Rechnerarchitektur

Sommersemester 2013

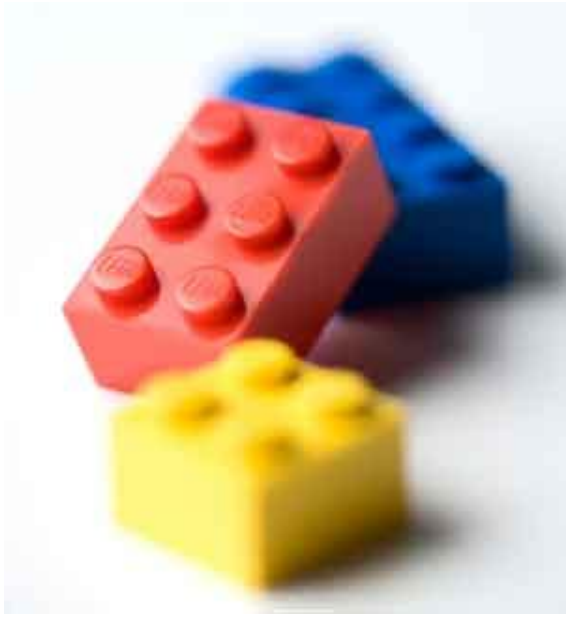
Beispielarchitekturen: Connection Machine

Michael Engel
Informatik 12
TU Dortmund

2013/07/09

Architekturen – Überblick

- FPGA – ??? – Multiprozessor
- Hochleistungsrechnen und eingebettete Systeme
- Die Connection Machine



Parallelität – aber wie?

- Welche Granularität ist die “richtige” zur Ausführung parallelisierbarer Probleme?
  - Nicht allgemein zu beantworten
- Wie müssen die Verbindungsnetze beschaffen sein?

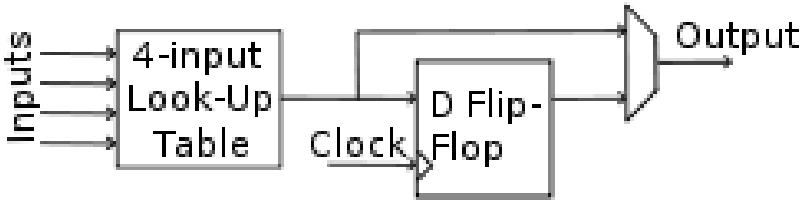
- Bandbreite:

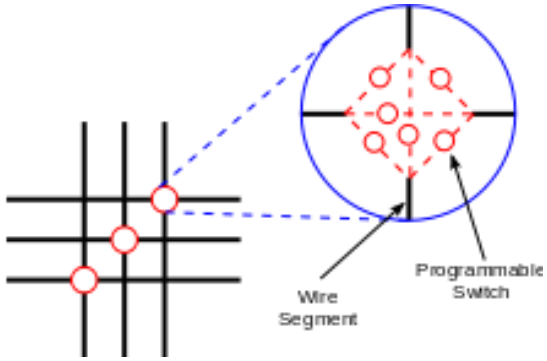
  FPGA – …… – Multiprozessor

- Gibt es zwischen FPGA und “traditionallen” Multiprozessor/Multicore-Systemen sinnvoll nutzbare Architekturen?

Entwurfsraum – FPGAs

- Recheneinheiten von FPGAs: LUTs
  - $y = f(x_0,x_1,x_2,x_3)$ – 4 (6) Bit Eingang, 1 Bit Ausgang
  - Funktion festgelegt für die Laufzeit der Anwendung
- Dynamische Rekonfiguration von FPGAs
  - Ändern der Funktion zur Laufzeit: experimentell
- Spezialeinheiten für effizientere Berechnungen
  - DSP-Blöcke
  - Schieberegister
  - “Hard core”-Prozessorkerne
- Konfigurierbares Verbindungsnetzwerk
  - Verbindungen zur Laufzeit fest





Entwurfsraum – MPSoCs

- Recheneinheiten von FPGAs: Mikroprozessoren
  - Ausführung beliebiger Programme
    - Beschränkt durch Speicher, Rechenleistung
  - Funktion während Laufzeit der Anwendung änderbar
    - Wird in eingebetteten Systemen oft nicht genutzt
- Spezialeinheiten für effizientere Berechnungen
  - DSP-Funktionalität
  - Gleitkommaeinheiten
- (Meist) fest konfiguriertes Verbindungsnetzwerk
  - Kommunikation über dedizierte Kanäle oder gemeinsamen Speicher

Beispiel: Erkennen von Verkehrszeichen

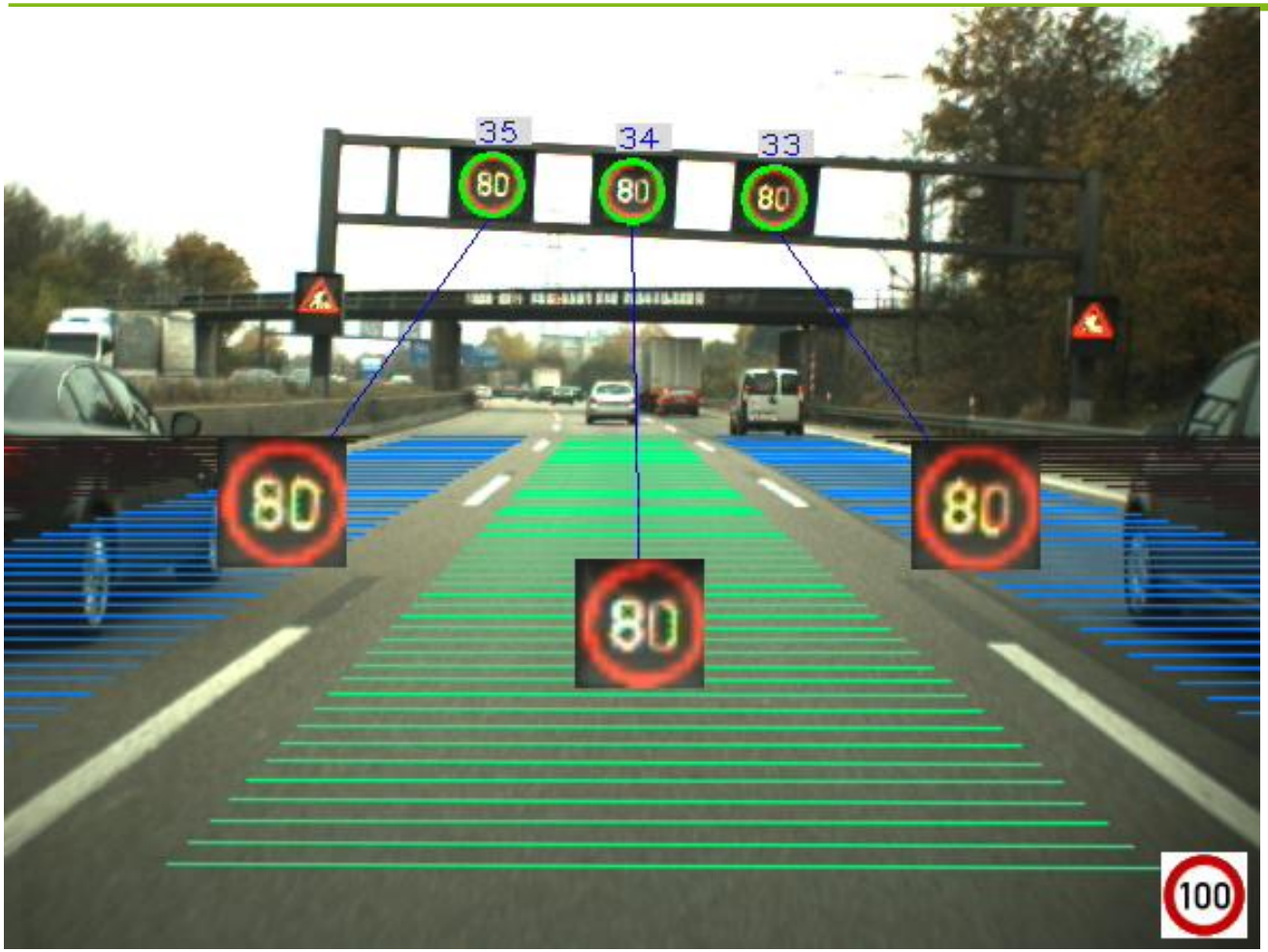
- Fahrerassistenzsysteme
  - ProPilot von Siemens-VDO
- Aufnehmen der Sicht aus der Frontscheibe mit CMOS-Kamera
- Vergleich der Bilder mit abgespeicherten Mustern von Geschwindigkeitszeichen
  - Auch Erkennung zeitabhängiger Beschränkungen
- Fusion mit Navigationsdaten
  - Innerhalb geschlossener Ortschaft oder auf der Autobahn?

