Praktische Informatik I

WS 2004/2005

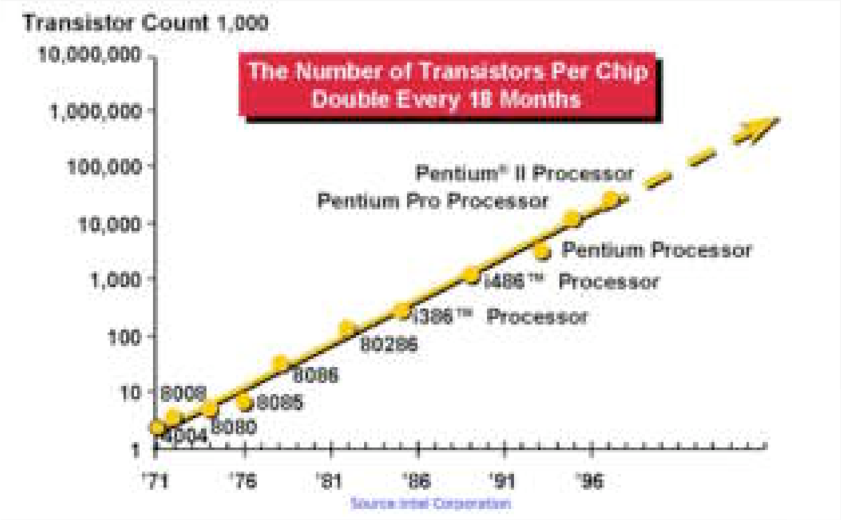
27

waste.informatik.hu-berlin.de/Lehre

3,5-Diskette (A:) formatieren

Der Datenträger in Laufwerk A:\ kann nicht formatiert werden, da er noch nicht formatiert ist.
Soll er jetzt formatiert werden?

Wachstum der Komplexität & das Problem der Skalierbarkeit



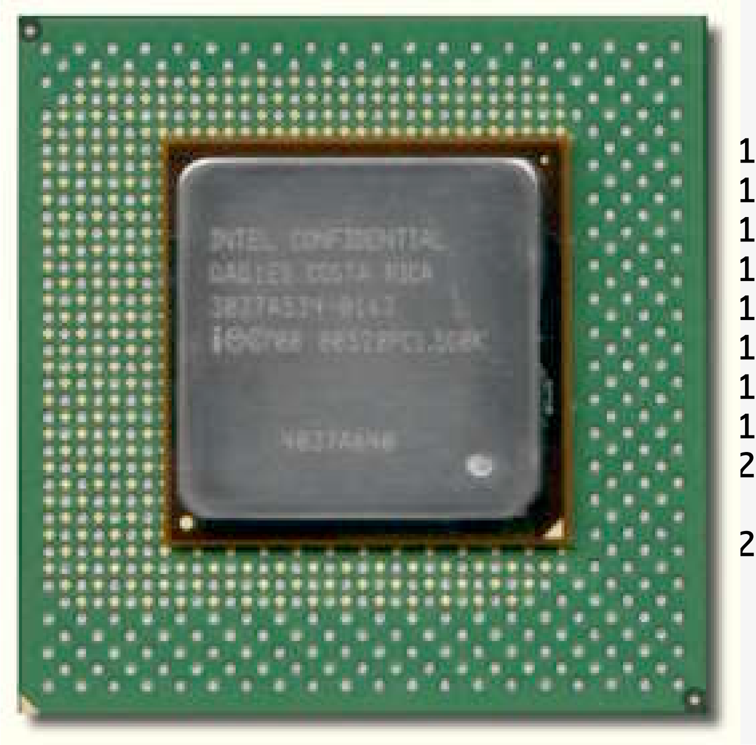
Source Intel Corporation

Moore's Law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will remain nearly constant for at least 10 years."

(Gordon Moore, *Electronics magazine*, April 1965)

Wachstum der Komplexität & das Problem der Skalierbarkeit



1975	8080	4500
1978	8086	29000
1982	80286	90000
1985	80386	229000
1989	80486	1,2 Mio
1993	pentium	3,1 Mio
1995	pentium *pro*	5,5 Mio
1998	pentium III	28 Mio
2000	pentium IV	42 Mio
2004		500 Mio

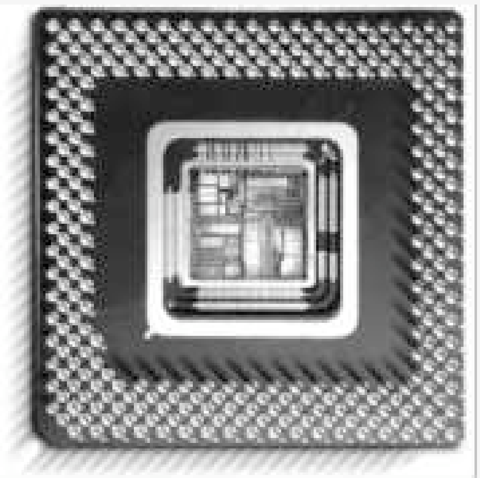
Prozessorchipfehler

386 A1 step	28	
386 B0 step	12	
386 B1 step	15	
386 D0 step	3	
386 "some versions"	19	
386 "all versions"	1	
Summe Intel 386		***78***
486 "early versions"	6	
486 "some versions"	8	
486 A-B4 step	3	
486 A-C0 step	2	
486 "all versions"	2	
Summe Intel 486		***21***
Pentium 60- und 66-MHz	21	
Pentium 75-, 90- und 100-MHz	42	
Summe Intel Pentium bis 1995		***56***

Der Pentium Fehler

The New York Times
ON THE WEB

November 1994



4195835/3145727=1,33373906 ??

oder

4195835/3145727=1.3338204491362

- Schaden 450 Mio. US-$

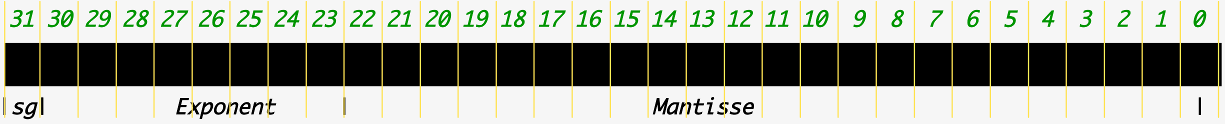
Der Pentium Fehler

- ›Because we had been marketing the Pentium brand heavily, there was a bigger brand awareness,‹ says Richard Dracott, Intel director of marketing. ›We didn't realize how many people would know about it, and some people were outraged when we said it was no big deal.‹
- Intel eventually offered to replace the affected chips, which Dracott says cost the company $450 million.
- To prove that it had learned from its mistake, Intel then started publishing a list of known ›*errata*,‹ or bugs, for all of its chips.

