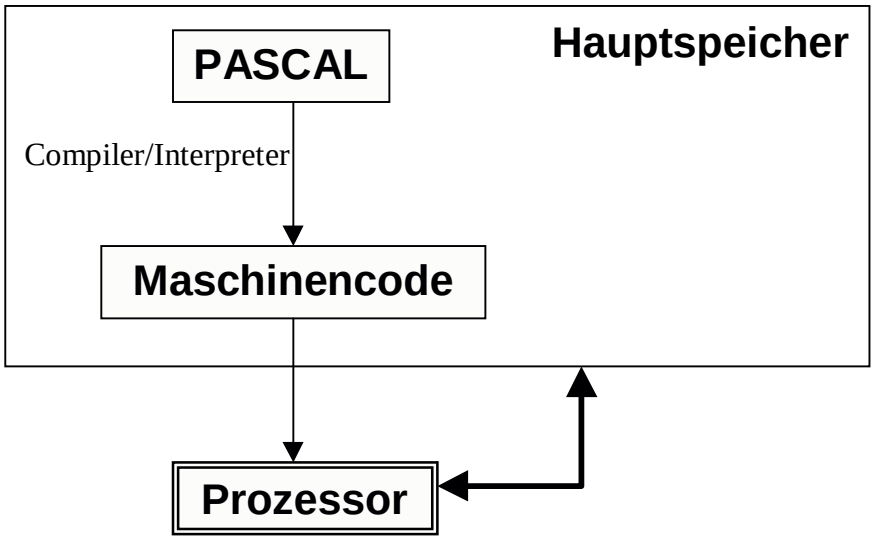
Funktionaler Aufbau eines Computers

Im Folgenden soll der Weg untersucht werden, wie ein Programm, das von einem Compiler/Interpreter in Maschinencode übertragen wurde, schließlich vom Prozessor abgearbeitet wird.



1. Zuweisungen

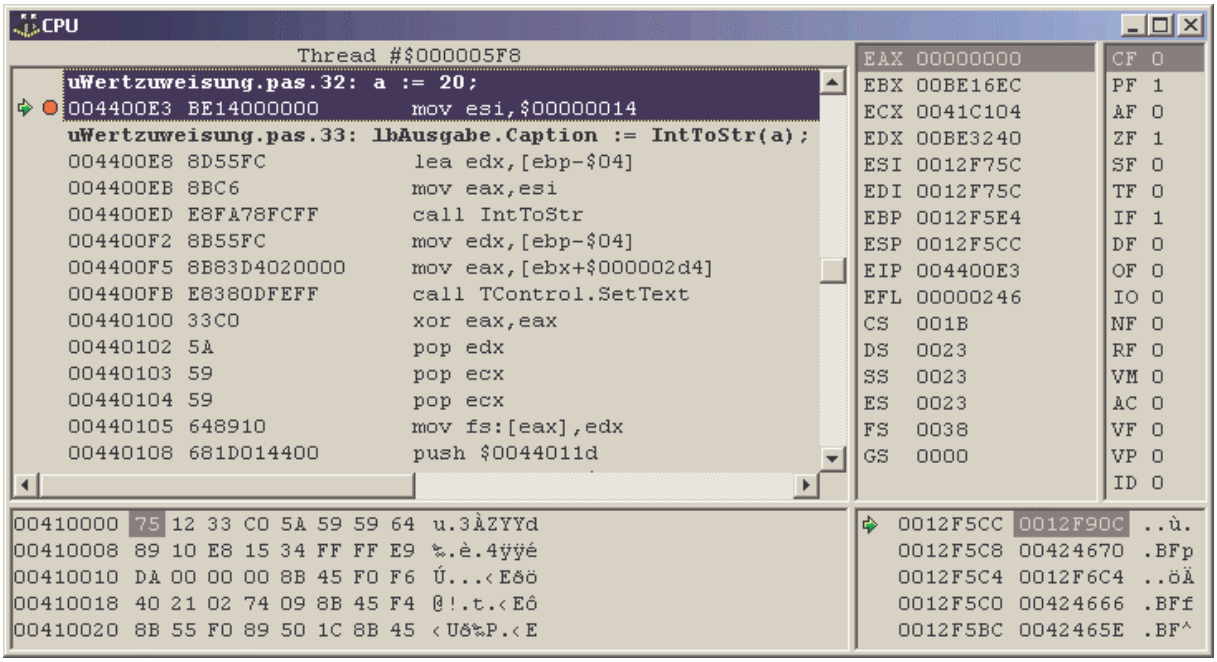
Es sollen zunächst einige einfache Anweisungen, wie Wertzuweisung an eine Variable, Selektion und Iteration betrachtet werden.
Wie verfährt der Compiler von Delphi etwa mit der Anweisung a := 20?
Delphi verfügt über ein CPU-Fenster, in welchem der Maschinencode angezeigt wird, sowie ein Blick in den Prozessor, genauer auf dessen interne Speichereinheiten, den Registern, ermöglicht wird. Der Maschinencode (Assemblerbefehle) kann Schrittweise ausgeführt werden und die Veränderungen der Registerinhalte studiert werden.