(Abb.: Siemens-VDO)

Beispiel: Erkennen von Verkehrszeichen (3)

(Abb.: nienhueser.de)

Beispiel: Erkennen von Verkehrszeichen (2)

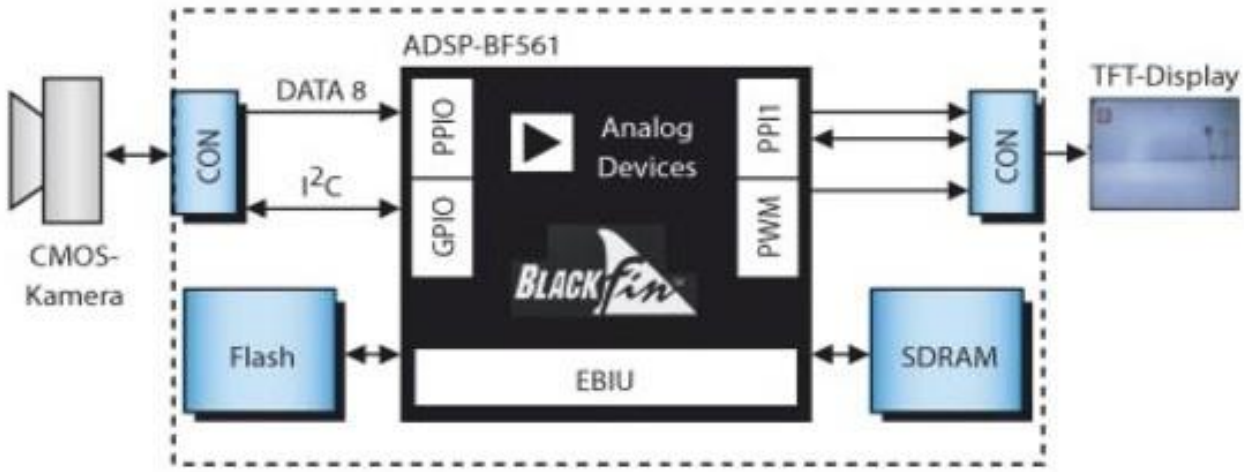


(Abb.: FZI)

Fusionierung von Daten: Spurerkennung und Verkehrszeichenerkennung arbeiten zusammen

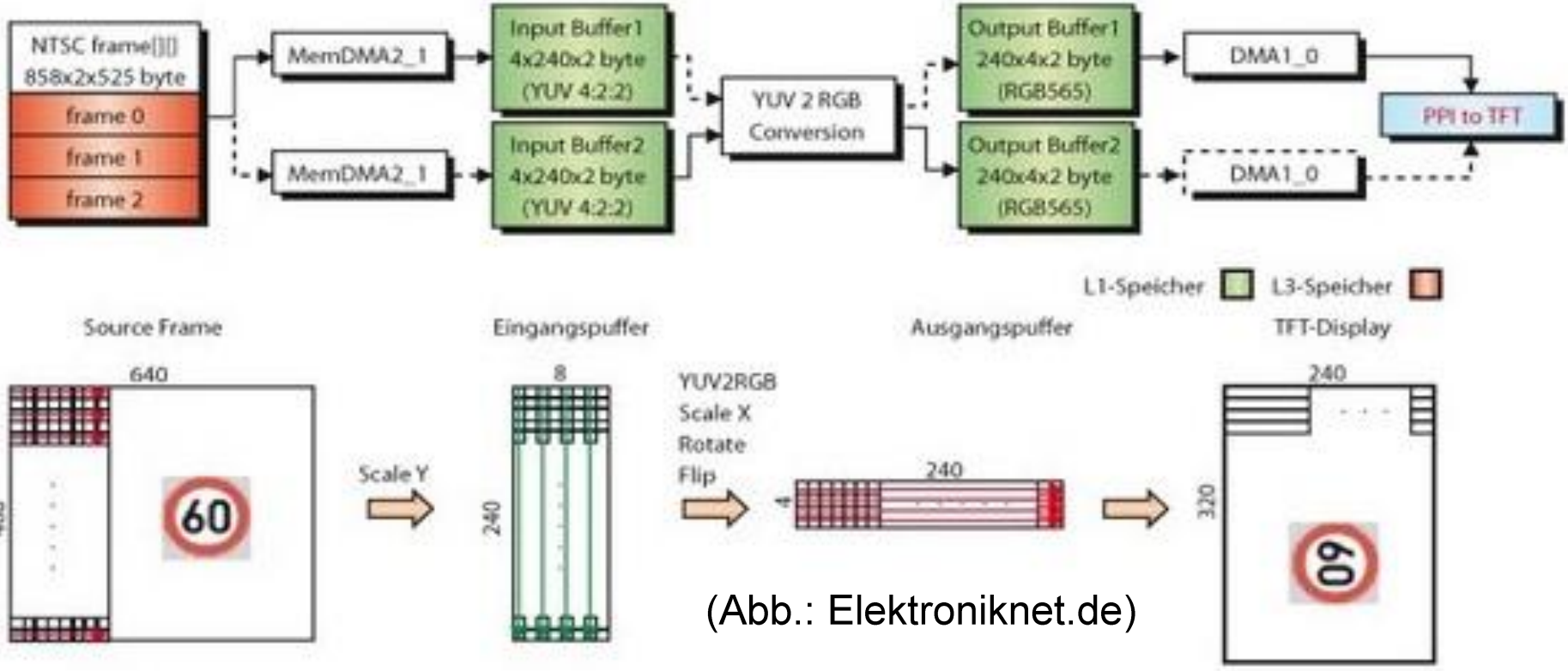
Beispiel: Erkennen von Verkehrszeichen (4)

- Beispielarchitektur
  - CMOS-Kamera
  - Analog Devices BlackFin ADSP-BF561
    - Symmetrischer Dual-Core-DSP
  - Externes SDRAM und Flash
  - TFT-Display zur Anzeige der erkannten Verkehrszeichen
  - Anbindung an KFZ-Netz
    - CAN-Bus



(Abb.: Elektroniknet.de)

Beispiel: Erkennen von Verkehrszeichen (5)



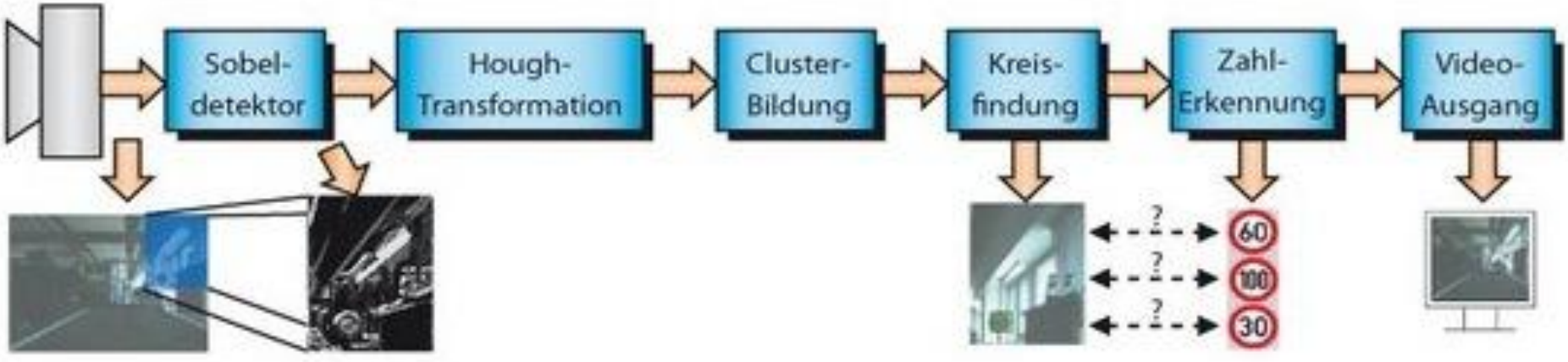
(Abb.: Elektroniknet.de)

