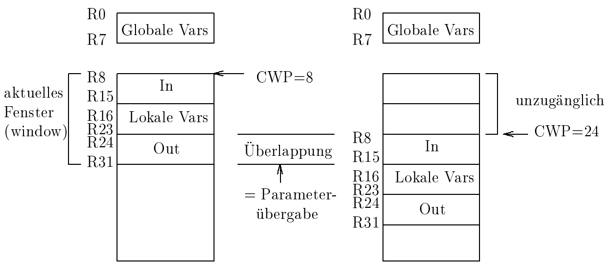
1.2.2.6 Die SPARC-CPU im Inneren

Die Register des SPARC V8-Prozessors (SUPER-SPARC) sind durch verschiedene Merkmale gekennzeichnet:

1. Registerstack



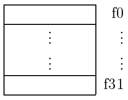
CWP steht für *current window pointer*. Die SPARCclassic besitzt z. B. acht solcher Fensterbereiche. Auf die Register greift man mit *logischen* Namen zu:

`%g0..%g7`	global vars
`%i0..%i7`	in
`%l0..%l7`	local vars
`%o0..%o7`	out

mit `%g0`≡`0` (fest), `%o6=%sp` (stack pointer), `%o7` *return adress* für Unterprogramm-Rücksprung.

2. $\boxed{}$ Y für Multiplikation und Division.

3. Gleikommaregister:



Je 32Bit-Gleitkommaregister für 16 *double precision* oder 8 *quad precision* Gleitkommazahlen.

4. State-Register sind unter anderem PSR *processor state register*, FSR *floating point state register*, PC *program counter*, nPC *next program counter*, WIM *window invalid maske*, TBR *trap base register*, IU-deferred trap queue, FP-deferred trap queue, ...

Befehle und Ausführungszeiten bei SPARC(RISC)-ALU's

LDSB	load signed byte sign extended
LDSH	load signed halfword sign extended
LDUB	load unsigned byte zero filled
LDUH	load unsigned halfword zero filled
LD	load word
LDD	load doubleword
LDF	load floating-point
LDDF	load double floating-point
LDFSR	load floating-point state register
STB	store byte
⋮	
LDSTUB	atomic load/store unsigned byte
SWAP	swap register with memory
SETHI	set 22 bit of register
NOP	no operation
AND	
ANDcc	... and modify icc (= integer condition codes) of PSR
OR	
ORN	A or not B
XOR	
XNOR	exclusive NOR = XOR not
SLL	shift left logical
SRL	shift right logical (fill zero)
SRA	shift right arithmetic (fill most significant bit)
ADD	A+B
ADDcc	A+B and modify icc
ADDX	add with carry = A + B + c
⋮	
SUB	subtract
⋮	
MULSss	multiply step (zusammen mit Register Y für 64Bit-Ergebnis)

UMUL	unsigned integer multiply	
SMUL	signed integer multiply	
UDIV	unsigned integer division	
SDIV	signed integer division	
SAVE	register stack	
RESTORE	register stack	
Bicc	branch on integer condition codes, z.B.	
	BA	branch always
	BN	branch never
	BNE	branch not equal
FBfcc	branch on floating point condition codes	
CALL	call and link	
JMPL	jump and link	
RETT	return from Trap	
Ticc	Trap on icc, z.B. TA, TN, ...	
⋮	(read/write state ... registers)	
UNIMP	unimplemented	
FLUSH	flush instruction memory	
FPop	(rounding direction, wahlweise *to nearrest*, *towards 0*, *towards* $-\infty$, *towards* $+\infty$)	
	FiTOs	integer to single
	FiTOd	integer to double
	FiTOq	integer to quad
	FsTOi	
	FdTOi	
	FqTOi	
	FsTOd	
	FsToq	
	⋮	
	FMOVs	
	FNEGs	
	FABSs	
	FSQRTs	
	FADDs	
	FADDd	
	FADDq	
	FSUBs	
	FMULs	
	FsMULd	entspricht $s \cdot s \to d$

FdMULq	entspricht $d \cdot d \to q$
FDIVs	
FCMPs	
FCMPEs	compare and exception if unordered
⋮	

Ein Beispiel-PASCAL Programm ...

```
program Tabelle(output);
    var
        i : integer ;

    function f (x : real) : real;
    const
        pi = 3.141592653589793 ;
    var
        arg : real;
    begin
        arg := 2.0 * pi * x + 5.0;
        f    := sin(arg);
    end;

begin
        writeln("f(arg)");
        writeln("——–");
        for i := 0 to 20 do
                writeln(i/10.0,f(i/10.0));
end.
```

... und was daraus in SPARC-Maschinenbefehlen wird (pc -s):

```
        .section   ".data"
        .align 4
        .common i,4,4
        .common PROGFRAME,4,4
        .align 4
.L15:
        .ascii     "%21.14e"
        .byte 0
        .section   ".text"
        .proc 010
        .global f
```

```
        .align 4
f:
        !#PROLOGUE# 0
        sethi   %hi(.LF2),%g1
        add     %g1,%lo(.LF2),%g1
        save    %sp,%g1,%sp
        !#PROLOGUE# 1
        st      %i0,[%fp+0x44]
        st      %i1,[%fp+0x48]
.L3:
.L4:
        .stabs "SC3.0.1 07 Nov 1994 Pascal 3.0.3 patch 101912-01 NN ",0x30,0,0xd,0x0
        .ident  "SC3.0.1 07 Nov 1994 Pascal 3.0.3 patch 101912-01 "
        .stabs  "Tabelle.p ",0x30,0,0x1,0x1
        .file   "Tabelle.p "
        .xstabs ".stab.index ", "V=2.0 ",0x3c,0,0,0x33b3734e
!       arg := 2.0 * pi * x + 5.0;
        sethi   %hi(.L2000000),%o0
        ldd     [%o0+%lo(.L2000000)],%f0
        ld2     [%fp+0x44],%f2
        fmuld   %f0,%f2,%f0
        sethi   %hi(.L2000001),%o1
        ldd [%o1+%lo(.L2000001)],%f4
        faddd   %f0,%f4,%f6
        std     %f6,[%fp+-0x20]
!       f := sin(arg);
        ldd     [%fp+-0x20],%o0
        call    __sin,2
        nop
        std     %f0,[%fp+-0x18]
.L6:
.L7:
        ldd     [%fp+-0x18],%f0
        b       .LE2
        nop
.LE2:
        ret
        restore
        .optim "-O Q R S "
        .LF2 = -128
        .LP2 = 96
        .LST2 = 96
```



```
        .LT2 = 96
        .type f,#function
        .size f,.-f
        .section ".rodata "
        .align 8
.L2000000: .word   0x401921fb,0x54442d18
        .align 8
.L2000001: .word   0x40140000,0x0
        .section ".text "
        .proc   022
        .global program

        .align 4
program:
        !#PROLOGUE# 0
        sethi   %hi(.LF1),%g1
        add     %g1,%lo(.LF1),%g1
        save    %sp,%g1,%sp
        !#PROLOGUE# 1
.L8:
.L9:
        .xstabs "'.stab.index ", "Tabelle ",0x2a,0,0,0
        call    __PC0__tb_init,0
        nop
        sethi   %hi(PROGFRAME),%o2
        st      %fp,[%o2+%lo(PROGFRAME)]
!       writeln( "f(arg) ");
        set     output,%o0
        call    __PC0__UNIT,1
        nop
        st      %o0,[%fp+-0x8]
        ld      [%fp+-0x8],%o3
        ld      [%o3+0xc],%o1
        mov     0xa,%o0
        call    fputc,2
        nop
!       writeln( "—— ");
        set     output,%o0
        call    __PC0__UNIT,1
        nop
        st      %o0,[%fp+-0x8]
        ld      [%fp+-0x8],%o4
```



```
        ld      [%o4+0xc],%o1
        mov     0xa,%o0
        call    fputc,2
        nop
!       for i := 0 to 20 do
        st      %g0,[%fp+-0x10]
.L12:
        ld      [%fp+-0x10],%o5
        sethi   %hi(i),%o7
        st      %o5,[%o7+%lo(i)]
!       writeln(i/10.0, f(i/10.0));
        set     output,%o0
        call    __PC0__UNIT,1
        nop
        st      %o0,[%fp+-0x8]
        ld      [%fp+-0x8],%l0
        ld      [%l0+0xc],%o0
        sethi   %hi(.L2000002),%l1
        ldd     [%l1+%lo(.L2000002)],%f8
        ld      [%fp+-0x10],%f10
        fitod   %f10,%f12
        fdivd   %f12,%f8,%f12
        st      %f12,[%sp+.LP1]
        ld      [%sp+.LP1],%o2
        st      %f13,[%sp+.LP1]
        ld      [%sp+.LP1],%o3
        set     .L15,%o1
        call    fprintf,4
        nop
        sethi   %hi(.L2000002),%l2
        ldd     [%l2+%lo(.L2000002)],%f14
        ld      [%fp+-0x10],%f16
        fitod   %f16,%f18
        fdivd   %f18,%f14,%f18
        st      %f18,[%sp+.LP1]
        ld      [%sp+.LP1],%o0
        st      %f19,[%sp+.LP1]
        ld      [%sp+.LP1],%o1
        call    f,2
        nop
        std     %f0,[%sp+.LP1+0x8]
        ld2     [%sp+.LP1+0x8],%o2
```



```
        ld      [%fp+-0x8],%l3
        ld      [%l3+0xc],%o0
        set     .L15,%o1
        call    fprintf,4
        nop
        ld      [%fp+-0x8],%l4
        ld      [%l4+0xc],%o1
        mov     0xa,%o0
        call    fputc,2
        nop
.L13:
        ld      [%fp+-0x10],%l5
        add     %l5,0x1,%l5
        st      %l5,[%fp+-0x10]
        ld      [%fp+-0x10],%l6
        cmp     %l6,0x14
        bg      .L16
        nop
        b       .L12
        nop
.L16:
.L14:
.L11:
.L17:
        b       .LE1
        nop
.LE1:
        ret
        restore
        .optim "'-O Q R S "
        .LF1 = -128
        .LP1 = 96
        .LST1 = 112
        .LT1 = 112
        .type   program,#function
        .size   program,.-program
        .section ".rodata "
        .align 8
.L2000002: .word   0x40240000,0x0
        .section ".data "
        .align 4
```

Anweisung	Fujitsu/LSI #1,#2	Cypress/ Fujitsu #3	MicroSPARC (&B5000)	Fujitsu SPARClite	Super SPARC
# of register windows	7	8	7	8	8
ld (32 bit integer)	2	2	1	1	1
ldd (64 bit integer)	3	3	2	2	1
ld (32 bit float)	2	2	1	1	1
ldd (64 bit double)	3	3	2(1)	2	1
st (32 bit integer)	3	3	2	1	1
std (64 bit integer)	4	4	3	2	1
st (32 bit float)	3	3	2(1)	1	1
std (64 bit double)	4	4	3(2)	2	1
taken branches	1	1	1	1	1
untaken branches	2	1	1	1	1
jmpl and rett	2	2	2	2	1
integer multiply	N/A	N/A	19	?	4
integer divide	N/A	N/A	39	?	18
issue FP operation	2	1	1	N/A	1

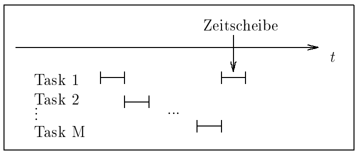
Tabelle 1.23: Anzahl der Register-Fenster und Integer-Zyklen pro Anweisung

Ein Ausflug in Multiprozess- und Multiprozessorsysteme

Prozesse (Tasks)

- Programmeinheiten, die unterbrech- und wiederfortsetzbar sind
- Multitasking:

  Ein Rechner erscheint M Benutzern als M jeweils dedizierte einzelne Rechner. Ein Rechner kann (scheinbar) zugleich mehrere Prozesse bearbeiten.



Beispiele:

Windows: Multitasking, nicht preemptiv

Windows NT / UNIX: preemptives Multitasking

Anweisung	FPC & TI 8847	Weitek 3170 & 3171	Texas 602	BIT B5000	MicroSPARC min	MicroSPARC typ	MicroSPARC max	Super SPARC
fitos	8	10	4	2	5	6	13	1
fitod	8	5	4	2	4	6	13	1
fstoir, fstoi	8	5	4	2	6	6	13	1
fdtoir, fdtoi	8	5	4	2	7	7	14	1
fstod	8	5	4	2	2	2	14	1
fdtos	8	5	4	2	3	3	16	1
fmovs	8	3	4	2	2	2	2	1
fnegs	8	3	4	2	2	2	2	1
fabss	8	3	4	2	2	2	2	1
fsqrts	15	60	22	24	6	37	51	6
fsqrtd	22	118	32	45	6	65	80	10
fadds, fsubs	8	5	4	2	4	4	17	1
faddd, fsubd	8	5	4	2	4	4	17	1
fmuls	8	5	4	3	5	5	25	1
fmuld	9	8	6	4	7	9	32	1
fdvis	13	38	16	14	6	20	38	4
fdivd	18	66	26	24	6	35	56	7
fcmps, fcmpes	8	3	4	2	4	4	15	1
fcmpd, fcmped	8	3	4	2	4	4	15	1

Tabelle 1.24: „floating point“-Zyklen pro Anweisung

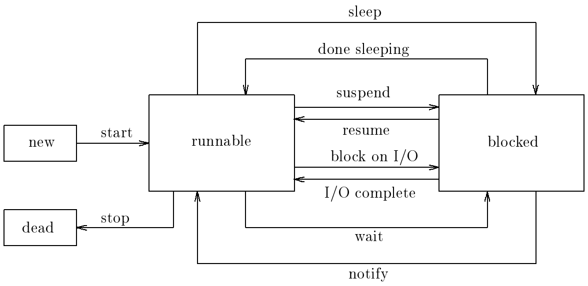
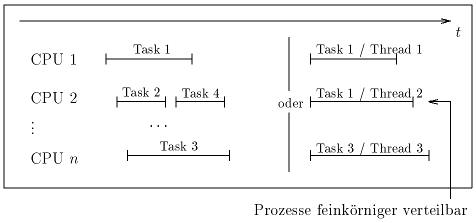


Abbildung 1.24: Thread/Prozeß-Stati

Preemtiv (=*wahlweise vergebbar*) heißt ein Multitasking-Betriebssystem dann, wenn der Betriebssystemkern allein für die Vergabe der Zeitscheiben verantwortlich ist, also nicht auf die kooperative Mitarbeit der Tasks angewiesen ist. Die Tasks werden von *außen* unterbrochen und später wieder fortgesetzt.

Threads – parallele, unabhängig durchführbare Teile eines Prozesses, die deshalb auf Multiprozessorsystemen ($\neq$ Multiprozesssystemen) durch verschiedene CPU's durchgeführt werden können: An der Aufgabe eines Prozesses arbeiten mehrere Prozessoren zugleich.



Wegen des gemeinsam benutzten Speichers ist ein gegenseitiger Ausschluß auf den Zugriff nötig:

⋮
while (Guthaben<Auszahlungsbetrag) do
  wait();
Guthaben := Guthaben - Auszahlungsbetrag;
Auszahle(Auszahlungsbetrag);
notify();
⋮

Threads sind weniger resourcenintensiv als die Erzeugung weiterer Prozesse. Mehr Probleme lassen sich in Threads zerlegen als in mehrere fast unabhängige Prozesse. Dafür sind Thread-Algorithmen schwieriger zu programmieren.

1.2.2.7 Einfügen und Löschen in verketteten Listen

Verkettete Listen (*linked list*) sind dynamische Datenstrukturen, die von multithreaded Anwengungen genutzt werden können. Ein Lock [3] kann dazu benutzt werden, den Zugriff auf die Daten zu synchronisieren. Einfache Locks zum Schutz von großen und häufig von mehreren Threads gemeinsam benutzten Datenstrukturen können jedoch uneffizient sein. Im folgenden Beispiel wird die CAS Anweisung dazu gebraucht, um auf einer verketteten Liste ohne explizite Locks zu operieren. Jedes Listenelement beginnt im Beispiel mit einem Adressenfeld, das entweder die Adresse des nächsten Listenelements enthält oder leer ist. Im Listenkopf ist die Adresse einer Variablen gespeichert, die die Adresse des ersten Listenelements enthält. Bei leeren Listen steht im Kopf eine Null.

```
ListInsert(Head Element) !%i0 = Head addr, %i1 = Element addr
retry:
  ld    [%i0],%l0
  st    %l0, [%i1]
  mov   %i1, %l1
  cas   [%i0],%l0,%l1
  cmp   %l0,%l1
  bne   retry
  nop
ListRemove(Head) !%i0 = Head addr
retry:
  ld    [%i0],%o0
  tst   %o0
```

[3] Lock=Sperrvorrichtung, sorgt dafür, daß einer Resource genau einem Thread zugeteilt wird, bis dieser sie wieder freigibt (gegenseitiger Ausschluß).

```
  be    empty
  nop
  ld    [%o0],%l0
  cas   [%i0],%o0,%l0
  cmp   %o0,%l0
  bne   retry
empty:
  nop
```

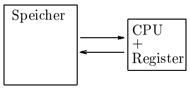
1.2.2.8 Adressierungsarten

- unmittelbare (immediate) Adressierung: MOV AX,0FFFFH, wobei der Bestandteil 0FFFFH die zu übertragenden Daten darstellt
- direkte Adressierung: MOV AX,MYDATA, dabei ist MYDATA die Adresse der zu übertragenden Daten
- (Register-) indirekte Adressierung: MOV AX,[BX], d.h. die Adresse der zu übertragenden Daten steht in BX
- indirekte Adressierung mit 8/16-Bit Displacement:

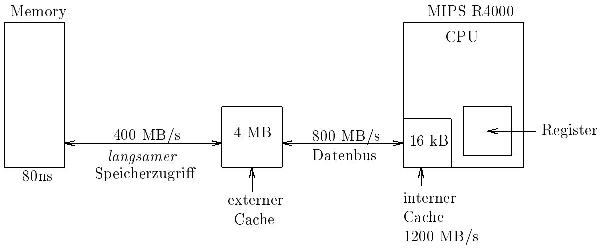
  MOV AX,4[SI]
  MOV AX,MYDATA[SI]
  MOV AX,ELEM[BX][DI]

1.2.2.9 Beschleunigter Datentransfer und beschleunigte Ausführung von Befehlen

Cache:
Es sei an den *von-Neumannschen-Falschenhals* erinnert:



Abhilfe schafft hier der **Cache-Speicher**:



Bemerkungen:

- $80ns \leftrightarrow \frac{1}{80 \cdot 10^{-9}s} = 12.5 \cdot 10^6$ Zugriffe/s
- Bus: 50 MHz und 8 Byte Busbreite führen zu 400 MB/s
- Im SIMM müssen also 32 Bit-Zugriffe *parallel* erfolgen
- Externer und interner Cache bilden einen Puffer, der mit *look ahead* bei einer Trefferquote von $\sim 90 - 95\%$ betrieben wird.
- Es handelt sich um eine *schnelle CPU* mit 100MHz; also ergeben sich bei einem Durchsatz von 1200 MB/s durchschnittlich $\leq$ 12B pro Taktperiode $\leq$ 3 Worte/Eintakt-Befehl (32 Bit Befehl und 64 Bit Daten).

Hoher Durchsatz wird erreicht bei:

- sequentieller Ausführung von Befehlen,
- Zugriff auf ganze Vektoren (Daten),
- Zugriff auf 64/128 Bit-Gleitkommazahlen bei 32 Bit Datenbusbreite,
- durchschnittlich geringem Bedarf an Daten (aus dem Speicher) pro Befehl

Geringer Durchsatz wird erzielt bei:

- wilden Sprüngen, ...
- Benutzung von verketteten Listen mit großen Knoten (CAD-Programme), ...

Bemerkung: Es werden SparcStation10's ohne SuperCache ausgeliefert, um bei speziellen Anwenderprogrammen das verlangsamende Cache-Füllen zu umgehen.

Pipelinebetrieb der CPU selbst:
FPU-Pipeline (baue ALU aus *unabhängigen* Einheiten auf):

		Beispiel: Subtraktion
1)	fetch operands	$1.082 \cdot 10^{12}$, $9.212 \cdot 10^{11}$
2)	adjust exponent	$1.082 \cdot 10^{12}$, $0.9212 \cdot 10^{12}$
3)	execute „-“	$0.1608 \cdot 10^{12}$
4)	normalize result	$1.608 \cdot 10^{11}$

$C_i = A_i + B_i, \forall i = 1..N$

Zyklus	1	2	3	4	5	...
fetch	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	B_5, C_5	
adjust	—	B_1, C_1	B_2, C_2	B_3, C_3	B_4, C_4	
execute	—	—	$B_1 - C_1$	$B_2 - C_2$	$B_3 - C_3$	
normalize	—	—	—	$B_1 - C_1$	$B_2 - C_2$	

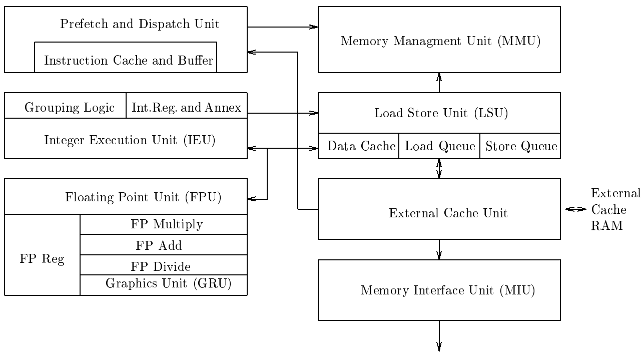
$\searrow$ $\searrow$
$B_1 - C_1$ $B_2 - C_2$

Resümee:

- Ein Einzelbefehl (z.B. Subtraktion) braucht 4 Zyklen Zeit.
- Nach einer Startup-Zeit von 4 Zyklen (Füllen der Pipe) wird dann jedoch pro Zyklus ein Endergebnis geliefert. (Das ist allerdings nur eine Komponente des Ergebnis-Vektors.)
- **Problem:** „Datenabhängigkeit“

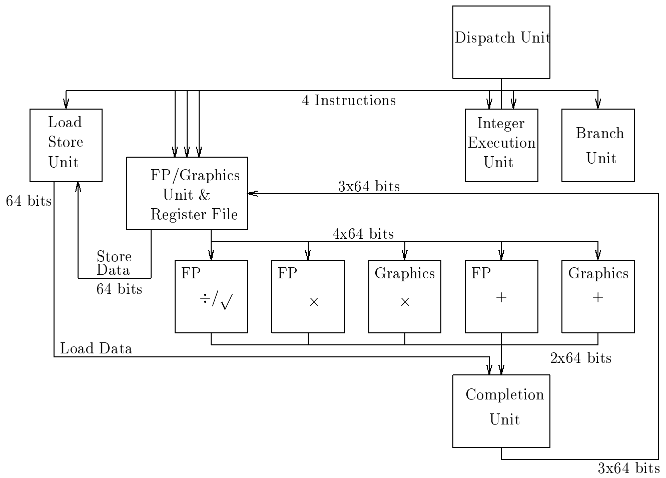
Ein Wert ist noch in der Pipe, obwohl er schon als neues Argument benötigt wird.

Übersicht über die 1996 marktreif gewordenen SPARC-Technologie UltraSPARCI



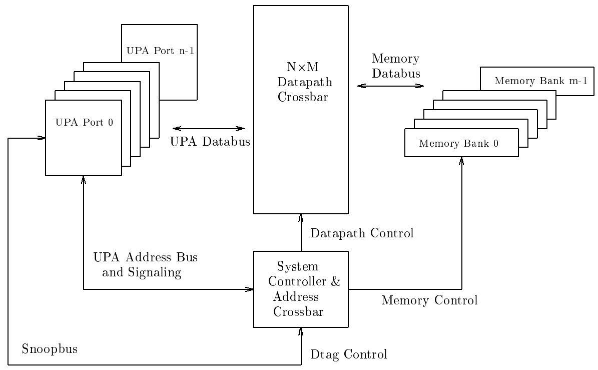
- IU-Pipeline (9 Phasen): fetch-decode-group-execute-cache access-load miss-integer pipe wait- trap resolution-writeback-
- 2 ALU's in IEU (= interger execute unit)
- FP-pipeline
- getrennte FP-execution units

Operation	Throughput (Cycles)	Latency (Cycles)
Add (Single Precision)	1	3
Add (Double Precision)	1	3
Multiply (Single Precision)	1	3
Multiply (Double Precision)	1	3
Divide (Single Precision)	12	12
Divide (Double Precision)	22	22
Square Root (Single Precision)	12	12
Square Root (Double Precision)	22	22



Die *Ultra Port Architekture* arbeitet statt mit einem Bus-System mit *packet switched* und *crossbar*-Datentransfer:

Ultra Port Architecture



1.2.2.10 Festkommazahlen

Festkommazahlen besitzen immer eine feste Anzahl von *Nachkommaziffern*, z.B. 5 Vor- und 3 Nachkommastellen wie bei kg:

$$\mathcal{F}_3^5 := \left\{ (-1)^s \sum_{i=-3}^{5} a_i \cdot 10^i : a_i \in \{0,..9\}; s \in \{0,1\} \right\}$$

Veranschaulichung am Zahlenstrahl: $\mathcal{F}_3^5$

Eigenschaften:

- fester Gitterabstand zwischen je zwei benachbarten Zahlen (im Beispiel 10^{-3})
- Addition bis auf Über-/Unterlauf exakt ausführbar
- Multiplikation mit ganzen Zahlen bis auf Über-/Unterlauf exakt ausführbar
- Multiplikation von echt gebrochenen Zahlen mit Rundungsfehlern behaftet

Überlegungen zeigen, daß z.B. die Realisierung von DM-Beträgen mit `real`-Variablen ungünstig ist, denn

$$1\text{Pf} = \frac{1}{2^2 5^2}\text{DM} = 1.\overline{47ae1}_{16}$$

Man rechnet also besser mit Pfennigbeträgen

type
  `Pfennig = integer;` $\{32\,Bit\}$
var
  `preis : Pfennig;` $\{-21\,474\,836, 48DM$ $...+21\,474\,836, 47DM\}$

Basisumwandlung von Nachkommastellen: Es sei $x \in [0,1)$. Dann gilt für beliebiges $b \in \mathbb{N} \setminus \{0,1\}$.

i := -1;
`Wiederhole`
  $z_i := \text{trunc}(x * b);$ $\{\in [0,..,b-1]\}$
  $x := x * b - z_i;$ $\{\in [0,1)\}$
  $i := \text{pred}(i)$
`bis((x=0) or (`$i < -$`maxStellenzahl))`

liefert mit

$$\sum_{j=\min\{i+1, -maxStellenzahl\}}^{-1} z_j b^j$$

eine *maxStellenzahl*-Näherungsdarstellung von x zur Basis b. Sei zum Beispiel $x := \frac{9}{64}$ und $b = 2$. Dann arbeitet der Algorithmus wie folgt:

$$\frac{9}{64} \cdot 2 = \frac{9}{32} \quad \Rightarrow \quad z_{-1} = 0$$

$$\begin{aligned}
\frac{9}{32} \cdot 2 = \frac{9}{16} &\quad \Rightarrow \quad z_{-2} = 0 \\
\frac{9}{16} \cdot 2 = \frac{9}{8} = 1 + \frac{1}{8} &\quad \Rightarrow \quad z_{-3} = 1 \\
\frac{1}{8} \cdot 2 = \frac{1}{4} &\quad \Rightarrow \quad z_{-4} = 0 \\
\frac{1}{4} \cdot 2 = \frac{1}{2} &\quad \Rightarrow \quad z_{-5} = 0 \\
\frac{1}{2} \cdot 2 = 1 &\quad \Rightarrow \quad z_{-6} = 1
\end{aligned}$$

Damit ergibt sich

$$\frac{9}{64} = (0.001001)_2$$

Aufgabe 1.4

1. *Berechne* $(\frac{1}{100})_{10}$ *zur Basis* $b = 16$.
2. *Wie ist der Algorithmus so zu ändern, daß der Fehler*

$$\left| x - \sum_{j=i+1}^{-1} z_j b^j \right| \leq \frac{1}{2} \cdot b^{i+1}$$

und nicht lediglich

$$\left| x - \sum_{j=i+1}^{-1} z_j b^j \right| \leq b^{i+1}$$

ist? (Runden statt Abschneiden)

Die Summe $\sum_{j=i+1}^{-1} z_j b^j$ berechnet man übrigens am besten mit dem Horner-Schema:

$$((\ldots(z_{i+1}b^{-1} + z_i) + \ldots + z_{-2})b^{-1} + z_{-1})b^{-1}$$

1.2.2.11 Gleitkommazahlen

Festkommazahlen sind für finanzielle Rechnungen sehr angebracht, solange man mit den Operatoren $+, -, *, /$ auskommt. Für Rechnungen, die Funktionen wie log, exp, ... benötigen, muß man im allgemeinen auf die Gleitkommazahlen ausweichen. Die Idee für Gleitkommazahlen kommt aus den Ingenieur- und Naturwissenschaften:

$$\mathcal{F}_3^4 = \{\min \mathcal{F} := -9999.999, -9999.998, \ldots, 0, \ldots, 9999.998, 9999.999 =: \max \mathcal{F}\}$$