Wir erstellen in Delphi folgende Click-Prozedur:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var a: integer;
begin
  a := 20;
  lbAusgabe.Caption := IntToStr(a);     {siehe Hinweis!}
end;
```

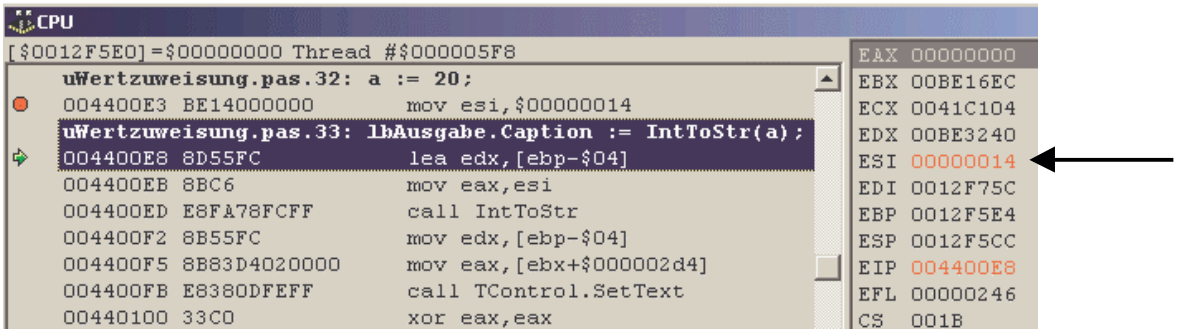
Hinweis: Damit die Zuweisung von Delphi überhaupt compiliert wird, muss der Wert auch benutzt werden, z.B. ausgegeben werden. Wird ein Wert nicht benutzt, so optimiert Delphi den Code dahingehend, dass unnötige Zuweisungen ignoriert werden.
Wir setzen in der Zuweisungszeile a := 20; einen Haltepunkt, damit ab dieser Stelle eine schrittweise Code-Ausführung ermöglicht wird.

Nach dem Starten hält das Programm an dieser Stelle an und wir öffnen das CPU-Fenster (Ansicht/Debugfenster/CPU bzw. Strg-Alt-C):



Der Bereich links oben zeigt den aktuellen Programmausschnitt im Hauptspeicher (Adresse Hexadezimalen Maschinencode und die Übersetzung in Assemblercode), unten links wird der Maschinencode fortlaufen hexadezimal angezeigt (Adresse in 8-er Schritten 8 Byte Code). Im rechten Bereich werden die Prozessorregister EAX (extendedAX) bis EFL, je 32 Bit (= 4 Byte groß) sowie CS bis GS (je 2 Byte groß) dargestellt. In der rechten Spalte finden wir so genannte Flags (CF bis ID). Der Bereich unten rechts zeigt einen Ausschnitt des Hauptspeichers im Datenbereich (Stack) (Adresse Inhalt (4 Byte groß)).
Blau hinterlegt wird die aktuelle Position im Programmcode dargestellt. Wir erkennen die auszuführende Pascalanweisung `a := 20;` der grüne Pfeil markiert den zugehörigen Maschinencode, der als nächstes ausgeführt wird.
Assemblercode: `mov esi,$00000014`
Bedeutung: Bewege nach (move Abk.: `mov`) Register ESI die HexZahl ($) 00000014 = 20_{dez}.
Maschinencode: BE14000000. BE ist der Code für mov esi, die 14 stellt die Zahlkonstante dar.
Der Assemblercode ermöglicht uns eine leichtere Lesbarkeit. Man müsste sonst z.B. die Bedeutung des Codes BE wissen.

Führt man nun über den Einzelschrittmodus in Delphi den Programmschritt aus, so erhält man folgendes CPU-Fenster:



Links wurde die nächste Anweisung ausgewählt.
Interessant ist aber der Registerbereich. Das Register ESI enthält nun tatsächlich den Wert 00000014.
Die folgenden Anweisungen betreffen nun die Aufbereitung und Ausgabe des Ergebnisses über das Label. Diesen Code zu verfolgen würde zu weit führen.
Es sollen aber weitere Beispiele in Übungen untersucht werden.

Als Ergebnisse der Zuweisungen (Aufgabe 1) können wir festhalten, dass

- der Name der Variable keine Rolle spielt,
- bei der Verwendung mehrerer Variablen eine Auslagerung in den Hauptspeicherbereich erfolgt,
- die Zuweisung des Wertes 0 durch die Anweisung `xor Register,Register` erfolgt,
- negative Werte als Zweierkomplement verarbeitet werden.

Bei der Verwendung großer Zahlen erkennt man, dass Zahlen im Speicher byteweise in der Reihenfolge von niederwertigen zu höherwertigen Stellen eingetragen werden. Die Zahl $305.419.896_{dez} = 12.34.56.78_{hex}$ steht im Speicher in der Reihenfolge 78.56.34.12.

2. Terme

Als nächsten Schritt betrachten wir die Verarbeitung von Termen.

Wir erstellen folgende Click-Prozedur:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var a, b: integer;
begin
  a := 5;
  b := 10;
  a := a + b;
  lbAusgabe.Caption := IntToStr(a);
end;
```