Erkennen von Verkehrszeichen – Ablauf (1)

- Ablauf
  - Videoschnittstelle nimmt über DMA-Kanals Videobilder auf und legt sie in den Speicher
  - *Sobeldetektor* filtert Bild: nur noch Kanten sichtbar
    - Berechnung von Kantenstärke und Kantenrichtung
  - *Hough-Transformation* erzeugt mit Vektorfeld vom Sobelfilter Feld mit Ansammlungen möglicher Kreise
  - Clusterbildung fasst Punkte in der Nachbarschaft zusammen und ermittelt Gewicht aller möglichen Kreise
  - Kreisfindung ermittelt die Kreise aus dem Sobel-Kantenbild
    - Hier wird ein Schild erkannt, aber noch nicht die Zahl im Inneren

Erkennen von Verkehrszeichen – Ablauf (2)

- Ablauf
  - Zahlerkennung
    - Schilder noch einmal im Original geladen, skaliert und mit Mustern aus einer Schilddatenbank verglichen
    - Ermittlung der Zahl mit der höchsten Wahrscheinlichkeit
  - Overlay-Erzeugung
    - Kopieren des Schildes in das zu sendende Videobild



(Abb.: Elektroniknet.de)

Erkennen von Verkehrszeichen – Ablauf (3)

- Beispielberechnungen

Anweisungs-Beispiele

Beispiel 1:

```
R4.L=(A0+=R0.L*R1.L), R4.H=(A1+=R0.H*R1.H) || R0=[I0++] || R1=[I1++];
```

Beispiel 2:

```
R4.L=(A0+=R0.L*R1.L), R4.H=(A1+=R0.H*R1.H) || R0=[I0++] || NOP MNOP || R0=[I0++]
|| R1=[I1++];

A0+=R0.L*R1.L || R0.L=w[I0++] || R1.L=w[I1++];
```

(Abb.: Elektroniknet.de)

- Oft Beschleunigung der Berechnungen durch FPGA

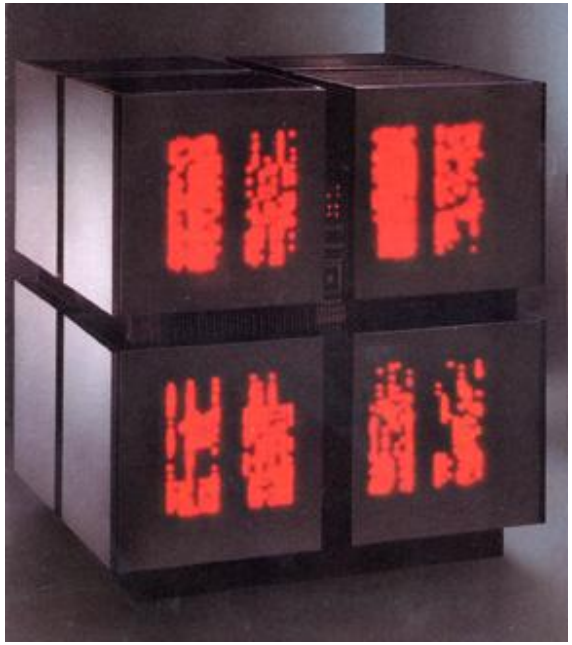
Entwurfsraum

- Komplexe eingebettete Anwendungen erfordern hohe Rechenleistung

- Fragestellungen:
  - Welche parallele Architektur eignet sich am besten?
  - Gibt es “etwas” zwischen FPGAs und MPSoCs?

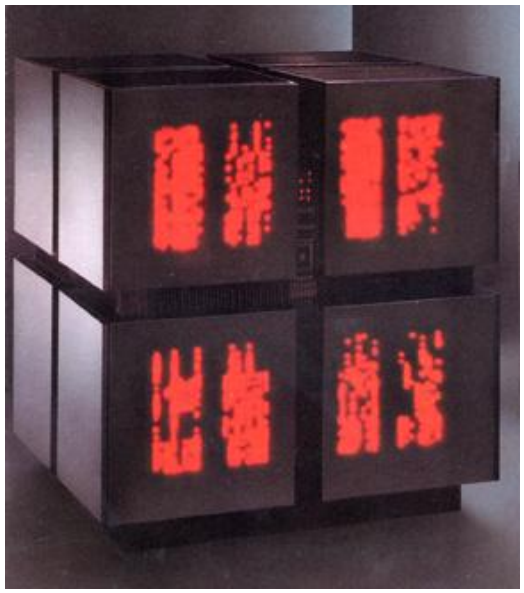
- Viele verschiedene Ansätze

- Beispiel: Connection Machine
- Supercomputer der 1980er Jahre



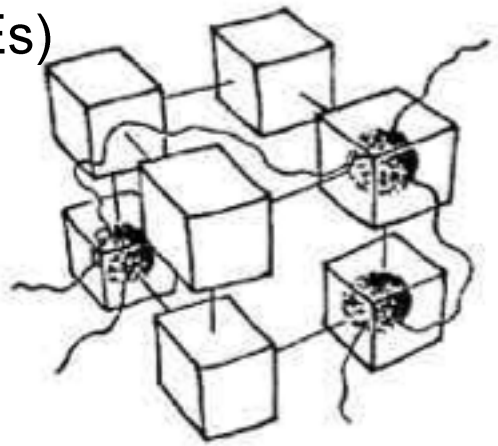
Hochleistungsrechnen vs. eingebettete Systeme?

- Anwendungen der CM
  - Künstliche Intelligenz (AI)
  - Komplexe Matrixoperationen
  - Signalverarbeitung
  - 3D-Grafik
- Vieles davon heute in ES zu finden
  - Automotive-Bereich
    - Abstandsradar
    - Parkassistenzsysteme
    - Verkehrszeichenerkennung
  - Spiele
    - 3D-Grafik, Surround-Sound
  - Mobiltelefone
    - Komplexe Signalverarbeitung im Baseband-Prozessor