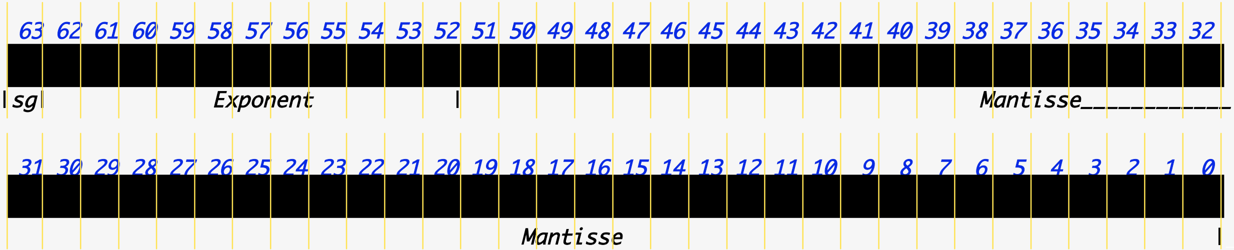
Software: Gleitkomma-Zahlen IEEE 754 Standard

- **Es gibt 32-bit und 64-bit Darstellungen.**
- <Gleitkommazahl≥ := <Mantisse><Exponent>
- <Mantisse> :=
  <Dezimalzahl>|
  <Dezimalzahl>.|.<Dezimalzahl>|
  <Dezimalzahl>.<Dezimalzahl>
- <Exponent> :=
  e<Ganzzahl> |
  E<Ganzzahl>
- float x **umfaßt den Bereich von gerundet $1,5*10^{-39} < x < 1,7*10^{38}$**
- double x **umfaßt den Bereich von gerundet $6*10^{-309} < x < 8*10^{307}$**

Software: Gleitkomma-Zahlen
IEEE 754 Standard



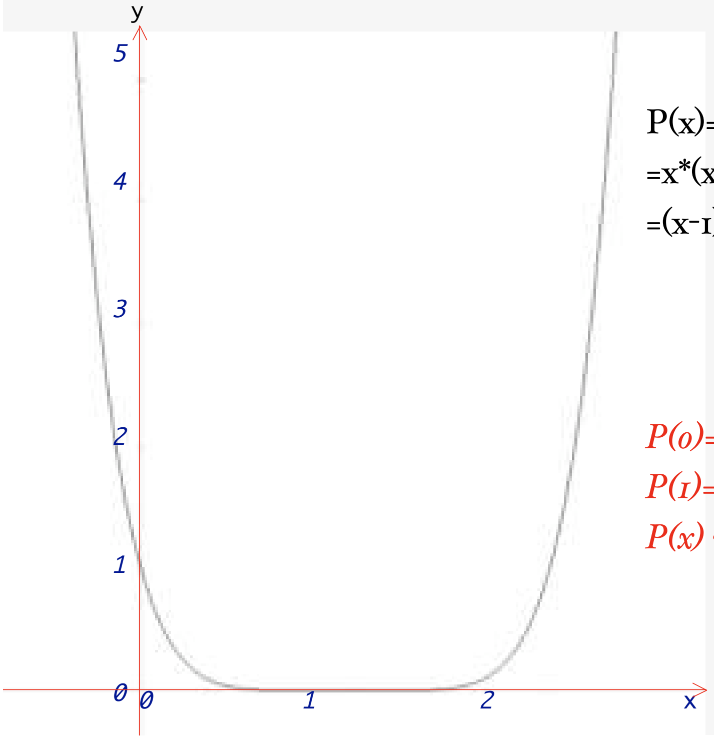
Einfache Genauigkeit



Doppelte Genauigkeit

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

Ein Polynom sechsten Grades P(*x*)



$P(x)=x^6-6*x^5+15*x^4-20*x^3+15*x^2-6*x+1$

$=x*(x*(x*(x*(x*(x-6)+15)-20)+15)-6)+1$

$=(x-1)^6$

P(0)= 1

P(1)= 1-6+15-20+15-6+1 = 0

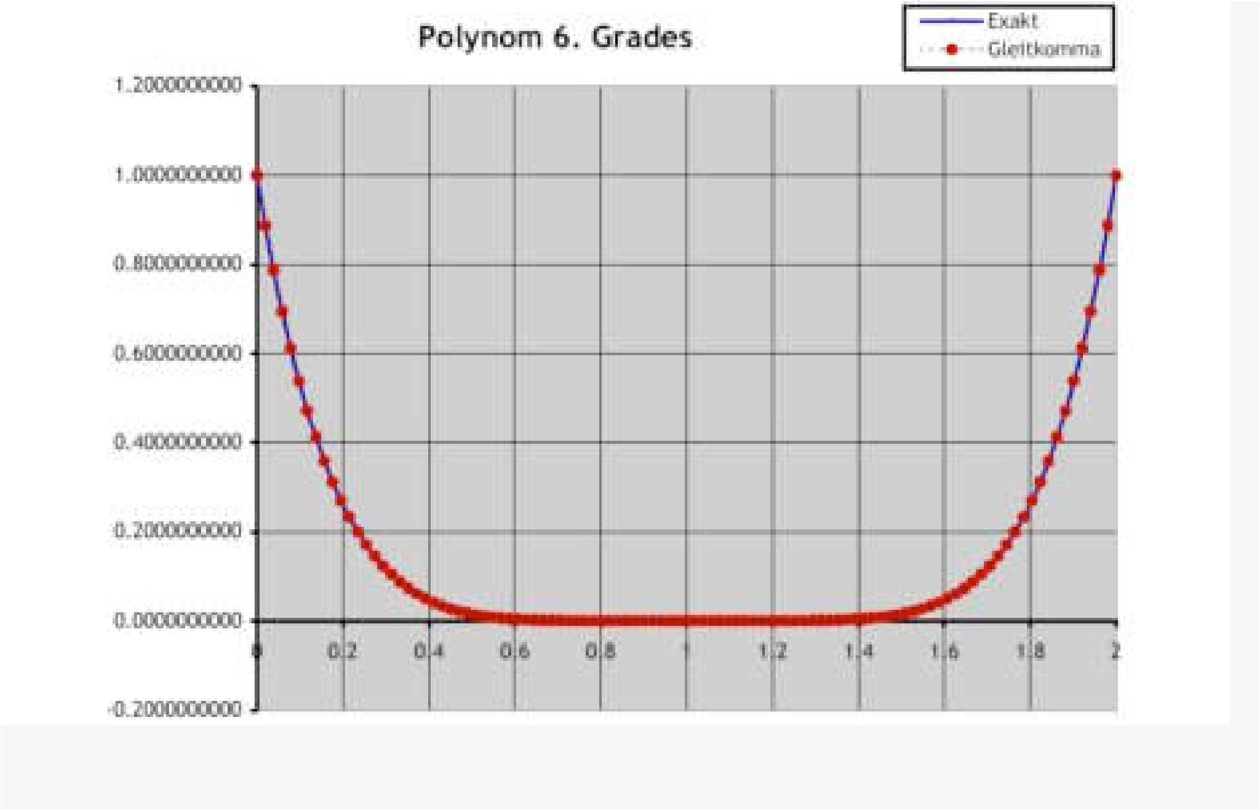
P(x) wird niemals negativ.

