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

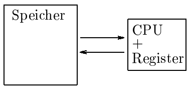
1.2.2.8 Adressierungsarten

- 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:

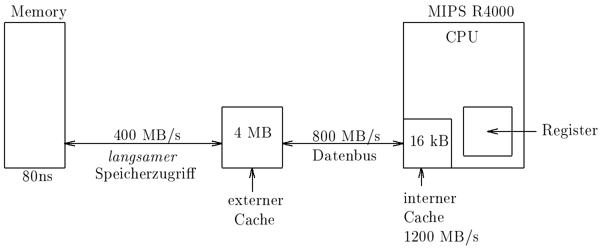
  MOV AX,4[SI]
  MOV AX,MYDATA[SI]
  MOV AX,ELEM[BX][DI]

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	—	—	—	$B_1 - C_1$	$B_2 - C_2$	

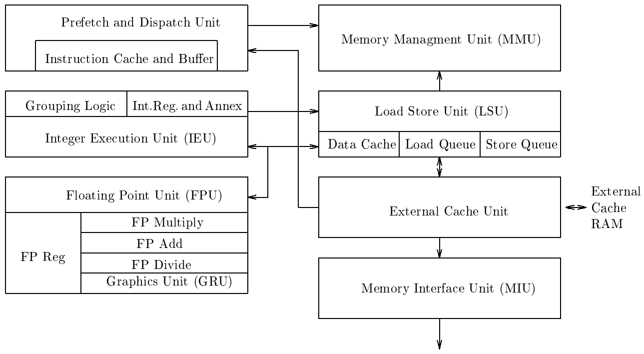
$\searrow$ $B_1 - C_1$ $\quad \searrow$ $B_2 - C_2$

$\boxed{\text{Resümee:}}$

- 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“

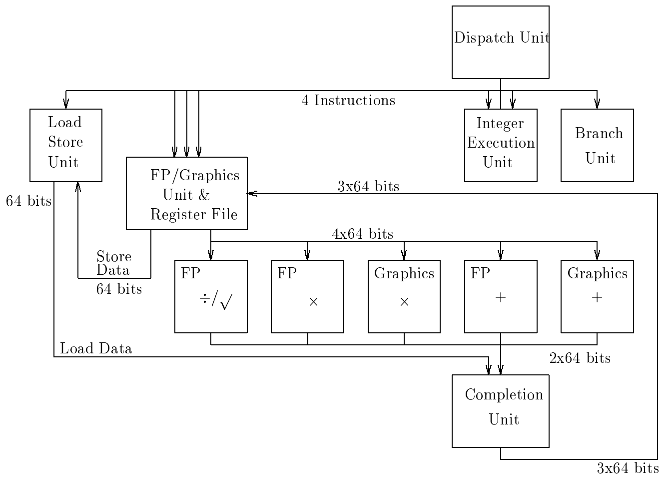
$\boxed{\text{Ein Wert ist noch in der Pipe, obwohl er schon als neues Argument benötigt wird.}}$

Übersicht über die 1996 marktreif gewordenen SPARC-Technologie UltraSPARCI



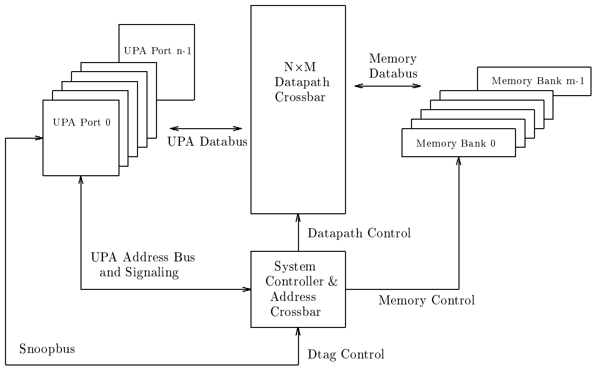
- IU-Pipeline (9 Phasen): fetch-decode-group-execute-cache access-load miss-integer pipe wait- trap resolution-writeback-
- 2 ALU's in IEU (= interger execute unit)
- FP-pipeline
- getrennte FP-execution units

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

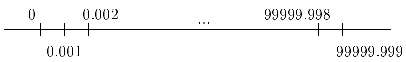


1.2.2.10 Festkommazahlen

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:

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

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

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*

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

und nicht lediglich

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

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 $\log, \exp, \ldots$ benötigen, muß man im allgemeinen auf die Gleitkommazahlen ausweichen. Die Idee für Gleitkommazahlen kommt aus den Ingenieur- und Naturwissenschaften:

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