Connection Machine

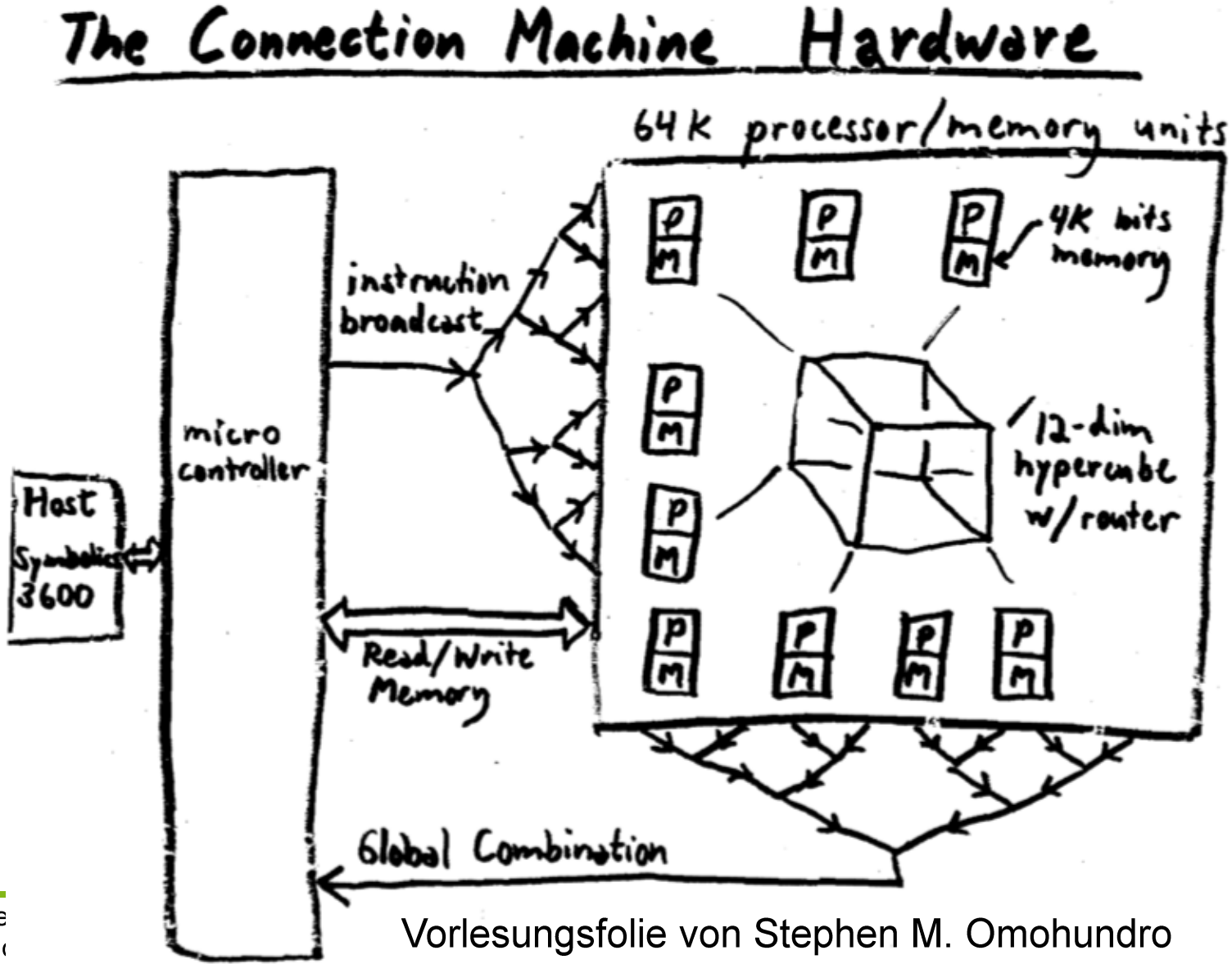
- Hochleistungs-Rechenanwendungen der 1980er Jahre sind heutige eingebettete Anwendungen!
- Connection Machine (CM1: 1985, CM-2: 1987)
  - Thinking Machines, Inc. (1983–1991)
  - Entwickelt zur Simulation von Intelligenz und Leben
- Idee von Daniel Hillis (Student am MIT) 1979:
  - Entwicklung: Rechner mit Struktur eines Gehirns
- Hohe Leistung soll erreicht werden
  - 1. durch viele einfache Rechenelemente (PEs)
  - 2. durch hohe Konnektivitat
- Resultierende Entwurfsentscheidungen:
  - Verwendung bitserieller PEs
  - Wahl der Topologie eines Hyperwurfels (Hyperkubus)



Connection Machine

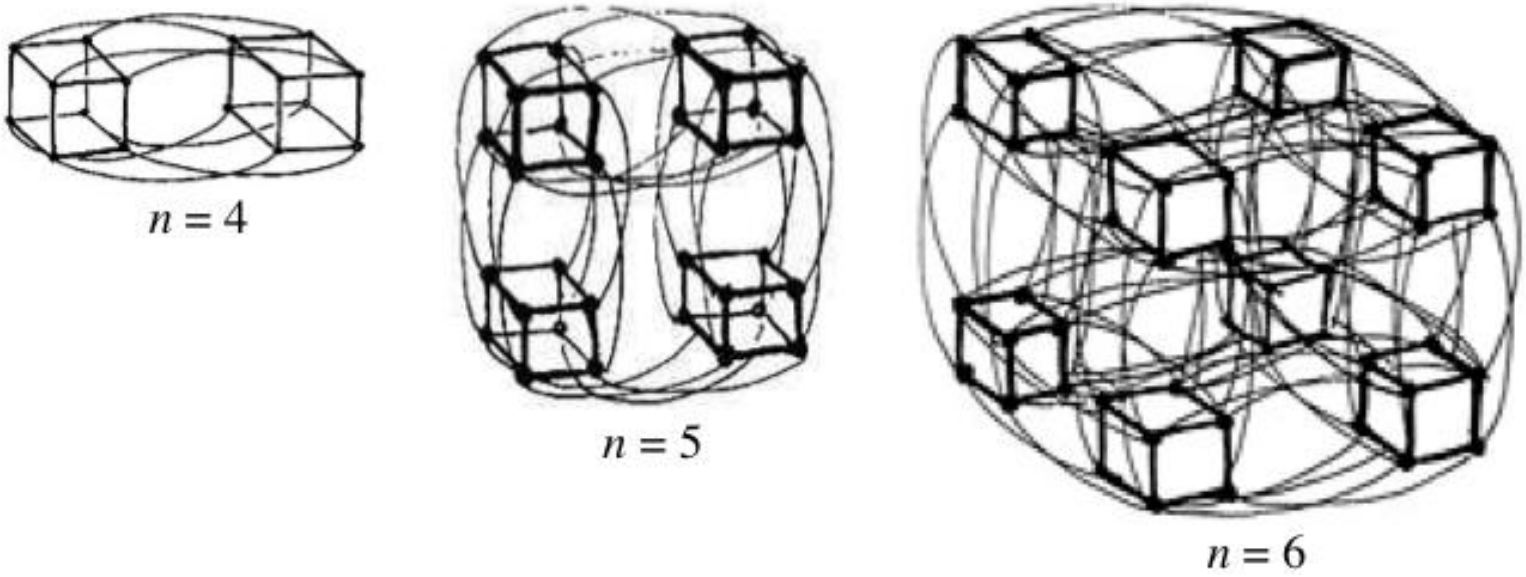
- Eingebettet? Nein!
  - Große 1,5m×1,5m×1,5m
  - 28kW Leistungsaufnahme
  - max.Taktfrequenz: 7MHz
  - Leistung mit 64k PEs:
    - 10 GFlops (32-Bit float Operation)
    - 2.5 GIps (32-Bit integer Addition) [*BF561: 2.4 GIps*]
- Jeder Teilwürfel enthält 16 Leiterplatten und einen "Sequenzer" genannten Hauptprozessor
- Jede Leiterplatte enthält 32 Chips mit je einem Kommunikationskanal ("Router"), 16 Processoren, 16 RAMs
- Die CM-1 besitzt ein hyperkubisches Routingnetzwerk, einen Hauptspeicher und einen I/O-Prozessor, verbunden mit einer Schaltmatrix (nexus)