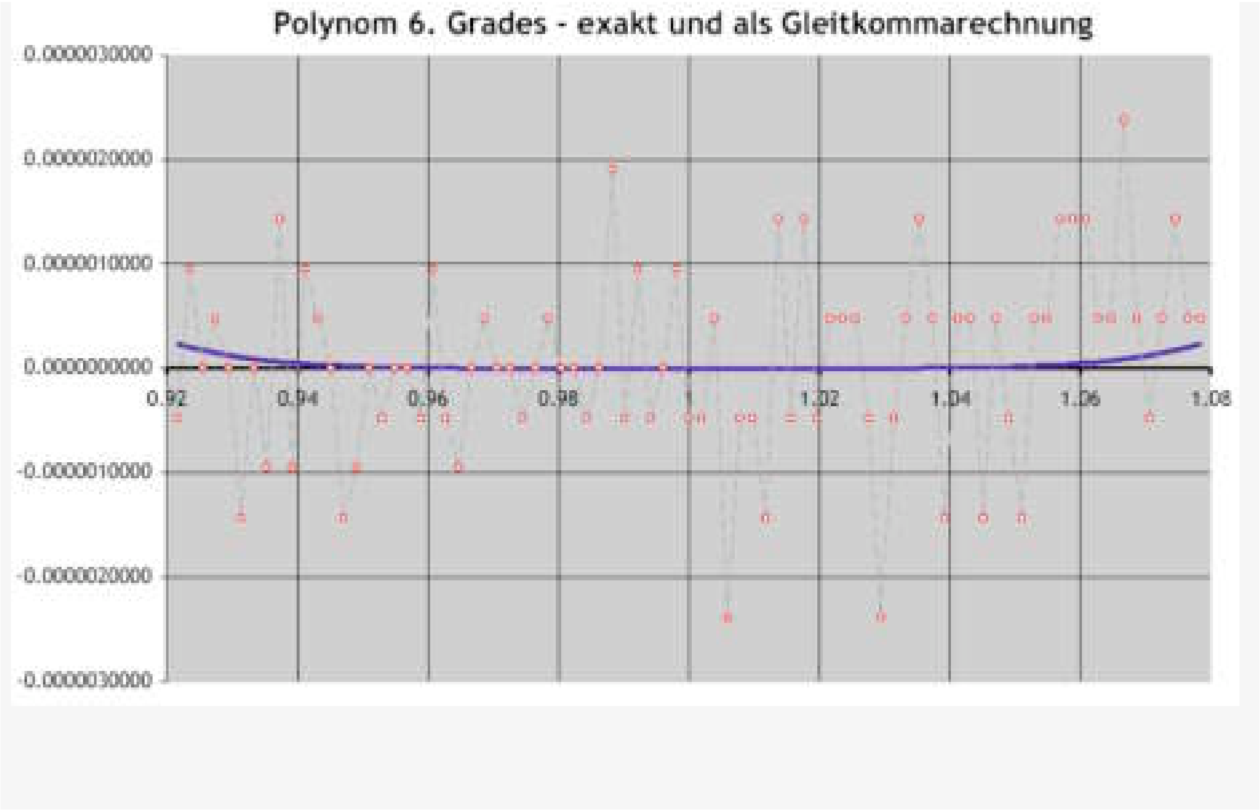
```
public class polynomial {
  static float Polynom(float x) {
       float x2,x3,x4,x5,x6,P;
  x2=x*x;
  x3=x2*x;
  x4=x3*x;
  x5=x4*x;
  x6=x5*x;
  P=x6-6*x5+15*x4-20*x3+15*x2-6*x+1;
  return P;}

public static void main(String args[ ]) {
  int i,n=54;float delta,x;
  x=0.5f; delta=1/(float)n;
  for (i=0;i<=n;i++) {
  System.out.println("P(" + x +")="+ Polynom(x));
  x=x+delta;}
            }
}
```

