Technische Informatik I

FPU, MMX, SSE, x86-64

Thorsten Thormählen

30. Januar 2018

Teil 10, Kapitel 3

Dies ist die Druck-Ansicht.

Aktiviere Präsentationsansicht

Steuerungstasten

→ nächste Folie (auch Enter oder Spacebar).
← vorherige Folie
d schaltet das Zeichnen auf Folien ein/aus
p wechselt zwischen Druck- und Präsentationsansicht
CTRL + vergrößert die Folien
CTRL - verkleinert die Folien
CTRL 0 setzt die Größenänderung zurück

Notation

Typ	Schriftart	Beispiele
Variablen (Skalare)	kursiv	a, b, x, y
Funktionen	aufrecht	$\mathrm{f}, \mathrm{g}(x), \max(x)$
Vektoren	fett, Elemente zeilenweise	$\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$, $\mathbf{B} = (x, y, z)^\top$
Matrizen	Schreibmaschine	$\mathtt{A}, \mathtt{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Mengen	kalligrafisch	$\mathcal{A}, B = \{a, b\}, b \in \mathcal{B}$
Zahlenbereiche, Koordinatenräume	doppelt gestrichen	$\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$

Inhalt

Fließkomma-Recheneinheit ("Floating Point Unit", FPU)

Multi Media Extension (MMX)

Streaming SIMD Extensions (SSE)

x86-64 Register

Fließkomma-Recheneinheit

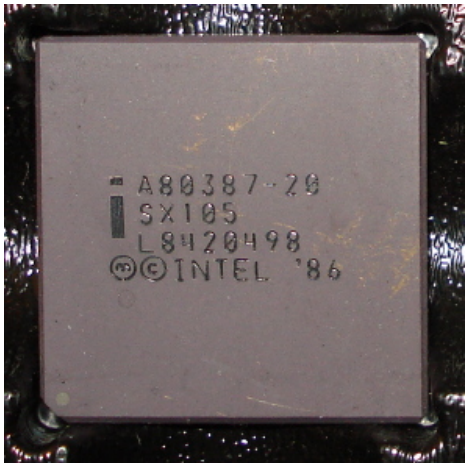
Bisher wurden nur Maschinensprachebefehle für das Rechnen mit ganzen Zahlen besprochen

Sollen Gleitkommazahlen verarbeitet werden, gibt es die Möglichkeit, die erforderlichen Operationen durch eigene Assemblerroutinen mit den bisherigen Befehlen zu implementieren

Die andere Möglichkeit ist, spezielle Gleitkomma-Befehl zu verwenden

Früher (bis zum Intel 80486) gab für die Berechnungen mit Gleitkommazahlen bei Intel die 80x87-Koprozessoren

Heutzutage ist die Fließkomma-Recheneinheit (engl. "Floating Point Unit", FPU) in die CPU integriert



Intel 80387 FPU

[**Bildquelle:** Intel 80387SX-20 SX105 by David Olsen, Creative Commons License]

FPU Register

Die Fließkomma-Recheneinheit hat acht 80-bit Register: `ST0`, `ST1`, ..., `ST6`, `ST7`

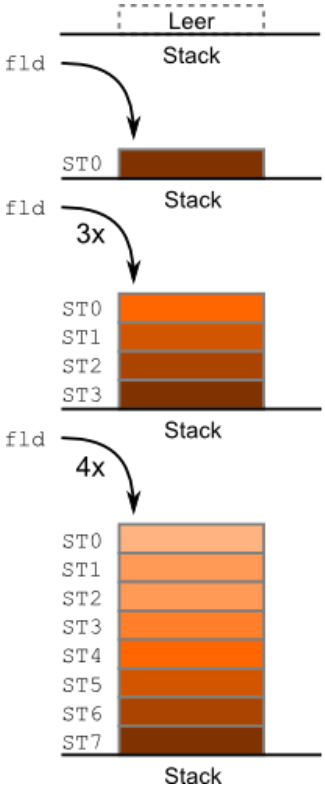
Der Assemblerbefehl `fld` zum Laden eines Registers bezieht sich immer auf `ST0`

Die Register verhalten sich so ähnlich, wie ein auf maximal acht Elemente begrenzter Stapelspeicher

Dabei bezeichnet `ST0` immer das oberste Element des Stapelspeichers

D.h. wird ein neues Element per Assemblerbefehl `fld` geladen (d.h. auf den Stapel gelegt) werden die Daten an das nächste Register weitergereicht:
`ST7=ST6`, `ST6=ST5`, ..., `ST1=ST0`, `ST0=neuer Wert`

Anzeigen der FPU-Register in MS Visual Studio:
Menü → Debuggen→ Fenster→ Register→ Kontext-Menü (rechte Maustaste)→Floating Point



FPU Register

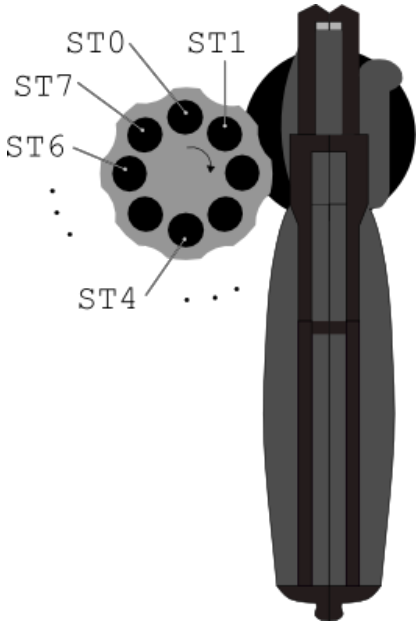
Als Analogie zur Funktionsweise der Register kann auch die Trommel eines Revolvers mit acht freien Kammern herangezogen werden

Der Inhalt von Register `ST0` kann immer an der "12 Uhr" Position abgelesen werden, von `ST1` bei "1:30 Uhr", von `ST2` bei "3 Uhr", usw.

Das Laden von Werten aus dem Speicher per Assemblerbefehl `fld` kann nur über die oberste Kammer an der "12 Uhr" Position (also über `ST0`) erfolgen

Bei einem Ladevorgang dreht sich die Trommel zunächst automatisch im Uhrzeigersinn und der Wert wird anschießend in das Register `ST0` geschrieben

Analogie: Die Revolvertrommel dreht sich und die oberste Kammer wird mit einer Kugel geladen



Eine Revolvertrommel als Analogie

FPU Register

Mit der Revolvertrommel als Analogie kann leicht nachvollzogen werden, was passiert, wenn mehr als acht Werte geladen werden. Es gilt: Wird versucht, eine Kammer mit einer Kugel zu laden, die schon besetzt ist, wird die Kugel beschädigt

D.h. bei einem Ladevorgang dreht sich die Trommel zunächst automatisch im Uhrzeigersinn aber der ST0 Wert wird ungültig. Beispiel (Quelldatei: main.cpp):

```
int main() {
  float a = 1.1f;
  float b = 0.3f;
  double c = 0.6f;
  __asm  {
    fld a;// ST0=1.1
    fld b;// ST1=1.1, ST0=0.3
    fld c;// ST2= 1.1, ST1=0.3, ST0=0.6
    fld a;// ST3= 1.1, ST2=0.3, ST1=0.6, ST1=0.1
    fld b;// ST4= 1.1, ST3=0.3, ST2=0.6, ST1=0.1, ST0=0.1
    fld c;// ST5= 1.1, ST4=0.3, ST3=0.6, ST2=0.1, ST1=0.1,  ST0=0.6
    fld a;// ST6= 1.1, ST5=0.3, ST4=0.6, ST3=0.1, ST2=0.1,  ST1=0.6, ST0=1.1
    fld b;// ST7= 1.1, ST6=0.3, ST5=0.6, ST4=0.1, ST3=0.1,  ST2=0.6, ST1=1.1, ST0=0.3
    fld c;// the stack (revolver cylinder) is full
    fld a;// values are still shifted but overwritten ones are invalid
  }
}
```

Der FPU Assemblerbefehl FLD

Zum Laden (engl. "load") des `ST0` Registers mit einer Gleitkommazahl kann der Befehl `fld` verwendet werden

```
fld quelle
```

Dabei kann `quelle` eine Speicheradresse sein, an der die Gleitkommazahl im RAM abgelegt ist, oder ein anderes FPU-Register: `ST1`, ..., `ST7`

Beispiel (Quelldatei: main.cpp):

```
int main() {
  float a = 1.1f;
  float b = 0.3f;
  double c = 0.6f;
  __asm  {
    fld a     ;// ST0=1.1
    fld b     ;// ST1=1.1, ST0=0.3
    fld c     ;// ST2= 1.1, ST1=0.3, ST0=0.6
    fld st(0) ;// ST3= 1.1, ST2=0.3, ST1=0.6, ST0=0.6
    fld st    ;// ST4= 1.1, ST3=0.3, ST2=0.6, ST1=0.6, ST0=0.6
    fld st(3) ;// ST5= 1.1, ST4=0.3, ST3=0.6, ST2=0.6, ST1=0.6, ST0=0.3
  }
  return 0;
}
```

Die FPU Assemblerbefehle FST und FSTP

Zum Speichern (engl. "store") des `ST0` Register als Gleitkommazahl können die Befehle `fst` und `fstp` verwendet werden

```
fst ziel
fstp ziel
```

Dabei kann `ziel` eine Speicheradresse sein oder ein anderes FPU-Register: `ST1`, ..., `ST7`

Während `fst` nur speichert, entfernt `fstp` das oberste Element vom Stack (engl. "store pop").

Analogie für `fstp`: Die Kugel wird an `ST0` entnommen und an `ziel` abgelegt und dann die Trommel gegen den Urzeigersinn gedreht

Die FPU Assemblerbefehle FST und FSTP

Beispiel:

```
int main() {
  float a = 1.1f;
  float b = 0.3f;
  double c = 0.6f;
  __asm  {
    fld a       ;// ST0=1.1
    fld b       ;// ST1=1.1, ST0=0.3
    fst c       ;// c=0.3
    fst st(2)   ;// ST2=0.3, ST1=1.1, ST0=0.3
    fstp st(3)  ;// ST2=0.3, ST1=0.3, ST0=1.1
  }

  return 0;
}
```

Quelldatei: main.cpp

FPU Assemblerbefehle für ganze Zahlen

Folgende FPU-Befehle kommen beim Laden und Speichern von ganzen Zahlen zum Einsatz:

Befehl	Beschreibung
`fild`	Laden einer vorzeichenbehafteten ganzen Zahl
`fist`	Speichern einer vorzeichenbehafteten ganzen Zahl
`fistp`	Speichern und herausschieben ("pop") einer vorzeichenbehafteten ganzen Zahl

Beispiel (Quelldatei: main.cpp):

```
int main() {
  int a = 10;
  int b = 20;
  __asm  {
    fild a      ;// ST0=10
    fild b      ;// ST1=10, ST0=20
    fistp b     ;// b=20;
    fistp b     ;// b=10;
  }
  return 0;
}
```

Einige Arithmetische FPU Assemblerbefehle

Befehl	Beschreibung
`fadd ziel, quelle`	Addiert `ziel` und `quelle`. Wenn kein Ziel angegeben, gilt `ziel=ST0`
`faddp ziel, quelle`	Wie `fadd`, anschließend herausschieben ("pop") von `ST0`
`fsub ziel, quelle`	Subtrahiert `ziel` und `quelle`. Wenn kein Ziel, gilt `ziel=ST0`
`fsubp ziel, quelle`	Wie `fsub`, anschließend herausschieben ("pop") von `ST0`
`fdiv ziel, quelle`	Dividiert `ziel` und `quelle`. Wenn kein Ziel, gilt `ziel=ST0`
`fdivp ziel, quelle`	Wie `fdiv`, anschließend herausschieben ("pop") von `ST0`
`fmul ziel, quelle`	Multipliziert `ziel` und `quelle`. Wenn kein Ziel, gilt `ziel=ST0`
`fmulp ziel, quelle`	Wie `fmul`, anschließend herausschieben ("pop") von `ST0`
`fabs`	Berechnet den Betrag von `ST0`
`fchs`	Ändert das Vorzeichen von `ST0`
`fyl2x`	Berechnet `ST1 = ST1` $\cdot \log_2(\texttt{ST0})$, und "pop" von `ST0`
`fsqrt`	Berechnet die Quadratwurzel von `ST0`

FPU Assemblerbefehle zum Laden einer Konstanten

Folgende FPU-Befehle können zum Laden einer Konstanten eingesetzt werden:

Befehl	Beschreibung
`fld1`	Laden von 1.0
`fldl2e`	Laden von $\log_2 e$
`fldl2t`	Laden von $\log_2 10$
`fldlg2`	Laden von $\log 2 \ (= \log_{10} 2)$
`fldln2`	Laden von $\ln 2 \ (= \log_e 2)$
`fldpi`	Laden von $\pi = 3.141592...$

FPU Assemblerbefehle

Beispiel: Berechnung des natürlichen Logarithmus $\ln(x)$

Gemäß den Rechenregeln für Logarithmen gilt:

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$$

und

$$\log_a(b) = \frac{1}{\log_b(a)}$$

D.h. für den natürlichen Logarithmus ergibt sich:

$$\ln(x) = \log_e(x) = \frac{\log_2(x)}{\log_2 e} = \log_e(2) \cdot \log_2(x)$$

FPU Assemblerbefehle

Lösung:

```
#include <stdio.h>
#include <math.h>

float myLog(float x) {
  float result;
  __asm {
    fldln2  ;// load log_e(2) in ST0
    fld x   ;// load ST0=x; ST1=log_e(2)
    fyl2x   ;// ST1 = log_e(2)*log_2(x) = log_e(x); pop;
    fstp result; // store ST0 in result; pop;
  }
  return result;
}

int main() {
  float x = 25.0;
  float result = myLog(x);
  printf("ln(%f) = %f \n", x, result); // output result of myLog
  printf("ln(%f) = %f \n", x, log(x)); // check result with C log function
  return 0;
}
```

Quelldatei: main.cpp

FPU Assemblerbefehle

Laut "cdecl"-Aufrufkonvention werden Gleitkommazahlen im `ST0` Register zurückgegeben

Verbesserte Lösung:

```
#include <stdio.h>
#include <math.h>

float myLog(float x) {
  __asm {
    fldln2  ;// load log_e(2) in ST0
    fld x   ;// load ST0=x; ST1=log_e(2)
    fyl2x   ;// ST1 = log_e(2)*log_2(x) = log_e(x); pop;
  }
}

int main() {
  float x = 25.0;
  printf("ln(%f) = %f \n", x, myLog(x)); // output result of myLog
  printf("ln(%f) = %f \n", x, log(x)); // check result with C log function
  return 0;
}
```

Quelldatei: main.cpp

Multi Media Extension (MMX)

Mit dem Pentium MMX führte Intel 1997 eine Befehlserweiterung ein, die es erlaubt, die acht FPU-Register für SIMD-Operationen ("Single Instruction Multiple Data") zu verwenden

Diese Register werden mit `MM0`, `MM1`, ..., `MM7` angesprochen

Dabei wird jeweils von einem 80-Bit FPU-Register nur 64-Bits für ein MMX-Register verwendet

Die MMX-Befehle arbeiten mit ganzzahligen Operanden

Es ist möglich, für acht 8-Bit-, vier 16-Bit-, oder zwei 32-Bit-Operanden parallel die gleiche Operation auszuführen

D.h. es kann mit einem maximalen theoretischen Geschwindigkeitsgewinn von Faktor 8 gerechnet werden

Der Name MMX motiviert sich daher, dass solche SIMD-Operationen vor allem bei der Verarbeitung von Audio-, Bild-, oder Videodaten eingesetzt werden, d.h bei Anwendungen bei denen häufig die gleiche Operation auf vielen Elementen ausgeführt werden müssen

Multi Media Extension (MMX)

Insgesamt umfasst die MMX-Erweiterung mehr als 40 Befehle

Beispiele sind: `padd`, `psub`, `pmul`, `pxor`, `pand`, `por`, usw.

Die Befehle werden teilweise mit einem Suffix versehen, das signalisiert ob es sich um

- `s` (= "signed") vorzeichenbehaftete Operanden oder
- `us` (= "unsigned") vorzeichenlose Operanden handelt

Außerdem kennzeichnet ein Suffix, wie viele Bytes die Operanden besitzen:

Suffix	Größe der Operanden
b	Byte (1 Byte)
w	Word (2 Bytes)
d	Double Word (4 Bytes)
q	Quad Word (8 Bytes)

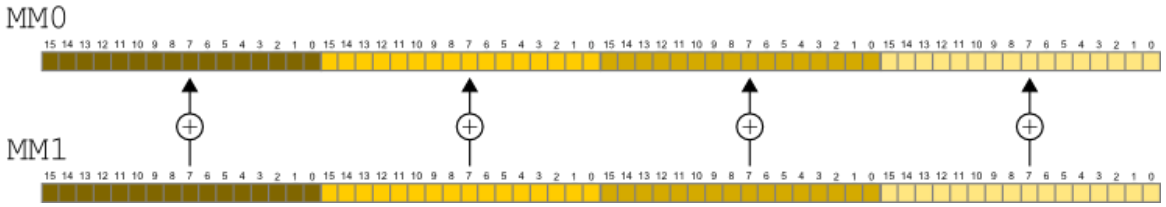
Frage: Der MMX-Befehl `paddusw` steht für welche Operation?

Multi Media Extension (MMX)

Der MMX-Befehl

```
paddusw mm0, mm1
```

addiert vier vorzeichenlose 16-Bit Zahlen



Die Ausführung erfolgt parallel, d.h. der Befehl berechnet zeitgleich vier Ergebnisse

Multi Media Extension (MMX)

Beispiel:

```
#include <stdio.h>

int main() {
  char text[] = "go fast!";
  char subMe[] = { 32, 32, 0, 32, 32, 32, 32, 0 };
  printf("%s \n", text);

  __asm {
    movq mm0, text     ;//move 64 bits into mm0 register
    psubsb mm0, subMe  ;//substract content of "text" and "subMe" in parallel
    movq text, mm0
    emms          ;// emms = Empty MMX Technology State (reactivate FPU registers)
  }

  printf("%s \n", text);
  return 0;
}
```

Quelldatei: main.cpp

Streaming SIMD Extension (SSE)

SSE ist eine weitere x86 Befehlserweiterung, die von Intel 1999 eingeführt wurde und auch von AMD unterstützt wird

SSE verfolgt die gleiche Idee wie MMX, dass die gleichen Operationen auf mehreren Operanden parallel ausgeführt werden ("Single Instruction Multiple Data")

Der Unterschied ist, dass bei SSE 8 neue 128-Bit breite Register spendiert wurden:
`XMM0`, `XMM1`, ..., `XMM7`

Ein weiterer wesentlicher Unterschied ist, dass SSE das Rechnen mit Gleitkommazahlen erlaubt

SSE2, SSE3, SSE4 sind spätere Befehlserweiterungen, die zusätzliche Operationen hinzufügen

Anzeigen der SSE-Register in MS Visual Studio:
Menü → Debuggen→ Fenster→ Register→ Kontext-Menü (rechte Maustaste)→SSE

Streaming SIMD Extension (SSE)

Hat ein SSE Befehl das Suffix `ps` ("Packed Single-precision"), z.B. `addps`, `subps`, `mulps`, `divps` usw., können vier 32-Bit-Gleitkommazahlen gleichzeitig von einem Befehl verarbeitet werden

Hat ein SSE Befehl das Suffix `pd` ("Packed Double-precision"), können zwei 64-Bit-Gleitkommazahlen gleichzeitig von einem Befehl verarbeitet werden

Des Weiteren gibt es die Suffixe `ss` ("Scalar Single-precision") und `sd` ("Scalar Double-precision"). Bei Befehlen mit diesen Suffixen ist der eine Operand ein Skalar und kein Vektor

Bei den `mov`-Befehlen kennzeichnet ein weiteres Suffix, ob die Speicheradresse, an der die Werte gelesen bzw. geschrieben werden sollen, auf ein Vielfaches von 16 Byte fällt:

- Wenn ja, kann der schnellere Befehl `movaps` ("move aligned") verwendet werden
- Wenn nein, der langsamere Befehl `movups` ("move unaligned")

Streaming SIMD Extension (SSE)

Beispiel:

```
#include <stdio.h>

int main() {
  float a[] = { 1.0f, 2.0f, 3.0f, 4.0f };
  float b[] = { 1.1f, 2.2f, 3.3f, 4.4f };
  float s = 2.0f;

  for (int i = 0; i < 4; i++) printf("a[%d] = %f\n", i, a[i]);

  __asm {
    movups xmm1, a   ;//load 4 floats = 128 bit
    movups xmm2, b   ;//load 4 floats = 128 bit
    addps xmm1, xmm2 ;//add all 4 floats in parallel
    addss xmm1, s    ;//add scalar s to 1st float, 2nd - 4th float remain unchanged
    movups a, xmm1   ;//store into a
  }

  for (int i = 0; i < 4; i++) printf("a[%d] = %f\n", i, a[i]);
  return 0;
}
```

Quelldatei: main.cpp

x86-64 Register

Beim Übergang auf die 64-Bit x86 Architektur (x86-64), wurden zunächst durch AMD und später auch durch Intel die Mehrzweckregister, das Flag Register und der Befehlszähler erneut erweitert und durch das Buchstabenpräfix `R` gekennzeichnet

Im 64-Bit Modus stehen dem Programmierer somit die Register `RAX`, `RBX`, ... `RSP` zur Verfügung. Die niederwertigen Teile davon können mit den älteren Bezeichnungen `EAX`, `AX`, `AH`, `AL`, etc. angesprochen werden

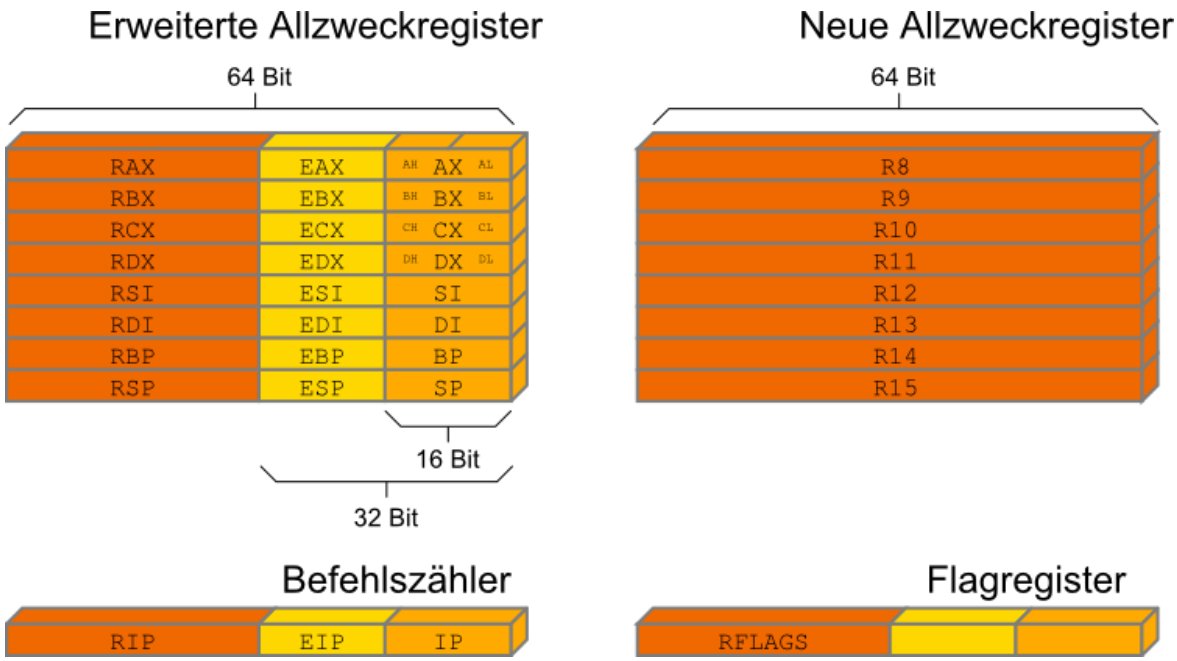
Neben diesen erweiterten Registern gibt es acht neue 64-Bit Mehrzweckregister mit den Bezeichnungen `R8`, .. , `R15`

Alle neueren Intel und AMD Prozessoren können wahlweise als 64-Bit Rechner oder im älteren 32-Bit Modus betrieben werden

Im 64-Bit Modus werden ältere 32-Bit Programme in einem Kompatibilitätsmodus ausgeführt

Davon machen auch die 64-Bit-Versionen der Betriebssysteme Windows 7 und 8 Gebrauch

x86-64 Register



Gibt es Fragen?

Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: Kontakt

Weitere Vorlesungsfolien