Connection Machine: Architektur



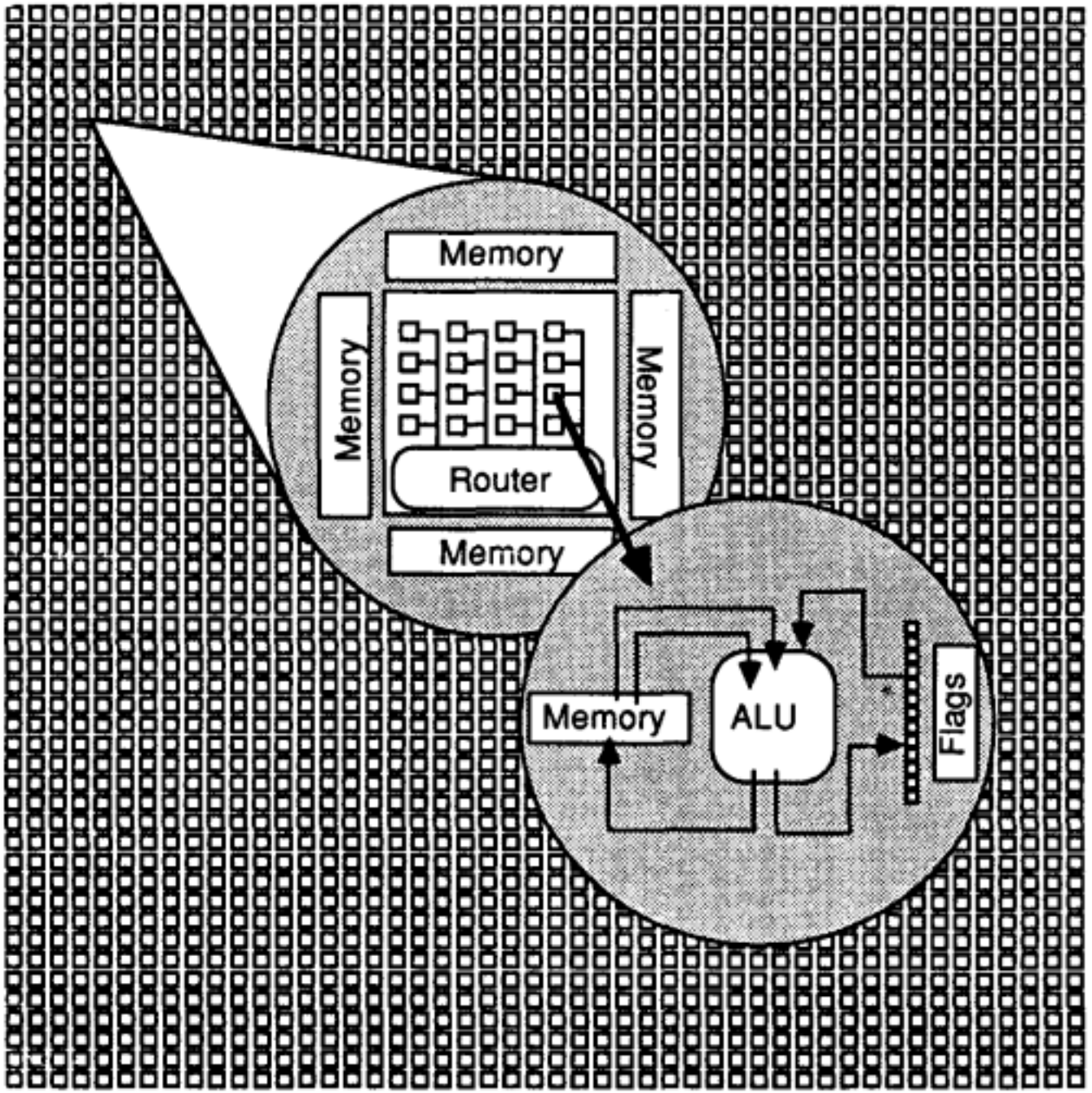
CM-Topologie

- Hyperkubus mit $N = 2^n$ *PEs:*



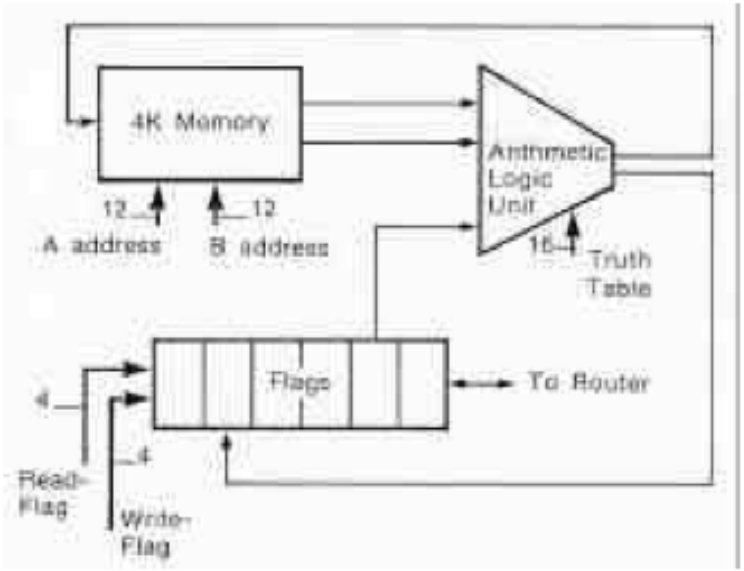
- Warum Hyperkubus?
  - jedes PE in maximal n=log(N) Kommunikations-Schritten erreichbar
  - Viele Topologien lassen sich nachbarschaftserhaltend auf Hyperkubus abbilden

CM-Architektur



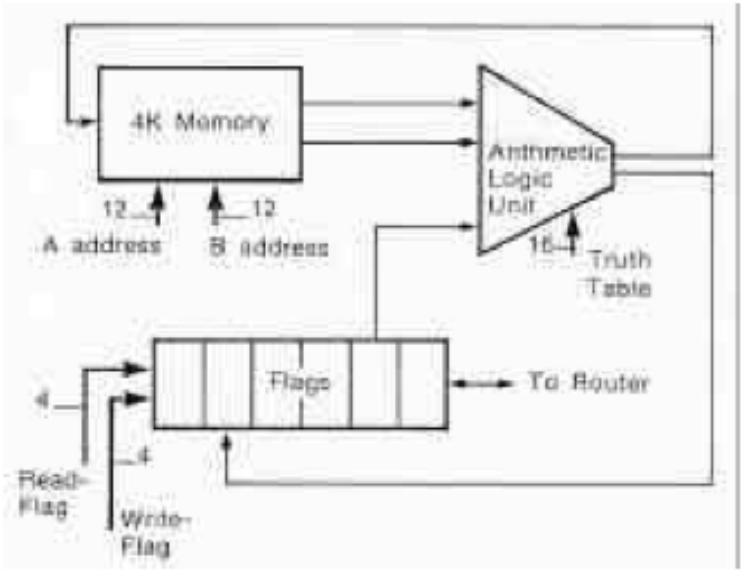
CM-Architektur

- Einzelnes Prozessorelement (PE)
  - 1-Bit-Architektur
  - 4 kBit Speicher, konfigurierbare ALU
  - bitserielle integer-Arithmetik
  - spätere Erhöhung auf 64 KBit Speicher je PE (CM2)
- three-input, two-output logic element and associated latches and memory interface
- ALU-Zyklus:
- 1. read two data bits from memory and one data bit from a flag
- 2. Logic element then computes two result bits from the three input bits.



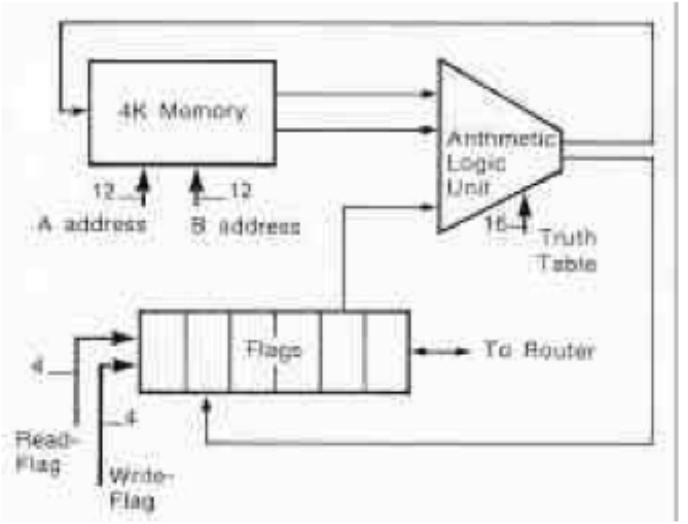
CM-Architektur

- 3. Finally, one of the two results is stored in memory and the other result in a flag
- Entire operation is conditional on the value of a context flag
  - if the flag is zero, then the results for that data processor are not stored
- Logic element can compute any two Boolean functions on three inputs.
- This simple ALU suffices to carry out all the operations of a virtual machine instruction set



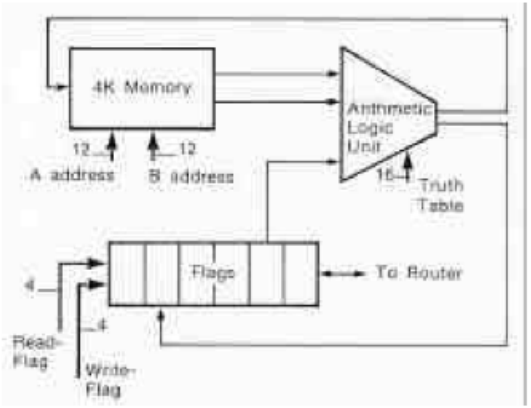
CM-Architektur

- Arithmetic is carried out in a bit-serial fashion
  - requiring 0.75 microsecond per bit plus instruction decoding and overhead
  - a 32-bit Add takes about 24 microseconds
- With 64K processors computing in parallel, this yields a aggregate rate of 2.000 million instructions per second (two billion 32-bit Adds per sec.)