Das CPU-Fenster liefert uns folgenden Maschinencode:

```
BB05000000      mov ebx,$00000005   Variable a
B80A000000      mov eax,$0000000a   Variable b
03D8            add ebx,eax         Ergebnis in EBX (Variable a)
```

Allgemein wird die Initialisierung zweier Variablen und deren Addition also nach folgendem Schema ausgeführt:

```
mov ebx,Wert1  (1. Variable)
mov eax,Wert2  (2. Variable)
add ebx,eax    Ergebnis in EBX (1. Variable)
```

Im zweiten Beispiel soll die Summe zweier Variablen einer dritten zugewiesen werden:

```
c := a + b;
```

Das CPU-Fenster liefert uns folgenden Maschinencode:

```
B805000000      mov eax,$00000005   Variable a
BA0A000000      mov edx,$0000000a   Variable b
8D3402          lea esi,[edx+eax]   Ergebnis in ESI (Variable c)
```

Hier bedeutet `lea` **l**ade die **e**ffektive **A**dresse, welche durch den Klammerterm gegeben ist. Damit enthält ESI die Summe von EDX und EAX.

Delphi löst die Initialisierung zweier Variablen und die Zuweisung deren Summe an eine dritte Variable also nach folgendem Schema:

```
mov eax,Wert1        (1. Variable)
mov edx,Wert2        (2. Variable)
lea esi,[edx+eax]    Ergebnis in ESI (3. Variable)
```

Multiplikationen werden wie folgt in Maschinencode übersetzt:

```
a := a * b;

mov ebx,Wert1  (1. Variable)
mov eax,Wert2  (2. Variable)
imul ebx,eax   Ergebnis in EBX  (1. Variable)

c := a * b;

mov eax,Wert1  (1. Variable)
mov edx,Wert2  (2. Variable)
mov esi,eax    (1. Variable als Faktor bereitstellen)
imul esi,edx   Ergebnis in ESI (3. Variable)
```

3. Verzweigungen

Wir wenden uns nun Programmstrukturen zu. Das erste Beispiel stellt eine einfache Entscheidung dar, welche nur eine *then*-Anweisung enthält:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var
  a, b: integer;
begin
  a := 5;