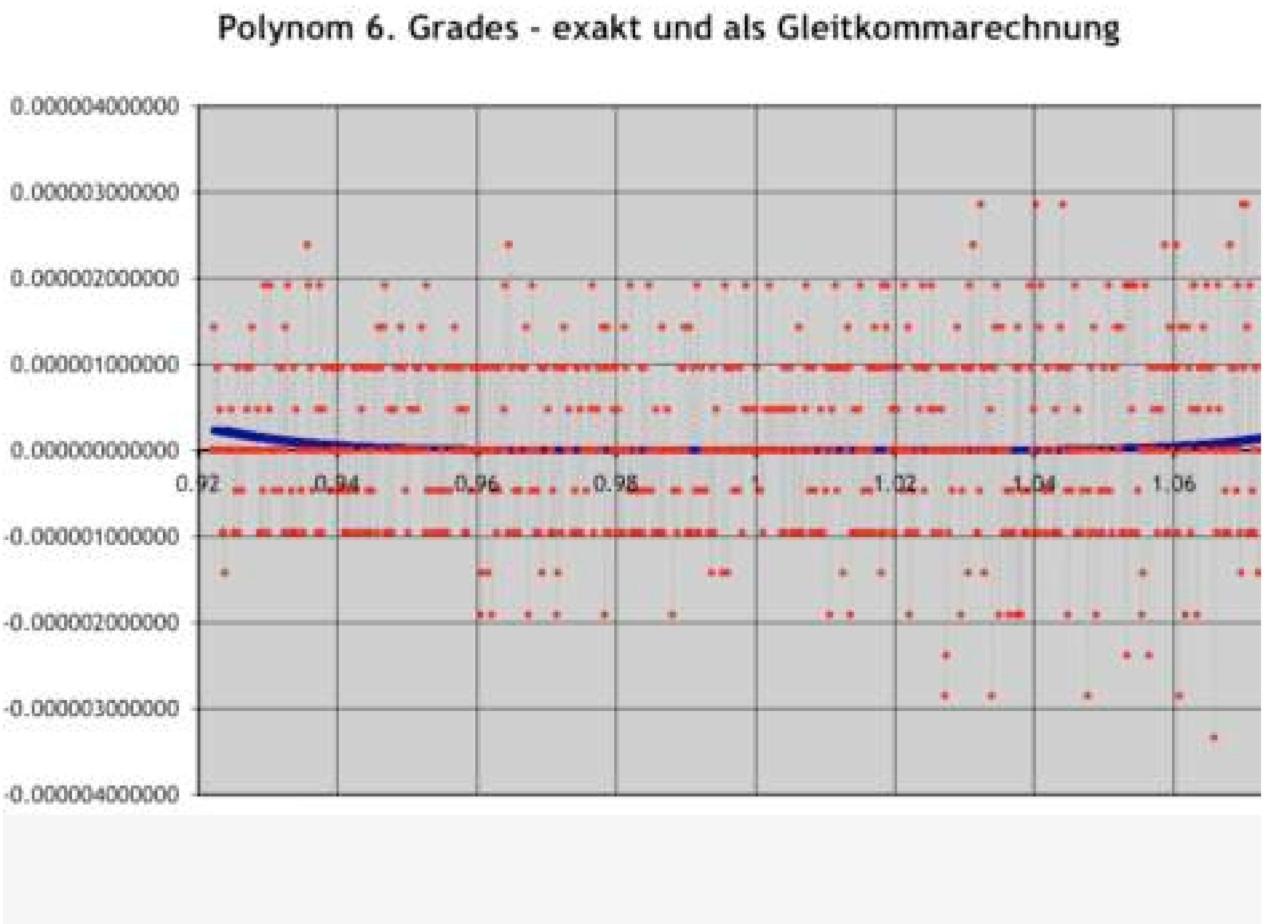
```
P(0.5)=0.015625
P(0.5185185)=0.012458801
P(0.537037)=0.009846449
P(0.5555555)=0.007707596
P(0.57407403)=0.0059702396
P(0.59259254)=0.0045723915
P(0.61111104)=0.0034589767
P(0.62962955)=0.0025815964
P(0.64814806)=0.001897335
P(0.66666657)=0.0013718605
P(0.6851851)=9.7322464E-4
P(0.7037036)=6.766319E-4
P(0.7222221)=4.6014786E-4
P(0.7407406)=3.0374527E-4
P(0.7592591)=1.9550323E-4
P(0.7777776)=1.206398E-4
P(0.7962961)=7.05719E-5
P(0.8148146)=4.005432E-5
P(0.83333313)=2.1457672E-5
P(0.85185164)=1.1444092E-5
P(0.87037015)=3.33786E-6
P(0.88888866)=1.9073486E-6
P(0.90740716)=0.0
P(0.9259257)=4.7683716E-7
P(0.9444442)=9.536743E-7
P(0.9629627)=-1.4305115E-6
P(0.9814812)=-9.536743E-7
```

```
P(0.9999997)=-9.536743E-7
P(1.0185182)=-1.4305115E-6
P(1.0370368)=9.536743E-7
P(1.0555553)=-9.536743E-7
P(1.0740739)=-9.536743E-7
P(1.0925925)=2.3841858E-6
P(1.111111)=9.536743E-7
P(1.1296296)=2.861023E-6
P(1.1481482)=9.536743E-6
P(1.1666667)=2.4318695E-5
P(1.1851853)=4.1007996E-5
P(1.2037039)=7.43866E-5
P(1.2222224)=1.2111664E-4
P(1.240741)=1.9693375E-4
P(1.2592596)=3.0326843E-4
P(1.2777781)=4.606247E-4
P(1.2962967)=6.7329407E-4
P(1.3148153)=9.7227097E-4
P(1.3333338)=0.0013742447
P(1.3518524)=0.0018959045
P(1.370371)=0.0025815964
P(1.3888896)=0.0034561157
P(1.4074081)=0.004573822
P(1.4259267)=0.00596714
P(1.4444453)=0.007709503
P(1.4629638)=0.009849548
P(1.4814824)=0.012465477
P(1.500001)=0.015623093
```

P(x) wird niemals negativ.


Polynom 6. Grades
Exakt
Gleitkomma
1.2000000000
1.0000000000
0.8000000000
0.6000000000
0.4000000000
0.2000000000
0.0000000000
-0.2000000000
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Polynom 6. Grades - exakt und als Gleitkommarechnung
0.0000030000
0.0000020000
0.0000010000
0.0000000000
-0.0000010000
-0.0000020000
-0.0000030000
0.92
0.94
0.96
0.98
1
1.02
1.04
1.06
1.08



Rundungsfehler

1/2+1/2=1
1/3+1/3+1/3=1
1/4+1/4+1/4+1/4=1
1/5+1/5+1/5+1/5+1/5=1
…
0,5+0,5=1
0,25+0,25+0,25+0,25=1
0,2+0,2+0,2+0,2+0,2=1
…

jedoch

0.3+0.3+0.3=0.9
0.33+0.33+0.33=0.99
0.333+0.333+0.333=0.999
…

Brüche zur Basis 2

1/2 = 0,1
1/3 = 0,1010101010101010…
1/4 = 0,01
1/5 = 0,0011001100110011…
1/6 = 0,0010101010101010…
1/7 = 0,0010010010010010…
1/8 = 0,001
1/9 = 0,0001110001110001…
1/10= 0,0001100110011001…
1/11= 0,0001011101000101…
1/12= 0,0001010101010101…
1/13= 0,0001001110110001…
1/14= 0,0001001001001001…
1/15= 0,0001000100010001…
1/16= 0,0001
…

bc

```
scale=32;
ibase=obase=2
n=1000;
h=1/n
for(i=1;i<=n;i++){
h=(n+1)*h-1
print h,"\n"
}
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
Beispiel n=5

h(0) = 1/5;
h(1) = 6*h(0) - 1 = 6/5 - 1 = 1/5
h(2) = 6*h(1) - 1 = 1/5
h(3) = 6*h(2) - 1 = 1/5
h(4) = 6*h(3) - 1 = 1/5
h(5) = 6*h(4) - 1 = 1/5

...
n*h(5) = 1    (oder allgemein n*h(i)=1)
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
n=8 als Dezimalbruch              n=8 als Binärbruch

h(0) = 0,125;                     h(0) = 0,001;
h(1) = 9*h(0)-1 = 0,125           h(1) = 1000*h(0)-1 = 0,001
h(2) = 9*h(1)-1 = 0,125           h(2) = 1000*h(1)-1 = 0,001
h(3) = 9*h(2)-1 = 0,125           h(3) = 1000*h(2)-1 = 0,001
h(4) = 9*h(3)-1 = 0,125           h(4) = 1000*h(3)-1 = 0,001
h(5) = 9*h(4)-1 = 0,125           h(5) = 1000*h(4)-1 = 0,001
h(6) = 9*h(5)-1 = 0,125           h(6) = 1000*h(5)-1 = 0,001
h(7) = 9*h(6)-1 = 0,125           h(7) = 1000*h(6)-1 = 0,001
h(8) = 9*h(7)-1 = 0,125           h(8) = 1000*h(7)-1 = 0,001
...                               ...
n*h(8) = 1                        n*h(8) = 1
```

Java

```
static float Funktion(int n) {
 float h,F; long i;
 h=1/(float)n;
   /*Schleifeninvariante h=1/n */
 for (i=1;i<=n;i++){h=(float)(n+1)*h-1;}
   /*Schleifeninvariante h=1/n */
 F=n*h;
 return F;
   /* F=n*1/n, also F=1 */
}
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
n=1,  f(n)=1.0
n=2,  f(n)=1.0
n=3,  f(n)=1.0000019
n=4,  f(n)=1.0
n=5,  f(n)=1.000309
n=6,  f(n)=1.0080142
n=7,  f(n)=1.09375
n=8,  f(n)=1.0
n=9,  f(n)=48.683716
n=10, f(n)=563.1785
n=11, f(n)=22144.375
n=12, f(n)=854569.75
n=13, f(n)=4.0550648E7
n=14, f(n)=2.32004582E9
n=15, f(n)=6.0129542E10
n=16, f(n)=1.0
```

$2{,}3 * 10^9$

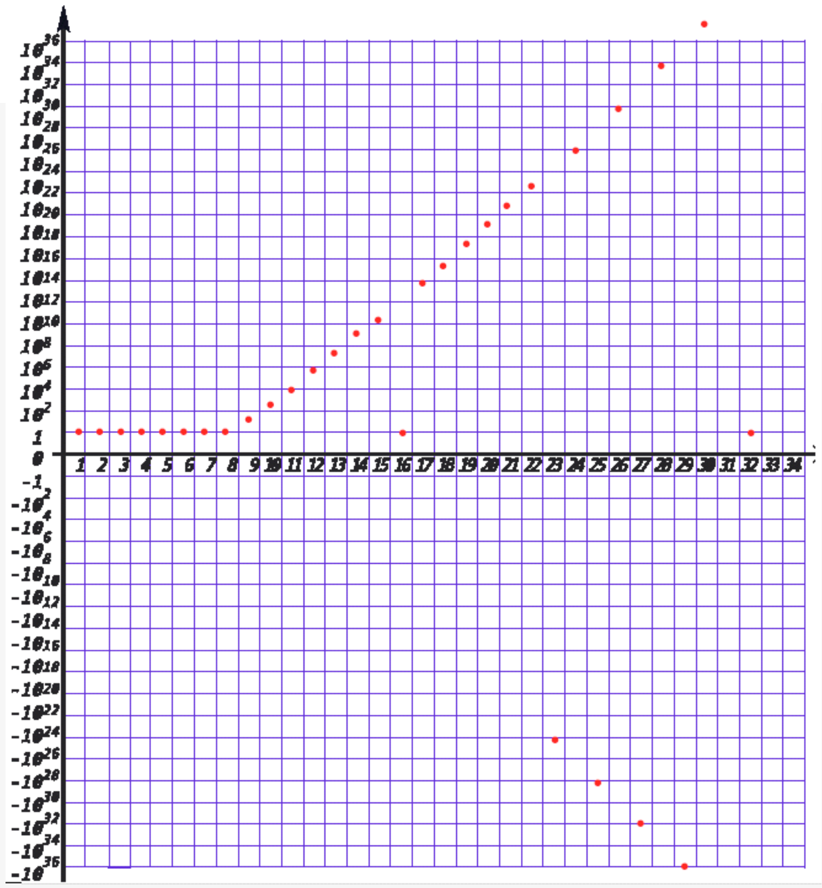
```
    class hFunktion {
    static float Funktion(int n) {
      float h,F; long i;
      h=1/(float)n;
    /*Schleifeninvariante h=1/n */
      for
(i=1;i<=n;i++){h=(float)(n+1)*h-1;}
    /*Schleifeninvariante h=1/n */
      F=n*h;
      return F;
    /* F=n*1/n, also F=1 */
    }

    public static void main(
String[ ] args ) {
        int  i=1;
      for (i=1;i<=34;i++){
      System.out.println("n="+i +",
f(n)="+ Funktion(i));
      } //Ende for
      }  //Ende main
    } //Ende class
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
n=17, f(n)=1.15813768E14
n=18, f(n)=2.61325164E15
n=19, f(n)=1.87499976E17
n=20, f(n)=1.895213E19
n=21, f(n)=8.4093746E20
n=22, f(n)=6.5875305E22
n=23, f(n)=-5.5206144E23
n=24, f(n)=2.7105054E26
n=25, f(n)=-8.684812E27
n=26, f(n)=4.3506622E29
n=27, f(n)=-2.5199321E31
n=28, f(n)=8.7350875E33
n=29, f(n)=-2.7271266E35
n=30, f(n)=4.65824E37
n=31, f(n)=-Infinity
n=32, f(n)=1.0
n=33, f(n)=-Infinity
n=34, f(n)=Infinity
```

$f(n) \in \{-\infty, +\infty\}$

```
    static float Funktion(int n) {
      float h,F; long i;
      h=1/(float)n;
    /*Schleifeninvariante h=1/n */
      for
(i=1;i<=n;i++){h=(float)(n+1)*h-1;}
    /*Schleifeninvariante h=1/n */
      F=n*h;
      return F;
    /* F=n*1/n, also F=1 */
    }

    public static void main(
String[ ] args ) {
        int  i=1;
      for (i=1;i<=34;i++){
      System.out.println("n="+i +",
f(n)="+ Funktion(i));
      } //Ende for
      }  //Ende main
    } //Ende class
```



$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

- Sei n=1. Dann gilt für h = 1/n = 1 und die for-Schleife rechnet max-mal h = 2*1 -1= 1 aus. Für F gilt dann F=1*1=1

- Bei n=2 gilt h = 1/2 und es wird 35-mal h = 3*(1/2)-1 = 1/2 berechnet. Für F gilt F=2*1/2=1.

  Ebenso für alle n, die Zweierpotenzen sind, so daß 1/n noch im Rechner darstellbar ist.

- Bei n=3 kann ein Binärrechner 1/3 nicht exakt mit endlich vielen Stellen darstellen. So liefert das Computerprogramm mit jeder Iteration der Schleife eine kleine Abweichung von der eigentlich geforderten „Invariante“ h=1/3. h ist also nicht invariant, sondern driftet von der besten Annäherung 0.3333333... in dezimaler Schreibweise bzw. 0.0101010101010101010101010... in binärer Schreibweise weg.

- Bei n=4 klappt das Rechnen wieder wie bei jeder Zweierpotenz, während 1/5 binär wiederum nur eine Annäherung, nämlich 0.0011001100110011001100110011001... erzeugt (anders als im dezimalen Fall, wo 1/5=0.2 gilt !).

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
n=5 als Dezimalbruch          n=5 als Binärbruch

h(0) = 0,2;                   h(0) = 0,00110011;
h(1) = 6*h(0)-1 = 0,2         h(1) = 110*h(0)-1 = 0,00110010
h(2) = 6*h(1)-1 = 0,2         h(2) = 110*h(1)-1 = 0,00101100
h(3) = 6*h(2)-1 = 0,2         h(3) = 110*h(2)-1 = 0,00001000
h(4) = 6*h(3)-1 = 0,2         h(4) = 110*h(3)-1 =-0,11010000
h(5) = 6*h(4)-1 = 0,2         h(5) = 110*h(4)-1 =-101,111000

...                           ...
n*h(5) = 1                    n*h(5) =-11101.01100 (=29,375)
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
scale=1000
ibase=2
obase=2
n=1111;
h=1/n
for(i=1;i<=n;i++){h=(n+1)*h-1
print h,"\t",i,"\n"
}
.000100010001000100001111010                                      1
.000100010001000011110100011                                      10
.000100010000111101000110111                                      11
.000100001111010001101110111                                      100
.000011110100011011101111110                                      101
-.000010111001000100000010110                                     110
-1.101110010001000000101100011                                    111
-11100.100100010000001011000111001                                1000
-111001010.000100000010110001110010111                            1001
-1110010100010.000000101100011100101111101                        1010
-11100101000100001.001011000111001011111010100                    1011
-111001010001000010011.110001110010111110101001111                1100
-1110010100010000100111101.011100101111101010011110011            1101
-11100101000100001001111011000.001011111010100111100110110        1110
-111001010001000010011110110000011.111110101001111001101100001    1111
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
scale=10000
ibase=2
obase=2
n=1111;
h=1/n
for(i=1;i<=n;i++){h=(n+1)*h-1
print h,"\t",i,"\n"
}
.000100010001000100010001000100010001000100010000110001            1
.000100010001000100010001000100010001000100001100010000            10
.000100010001000100010001000100010001000011000100001101            11
.000100010001000100010001000100010000110001000011010010            100
.000100010001000100010001000100001100010000110100100010            101
.000100010001000100010001000011000100001101001000101000            110
.000100010001000100010000110001000011010010001010001001            111
.000100010001000100001100010000110100100010100010011011            1000
.000100010001000011000100001101001000101000100110110011            1001
.000100010000110001000011010010001010001001101100111110            1010
.000100001100010000110100100010100010011011001111101000            1011
.000011000100001101001000101000100110110011111010000111            1100
-.001110111100101101110101110110010011000001011110001110           1101
-100.101111001011011101011101100100110000010111100011101000        1110
-1001100.110010110111010111011001001100000101111000111010000011    1111
```

$$\begin{cases} h(0) = 1/n \\ h(i+1) = (n+1) * h(i) - 1 \end{cases}$$

```
scale=1000000
ibase=2
obase=2
n=1111;
h=1/n
for(i=1;i<=n;i++){h=(n+1)*h-1
print h,"\t",i,"\n"
}
```

```
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000010100     1
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100000101000001     10
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010000101000001011 1     11
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000010100000101110111     100
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100000101000001011101111001     101
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010000101000001011101111001010 0     110
.000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010001000100010000101000001011101111001010010 01     111
```

Intervall-Arithmetik

$$x = x_{center} \pm \varepsilon$$

oder mengentheoretisch

$$x \in \left\{ [\min\{x\}\ldots\max\{x\}] \mid x \in Q,\ \min\{x\} \in Q,\ \max\{x\} \in Q,\ \min\{x\} \le \max\{x\} \right\}$$

z.B.

$x \in [0{,}992187\ldots0{,}995312]$

$x \in [0\ldots1]$

$x \in [100\ldots190]$

Intervallarithmetische Regeln

- [a,b] + [c,d] = [a+c,b+d]
- [a,b] − [c,d] = [a−d,b−c]
- [a,b] * [c,d] = [min{a*c,a*d,b*c,b*d},max{a*c,a*d,b*c,b*d}]
- [a,b] / [c,d] = [a,b]*[1/c,1/d], sofern 0 ∈ [c,d]
- [a,b] + [c,d] = [c,d] + [a,b]

Kommutativität

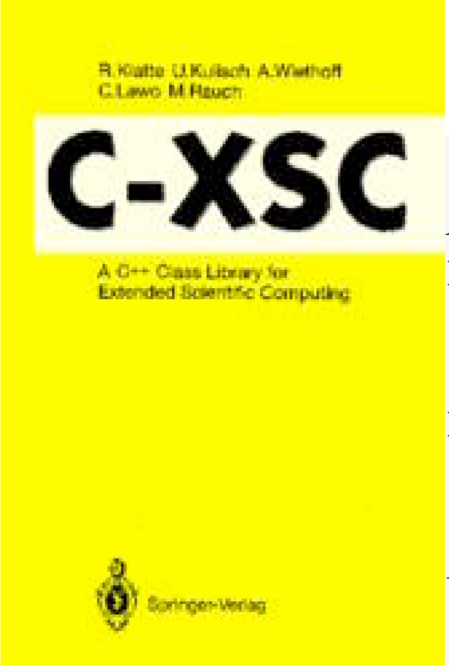
- [a,b] * [c,d] = [c,d] * [a,b]
- [a,b] + ([c,d] + [e,f]) = ([a,b] + [c,d]) + [e,f] =[a,b] + [c,d] + [e,f]

Assoziativität

- [a,b] * ([c,d] * [e,f]) = ([a,b] * [c,d]) * [e,f] =[a,b] * [c,d] * [e,f]

Neutrale Elemente

- [a,b] * 1 = [a,b]
- [a,b] + 0 = [a,b]



R.Klatte U.Kulisch A.Wiethoff
C.Lawo M.Rauch

C-XSC

A C++ Class Library for
Extended Scientific Computing

Springer-Verlag

Es gilt nicht:

Lösbarkeiten

$[a,b] + [c,d] = 0$

$[a,b] * [c,d] = 1$

Beide Gleichungen sind nicht eindeutig lösbar für $a<b$, $c<d$

Im allgemeinen gilt nicht:

$[a,b] - [a,b] = [0,0]$

$[a,b] / [a,b] = [1,1]$

Statt des Distributivgesetzes gilt ein Subdistributivgesetz:

$[a,b] * ([c,d] + [e,f]) \subseteq ([a,b] * [c,d]) + ([a,b] * [e,f])$

Pascal SC, Fortran SC, Calculus, Acrith-XSC, C-XSC

Erweiterung von Fortran, Pascal oder C durch Intervallarithmetik mit 29 intervallarithmetische Operatoren wie z.B.

+, −, *, /, div, mod, and, or,...

sowie

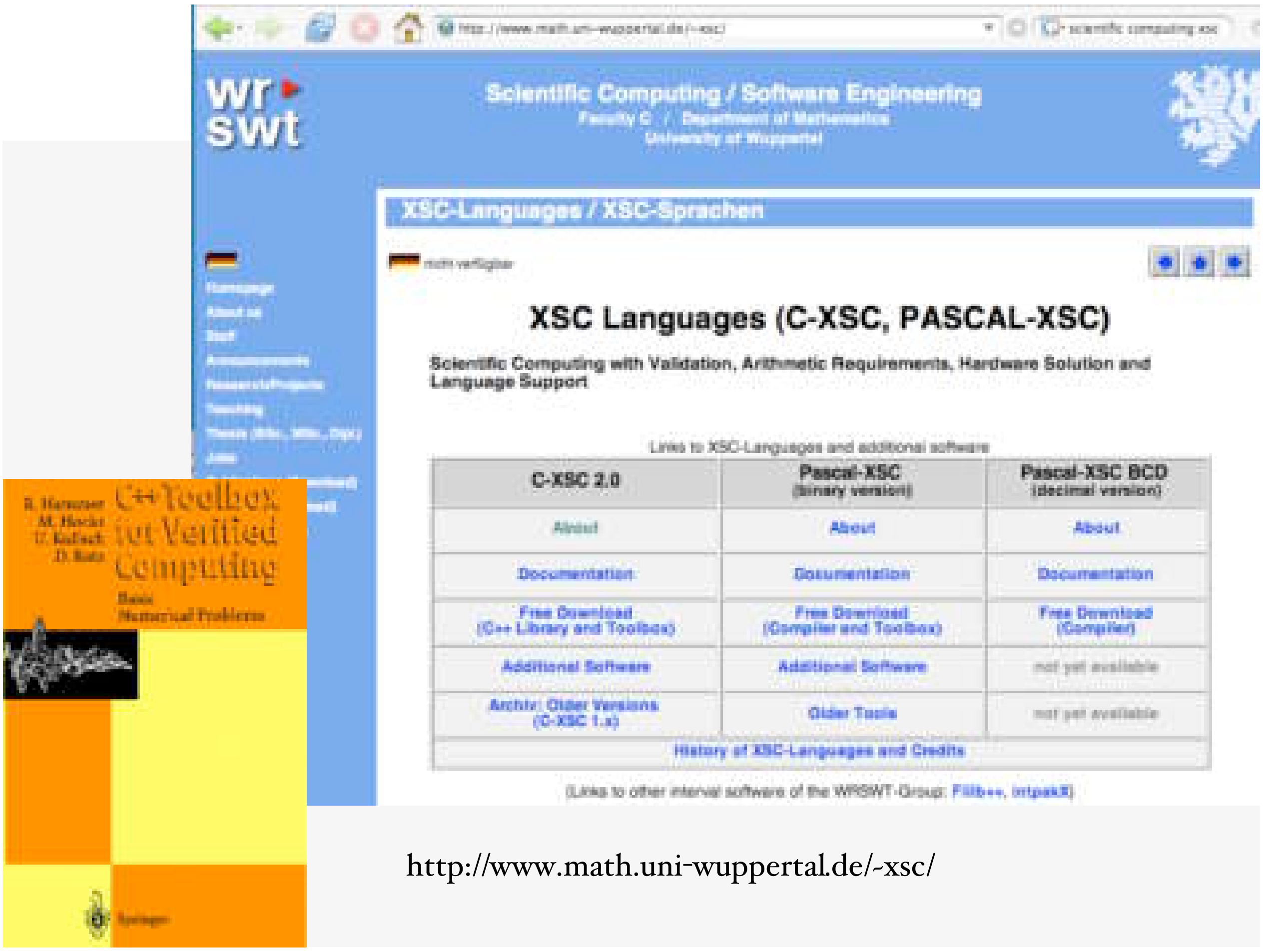
<, >, ≤, ≥, ...

und intervallarithmetische Grundfunktionen wie

sin, cos, exp, ln, arctan, sqrt

Hinzu kommen spezielle Intervalloperatoren z.B. für die Matrixmultiplikation

a*x+b



Scientific Computing / Software Engineering
Faculty C / Department of Mathematics
University of Wuppertal

XSC-Languages / XSC-Sprachen

XSC Languages (C-XSC, PASCAL-XSC)

Scientific Computing with Validation, Arithmetic Requirements, Hardware Solution and Language Support

Links to XSC-Languages and additional software

C-XSC 2.0	Pascal-XSC (binary version)	Pascal-XSC BCD (decimal version)
About	About	About
Documentation	Documentation	Documentation
Free Download (C++ Library and Toolbox)	Free Download (Compiler and Toolbox)	Free Download (Compiler)
Additional Software	Additional Software	not yet available
Archiv: Older Versions (C-XSC 1.x)	Older Tools	not yet available
History of XSC-Languages and Credits		

(Links to other interval software of the WRSWT-Group: Filib++, intpakX)

http://www.math.uni-wuppertal.de/~xsc/

SC Geschichte (*Scientific Computing*)

Institute of Applied Mathematics (Prof. U. Kulisch), University of Karlsruhe:

- 1967: An ALGOL-60 extension implemented on a Zuse Z 23 computer with operators and a number of elementary functions for a new data type interval.
- 1968/69: Implementation of the above language on a more powerful computer, an Electrologica X8.
- 1976: PASCAL-SC, a PASCAL extension implemented on a Z-80 microprocessor with 64 KB main memory (funded by the german company Nixdorf). The programming convenience of PASCAL-SC allowed a small group of collaborators to implement a large number of problem solving routines with automatic result verification within a few months.
- 1980: PASCAL-SC with a large number of problem solving routines was exhibited at the Hannover fair.
- 1980/89: ACRITH-XSC, a FORTRAN 77 extension for the /370 architecture was developed and implemented in cooperation with IBM.
- 1983: IBM shipped the first edition of the ACRITH library.
- 1986: ARITHMOS for BS 2000 (with Siemens)
- 1990: IBM shipped ACRITH-XSC
- 1990/91: Development of a new Runtime System (RTS) for PASCAL-XSC in C
- 1991: PASCAL-XSC shipped. The PASCAL-XSC system compiles a given PASCAL-XSC source code into C code which is passed to a C-Compiler.
- 1992: C++ class library C-XSC shipped, available for many computers with C++ compiler translating der AT&T language standard 2.0
- 1993: "Numerical Toolbox for Verified Computing" in PASCAL-XSC is published by Springer-Verlag.
- 1994: "C++ Toolbox for Verified Computing" in C-XSC is published by Springer-Verlag
- 1996: Begin of the implementation of Fortran-XSC (TU Dresden, Prof. Walter)
- 1997/98: Oberon-XSC (Universität Karlsruhe, Dr. P. Januschke; ETH Zürich, Prof. Gutknecht)
- 1997: XSC General Public License, *all XSC software has been available FREE OF CHARGE since 1997.*

WRSWT-Group (Prof. W. Krämer), University of Wuppertal:

- 1999/2001: Redesign of C-XSC (collaboration of the Institute of Applied Mathematics (Prof. Kulisch), University of Karlsruhe and the WRSWT-Group (Prof. Krämer), University of Wuppertal)
- 2000/02: New Web-Presentation of XSC-Languages and Additional Software
- 2002: First Beta Release C-XSC 2.0

Bewiesenermaßen sichere Hardware?

Das VIPER-Project

Verifiable Integrated Processor
for Enhanced Reliability

Avra J. Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1987.

Das VIPER-Projekt -

verifiable integrated processor for enhanced reliability

- VIPER is a 32-bit microprocessor architecture designed by the Royal Signals and Radar Establishment (RSRE) in Malvern, England.

Logische Verifikation eines Mikroprozessors 1995

- **The reader may be surprised that the majority of formally verified microprocessors have been part of University projects, but this is exactly the problem with hardware verification:**
- The industry is reluctant to adapt these methods. ...

Das VIPER-Projekt -
verifiable integrated processor for enhanced reliability

- **Various difficulties arise at this point. Firstly, because the theorem provers use a different language than the usual description language errors may occur in translating into the formal language. Secondly, most of the current theorem provers/checkers are** complex **to use,** slow **and do not have a standard language.** Engineers require a strong mathematical background to use them**. …**
- **In spite of new opportunities offered by hardware verification in industry, there are** limitations of formal proof in hardware verification**.**
- **Firstly, a proof does not show that the circuit will behave as intended by the designer, it only shows that it behaves in accordance to a possibly inaccurate specification. Or more generally expressed, as verifiers, designers and manufacturers are all working with models, inaccuracy in the models can lead to errors not detectable by a formal proof.**
- **Secondly,** the proof does not consider extra-logical factors, such as the reset switch being pressed manually, **for example. To ensure the safety of a critical system factors as staff training and performance of mechanical parts of the system have to be considered.**
- **Finally, manufacturing errors can always occur**. **Therefore, one has to be careful not to neglect these factors when designing a safety critical system with a verified processor.**

Murphys Law

- Was schief gehen kann, geht schief…

oder

Murphys Law

- Was schief gehen kann, geht schief…

oder

- Was nicht schief gehen kann, geht schief ?

Murphys Law

Capt. Edward A. Murphy,
Air Force Project MX981