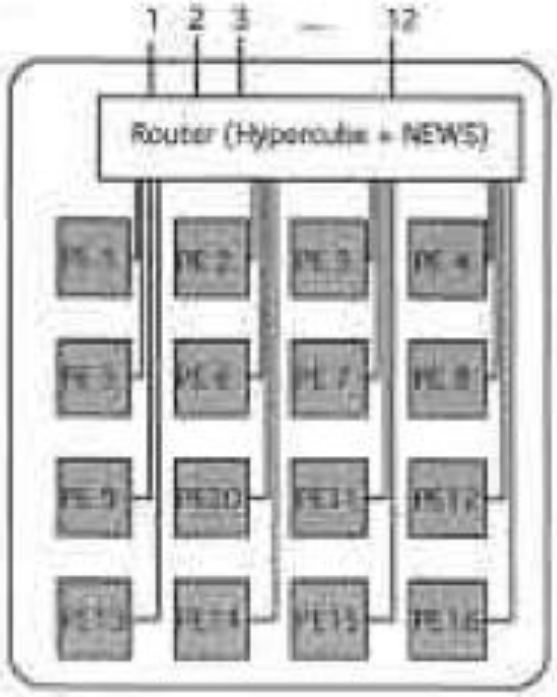
CM-Architektur

- The CM processing element is a reduced-instruction-set-computer processor (***really?***)
- Each ALU cycle breaks down into subcycles
  - On each cycle, data processors execute one low-level instruction (nanoinstruction) issued by the sequencer,
  - memories can perform one read or write operation
- The basic ALU cycle for a two-operand integer Add:
  - 1. LoadA to read memory operand A
  - 2. LoadB to read memory operand B
  - 3. and Store to store the result of the ALU operation



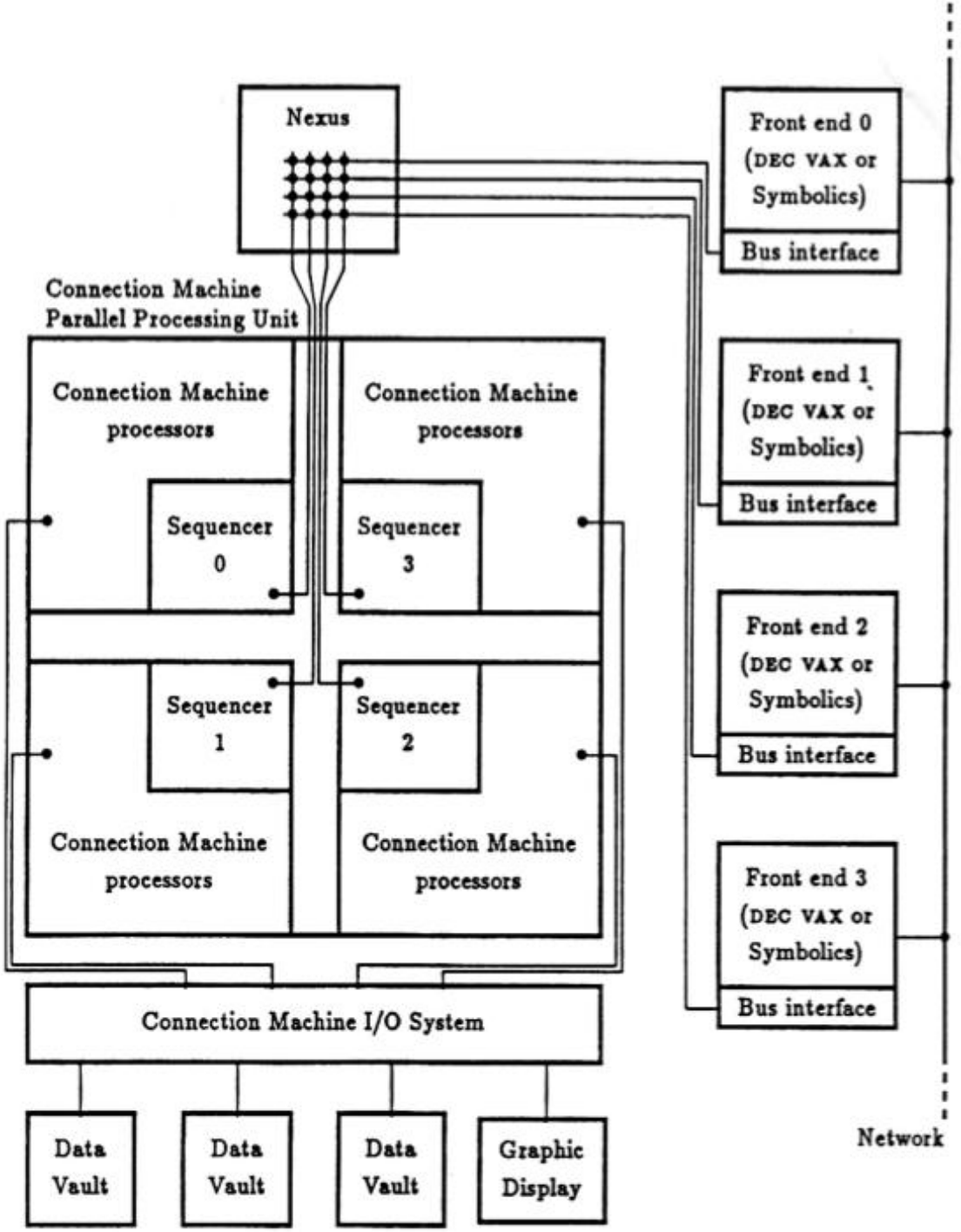
CM-Chips

- 16 PEs mit jeweils einem Router auf einem Chip
- Router sendet in Takt *i* ein Paket in Richtung *i = 1...12*
- spätere Ergänzung in CM-2: eine FPU für jeweils 32 PEs
- NEWS-Grid
  ("North, East, West, South")
  - Horizontale und vertikale Kommunikationsstrukturer zur Kommunikation mit direkten Nachbarn



CM-Systemarchitektur

- 12-dimensionaler Hyper-kubus mit 16 PEs je Knoten
- 32 PE-Chips je Board, 32 Boards je Rack, 4 Racks je Maschine
- Sequenzer generiert Mikrocode-Sequenzen aus Maschinen-Instruktionen
- Anwendungsprogramm läuft auf Host („*Front End*“)
- skalare Operationen auf Host, parallele Operation auf PEs
- 4 Sequencer und 4 Hosts gestatten Multiuser-Betrieb



CM: Skalierbarkeit

- Wichtige Eigenschaft der Connection Machine : *Skalierbarkeit*
- Connection Machine-Systeme haben zwischen 4096 und 65536 physikalische (Hardware-)Prozessoren
- Normalerweise kann Software ohne Änderungen auf CMs mit unterschiedlichen Anzahlen von physikalischen Prozessoren ausgeführt werden
- Zitat: “Using twice as many physical processors, a problem will run in half the time”
  - CM führt nur die datenparallelen Teile einer Anwendung aus!

CM: Befehlssatz

- “Parallel Instruction Set”: Paris
- Paris besitzt eine große Anzahl an Operationen, die den Maschinensprache-Instruktionen eines “gewöhnlichen” Computers ähnlich sind:
  - primitive operations on
    - signed and unsigned integers
    - floating point numbers,
    - and complex numbers
  - communication operations
  - facilities for transferring data between the Connection Machine processors and the front- end computer.