  b := 0;
  if a = 7 then b := 8;
  lbAusgabe.Caption := IntToStr(b);
end;
```

Delphi erzeugt hierzu folgenden Maschinencode:

```
B805000000     mov eax,$00000005   Variable a
33DB           xor ebx,ebx         Variable b auf 0 setzen
83F807         cmp eax,$07         Vergleiche EAX mit 7
7505           jnz +$05            Springe, falls nicht 0 um 5 weiter
BB08000000     mov ebx,$00000008   Variable b 8 zuweisen
```

Wie erfolgt der Sprung und was bedeutet dabei "nicht 0"?

Diese Überprüfung bezieht sich auf den vorherigen Vergleich `cmp` (= co**mp**are). Hierbei wird die Differenz (EAX – 7) gebildet. Das Ergebnis selbst wird verworfen. Allerdings wird in den Flags unter anderem festgehalten, ob das Ergebnis 0 (Zero-Flag) war, bzw. negativ (Sign-Flag). In diesen Fällen werden die Flags jeweils gesetzt, d.h. mit 1 belegt.
Die Anweisung `jnz` (= **j**ump **n**ot **z**ero) überprüft nun, ob das Zero-Flag (ZF) gesetzt ist. Falls dies *nicht* der Fall ist (ZF = 0), wird der Sprung ausgeführt.
In diesem Fall wird 5 Byte weitergesprungen im Programmcode. Das bedeutet, es wird zur aktuellen Adresse 5 addiert, womit sich die Adresse für die nächste Anweisung ergibt. Diese

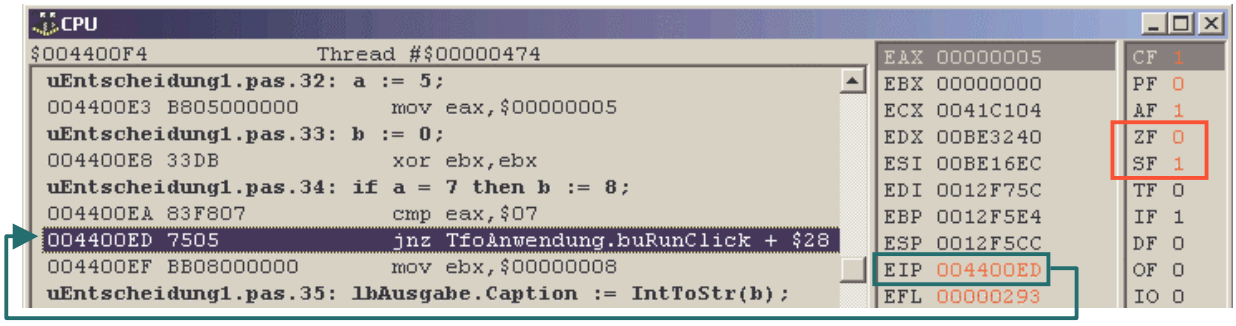
Addition erfolgt ins EIP (Instruction-Pointer-Register), wo die jeweils aktuelle Adresse für die nächste auszuführende Anweisung steht.
Hinweis: Delphi gibt in manchen Versionen als Sprungziel
`TfoAnwendung.buRunClick + $28` an.
Hierbei wird die Sprungzieladresse aus der Startadresse der Klickprozedur (**begin**) ermittelt. Der Maschinencode enthält jedoch die relative Entfernung von der aktuellen Adresse aus (hier 05; zweites Byte im Code 75 05). Die Sprungentfernung hängt natürlich von den Anweisungen im then-Teil ab. In diesem Fall lautet die Anweisung