CM: Befehlssatz

- Klassen von Instruktionen:
  - Arithmetisch: lokale arithm./logische Operation auf jedem Prozessor
  - Cube Swap: Senden und Empfangen von Nachrichten über die Dimensionen des Hyperkubus
    - Gleichzeitiges Senden über alle 12 Dimensionen
  - Send: Direktes Senden von Nachrichten durch das Routing-Netzwerk an einen bestimmten Prozessor
  - Scan: parallele Berechnung auf einem Vektor an Eingabedaten (Integer), eine pro Prozessor: aus

$$[\ 4 \quad 7 \quad 1 \quad 0 \quad 5 \quad 2 \quad 6 \quad 4 \quad 8 \quad 1 \quad 9 \quad 5\ ]$$

wird für Add: $[\ 0 \quad 4 \quad 11 \quad 12 \quad 12 \quad 17 \quad 19 \quad 25 \quad 29 \quad 37 \quad 38 \quad 47\ ]$

CM: Befehlssatz-Beispiel

- LISP-ähnliche Notation, Kommunikations-instruktionen

```
(defun *applya (u v)
   (let* ((slc::stack-index *stack-index*)
          (-!!-index-2 (+ slc::stack-index 32))
          (pvar-location-u-11 (pvar-location u))
          (pvar-location-v-12 (pvar-location v)))

     (cm:get-from-west-always -!!-index-2 pvar-location-u-11 32)
     (cm:get-from-east-always *!!-constant-index4 pvar-location-u-11 32)
     (cmi::f+always -!!-index-2 *!!-constant-index4 23 8)
     (cmi::get-from-east-with-f-add-always -!!-index-2 pvar-location-u-11 23 8)

     (cmi::f-multiply-constant-3-always pvar-location-v-12 pvar-location-u-11 s 23 8)
     (cmi::f-subtract-multiply-constant-3-always pvar-location-v-12
                                  pvar-location-v-12 -!!-index-2 r 23 8)

     (cm:get-from-north-always -!!-index-2 pvar-location-u-11 32)
     (cmi::f-always slc::stack-index -!!-index-2 23 8)
     (cmi::get-from-north-with-f-subtract-always pvar-location-v-12 pvar-location-u-11 23 8)

     (cm:get-from-south-always -!!-index-2 pvar-location-u-11 32)
     (cmi::float-subtract pvar-location-v-12 slc::stack-index -!!-index-2 23 8)
     (cmi::get-from-south-with-f-subtract-always pvar-location-v-12 -!!-index-2 23 8)
     )
)
```

CM: Virtuelle Maschine

- Skalierbarkeit der CM durch Modellierung eines abstrakten Version der CM-Hardware in Paris
- Wichtigste Eigenschaft: *virtuelle Prozessoren*
  - Jeder physikalische Prozessor simuliert ein oder mehrere virtuelle Prozessoren
  - Programme können eine beliebige, passende Anzahl von Prozessoren annehmen
  - Diese virtuellen Prozessoren werden dann auf die physikalischen Prozessoren abgebildet
- Linearer trade-off zwischen Anzahl physikalischer Prozessoren und Ausführungszeit
- Speicher trade-off: der Speicher eines physikalischen Prozessors wird unter den zugeordneten virtuellen Prozessoren aufgeteilt

CM: Virtuelle Maschine

- Paris modelliert *parallele Instruktionen:*
  - Wenn eine “add”-Instruktion in Paris ausgeführt wird, kann jeder physikalische Prozessor viele Additionen ausführen – eine für jeden zugeordneten virtuellen Prozessor
  - Ausführung bestimmter Instruktionen kann von Bedingungen (Flags) abhängig gemacht werden: *predicated execution*
- Paris verfügt auch über für virtuelle Prozessoren angepasste Versionen der drei hardware-unterstützten Kommunikationsmechanismen (routing, NEWS grids, und scanning)

CM: Virtuelle Maschine

- Virtuelle Prozessoren unterstützen datenparallele Programmierung durch Zuweisung eines virtuellen Prozessores (VP) zu jedem Element einer Datenmenge
  - Die Menge aller VPs, die einer Datenmenge zugeordnet sind, heisst auch VP set
- Beispiel: Bildverarbeitung auf einem Bild mit 65.536 Pixeln im Format 512 x 128:
  - Jedes Pixel ist eine Element der Datenmenge, die das Bild beschreift
    - Programm nutzt ein VP set der Größe 65.536: ein VP für jedes Pixel
    - Konfiguration des NEWS grid für dieses VP set ist zweidimensional mit der Form 512 x 128

CM: Abbildung virtueller Maschinen

- Die Größen der VP sets und die Größe der physikalischen Maschine bestimmen, wie viele virtuelle Prozessoren auf jeden physikalischen Prozessor abgebildet werden
- Wenn eine Anwendung eine Million Datenelemente verarbeiten muss, fordert sie $v = 2^{20}$ virtuelle Prozessoren an
- Die CM-Hardware besitzt (in Maximalkonfiguration) $P = 2^{16}$ physikalische Prozessoren mit jeweils $M = 2^{16}$ Bit Speicher (bei der CM-2; $M = 2^{12}$ Bit für die CM-1)
- Damit werden auf jeden physikalischen Prozessor $V/P = 16$ virtuelle Prozessoren abgebildet

CM: Programmierung

- Programmierung in Paris: aufwendig
- Explizite datenparallele Programmierung in C*:

– Definition von **mehrdimensionalen Datenräumen** mittels `shape`, z.B.:

```
shape [10000] vec;
shape [10][50] mat;
shape [6][6][6] dim3;
```

– Deklaration **paralleler Variablen** eines Raumes, z.B.:

```
int:vec x,y,z[10];  int:mat a,b;  float:dim3 g,h;
```

– **parallele Ausführung** durch Selektion eines Raumes mittels `with` und Verwendung überladener Operatoren `+,-,*,...`, z.B.:

```
with (vec)                    with (dim3)
  z[3]= x+y;                    g = g*2h;
```

– Ausführung **paralleler Bedingungen** mittels `where`, z.B.:

```
with (mat)
  where (a != 0)
    b = b / a;
```

CM: Programmierung

- Programmierung in C*

– Funktion `pcoord(i)` liefert jedem PE seinen **Index in Dimension** `i` des mittels `shape` definierten Datenraumes

– synchroner Datentransport um eine Distanz `dist` in Dimension `i` durch **linksseitigen Indexausdruck**, z.B.:

```
y = [pccord(0)+dist]x;
b = [pccord(0)+1][pccord(1)-2]a;
```

(*Hinweis*: `pccord(i)` kann hier auch durch `.` ersetzt werden)

– beliebige, **irreguläre Kommunikation** durch Verwendung einer parallelen Indexvariablen im linksseitigen Indexausdruck, z.B.:

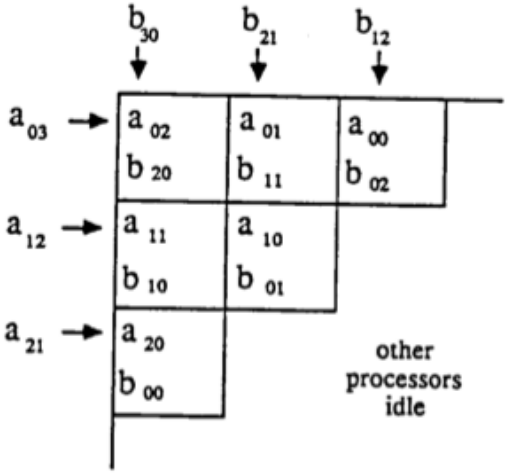
```
b = [index]a;
```

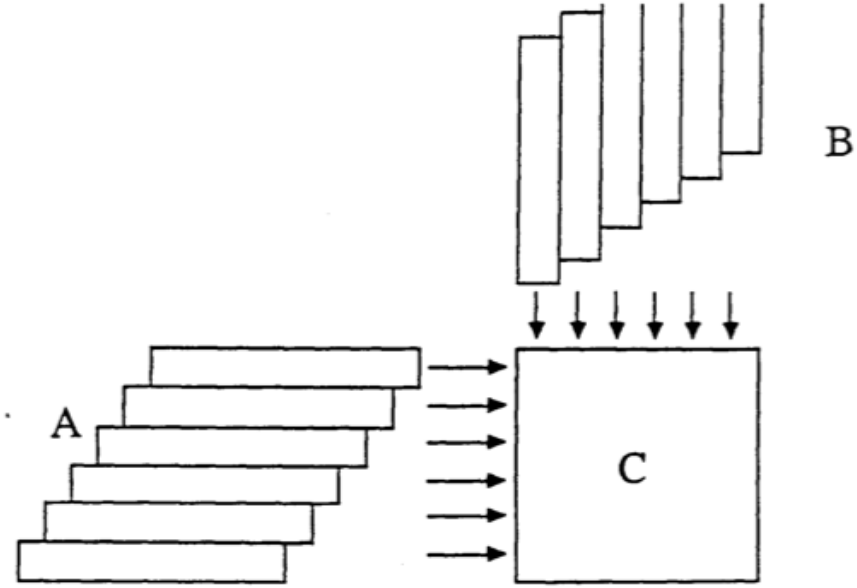
(*Hinweis*: `index` enthält hier Quellkoordinaten und muß zum gleichen Datenraum gehören wie `a`)

- Hardwareunterstutzung von $V = 2^v$ virtuellen PEs, indem Speicher in V/N Teile aufgeteilt wird

CM-Anwendung: Matrixmultiplikation

- O(n)-Algorithmus (vergl. Systolische Arrays)
- Wechselt zwischen Kommunikationsphase und Berechnungsphase
- Optimierung für CM: Verschieben von Koeffizienten ist nur Folge der Struktur systolischer Arrays
  - PEs der CM können Koeffizienten von A und B direkt laden

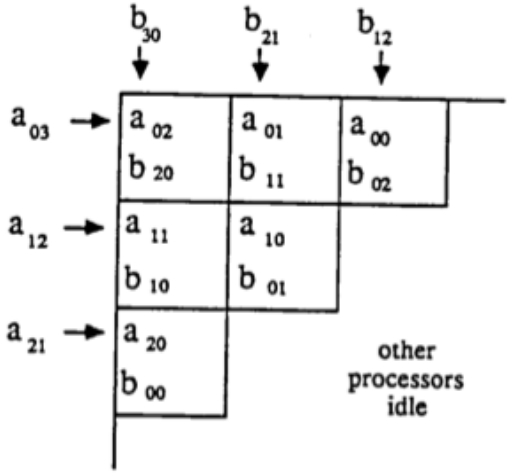




CM-Anwendung: Matrixmultiplikation (2)

- Problem des vorigen Algorithmus: immer noch schlechte Ausnutzung der PEs
  - Aktivierung von Prozessoren breitet sich von NW nach SO aus
  - Nie mehr als 50% aller Prozessoren aktiv
  - Durchschnittliche Prozessorauslastung: 1/3
- Wie kann die Auslastung verbessert werden?
- Idee: Innere Produkte der Elemente von C in anderer Reihenfolge berechnen:

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$



CM-Anwendung: Matrixmultiplikation (3)

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \times B_{k,j}$$

- Addition ist kommutativ
- Beginn der Summe mit jedem $k_{i,j}$ aus [0…m-1] machbar
- Aufsummieren bis m-1, dann den Rest der Summe von 0…$k_{i,j}$-1 berechnen
  - Für 4x4-Matrizen ergibt sich dann folgende Anfangskonfiguration:

$K^{(1)}_{i,j}$

0	1	2	3
1	2	3	0
2	3	0	1
3	0	1	2

- O(log n)-Algorithmus:
  - Verwendung von n^3 Prozessoren, um alle n^3 Produkte gleichzeitig zu berechnen
  - In Folge dann n^2 parallele Additionen
  - Multiplikation: konstante Zeit
  - Summenreduktion: log n

CM-Entwicklung

- ***"That is positively the dopiest idea I ever heard"*** (Richard Feynman about the CM)
- Eine kleine Anekdote zur Beteiligung von Prof. Richard Feynman an der Entwicklung der CM (von Danny Hillis):
- The router of the Connection Machine was the part of the hardware that allowed the processors to communicate
- It was a complicated device; by comparison, the processors themselves were simple
- Connecting a separate communication wire between each pair of processors was impractical since a million processors would require 10^{12} wires
  - (using the 64k Processors of the CM: $2^{32} \approx 4*10^9$)

CM-Entwicklung

- Instead, we planned to connect the processors in a 20-dimensional hypercube so that each processor would only need to talk to 20 others directly.
- Because many processors had to communicate simultaneously, many messages would contend for the same wires.
- The router's job was to find a free path through this 20-dimensional traffic jam or, if it couldn't, to hold onto the message in a buffer until a path became free.
- ***Our question to Richard Feynman was whether we had allowed enough buffers for the router to operate efficiently.***

CM-Entwicklung

- By the end of that summer of 1983, Richard had completed his analysis of the behavior of the router, and much to our surprise and amusement, he presented his answer in the form of a *set of partial differential equations*.
- To a physicist this may seem natural, but to a computer designer, treating a set of boolean circuits as a continuous, differentiable system is a bit strange.
- Feynman's router equations were in terms of variables representing continuous quantities such as "the average number of 1 bits in a message address.”

CM-Entwicklung

- I was much more accustomed to seeing analysis in terms of inductive proof and case analysis than taking the derivative of "the number of 1's" with respect to time.
- Our discrete analysis said we needed seven buffers per chip
- Feynman's equations suggested that we only needed five.
- ***We decided to play it safe and ignore Feynman.***
- The decision to ignore Feynman's analysis was made in September, but by next spring we were up against a wall…

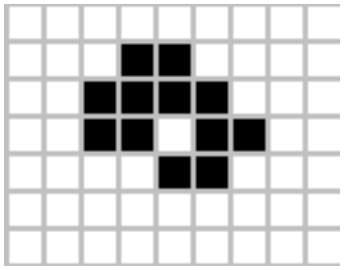
CM-Entwicklung

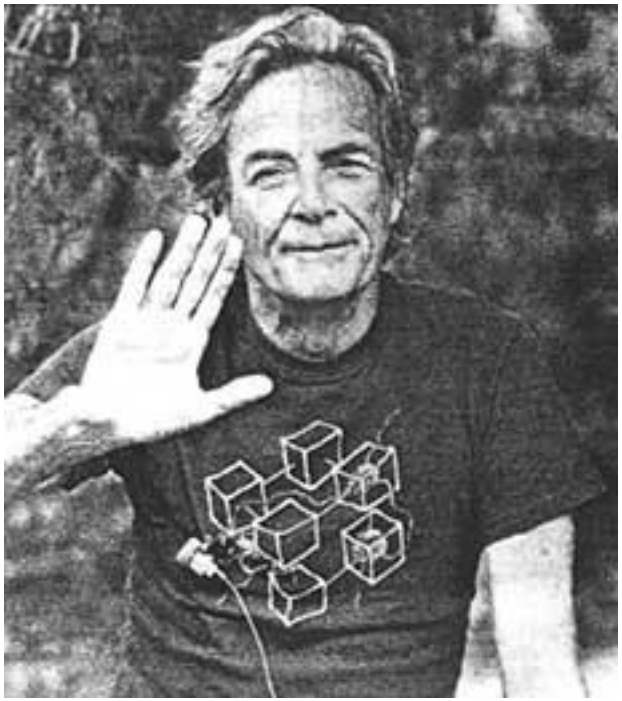
- The chips that we had designed were slightly too big to manufacture and the only way to solve the problem was to cut the number of buffers per chip back to five.
- Since Feynman's equations claimed we could do this safely, his unconventional methods of analysis started looking better and better to us.
- We decided to go ahead and make the chips with the smaller number of buffers.

CM-Entwicklung

- ***Fortunately, he was right.***
- When we put together the chips the machine worked.
- The first program run on the machine in April of 1985 was Conway's game of Life.

- **Video** von Danniel Hills...





Zusammenfassung

- Einige Eingebettete Anwendungen erfordern heute so viel Rechenleistung, wie ein Supercomputer der 1980er Jahre zur Verfügung gestellt hat
- Konzept der CM:
  - Einzelbitoperationen, SIMD
  - große Anzahl von Prozessorelementen
  - Hochkomplexes Verbindungsnetzwerk
  - Sequenzer steuert Kontrollfluss der PEs

- Sinnvoll für eingebettete Systeme?
  - Passt eine CM auf einen heutigen FPGA?