```
BB 08 00 00 00     mov ebx,$00000008
```

Man erkennt, dass der Maschinencode aus 5 Byte besteht, welche übersprungen werden müssen.

Führt man den Code im Einzelschrittmodus aus, kann man die Änderung der Flags erkennen, sowie die entsprechende Änderung der Befehlsadresse in EIP.

Das folgende Bild zeigt den Zustand der CPU nach der Ausführung der cmp-Anweisung:



`cmp eax,$07` → EAX – $07 → 5 – 7 = –2
⇒ ZF := 0, da Ergebnis ≠ 0 und SF := 1, da Ergebnis negativ.

`jnz +$05` prüft `not(ZF)` → `not(0) = 1`
⇒ Der Sprung wird ausgeführt.
Vergleichen wir nochmals mit der Anweisung in Pascal:

```
if a = 7 then b := 8;
```

so gilt für die Differenz-Darstellung: a – 7 = 0

```
if a - 7 = 0 then b := 8;
```

Die then-Anweisung ist also auszuführen, wenn a – 7 = 0 ist. Sie ist daher zu überspringen, wenn dies nicht der Fall ist → `jnz`.

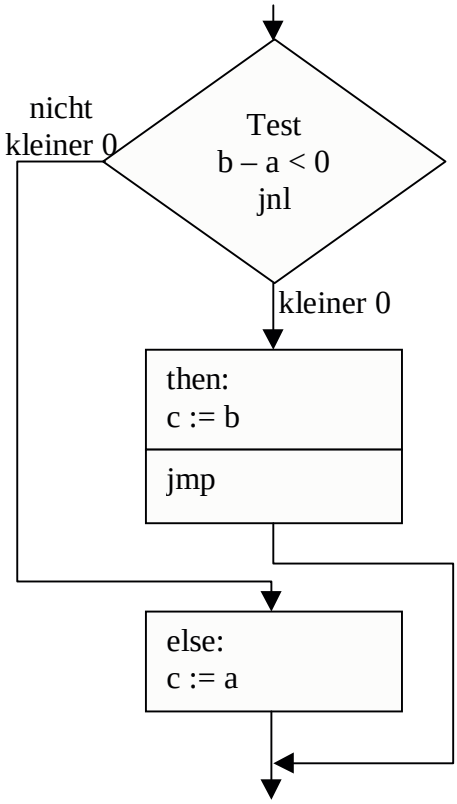
Im zweiten Beispiel betrachten wir eine zweiseitige Verzweigung:

```
a := 14;
b := 12;
if a > b then c := b else c := a;
```

Delphi erzeugt folgenden Maschinencode hierzu:

```
B80E000000      mov eax,$0000000e    (Variable a)
BA0C000000      mov edx,$0000000c    (Variable b)
3BD0            cmp edx,eax          (Vergleiche EDX – EAX (= b – a))
7D04            jnl +$04             (Springe falls nicht kleiner 0 um 4 Byte)
8BDA            mov ebx,edx          (c := b)
EB02            jmp +$02             (Springe um 2 Byte weiter)
8BD8            mov ebx,eax          (c := a)
```

Die Untersuchung zeigt, dass die Differenz b – a getestet wird.
Aus der Umformung `a > b` ⇔ `b – a < 0` ergibt sich, dass die then-Anweisung zu überspringen ist, wenn `b – a < 0` nicht erfüllt ist, bzw. wenn `b – a` nicht kleiner 0 ist. Genau dies bewirkt `jnl` (= **j**ump **n**ot **l**ess 0). Diese Anweisung sorgt für den Sprung zum Anfang des else-Teils. Hierbei sind die nächsten zwei Anweisungen zu je 2 Byte also insgesamt 4 Byte zu überspringen. Diese Sprunganweisung wertet das Sign-Flag aus. Dabei kann `jnl` übersetzt werden in `jump not(SF)`.
Wird der then-Teil ausgeführt, so ist zu verhindern, dass der anschließend folgende else-Teil ebenfalls noch ausgeführt wird. Hierfür sorgt wieder ein Sprung, der jedoch keiner Bedingung unterliegt und in jedem Fall ausgeführt wird. Dieser *unbedingte Sprung* `jmp` (= **j**u**mp**) übergeht den else-Teil, der in diesem Fall aus einer Anweisung zu 2 Byte besteht.
Den gesamten Ablauf illustriert folgendes Diagramm:



Die anderen Sprungarten, die unter bestimmten Voraussetzungen ausgeführt werden, nennt man *bedingte Sprünge*. Beispiele sind `jnz, jnl, jz, jg` usw.

4. Schleifen

Im letzten Teil untersuchen wir, wie Schleifen auf Maschinenebene realisiert weden.

a) For-Schleife

Als Klickprozedur wird nachfolgende Zählschleife programmiert:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var
  i, a: integer;
begin
  a := 1;
  for i := 1 to 10 do a := a + 1;
  lbAusgabe.Caption := IntToStr(a);
end;
```

Delphi übersetzt die Prozedur in folgenden Maschinencode:

```
BB01000000      mov ebx,$00000001   (Variable a)
B80A000000      mov eax,$0000000a   (Variable i)
43              inc ebx             (Erhöhe a um 1)
48              dec eax             (Erniedrige i um 1)
75FC            jnz -$04            (Springe, falls nicht 0, 4 Byte zurück)
```

Zunächst fallen einige Besonderheiten auf:

Die Anweisung `a := a + 1` wird durch die effektive Anweisung `inc` (= increment) realisiert.
Der Schleifenzähler wird auf den Endwert initialisiert und dann erniedrigt. Auch hierbei kommt die effektive Anweisung `dec` (= decrement) zum Einsatz.
Der Sprung zum Anfang der Schleife erfolgt mit `jnz -$04`. Dabei wird der EIP um 4 erniedrigt, da FC ein negativer Wert, nämlich –4 darstellt. Wir wissen bereits, dass `jnz` das Zero-Flag auswertet. Wie aber wird dieses gesetzt? Es fehlt scheinbar eine compare-Anweisung.
Hier kommt eine weitere Besonderheit von `inc` bzw `dec` zum Tragen. Diese Anweisungen setzen je nach Ergebnis unter anderem das Zero-Flag (ZF) und das Parity-Flag (PF). Beim Einzelschrittmodus lässt sich das auch gut verfolgen. Delphi stellt im CPU-Fenster Veränderungen, die bei einer Anweisung auftreten, rot dar.
So kann tatsächlich der nötige Rücksprung ermittelt werden ohne eine weitere compare-Anweisung. Jetzt wird auch klar, warum die Schleife rückwärts gezählt wird, da die Auswertung sonst nicht auf diese Weise erfolgen könnte.

b) Repeat-Schleife

Als nächstes betrachten wir die Repeat -Schleife:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var
a, b: integer;
begin
  a := 11;
  b := 2;
  repeat
    a := a - b;
  until a < b;
  lbAusgabe.Caption := IntToStr(a);
end;
```

Der Maschinencode, den Delphi hierzu erzeugt, sieht folgendermaßen aus:

```
BB0B000000      mov ebx,$0000000b   (Variable a)
B802000000      mov eax,$00000002   (Variable b)
2BD8            sub ebx,eax         (Subtrahiere EBX – EAX (= a – b),
                                     Ergebnis in EBX (= a))
3BC3            cmp eax,ebx         (Vergleiche EAX – EBX (= b – a))
7EFA            jle -$06            (Springe, falls kleiner oder gleich 0 um 6
                                     Byte zurück)
```

Zunächst wird der Schleifenkörper ausgeführt (`sub`-Anweisung).

Danach erfolgt ein compare von b – a. Die Abbruchbedingung a < b ergibt umgeformt b - a > 0. Da der Sprung jedoch als Wiederholung ausgeführt wird, nämlich zurück zum Schleifenbeginn, hat dieser für b – a <= 0 zu erfolgen. Das bedeutet bei compare müsste in diesem Fall SF oder ZF gesetzt sein: `jle` (= **j**ump **l**ess or **e**qual 0) führt aber genau in diesem Fall zur Sprungausführung. Es wird um 6 Byte zurückgesprungen, also zum Schleifenkörperbeginn (`sub`-Anweisung).

c) While-Schleife

Als letztes betrachten wir die While-Schleife:

```
procedure TfoAnwendung.buRunClick(Sender: TObject);
var
a, b: integer;
begin
  a := 11;
  b := 2;
  while a >= b do a := a - b;
  lbAusgabe.Caption := IntToStr(a);
end;
```

Der Maschinencode, den Delphi hierzu erzeugt, sieht folgendermaßen aus:

`BB0B000000`	`mov ebx,$0000000b`	(Variable `a`)
`B802000000`	`mov eax,$00000002`	(Variable `b`)
`3BC3`	`cmp eax,ebx`	(Vergleiche EAX – EBX (= `b` – `a`))
`7F06`	`jnle +$06`	(Springe, falls nicht kleiner oder gleich 0 um 6 Byte weiter)
`2BD8`	`sub ebx,eax`	(Subtrahiere EBX – EAX (= `a` – `b`), Ergebnis in EBX (= a))
`3BC3`	`cmp eax,ebx`	(Vergleiche EAX – EBX (= `b` – `a`))
`7EFA`	`jle -$06`	(Springe, falls kleiner oder gleich 0 um 6 Byte zurück)

Die Abfrage a >= b wird umgeformt in b – a <= 0.
Die Anweisung `jnle` (= **j**ump **n**ot **l**ess or **e**qual 0) führt also den Sprung aus, falls not (SF or ZF). Hieraus folgt SF = 0 und ZF = 0, d.h. der Vergleich b – a war größer 0. Man erkennt, dass der Sprung ausgeführt wird, wenn die Abfrage b – a <= 0 falsch ist, also die Schleife nicht ausgeführt werden darf. Das Sprungziel ist die nächste Anweisung nach dem Schleifenkörper. Es werden 6 Byte übersprungen. Dies sind die nächsten drei Anweisungen zu je 2 Byte. Damit ist die Ausführungsbedingung realisiert.
Man sollte nun annehmen, dass am Ende des Schleifenkörpers ein unbedingter Rücksprung zur Ausführungsbedingung erfolgt. Die Lösung sieht hier anders aus.
Nach einem nochmaligen compare-Vergleich erfolgt dieses Mal der Sprung im entgegengesetzten Fall wie bei der Ausführungsbedingung: `jle` wird also ausgeführt, wenn der Schleifenkörper wiederholt werden muss. Es wird um 6 Byte nach vorne übersprungen, was zum Beginn des Schleifenkörpers führt (`sub`-Anweisung).
Damit ist der zweite Teil der Schleife eigentlich eine Schleife mit Wiederholbedingung am Schluss. Dies entspricht aber der Repeat-Schleife.

Die While-Schleife wird also als Repeat-Schleife mit zusätzlicher Eintrittsbedingung realisiert.

Die nachfolgenden Diagramme zeigen den Ablauf im Vergleich: