

Hochleistungsrechner

Skript zur Vorlesung

Ausgabestand November 2000

Prof. Dr.-Ing. C. Müller-Schloer
Institut für Rechnerstrukturen und Betriebssysteme
Universität Hannover
Appelstraße 4
30167 Hannover

email: cms@irb.uni-hannover.de
<http://www.irb.uni-hannover.de>
Tel.: 0511-762-19730
FAX: 0511-762-19733

1 Motivation und Inhalt	1
2 Prozessoren	2 - 1
2.1. Computerarithmetik	2 - 1
2.1.1. Zahlendarstellung..... (Wiederholung)	2 - 1
2.1.2. Befehlssatz MIPS	2 - 4
2.1.3. ALU- Entwurf	2 - 8
2.1.4. Beschleunigung der Addition	2 - 16
2.1.5. Multiplikation	2 - 24
2.1.6. Division	2 - 27
2.1.7. Floating Point Unit	2 - 27
2.2. Datenpfad	2 - 27
2.2.1. Taktschemata	2 - 27
2.2.2. Ein-Takt-Datenpfad	2 - 30
2.2.3. Steuerung des Ein-Takt-Pfades	2 - 34
2.2.3.1 ALU-Control	2 - 34
2.2.3.2 Main Control	2 - 36
2.2.4. Mehr-Takt-Datenpfad	2 - 42
2.3. Pipeline-Datenpfad	2 - 46
2.3.1. Befehlszyklus	2 - 46
2.3.2. Umformung	2 - 46
2.3.3. PL-Register	2 - 47
2.3.4. Ablauf des Befehls Load Word (lw)	2 - 48
2.3.5. Datenpfad-Modifikation für lw	2 - 53
2.3.6. Graphische PL-Darstellung	2 - 54
2.3.7. PL-Steuerung	2 - 56
2.3.8. Datenkonflikte	2 - 57
2.3.9. Forwarding	2 - 60
2.3.10. Steuerkonflikte	2 - 63
2.3.11. Exceptions	2 - 63
2.4. Superskalarität	2 - 63
2.4.1. Mehrzyklenbefehle	2 - 63
2.4.2. Nichtlineare Einspeisung	2 - 65
2.4.3. Scoreboard	2 - 67
2.4.4. Entkoppelte Architektur	2 - 71
2.5. Ein/Ausgabe	2 - 75
2.5.1. I/O-Performance	2 - 76
2.5.2. I/O-Geräte (Maus, HD, Netz)	2 - 77
2.5.3. Busse	2 - 78
2.5.4. Kommunikation	2 - 82
2.6. Fallstudie: Mikroprozessor Alpha 21064	2 - 84

3 Parallelrechner	3 -1
3.1. Anwendungen	3 -1
3.1.1. Überblick	3 -1
3.1.2. Anwendung: Transaktionssysteme	3 -4
3.1.3. Anwendung AI	3 -8
3.1.3.1 Regelbasierte Systeme	3 -8
3.1.3.2 Semantische Netzwerke	3 -10
3.1.4. Grundbegriffe der Parallelarbeit	3 -12
3.1.5. Anwendung Numerik: Wellengleichung	3 -17
3.1.6. Zusammenfassung wichtiger Begriffe	3 -22
3.2. Merkmale und Klassifikation	3 -23
3.2.1. Vorgehen	3 -23
3.2.2. Allgemeine Merkmale der Parallelarbeit	3 -23
3.2.3. Merkmale aus Benutzersicht	3 -26
3.2.4. Merkmale aus HW-Sicht	3 -29
3.2.5. Wichtige Klassifizierungssysteme	3 -31
3.2.5.1 Flynn	3 -31
3.2.5.2 Snyder	3 -32
3.2.5.3 ECS	3 -33
3.2.5.4 Hockney	3 -35
3.2.5.5 Treleaven	3 -36
3.3. Verbindungsnetzwerke	3 -40
3.3.1. Grundstrukturen und Klassifikation	3 -40
3.3.1.1 Bus-Verbindungsnetzwerke	3 -40
3.3.1.2 Direkte Verbindungsnetzwerke	3 -41
3.3.1.3 Indirekte Verbindungsnetzwerke	3 -42
3.3.1.4 Virtuelle Verbindungsnetzwerke	3 -44
3.3.2. Dynamische Bewertungskriterien	3 -45
3.3.3. Statische Bewertungskriterien: Topologie	3 -47
3.3.4. Wegesuche (Routing)	3 -49
3.3.5. Vermittlungsprinzip	3 -51
3.3.6. Flußmechanismen	3 -52
3.3.7. Lastverteilung und Hot Spots	3 -53
3.3.8. Kombininierende Netzwerke	3 -55
3.4. Quantitative Betrachtung der Parallelarbeit	3 -60
3.4.1. Verlustmechanismen	3 -60
3.4.2. Kenngrößen	3 -61
3.4.3. Gesetz von Amdahl	3 -69
3.4.4. Speed-up vs. Scale-up	3 -70
3.4.4.1 Idealisierte Annahmen	3 -70
3.4.4.2 Realistische Annahmen	3 -70
3.4.5. Overhead	3 -71
3.4.5.3 Definition	3 -71
3.4.5.4 Overhead-Berechnung	3 -73
3.4.6. Speed-up mit Latenzzeit	3 -75

3.4.6.1	Untere Schranke (Worst Case)	3 -76
3.4.6.2	Obere Schranke (Best Case)	3 -77
3.4.6.3	Diskussion	3 -81
3.5.	Parallelrechnerarchitekturen am Beispiel	3 -83
3.5.1.	SIMD Parallel	3 -83
3.5.1.1	ILLIAC IV Projekt-Start 1966, University of Illinois	3 -83
3.5.1.2	Connection Machine (D.Hillis)	3 -85
3.5.2.	MIMD Shared Memory	3 -90
3.5.2.3	CMU-Maschinen	3 -90
3.5.2.4	NYU Ultracomputer	3 -92
3.5.2.5	Sequent	3 -94
3.5.3.	MIMD Message Passing	3 -95
3.5.3.1	Hypercube-basierte Multiprozessoren	3 -95
3.5.3.2	Parsytec	3 -99
3.5.3.3	Paragon XP/S (Touchstone)	3 -100
3.5.3.4	CM-5	3 -101
3.5.4.	Hybrid-Architektur	3 -101

1 Motivation und Inhalt

Vorbemerkung:

Ab dem WS 97/98 wird die Vorlesung Parallelrechner durch die Vorlesung Hochleistungsrechner ersetzt. Die neue Vorlesung HLR enthält wesentliche Teile der alten Vorlesung PAR, es fallen aber die Kapitel *Programmierung von Parallelrechnern (am Beispiel Hypercube)* und *Programmierung komplexerer Anwendungen* weg. Stattdessen werden moderne Prozessorarchitekturen tiefergehend behandelt. Neu hinzu kommen die Kapitel *Computerarithmetik*, *Datenpfad*, *Pipelining*, *Superskalarität*, *Fallstudie Mikroprozessor*.

Voraussetzungen:

Grundlagen der Informatik I-II (oder äquivalente Vorlesung). Die Kenntnis der Inhalte der Vorlesung Betriebssysteme ist hilfreich, die der Vorlesung Rechnerstrukturen notwendig.

Literatur HLR:

Patterson, Hennessy: Computer Organization & Design: The Hardware/Software Interface, Morgan Kaufmann Publishers 1994

Hennessy, Patterson: Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers 1996

Almasi, Gottlieb: Highly Parallel Computing, Benjamin/Cummings Publishing Comp. 1989

Hockney, Jesshope: Parallel Computers 2, Adam Hilger 1988

$$P = \frac{f_c}{CPI + MEMD} \cdot n_{ref} \cdot P_{eff}$$

2 Wege zur Erhöhung der Performance:

- a) innerhalb eines Prozessors (f_c , CPI, MEMD, n_{ref})
- b) mit mehreren Prozessoren (p_{eff})

Neben technologischen Verbesserungen (f_c) spielt dabei die Parallelisierung die wichtigste Rolle.

Innerhalb eines Prozessors:

- Wortparallelität (64 bit)
- Pipelining
- Superskalarität

Multiprozessoren: Zusammenschaltung vieler Monoprozessoren

Beide Wege werden gleichzeitig beschritten. Im Breitenmarkt (Mikroprozessoren) herrscht heute die prozessor-interne Parallelität vor. Multiprozessoren sind schwerer zu handhaben, sie werden daher nur für höchste Leistungsanforderungen eingesetzt. Eine Sättigung des Monoprozessor-Leistungsanstiegs ist nicht zu beobachten. Erst wenn es einfacher (billiger) wird, einen bestimmten Leistungszuwachs durch Prozessorvervielfachung zu erzielen (im Vergleich zur Monoprozessor-Leistungssteigerung), wird ein breites Umschwenken zu Multiprozessoren erfolgen.

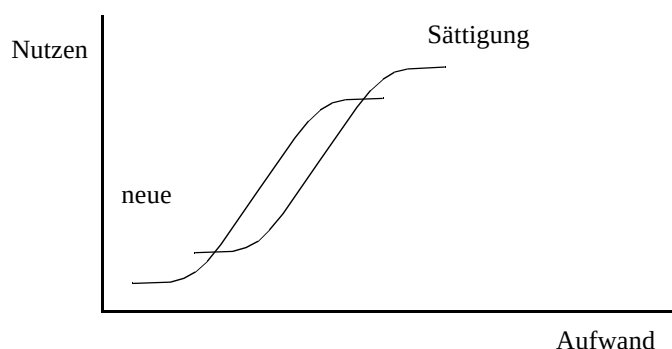


Bild 1.1. S-Kurve zur Aufwand/Nutzen-Relation

Grenzen für Monoprozessoren

- Taktfrequenz
 Lichtgeschwindigkeit = $3 \cdot 10^8$ m/s
 $\Delta s = 3$ cm: $\Delta t = 0,1$ ns

$$\Delta s = 3 \text{ m:} \quad \Delta t = 10 \text{ ns}$$

- steigende Kosten für Technologieverbesserungen
- CPI-Senkung durch Pipelining und Superskalarität in Monoprozessoren: begrenzte Parallelität auf Instruktionsebene
- Leistungssteigerung durch Kombination von Maßnahmen (DEC-Behauptung):

Taktfrequenz	10
Superskalarität	10
Parallelität	10

Mittel

- Wo bringt zusätzlicher Aufwand den größten Nutzen?
These: Zentralisierung führt zum Dinosaurier
- Kombinationen mehrerer Technologien, welche sich nicht in der Sättigung befinden, ist am billigsten.
- Komplexitätsvermeidung durch Modularität, Lokalität

Beispiel 1-1:

Effektivster Einsatz von 2 Millionen Transistoren

zentral

Monoprozessor

Superskalar

$f = 100 \text{ MHz}$

$\text{CPI} = 0,5$

$P = 200 \text{ MIPS}$

dezentral

Multiprozessor, 16 Knoten

je 125 000 Transistoren

$f = 100 \text{ MHz}$

$\text{CPI} = 1,5$

$P = 1000 \text{ MIPS (peak)}$

$P = 500 \text{ MIPS } (\epsilon = 50 \%)$

Hindernisse für Multiprozessoren

- Zusätzlicher freier Entwurfparameter: Topologie
- Anwendungsabhängigkeit
- Auslastung von Parallelrechnern i. A. schlecht (20% ... 50%)
- Programmierung schwierig, ungewohnt
- Verhaltensanalyse schwierig
- Systemsoftware (Compiler, Lastverteiler, Optimierer) noch rudimentär
- Dusty Deck-Software erschwert das Umsteigen

Aufbau dieser Vorlesung

Teil 1: Prozessoren für höchste Leistungen

- Schrittweiser Aufbau eines Prozessors (Computerarithmetik, Datenpfad, Pipeline-Datenpfad, Superskalarität)
- Fallstudie Mikroprozessor Alpha 21 064 (schnellster μP !)

Teil 2: Parallelrechner

- Anwendungen
- Merkmale und Klassifikation
- Verbindungsnetzwerke
- Quantitative Betrachtung
- Parallelrechner am Beispiel

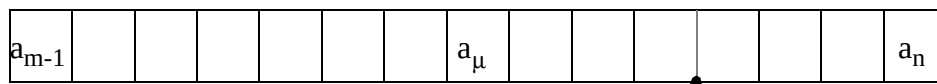
2 Prozessoren

2.1. Computerarithmetik

2.1.1. Zahlendarstellung (Wiederholung)

1 Dualzahlen

Grundlage: Polyadisches Zahlensystem, Stellungsnotation



$$\mu = 10 \quad 9 \quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \quad -1 \quad -2 \quad -3 \quad -4$$

b : Basis des Zahlensystems

a_μ : Ziffer an der Stelle μ . ($0 \leq a_\mu \leq b - 1$)

$$A = (a_{m-1} a_{m-2} \dots a \dots a_2 a_1 a_0 \cdot a_{-1} a_{-2} \dots a_n) = \sum_{\mu=n}^{m-1} a_\mu \cdot b^\mu$$

MSB (most significant Bit): $\mu = m - 1$

LSB (least significant Bit): $\mu = n$

speziell Dualzahlensystem: $b = 2$: $a_\mu = (0, 1)$

Festkommazahl: Dezimalkomma steht an festgelegter Stelle im Bitmuster und legt damit den Nullpunkt für die Stellenwertigkeit ($\mu = 0$) fest.

Integerzahl: $\mu \geq 0$

1 Negative Zahlen

Wortlänge: m bit ----> 2^m Bitmuster

nur positive Zahlen: $0 \leq A \leq 2^m - 1$

positive und negative Zahlen: möglichst 50% der Bitmuster für positive Zahlen, 50% für negative. (zzgl. Null!)

Symmetrie erwünscht.

a) Sign / Magnitude:

a_{m-1} für Vorzeichen:

$a_{m-1} = 0$: positiv

$a_{m-1} = 1$: negativ

Folge: +0, -0

komplizierte Subtraktion

b) Zweierkomplement:

Wortlänge m bit:

$$A + A_{2\text{Kompl}} = 2^m$$

Bildung des Zweierkomplements:

$$A_{2\text{Kompl}} = \bar{A} + 1$$

Zuordnung: MSB = 0 -----> positiv

MSB = 1 -----> negativ

Folge: 0 ist positiv.

Beispiel 2-2:

Wortlänge m = 32 bit

positive Zahlen: 0 bis $2^{31}-1$ negative Zahlen: -1 bis -2^{31} keine Symmetrie: -2^{31} darstellbar $+2^{31}$ nicht darstellbar

c) Excess K-Code

Zuordnung von 00...0 zur kleinsten darstellbaren Zahl, von 11...1 zur größten, d.h. Verschiebung um K (K = Verschiebung oder Bias):

$$A + K = \text{Bitmuster}$$

 $K \approx 2^{m-1}$, damit Aufteilung pos. / neg. symmetrisch wird.**Beispiel 2-3:**

m = 7 ----> 128 Bitmuster

K = 64

$$\Rightarrow -64 \leq A \leq 63$$

Bitmuster ₁₀	Bitmuster ₂	A
0	0000000	-64
1	0000001	-63
64	1000000	0
127	1111111	63

Anwendung: Floating Point Darstellung, Exponent

l Verwendung im Rechner

a) Integer-Zahlen (Deklaration: int) in Zweierkomplement-Darstellung, positiv und negativ.

b) Unsigned Integer-Zahlen (Deklaration: unsigned int) nur positiv, z.B. für Adressen.

Maschinenbefehle müssen beide Typen verarbeiten können, insbesondere Vergleichsbefehle.

1 Sign Extension

Problem: Verlängerung einer Immediate-Konstanten (z.B. 15 bit) auf volle Wortlänge (32 bit).

Lösung: Auffüllen nach links mit MSB.

Beispiel 2-4:

$$A = 2_{10}$$

$$\Rightarrow A = 0000\ 0010 \quad (8\text{ bit})$$

$$A = 0000\ 0000\ 0000\ 0010 \quad (16\text{ bit})$$

$$A = -2_{10}$$

$$\Rightarrow A = 1111\ 1101$$

$$\begin{array}{r} + \quad 1 \\ \hline \end{array}$$

$$1111\ 1110 \quad (8\text{ bit})$$

$$A = 1111\ 1111\ 1111\ 1110 \quad (16\text{ bit})$$

1 Addition und Subtraktion

Wie im Dezimalsystem: bitweise Addition mit Übertrag (Carry):

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0; \quad C = 1$$

Beispiel 2-5:

Addition: $A + B = ?$

$$A: \quad \quad \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1$$

$$B: \quad \quad \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0$$

$$\begin{array}{r} \text{Carry:} \quad \quad \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \\ \hline (0)0 \quad (0)0 \quad (0)1 \quad (1)1 \quad (1)0 \quad (0)1 \end{array}$$

Beispiel 2-6:

Subtraktion: $A - B = ?$

$$A: \quad \quad \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1$$

$$-B: \quad \quad \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0$$

$$\quad \quad \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1$$

Beispiel 2-7:

Subtraktion (Zweierkomplement):

$$\begin{array}{rcccccc}
 A: & 0 & 0 & 0 & 1 & 1 & 1 \\
 + B_{2K}: & 1 & 1 & 1 & 0 & 0 & 1 \\
 \hline
 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array}$$

l Overflow

A, B: int

$$\begin{array}{r}
 A: \quad 0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \\
 + B: \quad \underline{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010} \\
 \quad 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001
 \end{array}$$

Ergebnis: negativ (MSB = 1)

- 2 147 483 647 (falsch)

2 147 483 649 (richtig)

analog: negativer Overflow bei Subtraktion

Indikator für Overflow:

	A	B	Ergebnis
A + B	> 0	> 0	< 0
A + B	< 0	< 0	> 0
A - B	> 0	< 0	< 0
A - B	< 0	> 0	> 0

Maßnahmen bei Overflow: Exception (interner Interrupt), Weiterbehandlung durch SW.

2.1.2. Befehlssatz MIPS

Für die folgende Entwicklung von ALU, Datenpfad, Controller und Befehlspipeline werden Maschinenbefehle als Beispiele verwendet. Diese entsprechen dem Befehlssatz des MIPS-Mikroprozessors R2000. Sie sind ähnlich dem DLX-Befehlssatz (siehe Skript Rechnerstrukturen). Die Floating Point-Befehle sind nicht aufgeführt.

MIPS operands:

Name	Example	Comments
32 registers	\$0, \$1, \$2, . . . , \$31, Hi, Lo	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register \$0 always equals 0. Register \$1 is reserved for the assembler to handle pseudoinstructions and large constants. Hi and Lo are 32-bit registers containing the results of multiply and divide.
2 ³⁰ memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; exception possible
	subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; exception possible
	add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; exception possible
	add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	3 operands; no exceptions
	subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	3 operands; no exceptions
	add imm. un-sig.	addui \$1,\$2,100	$\$1 = \$2 + 100$	+ constant; no exceptions
	Move fr. copr. reg.	mfc0 \$1,\$epc	$\$1 = \epc	Used to get exception PC
	multiply	mult \$2,\$3	$Hi, Lo = \$2 \times \3	64-bit signed product in Hi,Lo
	multiply unsigned	multu \$2,\$3	$Hi, Lo = \$2 \times \3	64-bit unsigned product in Hi,Lo
	divide	div \$2,\$3	$Lo = \$2 / \$3, Hi = \$2 \bmod \3	Lo = quotient, Hi = remainder
	divide unsigned	divu \$2,\$3	$Lo = \$2 / \$3, Hi = \$2 \bmod \3	Unsigned quotient and remainder
	Move from Hi	mfhi \$1	$\$1 = Hi$	Used to get copy of Hi
	Move from Lo	mflo \$1	$\$1 = Lo$	Used to get copy of Lo

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	3 register operands; logical AND
	or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	3 register operands; logical OR
	and immediate	and \$1,\$2,100	$\$1 = \$2 \& 100$	Logical AND register, constant
	or immediate	or \$1,\$2,100	$\$1 = \$2 100$	Logical OR register, constant
	shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant
	shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant
Data transfer	load word	lw \$1,100(\$2)	$\$1 = \text{Memory}[\$2 + 100]$	Data from memory to register
	store word	sw \$1,100(\$2)	$\text{Memory}[\$2 + 100] = \1	Data from register to memory
	load upper imm.	lui \$1,100	$\$1 = 100 \times 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$1,\$2,100	if ($\$1 == \2) go to PC+4+100	Equal test; PC relative branch
	branch on not eq.	bne \$1,\$2,100	if ($\$1 \neq \2) go to PC+4+100	Not equal test, PC relative
	set on less than	slt \$1,\$2,\$3	if ($\$2 < \3) $\$1 = 1$; else $\$1 = 0$	Compare less than; 2's complement
	set on less than imm.	slti \$1,\$2,100	if ($\$2 < 100$) $\$1 = 1$; else $\$1 = 0$	Compare < constant; 2's comp.
	set less than uns.	sltu \$1,\$2,\$3	if ($\$2 < \3) $\$1 = 1$; else $\$1 = 0$	Compare less than; natural number
	set l. t. imm. uns.	sltiu \$1,\$2,100	if ($\$2 < 100$) $\$1 = 1$; else $\$1 = 0$	Compare < constant; natural
Unconditional jump	jump	j 10000	go to 10000	Jump to target address
	jump register	jr \$31	go to \$31	For switch, procedure return
	jump and link	jal 10000	$\$31 = \text{PC} + 4$; go to 10000	For procedure call

MIPS machine language

Name	Format	Example						Comments
		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	
add	R	0	2	3	1	0	32	add \$1,\$2,\$3
sub	R	0	2	3	1	0	34	sub \$1,\$2,\$3
addi	I	8	2	1	100			addi \$1,\$2,100
addu	R	0	2	3	1	0	33	addu \$1,\$2,\$3
subu	R	0	2	3	1	0	35	subu \$1,\$2,\$3
addui	I	9	2	1	100			addui \$1,\$2,100
mfc0	R	16	0	1	14	0	0	mfc0 \$1,\$epc
mult	R	0	2	3	0	0	24	mult \$2,\$3
multu	R	0	2	3	0	0	25	multu \$2,\$3
div	R	0	2	3	0	0	26	div \$2,\$3
divu	R	0	2	3	0	0	27	divu \$2,\$3
mfhi	R	0	0	0	1	0	16	mfhi \$1
mflo	R	0	0	0	1	0	18	mflo \$1
and	R	0	2	3	1	0	36	and \$1,\$2,\$3
or	R	0	2	3	1	0	37	or \$1,\$2,\$3
andi	I	12	2	1	100			andi \$1,\$2,100
ori	I	13	2	1	100			ori \$1,\$2,100
sll	R	0	0	2	1	10	0	sll \$1,\$2,10
srl	R	0	0	2	1	10	2	srl \$1,\$2,10
lw	I	35	2	1	100			lw \$1,100(\$2)
sw	I	43	2	1	100			sw \$1,100(\$2)
lui	I	15	0	1	100			lui \$1,100
beq	I	4	1	2	100			beq \$1,\$2,100
bne	I	5	1	2	100			bne \$1,\$2,100
slt	R	0	2	3	1	0	42	slt \$1,\$2,\$3
slti	I	10	2	1	100			slti \$1,\$2,100
sltu	R	0	2	3	1	0	43	sltu \$1,\$2,\$3
sltiu	I	11	2	1	100			sltiu \$1,\$2,100
j	J	2	10000					j 10000
jr	R	0	31	0	0	0	8	jr \$1
jal	J	3	10000					jal 10000

MIPS machine language

Format R	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
Format I	I	op	rs	rt	address/immediate			Transfer, branch, imm. format
Format J	J	op	target address					Jump instruction format
Field size		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	All MIPS instructions 32 bits

op und funct:	Opcode	6 + 6 bit
rs:	Source Register	5 bit
rt:	Source Register (meist)	5 bit
rd:	Destination Register	5 bit
shamt:	Shift amount	5 bit

Beispielzahlen in der Tabelle: dezimal.

2.1.3. ALU- Entwurf

Es soll eine Arithmetik-Logik-Einheit (ALU) entworfen werden, welche folgende MIPS-Befehle unterstützt:

a) Logical

and, or, and immediate, or immediate, not

b) Arithmetic

add, sub

c) Conditional set

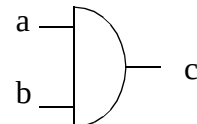
set less than unsigned

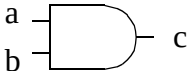
set less than immediate unsigned

HW-Bausteine:

1. AND gate

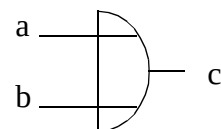
$$c = a \wedge b \text{ oder } c = a \cdot b$$



US: 

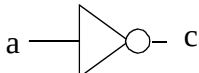
2. OR gate

$$c = a \vee b \text{ oder } c = a + b$$



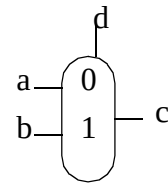
US: 

3. Inverter

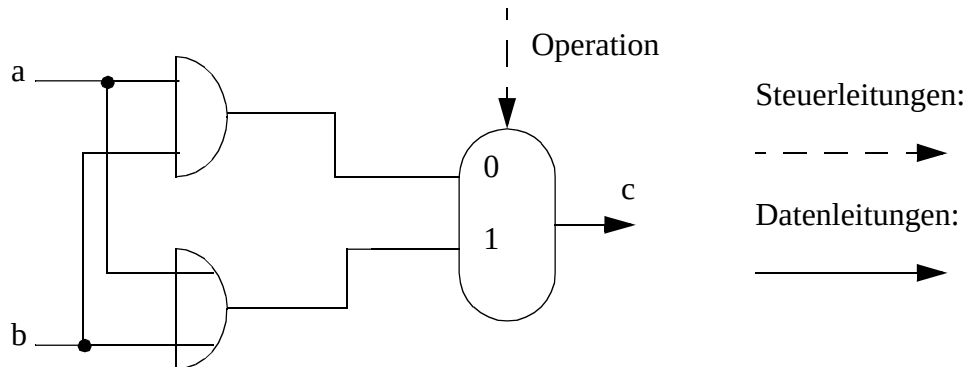
$$c = \bar{a}$$


4. Multiplexer

if (d == 0) c := a;
else c := b

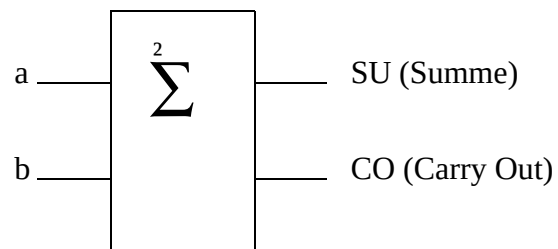


1 1 bit-(A)LU (nur Logikfunktionen AND und OR):



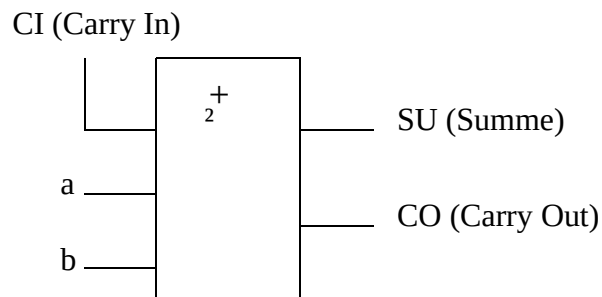
1 Addierer

Halbaddierer (2,2 - Addierer)



a	b	SU	CO
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Volladdierer (3,2 - Addierer)



$$SU = (a \cdot \bar{b} \cdot \bar{CI}) + (\bar{a} \cdot b \cdot \bar{CI}) + (\bar{a} \cdot \bar{b} \cdot CI) + (a \cdot b \cdot CI)$$

$$CO = (b \cdot CI) + (a \cdot CI) + (a \cdot b)$$

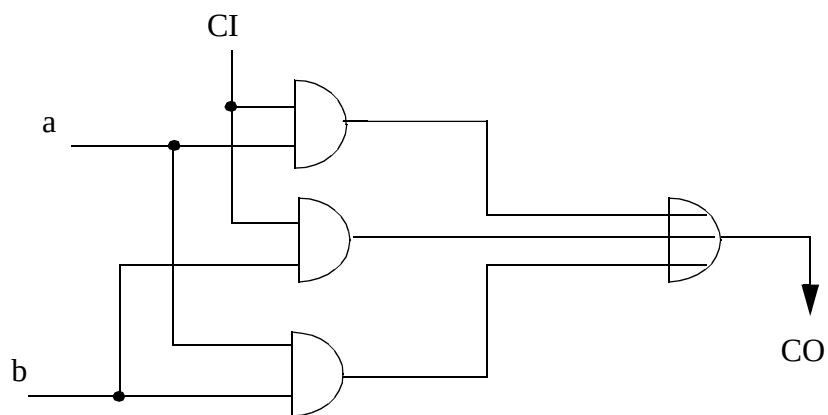


Bild 2.1. CO-Schaltung

Kombination der Elemente und Erweiterung des Multiplexers ergibt:

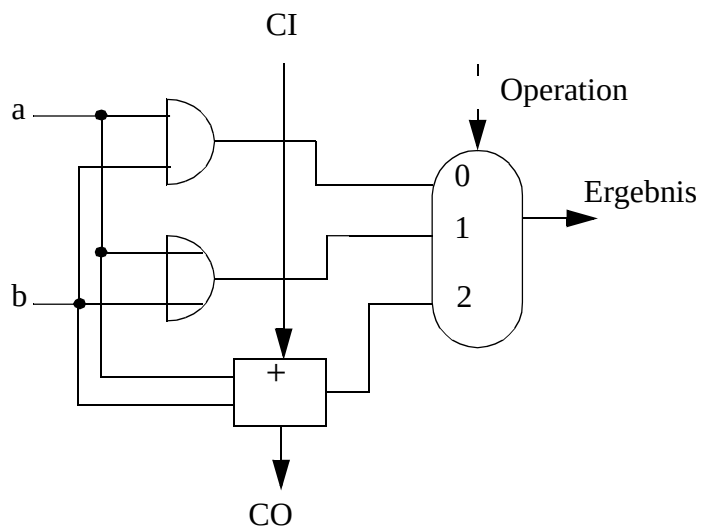


Bild 2.2. 1 bit-ALU (AND, OR, Addition)

- 1 Funktionserweiterung der ALU durch Erweiterung des MUX und Hinzufügen der zusätzlichen Logik.

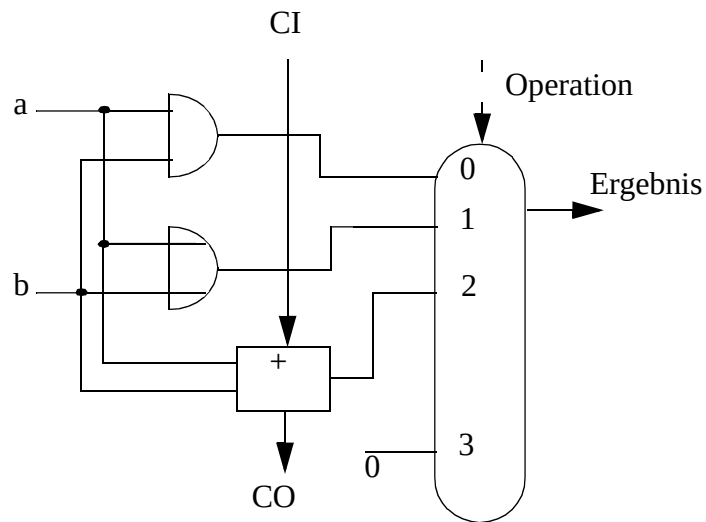


Bild 2.3. Hinzufügen von "generate 0"

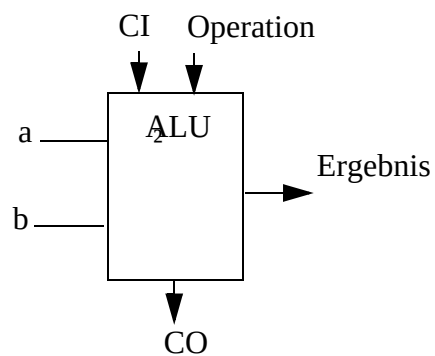


Bild 2.4. Symbol 1 bit-ALU

1 32 bit-ALU

Kombination von $32 \cdot 1$ bit-ALU-Blöcken durch Verbindung von CO_i mit CI_{i+1} .

Folge: Carry durchläuft Bits 0 bis 31 (ripple through).

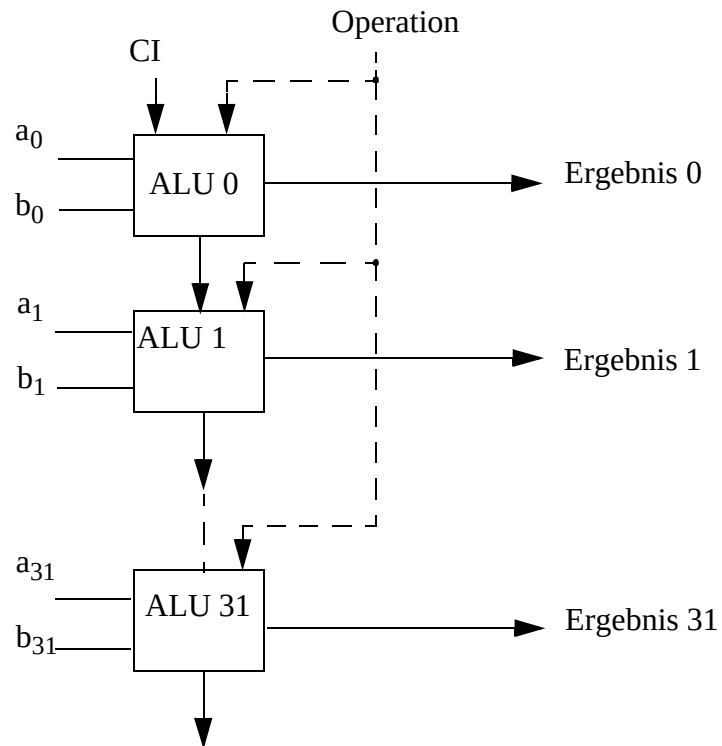


Bild 2.5. 32 bit-Ripple Carry ALU

Zeitbedarf für Addition wird bestimmt durch Carry-Durchlauf:

$$t_{\text{Add}} = 32 \cdot 2 \cdot t_d$$

mit t_d = Schaltzeit für AND, OR, INV

1 Erweiterung um Subtraktion

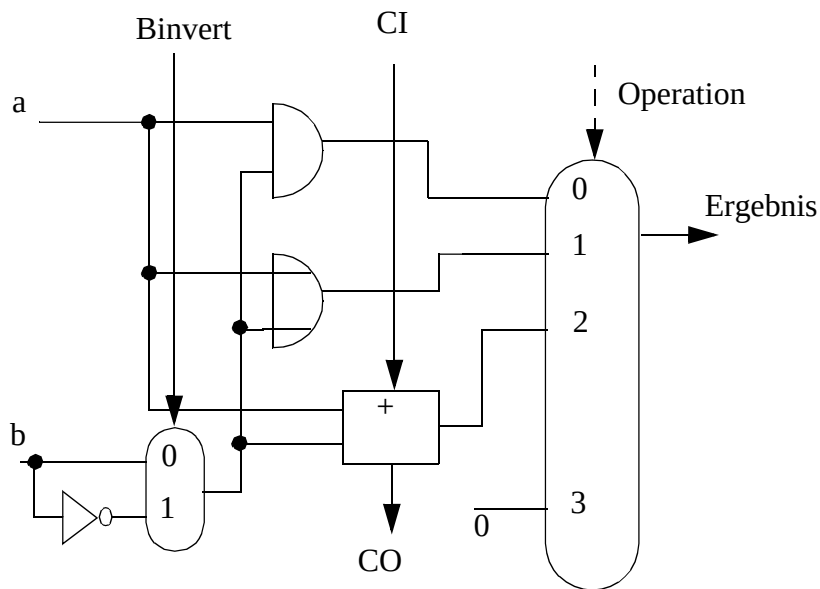


Bild 2.6. Erweiterung um Subtraktion

Soll b subtrahiert werden, so ist es zu invertieren (Inverter, 2:1-MUX). Zum Gesamtergebnis ist 1 zu addieren:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

Subtraktion: Steuersignal $\text{Binvert} = 1$ und $\text{CI}_0 = 1$.

1 Erweiterung um Vergleichsoperationen

Maschinenbefehl: set-on-less-than

`slt $1, $2, $3` if ($\$2 < \3) $\$1 = 1$; else $\$1 = 0$

Realisierung durch Subtraktion und Auswertung des MSB (Vorzeichen).

`slt` benutzt das R-Format, daher

$\text{Rs} \leftarrow \$2$

$\text{Rt} \leftarrow \$3$

Wenn $\text{Rs} < \text{Rt} \Rightarrow \text{Rs} - \text{Rt} < 0 \Rightarrow \text{MSB} = 1$.

Maßnahmen:

1. ALU-Erweiterung um Less-Eingang
2. ALU-Erweiterung um Adder-Ausgang (Set)
3. Verbinde Set-Ausgang 31 mit Less-Eingang 0
4. Bits 0 - 30 : 0 an Less-Eingang

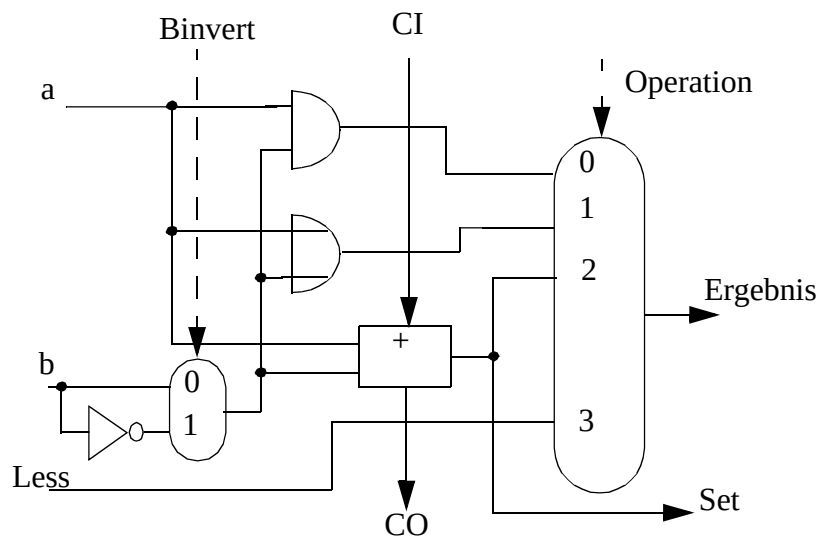


Bild 2.7. 1 bit-ALU mit slt-Erweiterung (nur für Bit 31). Die 1 bit-ALUs für die Bits 0 bis 30 brauchen keinen Set-Ausgang.

1 Overflow

Die MSB-ALU benötigt außerdem eine Logik zur Erkennung von Overflow-Bedingungen. Overflow kann auch den Set-Ausgang verfälschen und ist entsprechend zu berücksichtigen! (Hier nicht behandelt.)

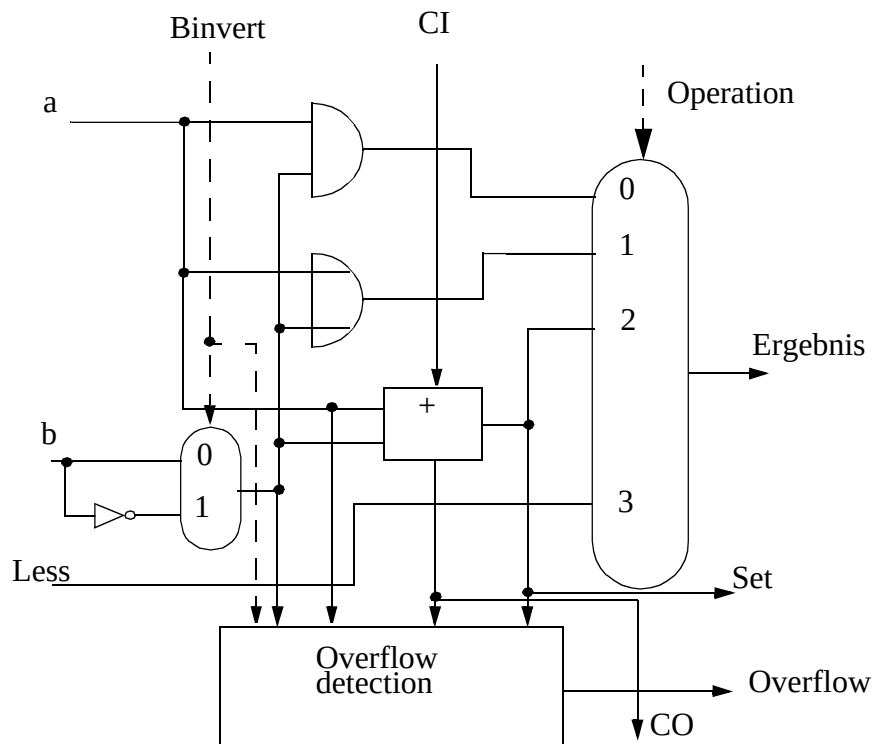


Bild 2.8. MSB-ALU mit Overflow-Logik

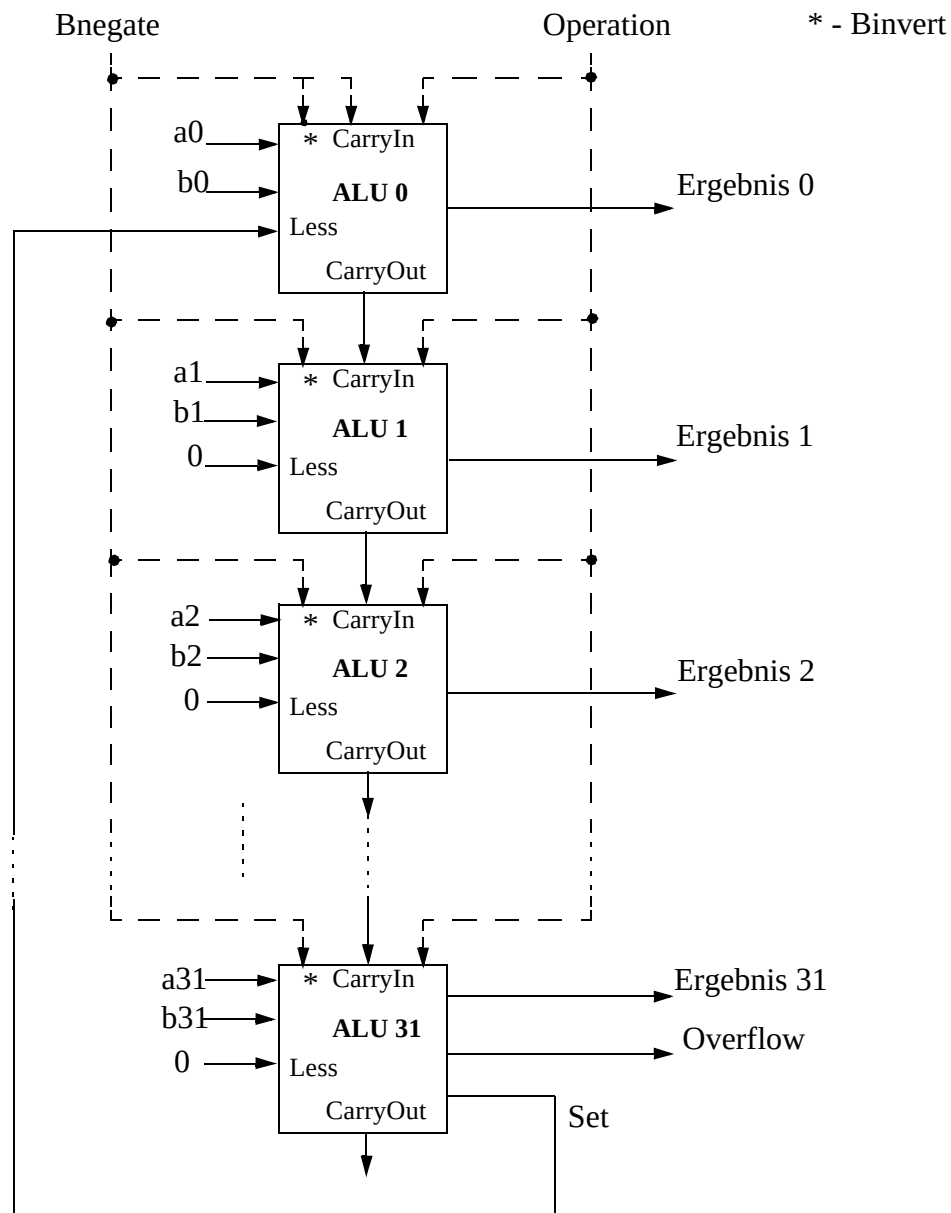


Bild 2.9. 32 bit-ALU (Ripple Carry)

CarryIn für ALU 0 wird genau dann aktiviert (auf 1 gesetzt), wenn b subtrahiert werden soll, wenn also auch Binvert = 1 ist. Daher wird Binvert und CarryIn 0 verbunden und mit dem Steuersignal Bnegate angesteuert.

1 Zero Detection

Zusätzlicher ALU-Ausgang Zero zur Null-Erkennung, z.B. für bedingte Sprünge, if a = b => a - b = 0, Zero = 1.

Erkennung durch NOR-Gatter über alle ALU-Ergebnis-Ausgänge:

$$\text{Zero} = \overline{(\text{Erg } 0 \vee \text{Erg } 1 \vee \dots \vee \text{Erg } 31)}$$

1 Zusammenfassung: 32 bit-ALU-Symbol

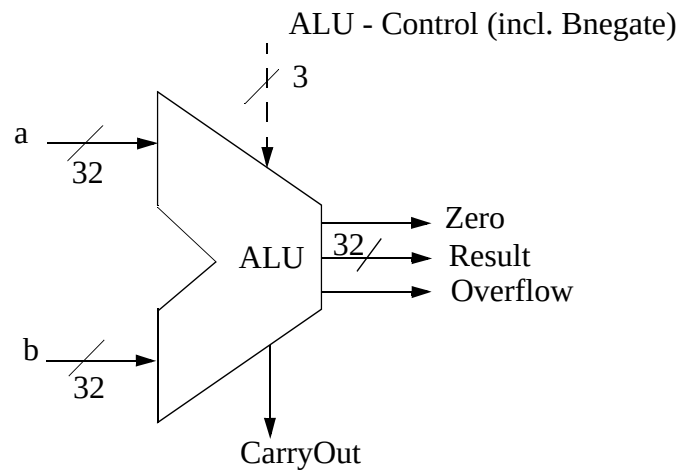


Bild 2.10. 32 bit-ALU

Für die Steuerung der ALU werden also 3 Signale (Bnegate, Operation 1, Operation 0) benötigt:

ALU-Steuerleitungen (ALU-Control)	Funktion
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

2.1.4. Beschleunigung der Addition

Ripple Carry ist zu langsam.

Lösung 1: Jede Schaltfunktion lässt sich als DNF darstellen => Schaltzeit = $2t_d$.

Aber: Funktionstabelle hat $\geq 2^{64}$ Zeilen!

Lösung 2: Berechnung von Carry zwischen Bitgruppen, Ripple innerhalb der Bitgruppen.

1 Carry Lockahead Adder (CLA)

Definition zweier Signale für Bit i:

$$\text{generate} \quad g_i = a_i \cdot b_i$$

$$\text{propagate} \quad p_i = a_i + b_i$$

$g_i = 1$: Addiererstufe i erzeugt ein CarryOut unabhängig von CarryIn.

$p_i = 1$: Addierstufe i reicht ein CarryIn = 1 als CarryOut =1 weiter.

Damit wird

$$ci_1 = g_0 + (p_0 \cdot ci_0)$$

$$ci_2 = g_1 + (p_1 \cdot g_0) + (p_1 \cdot p_0 \cdot ci_0)$$

$$ci_3 = g_2 + (p_2 \cdot g_1) + (p_2 \cdot p_1 \cdot g_0) + (p_2 \cdot p_1 \cdot p_0 \cdot ci_0)$$

usw.

Komplexität wächst schnell!

Zusammenfassung in 4 bit-Gruppen, wobei jede Gruppe ein Propagate-($P_{i,j}$) und Generate-($G_{i,j}$) Signal besitzt:

Propagate: Alle vier 1 bit-Adder müssen weiterreichen, damit $P_{i,j} = 1$:

$$P_{0,3} = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

$$P_{4,7} = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

$$P_{8,11} = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$

$$P_{12,15} = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

Generate: $G_{i,j} = 1$, wenn ein Carry aus irgendeinem der 1-bit-Adder ankommt:

$$G_{0,3} = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$$

$$G_{4,7} = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)$$

$$G_{8,11} = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8)$$

$$G_{12,15} = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})$$

Die Carry-Signale zwischen den 4-bit-Gruppen (c_4, c_8, c_{12}, c_{16}) errechnen sich nach folgenden Gleichungen:

$$c_4 = G_{0,3} + (P_{0,3} \cdot c_0)$$

$$c_8 = G_{4,7} + (P_{4,7} \cdot G_{0,3}) + (P_{4,7} \cdot P_{0,3} \cdot c_0)$$

$$c_{12} = G_{8,11} + (P_{8,11} \cdot G_{4,7}) + (P_{8,11} \cdot P_{4,7} \cdot G_{0,3}) \\ + (P_{8,11} \cdot P_{4,7} \cdot P_{0,3} \cdot c_0)$$

$$c_{16} = G_{12,15} + (P_{12,15} \cdot G_{8,11}) + (P_{12,15} \cdot P_{8,11} \cdot G_{4,7}) \\ + (P_{12,15} \cdot P_{8,11} \cdot P_{4,7} \cdot G_{0,3}) + (P_{12,15} \cdot P_{8,11} \cdot P_{4,7} \cdot P_{0,3} \cdot c_0)$$

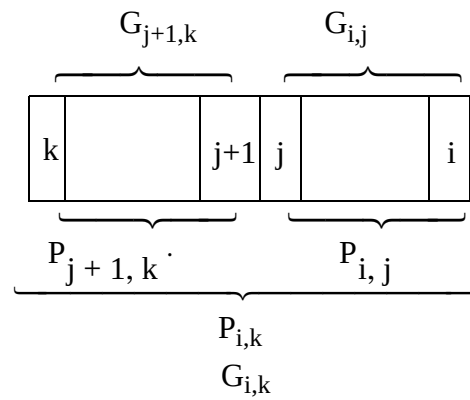
Es gilt für $i < j, j+1 < k$: (Konkatenierungssätze)

$$P_{i,k} = P_{i,j} \cdot P_{j+1,k} \quad (*)$$

$$G_{i,k} = G_{j+1,k} + P_{j+1,k} \cdot G_{i,j} \quad (**)$$

und

$$P_{i,i} = p_i; \quad G_{i,i} = g_i$$

**Beispiel 2-8:**

$$\text{Ges.: } G_{0,3} = f(p_i, g_i)$$

$$P_{0,3} = f(p_i, g_i)$$

$$P_{0,3} = P_{0,1} \cdot P_{2,3} = P_{0,0} \cdot P_{1,1} \cdot P_{2,2} \cdot P_{3,3} = p_0 \cdot p_1 \cdot p_2 \cdot p_3$$

$$G_{0,3} = G_{2,3} + P_{2,3} \cdot G_{0,1}$$

$$= (G_{3,3} + P_{3,3} \cdot G_{2,2}) + (P_{2,2} \cdot P_{3,3}) \cdot (G_{1,1} + P_{1,1} \cdot G_{0,0})$$

$$= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Mit den rekursiven Gleichungen (*) und (**) können baumartig die höheren $G_{i,j}$ und $P_{i,j}$ berechnet werden:

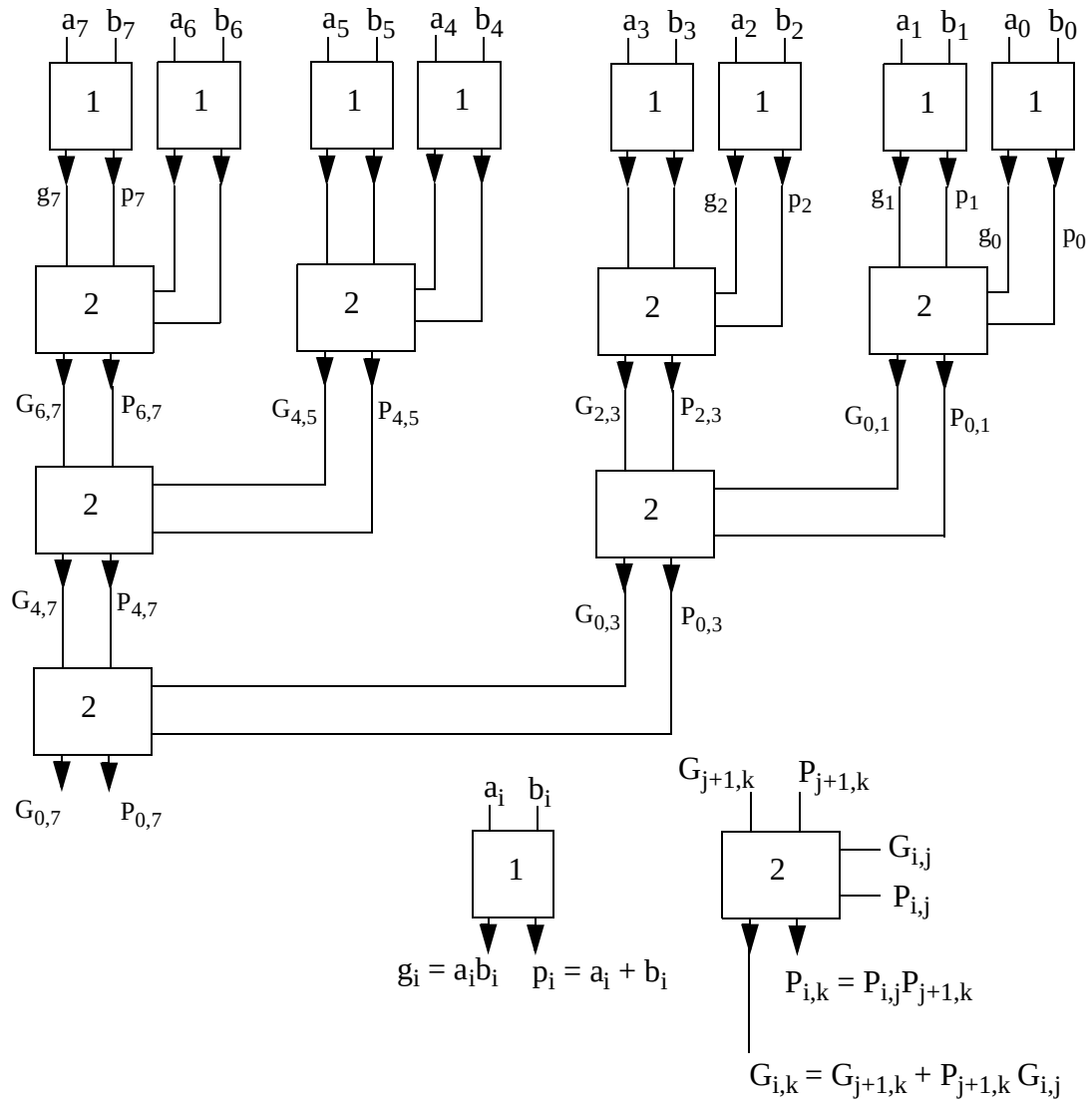


Bild 2.11. Generate / Propagate - Baum (Teil 1) eines CLA.

Zahlen in den Kästchen: Anzahl t_d .

Aus den $P_{i,j}$ und $G_{i,j}$ - Signalen errechnet sich das jeweilige CarryIn c_{k+1} :

$$c_{k+1} = G_{i,k} + P_{i,k} \cdot c_i \quad (***)$$

Die c_{k+1} ergeben sich also durch Durchlaufen des G/P-Baumes von unten nach oben:

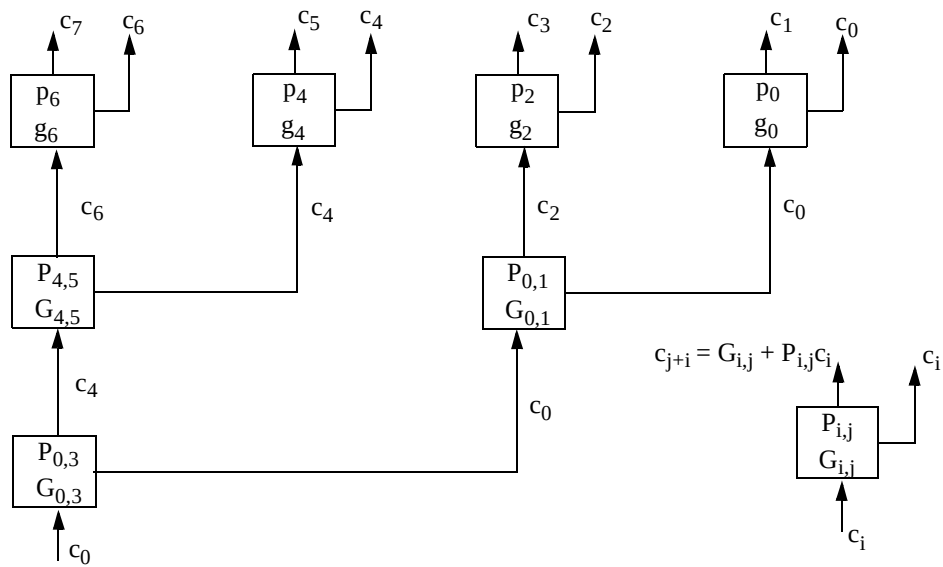


Bild 2.12. Rückwärts-Durchlauf durch den G/P-Baum zur Erzeugung der c_i (Teil 2).

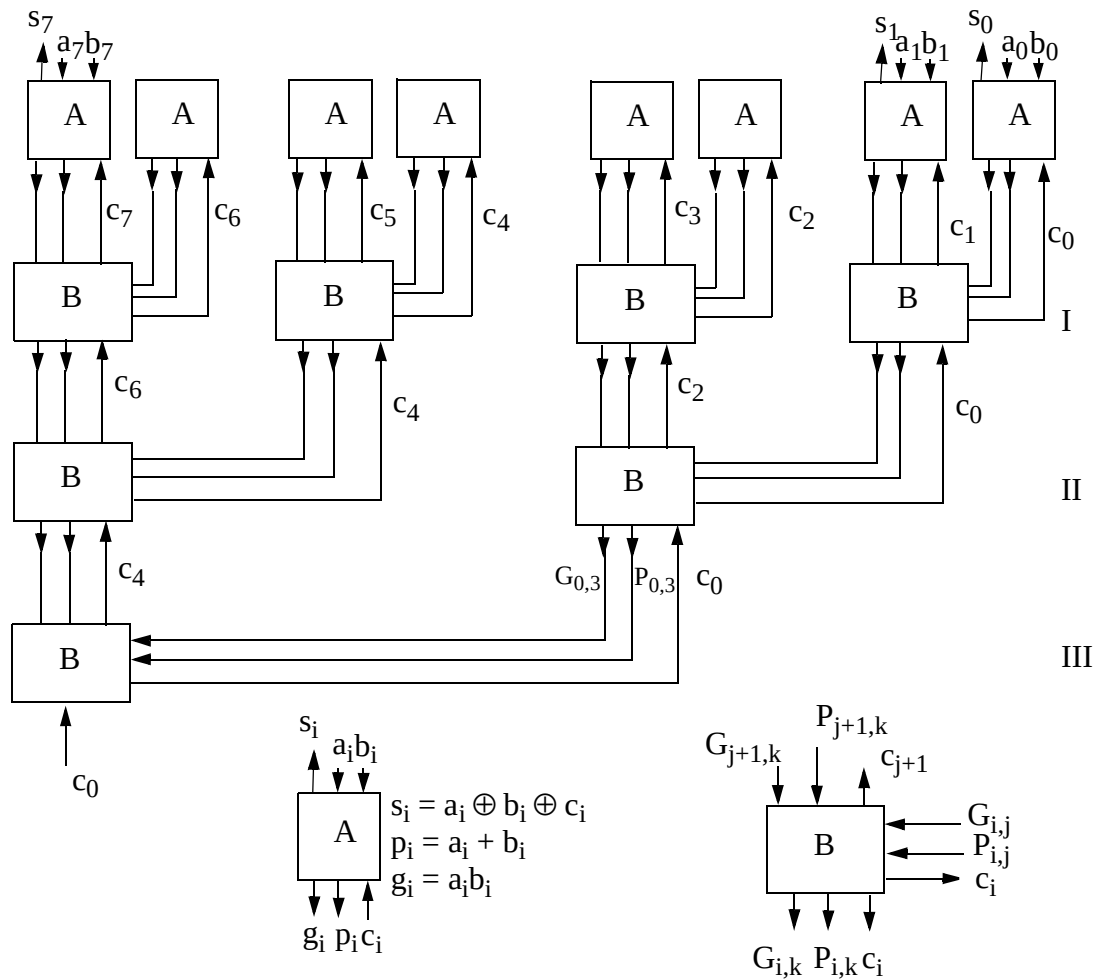


Bild 2.13. Gesamter CLA - Baum.

Je nach Zeitbedarf für die Schaltungen A, B und C kann es auch günstiger sein, innerhalb der 4 bit-Blöcke zu rippeln und außerhalb mit Lookahead-Technik zu arbeiten.

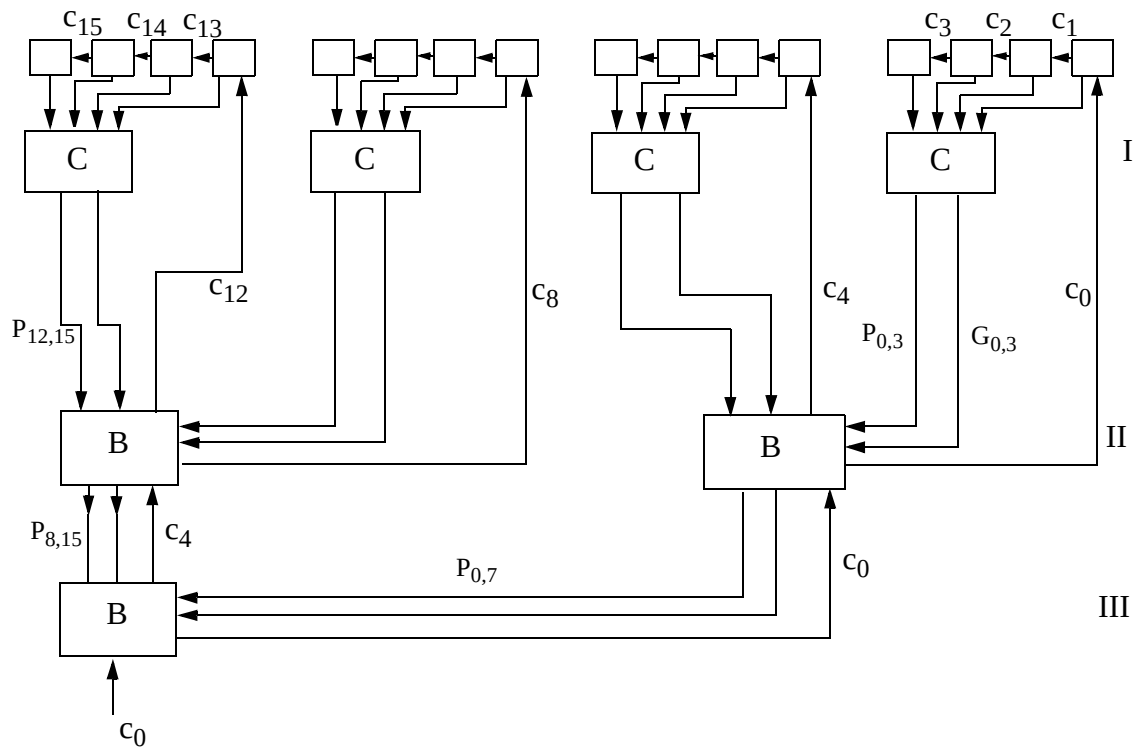


Bild 2.14. Kombination von Ripple Carry und CLA.

I Zeitbedarf für 8 bit-Addierer

Ripple-Carry: $t_{\text{add}}^{\text{RC}} = 8 \cdot 2 \cdot t_d$

CLA:

Abstieg:	$5 t_d$
Aufstieg:	$6 t_d$
MSB-Addition:	$2 t_d$
$t_{\text{add}} = 13 t_d$	

allg.: Ripple Carry: $t_{\text{add}}^{\text{RC}} = n \cdot 2 t_d [O(n)]$
 $t_{\text{add}}^{\text{CLA}} \approx \log(n) \cdot 2 \cdot 2 t_d [O(\log n)]$

I Aufwand/Nutzen-Trade-off durch Carry Skip Adder und Carry Select Adder.

Carry Skip: $G_{i,j}$ -Berechnung ist aufwendiger als P-Berechnung.

Daher: c_{in} überspringt einen Addierer-Block, wenn $P = 1$.

Voraussetzung: c_{out} jedes Blocks muß zu Beginn auf 0 gesetzt sein.

Geschwindigkeit: $O(\sqrt{n})$. Aufwand: $O(n)$.

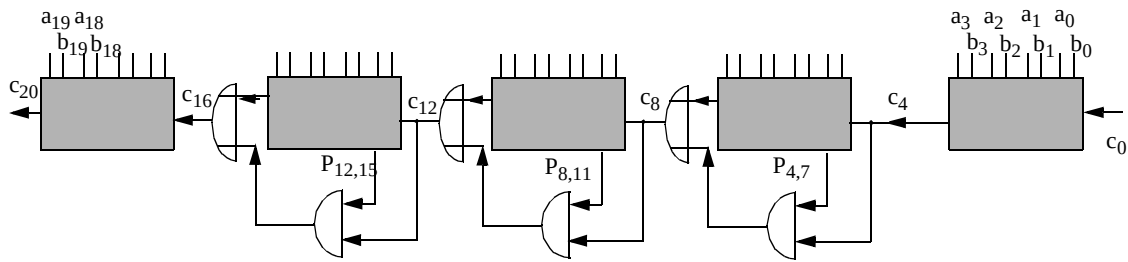


Bild 2.15. Carry Skip Addierer.

Carry Select: Idee: Höherwertige Hälfte des Addierers wird doppelt ausgeführt, berechnet (1) Ergebnis für $c_{in} = 0$, (2) für $c_{in} = 1$. Das wirkliche c_{in} aus der unteren Hälfte selektiert dann das richtige Ergebnis.

Geschwindigkeit: $O(\sqrt{n})$.

Aufwand: $O(n)$.

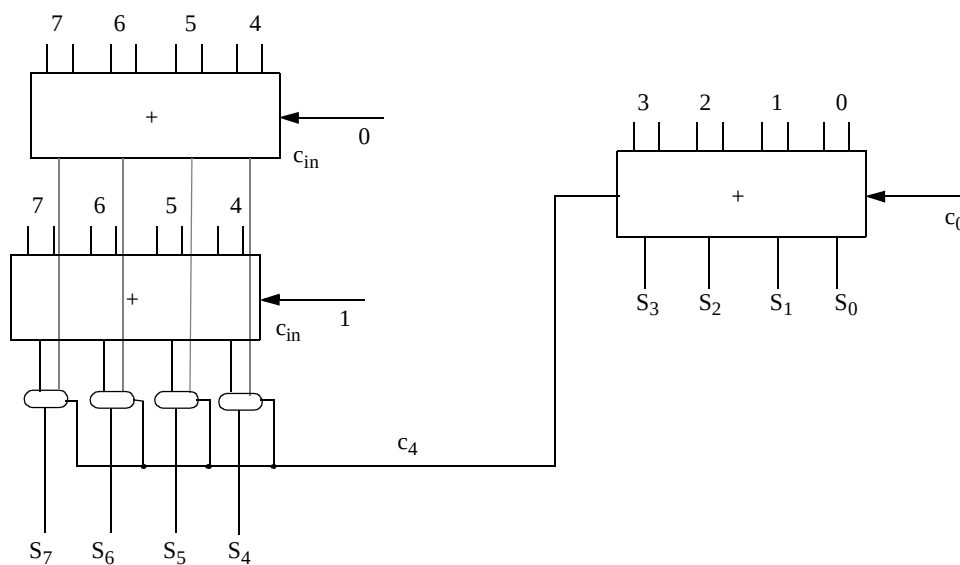


Bild 2.16. Carry Select Addierer (n = 8 bit).

2.1.5. Multiplikation

Algorithmus wird in 3 Stufen entwickelt.

Ausgangspunkt: manuelle Multiplikation.

Beispiel 2-9:

Dezimalzahl nur aus 0 und 1 bestehend:

$$\begin{array}{r}
 1000_{10} \quad * \quad 1001_{10} \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 1001000_{10}
 \end{array}$$

Multiplikand * Multiplikator = Produkt

Platzbedarf Produkt: Multiplikand: m bit
 Multiplikator: n bit
 Produkt: n + m bit

Algorithmus:

Für alle Ziffern des Multiplikators (von rechts nach links):

if Ziffer = 1: addiere Multiplikand

if Ziffer = 0: addiere 0

verschiebe Multiplikand um eine Stelle nach links.

l Version 1

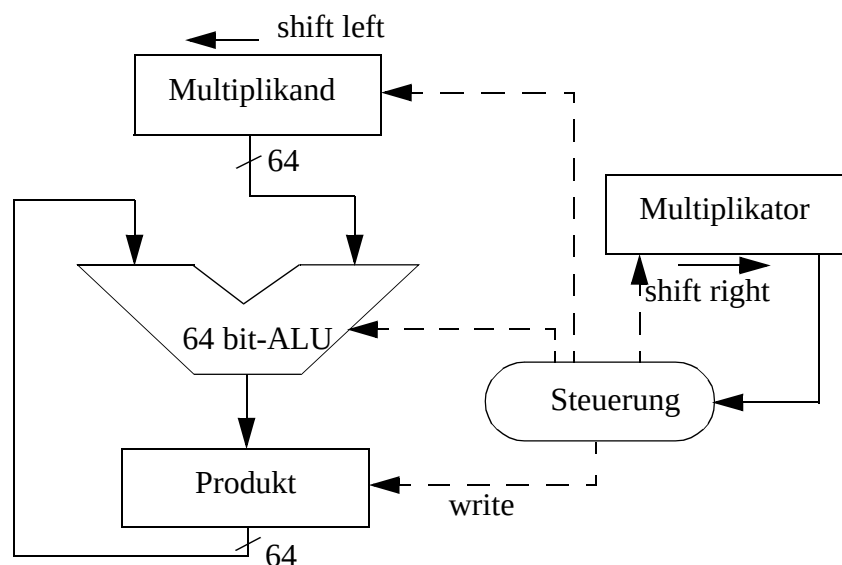


Bild 2.17. Multiplizierer-Hardware (Version 1) für n = m = 32 bit.

Multiplikand (32 bit) ist zu Beginn rechts ausgerichtet, 32 MSBs = 0. Pro Runde wird er um 1 bit nach links geschoben.

LSB des Multiplikators bestimmt, ob 0 oder Multiplikand addiert wird.

Zeitbedarf: $(\text{Test Multiplikator} = 0; \text{Add; Shift}) * 32 \Rightarrow \text{ca. 100 Takte.}$

1 Version 2

Äquivalenter Algorithmus: Statt den Multiplikanden nach links zu schieben wird das Produkt nach rechts geschoben. ALU liefert jeweils einen 32 bit breiten Ausschnitt des Produkts, LSBs des Produkts bleiben unverändert.

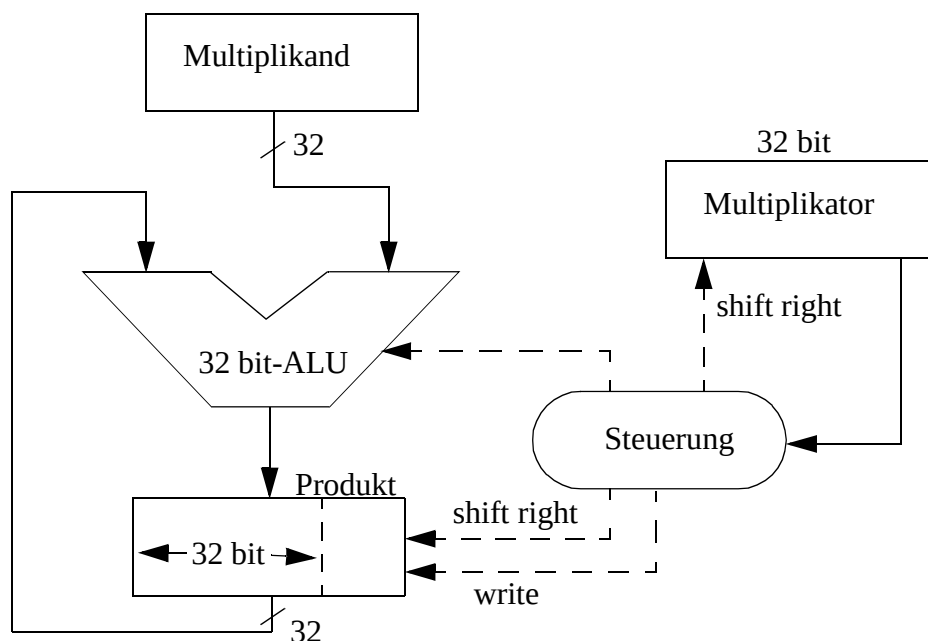


Bild 2.18. Multiplizierer-Hardware (Version 2) für $n = m = 32$ bit.

Ergebnis: HW-Ersparnis, nur 32 bit-Multiplikanden-Register und 32 bit-ALU nötig.

1 Version 3

Beobachtung: Rechter Teil des Produktregisters enthält immer 0 (zu Anfang 32 bit, pro Runde 1 bit weniger).

Dieser Platz kann für den Multiplikator genutzt werden.

Ergebnis: 32 bit-Multiplikator-Register entfällt.

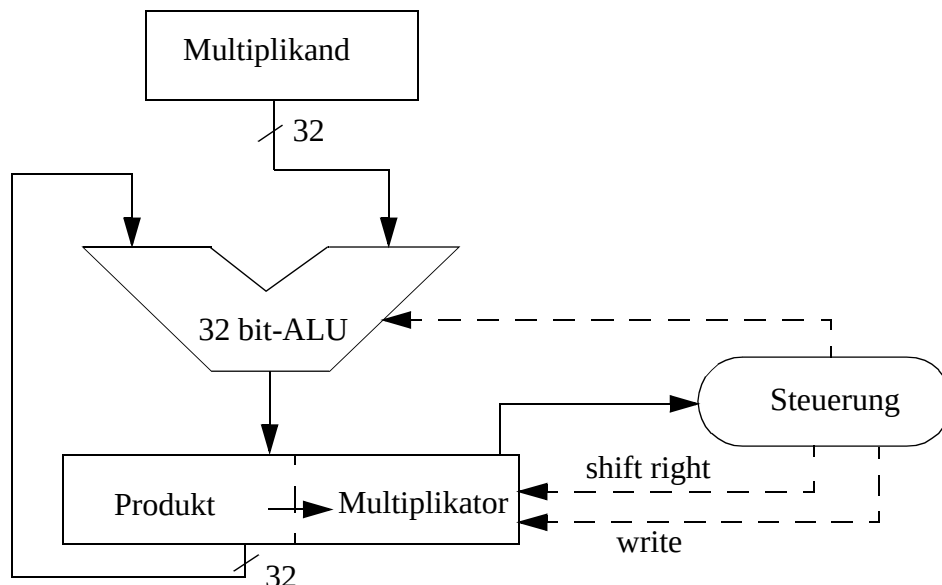


Bild 2.19. Multiplizierer-Hardware (Version 3) für $n = m = 32$ bit.

1 Booth's Algorithmus

Annahme: Multiplikator besteht nur aus 11...1. => Es finden n Additionen des Multiplikanden statt.

Annahme: Multiplikator = 100...0. => Multiplikand wird 1 mal addiert und $n-1$ mal nach links geschoben.

Daher: Es wird nicht mit 11...1 multipliziert, sondern mit 100...0 (d.h. 1 Addition zu viel). Korrektur durch Subtraktion.

Die Überlegung gilt analog für Folgen von 1 innerhalb des Multiplikators.

Realisierung: HW muß 0-Strings und 1-Strings erkennen.

Dazu: Auswertung von 2 bit-Ausschnitt (aktuelles Bit, Bit rechts davon).

Algorithmus:

Teste (aktuelles Bit, Bit rechts davon)

00: innerhalb 0-String, keine arithmetische Operation

01: Ende eines 1-String. Addiere Multiplikand zur linken Hälfte des Produktes.

10: Anfang eines 1-Strings. Subtrahiere Multiplikand von der linken Hälfte des Produkts (Korrekturschritt).

11: innerhalb eines 1-Strings, keine arithmetische Operation.

aktuelles Bit	Bit rechts	Erläuterung	Beispiel
1	0	Anfang 1-String	0000 1111 0 00
1	1	innerhalb 1-String	0000 111 1 000
0	1	Ende 1-String	0000 1 111 000
0	0	innerhalb 0-String	0000 1111 000

Problem: Beschleunigungseffekt um so größer, je längere 1-Strings vorhanden. Für alternierende Bitfolgen (010101...) ist der Algorithmus langsamer!

2.1.6. Division

- zu ergänzen -

2.1.7. Floating Point Unit

- zu ergänzen -

2.2. Datenpfad

Im folgenden soll ein einfacher Datenpfad aufgebaut werden, welcher folgende MIPS-Instruktion unterstützt:

- Speicherzugriff lw und sw
- ALU-Befehle add, sub, and, or, slt
- Sprünge beq und j

Wichtig: Einfachheit und Regularität des Instruktionsatzes ermöglichen einen einfachen Aufbau der HW.

RISC-Grundregeln:

- Mache die häufigen Fälle schnell!
- Einfachheit ermöglicht Regularität.

Sprachregelung: Signale können beim Pegel 0V true oder false sein.

Eindeutige Ausdrucksweise:

Signal aktiv = true = asserted

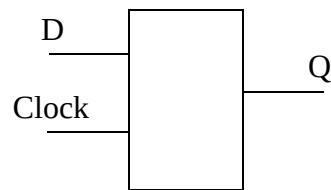
Signal inaktiv = false = deasserted

Zuordnung: Pegel <-> Zustand ist technologieabhängig.

2.2.1. Taktschemata

Prozessoren benutzen sequentielle Schaltungen. Diese lassen sich aufteilen in speichernde und kombinatorische Elemente.

Speichernde Elemente enthalten den Zustand des Systems. Beispiele: Register, Datenspeicher, Befehlsspeicher. Statische Speicherelemente werden mit Flip Flops realisiert.



Der Takt (Clock) bestimmt den Zeitpunkt der Übernahme der Information am D-Eingang zum Ausgang Q. Ein Speicherelement kann immer gelesen werden.

Ein Taktschema legt eine Abfolge von Zeitpunkten für das Lesen und Schreiben fest.

Hier Annahme: flankengesteuerte FFs, d.h. Übernahme erfolgt zum Zeitpunkt der steigenden (oder fallenden) Taktflanke.

Alternative (hier nicht): zustandsgesteuert.

Folge: Speicher- und Kombinatorikelemente wechseln einander ab:

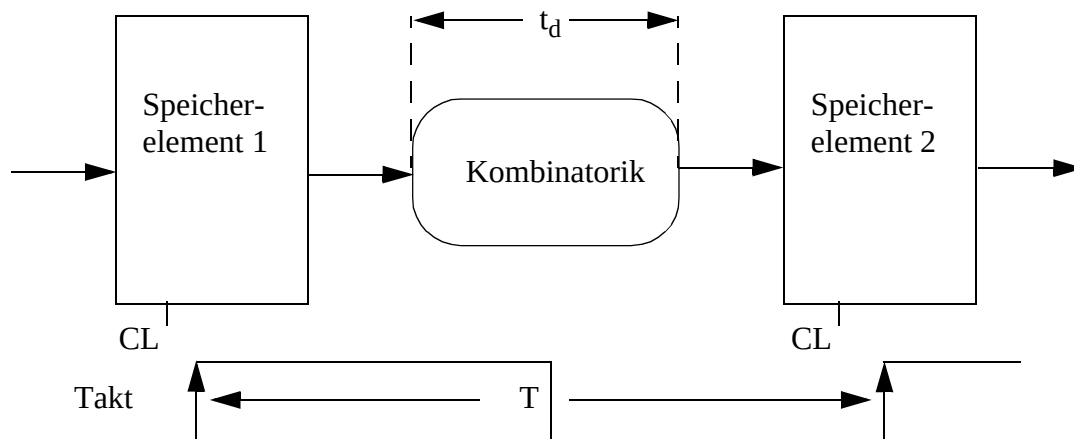


Bild 2.20. Ein-Takt-Anordnung: Steigende Flanke ist aktiv. Zu diesem Zeitpunkt wird das am Eingang anliegende Signal in den Speicher übernommen und erscheint am Ausgang. Die Taktperiode T muß mindestens so groß sein wie die Durchlaufzeit t_d durch die Kombinatorik.

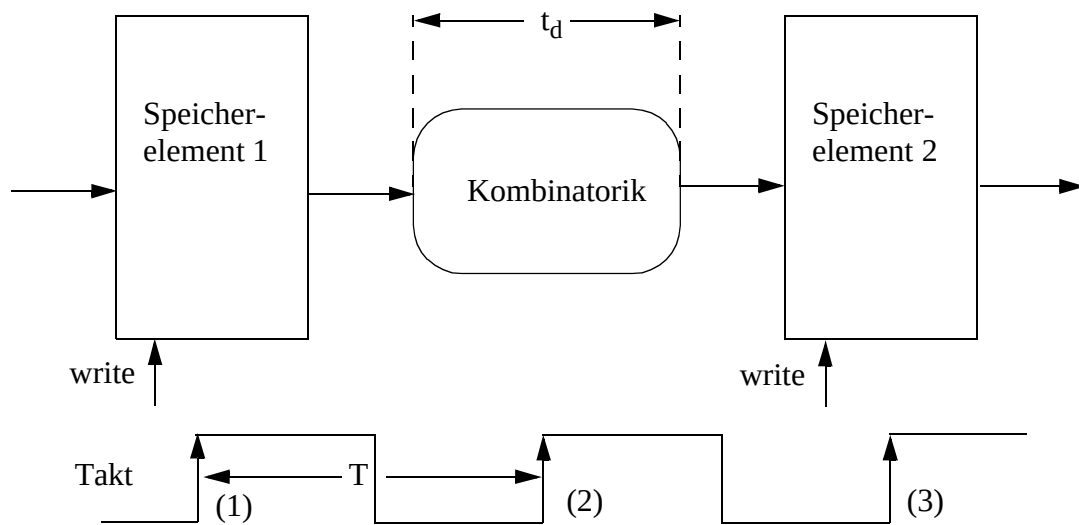


Bild 2.21. Mehr-Takt-Anordnung: Kombinatorik benötigt ≥ 1 Takt. Write-Signal nötig, welches nur zu den Zeitpunkten (1) und (3) aktiviert wird.
Wichtig: Output von Speicherelement 1 muß stabil bleiben bis 3! $t_d > T$.

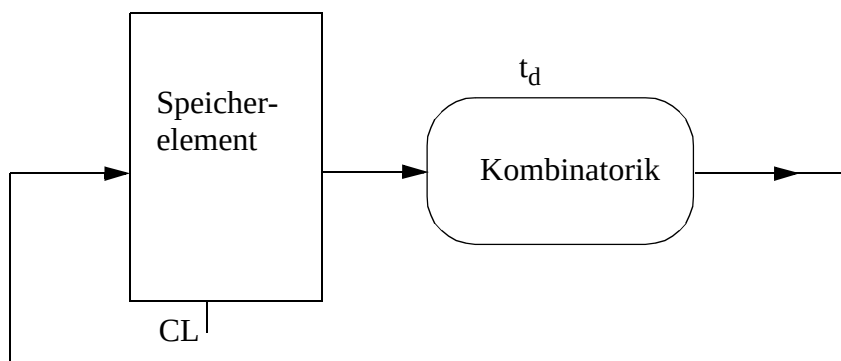


Bild 2.22. Zulässige Anordnung bei flankengetriggerten FFs. Zum Zeitpunkt der aktiven Flanke wird der (stabile) Kombinatorikoutput in den Speicher übernommen.

2.2.2. Ein-Takt-Datenpfad

Durch die während einer Befehlsausführung ablaufenden Aktionen (Befehlszyklus) wird ein grundsätzlicher Aufbau wie folgt nahegelegt:

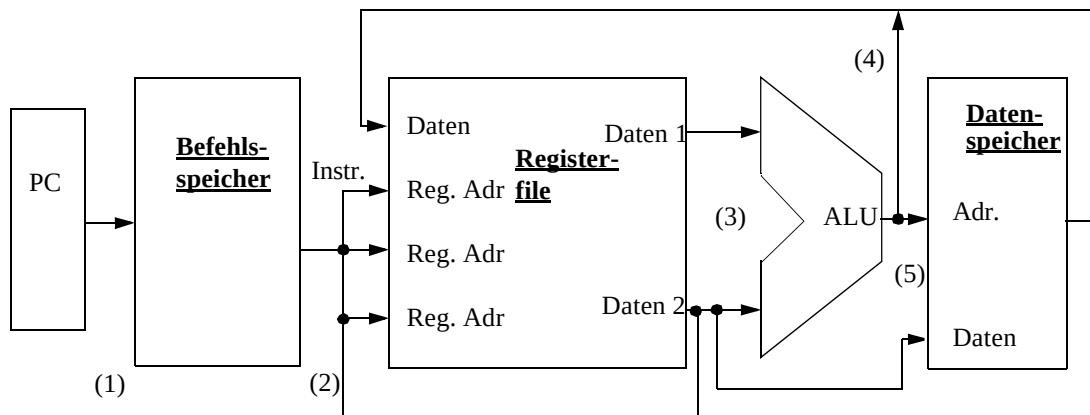


Bild 2.23. Datenpfad-Schema. (1) PC adressiert Befehlsspeicher. (2) Registeradressen aus dem Befehl (2 * Read, 1 * Write) werden an das Registerfile angelegt. (3) Registerinhalte werden an ALU-Eingang gelegt. (4) ALU Ergebnis wird in Register zurückgeschrieben. (5) Nur bei Store-Befehl: Registerinhalt wird in Datenspeicher geschrieben.

Der Datenpfad wird nun im Detail stufenweise aufgebaut. Wir verfolgen dabei den Ablauf einzelner Befehle.

- 1 IFETCH (gleich für alle Befehlstypen)
Beteiligte Elemente: Befehlsspeicher, PC, Addierer.

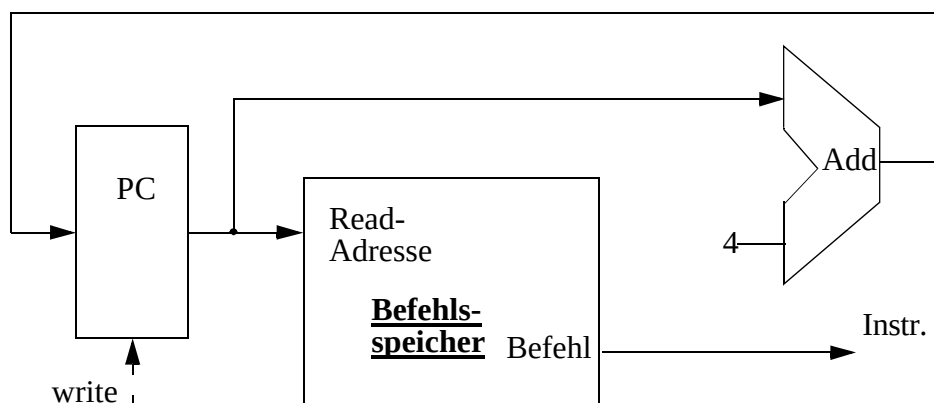


Bild 2.24. Teil-Datenpfad für IFETCH:

```
Instr <-- Mem[PC]
PC <-- PC + 4
```

1 R-Format-Befehle (add, sub, and, or, slt)

z.B. add \$1,\$2,\$3; /* \$1 <-- \$2 + \$3

Zugriff auf 3 Register nötig, d.h.

2 Read Register Adressen (Inputs)

1 Write Register Adresse (Input)

1 Write Daten (Input)

2 Read Daten (Output)

1 Write Steuersignal (Input)

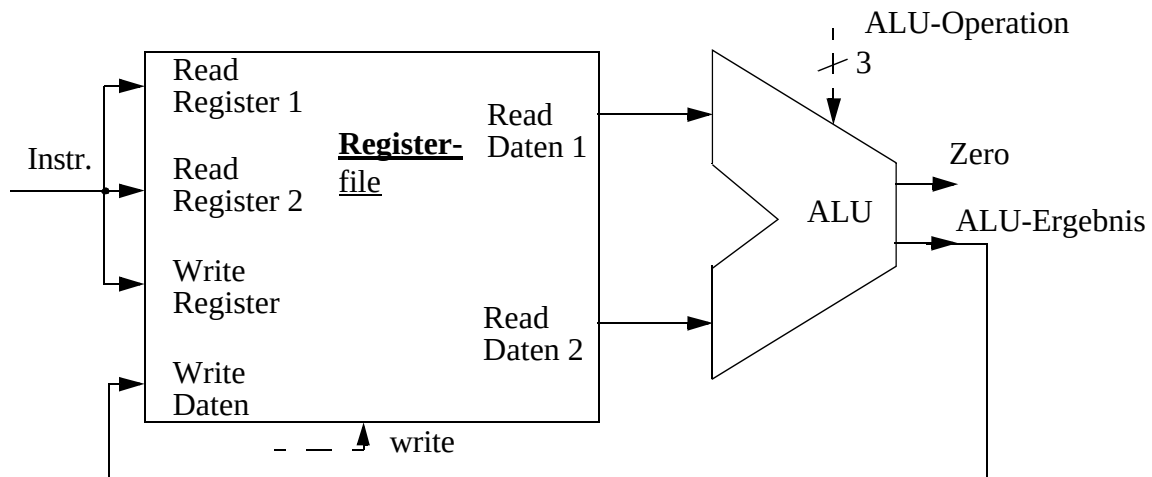


Bild 2.25. Teil-Datenpfad für Register-Zugriff und ALU-Operation von R-Format-Befehlen

1 Load- und Store-Befehle

lw \$1, offset (\$2); /* \$1 <-- M[\$2 + offset]

sw \$1, offset (\$2); /* M[\$2 + offset] <-- \$1;

offset = 16 bit signed

Beteiligte Elemente: Registerfile, Datenspeicher, ALU (zur Adreßrechnung),

Vorzeichenauffüller (Sign Extension Unit, 16 auf 32 bit).

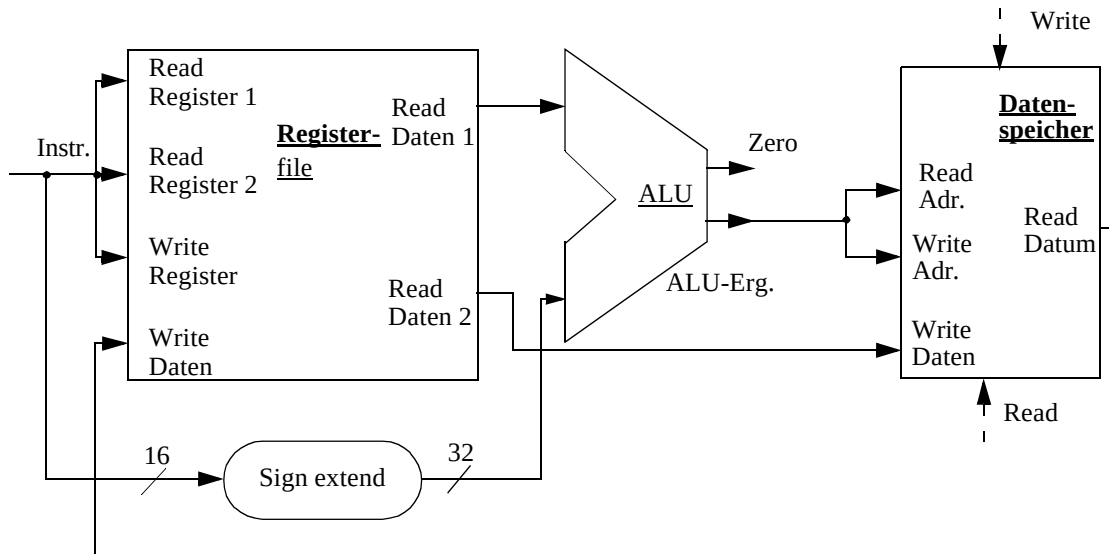


Bild 2.26. Teil-Datenpfad für Load und Store.

1 Branch-Befehl beq

beq \$1,\$2,offset

Ablauf:

(1) berechne Branch Target

Branch Target $\leftarrow$ PC + SignExt(offset)

(2) if (\$1 == \$2) PC $\leftarrow$ Branch Target

else PC $\leftarrow$ PC + 4

Zu beachten: a) Basis für Adreßrechnung ist der neue (inkrementierte) PC.

b) offset ist Word Offset, muß also um 2 bit nach links geschiftet werden.

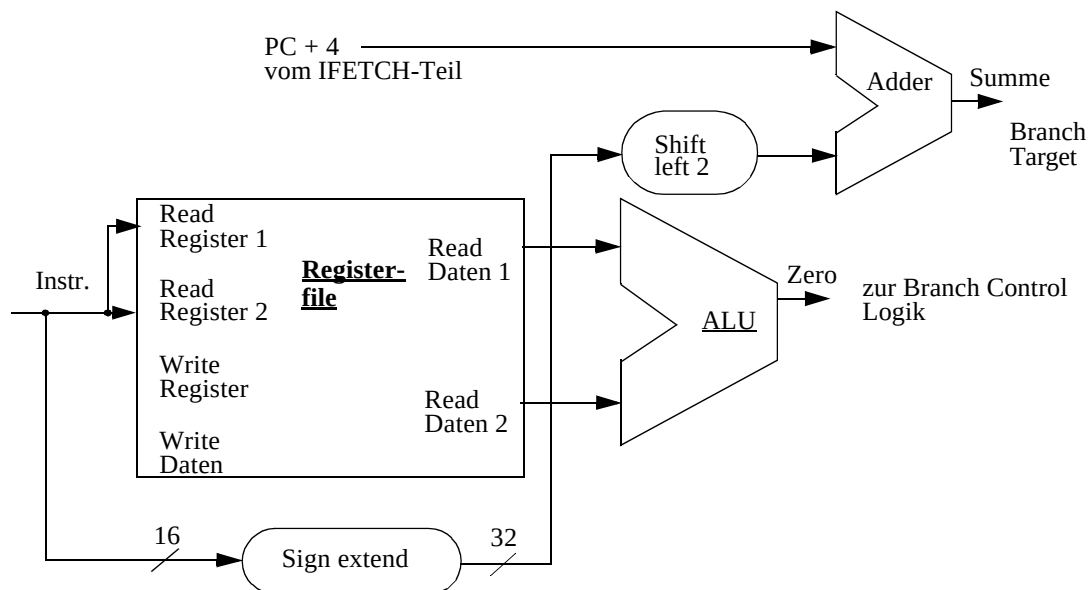


Bild 2.27. Teil-Datenpfad für Branch.

1 Kombination der Teil-Datenpfade

Überlagerung zu einem Ein-Takt-Datenpfad: Jedes Element kann pro Taktzyklus nur einmal genutzt werden.

Folge: Es werden je ein separater Befehls- und Datenspeicher benötigt. Zur Auswahl unterschiedlicher Eingangsinformationen werden Multiplexer benutzt.

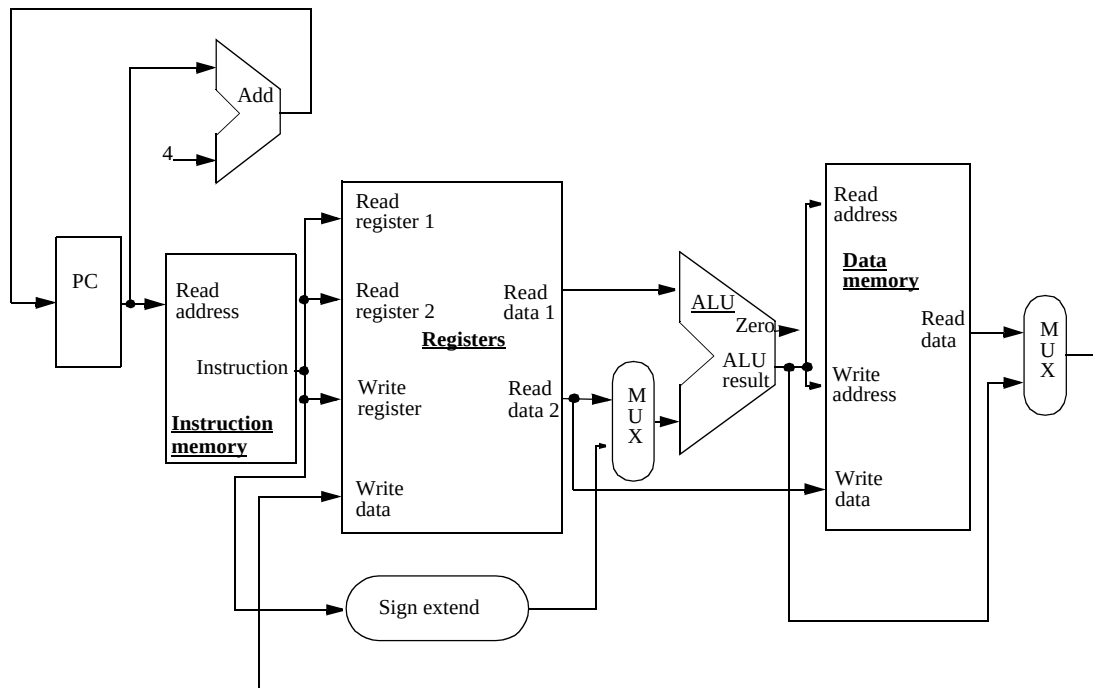


Bild 2.28. Kombination der Teile "IFETCH", R-type und Load/Store (für add, sub, and, or, slt, lw, sw)

Erweiterungen:

- MUX für den 2. ALU-Eingang wählt zwischen Register und Immediate.
- MUX für Register Write wählt zwischen ALU Result und Datum vom Datenspeicher.

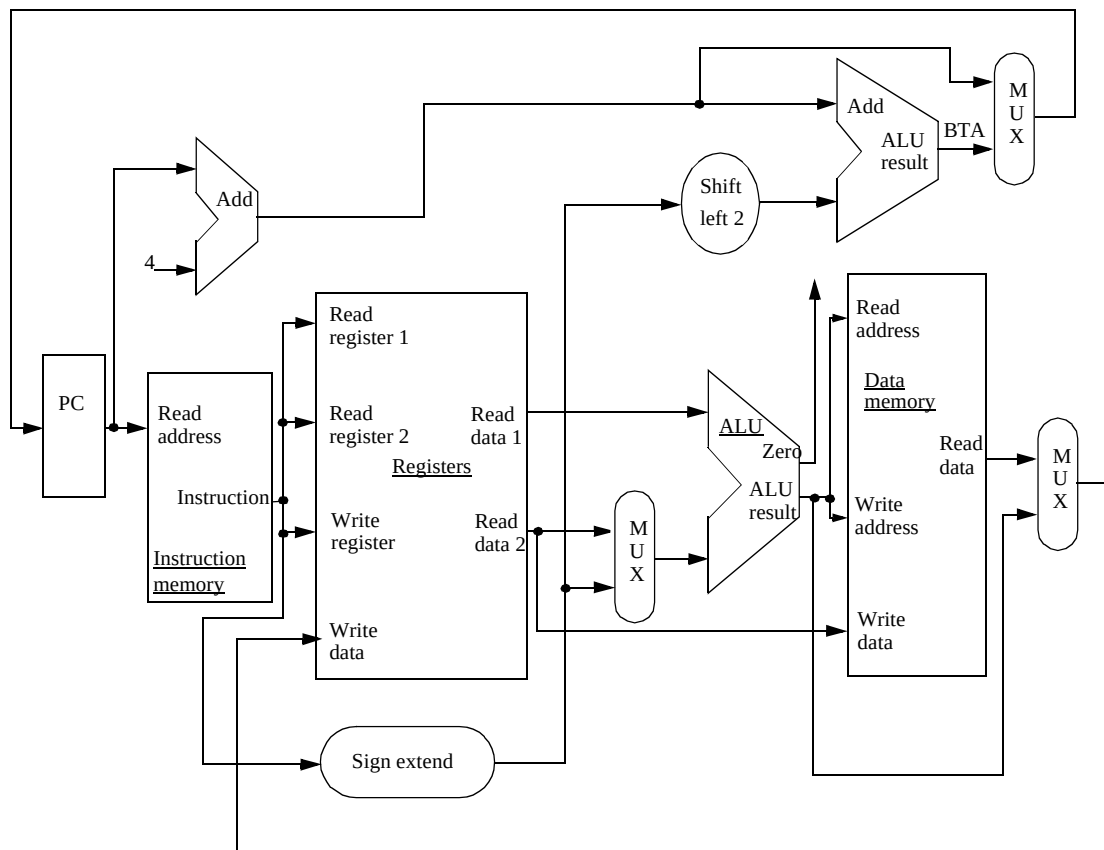


Bild 2.29. Erweiterung um Logik für bedingte Sprünge (beq).

Erweiterungen:

- Shift left 2 - Einheit für Adress-Offset
- Addierer zur Berechnung der Branch Target Adresse BTA.
- MUX zur Auswahl zwischen PC+4 und BTA.

2.2.3. Steuerung des Ein-Takt-Pfades

2.2.3.1 ALU-Control

Die ALU besitzt 3 Steuerleitungen (ALU control). Von den codierbaren 8 Möglichkeiten sind 5 belegt (s. Kap. 2.1.3):

- 1.- Load/Store: Addition (Adreßrechnung)
- 2.- R-type: sub, add, and, or, slt
Unterscheidung durch funct-Feld (6 low order Bits der Instruktion)
- 3.- Branch beq: sub

Ein Steuersignal ALUop muß zunächst zwischen 3 Möglichkeiten unterscheiden:

ALUop	Gruppe	Aktion	ALU control
00	Load/Store	add	010
01	Branch equal	sub	110
10	R-type	gemäß funct	s. Tabelle

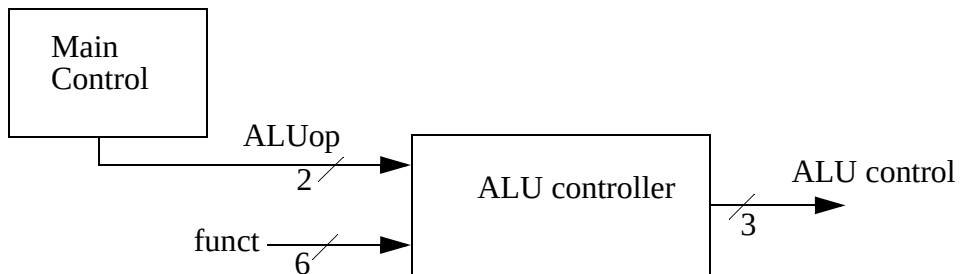


Bild 2.30. Zusammenwirken von Main Control und ALU Control

Instr.	ALUop	Aktion	funct	ALU control
lw	00	load word	xxx xxx	010
sw	00	store word	xxx xxx	010
beq	01	branch equal	xxx xxx	110
R-type	10	add	100 000	010
		sub	100 010	110
		and	100 100	000
		or	100 101	001
		slt	101 010	111

Bild 2.31. Zuordnungstabelle (ALUop, funct) --> (ALU control)

ALUop		funct	ALU control		
ALUop 1	ALUop 0	F5 F4 F3 F2 F1 F0	Actrl 2	Actrl 1	Actrl 0
0	0	x x x x x x	0	1	0
0	1	x x x x x x	1	1	0
1	0	x x 0 0 0 0	0	1	0
1	0	x x 0 0 1 0	1	1	0
1	0	x x 0 1 0 0	0	0	0
1	0	x x 0 1 0 1	0	0	1
1	0	x x 1 0 1 0	1	1	1

Bild 2.32. Funktionstabelle ALU control mit don't cares (x).

$$\text{Actrl 2} = \text{ALUop 0} \vee (\text{ALUop 1} \wedge \text{F1})$$

$$\text{Actrl 1} = \overline{\text{ALUop 1}} \vee \overline{\text{F2}}$$

$$\begin{aligned} \text{Actrl 0} &= (\text{ALUop 1} \wedge \text{F0}) \vee (\text{ALUop 1} \wedge \text{F3}) \\ &= \text{ALUop 1} \wedge (\text{F0} \vee \text{F3}) \end{aligned}$$

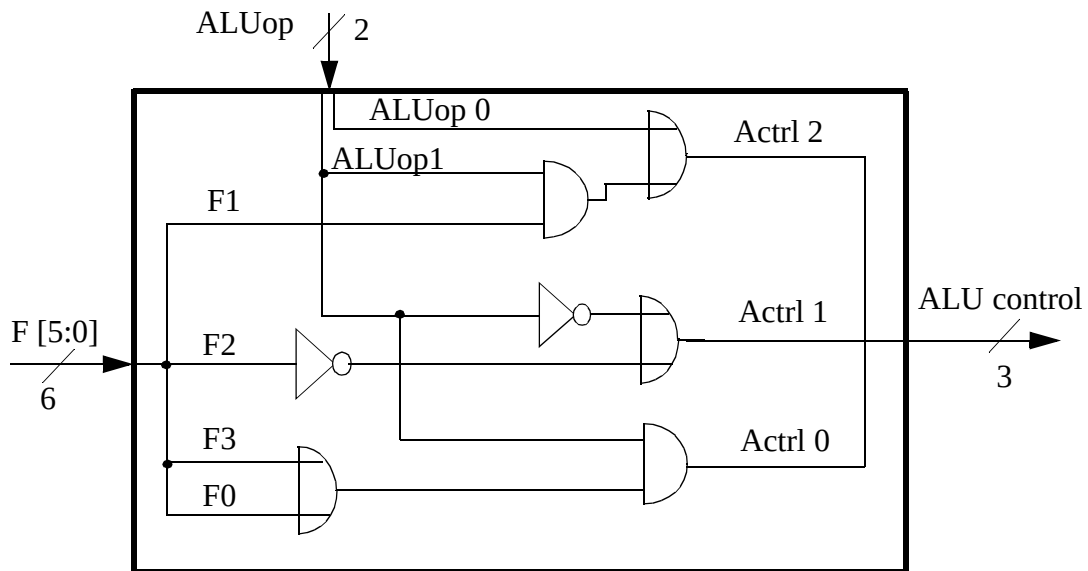


Bild 2.33. Logik für ALU Control.

2.2.3.2 Main Control

Vorgehen:

1. Festlegung aller Steuersignale.
2. Entwurf einer Kombinatorik, welche in einem Schritt pro Befehl alle Steuersignale erzeugt.

Voraussetzungen (vgl. Formate in Kap. 2.1.2):

- Opcode befindet sich immer im Instruktionsfeld 31-26. Bezeichnung Op[5-0].
- Für R-type, beq, store: 2 Leseregister rs und rt in Instruktionsfeldern 25-21 und 20-16.
- Basisregister rs für Load/Store im Instruktionsfeld 25-21.
- 16 bit-Offset für beq, lw, sw im Instruktionsfeld 15-0.
- Zielregister
 - für Load (rt): Instruktionsfeld 20-16
 - für R-type (rd): Instruktionsfeld 15-11
 - => MUX nötig.

- Steuersignale

RegDst	MUX (Write Register Adresse)
RegWrite	Register schreiben

ALUSrc	MUX (ALU input)
PCSrc	MUX (next PC)
ALUop	2 bit zur ALU-Steuerung (s.o.)
MemWrite	Datenspeicher schreiben
MemRead	Datenspeicher lesen
MemtoReg	Quelle für Register schreiben
Branch	Aktiviert bei Branch-Befehl

Alle Steuersignale (außer PCSrc) werden direkt von Main Control erzeugt.

PCSrc = 1 if (Branch $\wedge$ Zero) wählt BTA im Falle eines genommenen Sprunges als Quelle für den nächsten PC, PCSrc = 0 wählt PC + 4.

Die Steuersignale ergeben sich eindeutig aus den 4 möglichen Fällen R-type, lw, sw, beq gemäß folgender Tabelle:

Instruction	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop1	ALUop0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

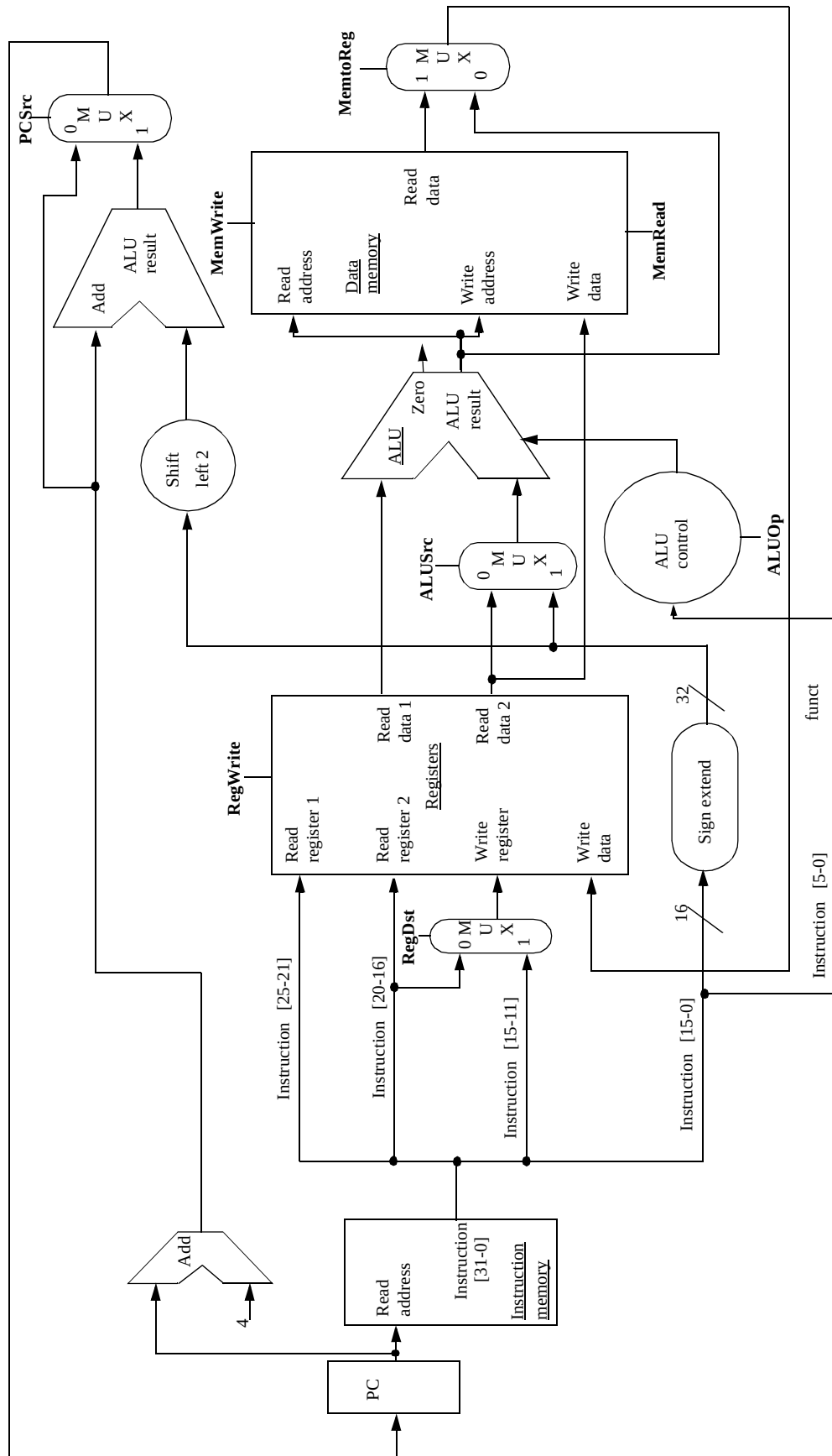


Bild 2.34. Vollständiger Datenpfad (Ein-Takt) für R-Type, lw, sw, beq (noch ohne jmp). Eingezeichnet sind auch die Steuerleitungen.

Beispiel 2-10:

R-type

RegDst = 1 : Instr[15-11] enthält rd (Write Register Adresse)
 ALUSrc = 0 : Read data 2 ist ALU-Input.
 MemtoReg = 0 : ALU result wird ins Zielregister geschrieben.
 RegWrite = 1 : Zielregister soll geschrieben werden.
 MemRead = 0 : Kein Zugriff auf Datenspeicher.
 MemWrite = 0 : Kein Zugriff auf Datenspeicher.
 Branch = 0 : Kein Branch-Befehl.
 ALUOp = 10 : Zeigt R-type-Befehl an.

OpCode-Zuordnung:

Op ₁₀	Op ₅	Op ₄	Op ₃	Op ₂	Op ₁	Op ₀	Instruktion
0	0	0	0	0	0	0	R-type
35	1	0	0	0	1	1	lw
43	1	0	1	0	1	1	sw
4	0	0	0	1	0	0	beq

Realisierung von Main Control:

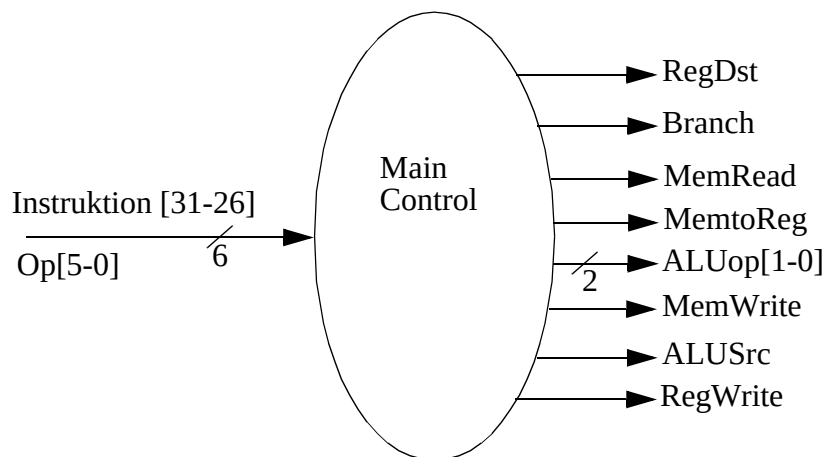


Bild 2.35. Input- und Output-Signale der Main Control Unit.

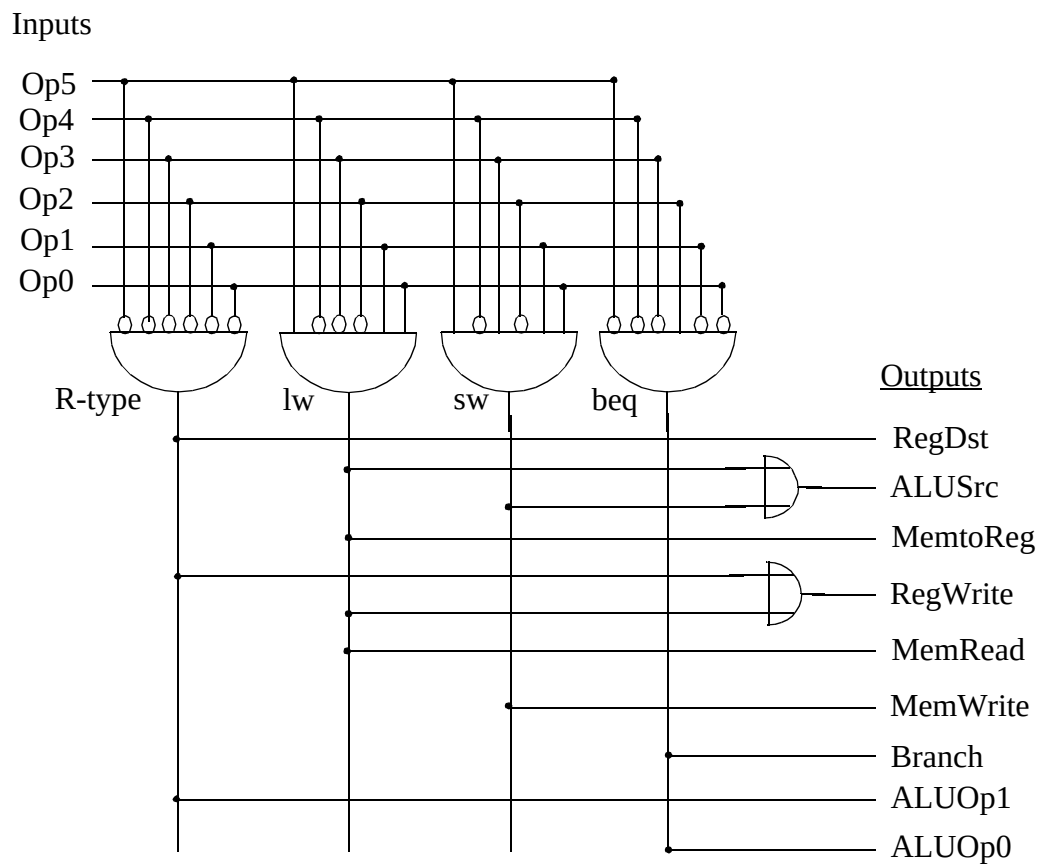


Bild 2.36. Logik der Main Control Unit.

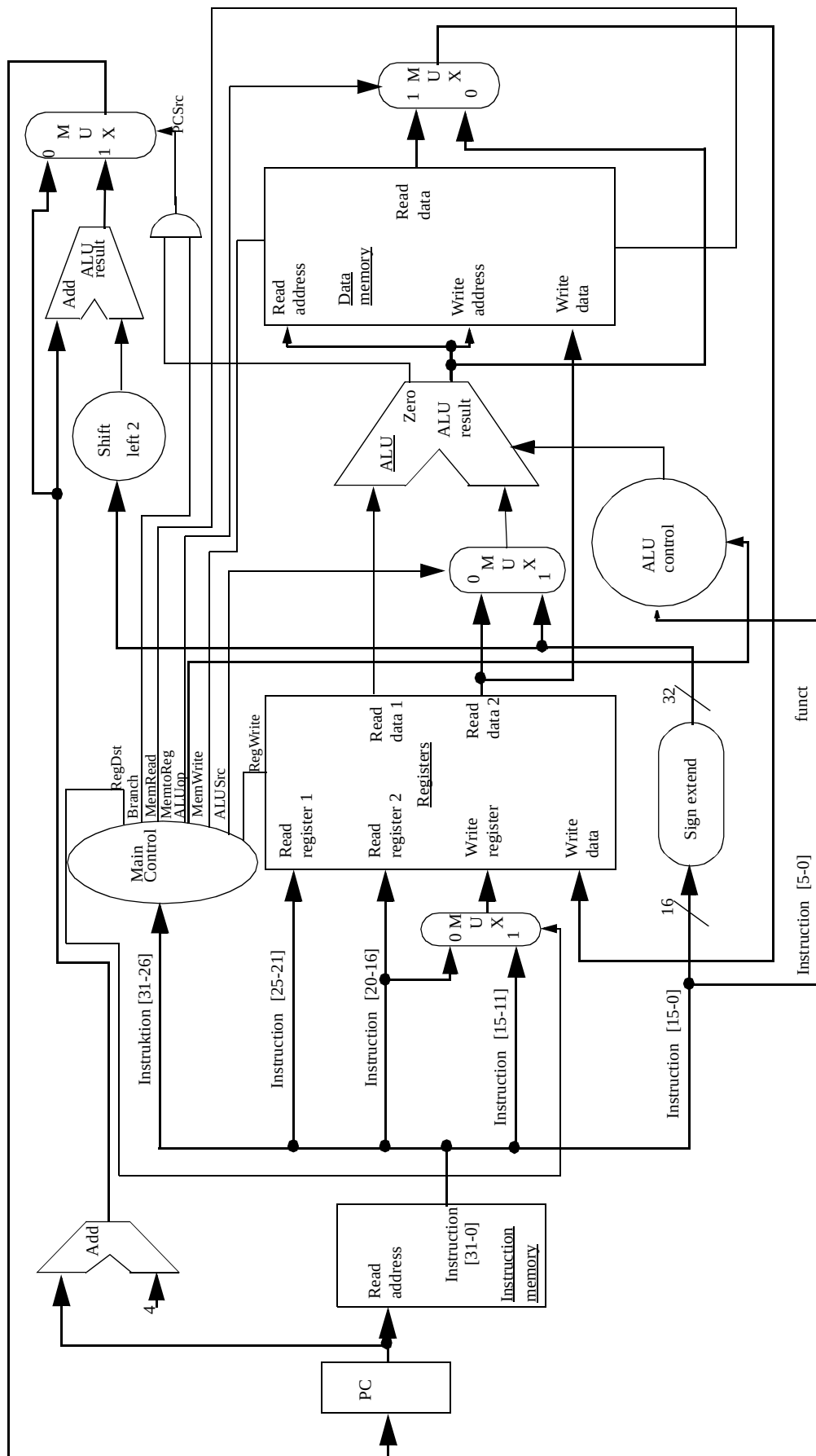


Bild 2.37. Vollständiger Ein-Takt-Datenpfad incl. Main Control Unit (ohne jmp).

2.2.4. Mehr-Takt-Datenpfad

l Problem beim Ein-Takt-Datenpfad

Taktperiode wird bestimmt durch den längsten Pfad: Load. CPI = 1, aber Takt ist gleich lang für alle Befehle.

Beispiel 2-11:

Befehlsausführungszeiten:

Annahmen: Speicherzugriff: 10 ns
 Registerzugriff: 5 ns
 ALU: 10 ns

Häufigkeit	Befehlstyp	I-mem	Reg. read	ALU	D-mem	Reg. write	Summe
49%	R-type	10	5	10		5	30
22%	lw	10	5	10	10	5	40
11%	sw	10	5	10	10		35
16%	branch	10	5	10			25
2%	jump	10					10

=> durchschnittliche Befehlsausführungszeit: 31,6 ns

=> hypothetische Maschine mit variabler Taktperiode ist

$$\frac{40}{31.6} = 1.27 \text{ mal schneller als der Ein-Takt-Datenpfad.}$$

Der Performancevorteil einer Mehrzyklenimplementierung steigt, wenn der Unterschied zwischen dem schnellsten und langsamsten Befehl größer wird, z.B. mit FPU.

Außerdem: Ein-Takt-Datenpfad erlaubt kein Pipelining. Jede Einheit wird pro Befehl höchstens 1 mal genutzt.

l Maßnahmen für Mehr-Takt-Datenpfad

- nur eine Speichereinheit für Daten und Befehle.
- Zwischenspeicherung des Befehls im Instruction Register IR (Grund: Befehlswort wird noch benötigt, während ein erneuter Zugriff auf den Speicher erfolgt.).
- Zusammenfassung von ALU + 2 * Adder zu einer ALU (Mehrfachnutzung).
- Zusätzliche Multiplexer (incl. Steuerleitungen)

Memory Read Address Selector (PC, ALU result)

ALU Input Selector A (PC, Register)

ALU Input Selector B (Register, 4, sign-ext. immediate, sign-ext. left-shift branch offset)

Register Write Data Selector (ALU result, Mem Data)

- Zusätzliche Steuersignale

Write-Signale für alle Speicherelemente, da nicht mehr in jedem Taktzyklus geschrieben wird:

PCWrite

MemWrite

MemRead

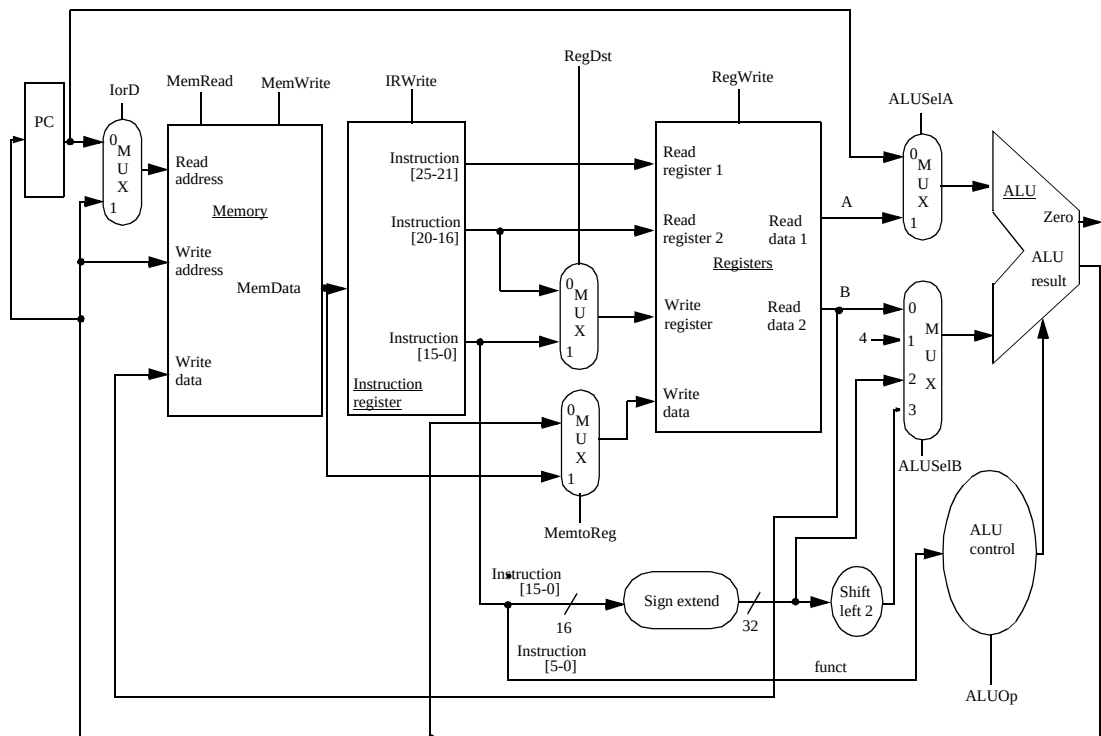
IRWrite

RegWrite

Read-Signale für Hauptspeicher: MemRead

- ALU Control: keine Änderung

- Zusätzliche Steuersignale für Branch und Jump zur Auswahl von next PC.



(s 315)

Bild 2.38. Mehr-Takt-Datenpfad mit Steuersignalen. Es fehlen hier noch die Logik zur Auswahl des nächsten PC sowie die Main Control Unit.

Anmerkung 1: Anstatt der ALU-Input-Selektoren könnte eine 3-Bus-Struktur (Source1, Source 2, Destination) realisiert werden (vgl. DLX). Es ist dann dafür zu sorgen, daß nur jeweils 1 Sender pro Bus aktiv ist.

Anmerkung 2: Bei der Umwandlung einer Kombinatorik (Ein-Takt-Datenpfad) in einen Mehr-Takt-Datenpfad müssen zusätzliche Register eingefügt werden, wenn sich das Eingangssignal für ein Speicherelement ändert, bevor es benutzt (d.h. in das Speicherelement geschrieben) wird.

Beispiel: Der Befehl wird aus dem Speicher gelesen und mehrfach (für Registerzugriff, Branch, ALU) benutzt. Er wird in IR zwischengespeichert, da während dieser Benutzungszeit der Speicher erneut benötigt wird (zum Lesen von Daten).

1 Befehlszyklus

Die Taktperiode wird durch den längsten Einzelschritt bestimmt. Daher wird die Befehlsabarbeitung in (hier 5) möglichst gleichlange Schritte eingeteilt.

Befehlszyklus und zugeordnete RT-Operationen für die 3 Befehlsklassen:

	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches
IF	Instruction fetch	IR = Memory[PC] PC = PC+4		
ID	Instruction decode/ register fetch	A = Registers[IR[25-21]] B = Registers[IR[20-16]] Target = PC + (sign-extend(IR[15-0]) << 2)		
EX	Execution, address computation, or branch completion	ALUoutput = A op B	ALUoutput = A + sign-extend (IR[15-0])	if (A == B) then PC = Target
MEM	Memory access or R-type completion	Reg(IR[15-11]) = ALUoutput	memory-data = Memory[ALUoutput] or Memory [ALUoutput] = B	
WB	Write-back		Reg[IR[20-16]] = memory-data	

1 Erweiterung um Jump- und Branch-Logik

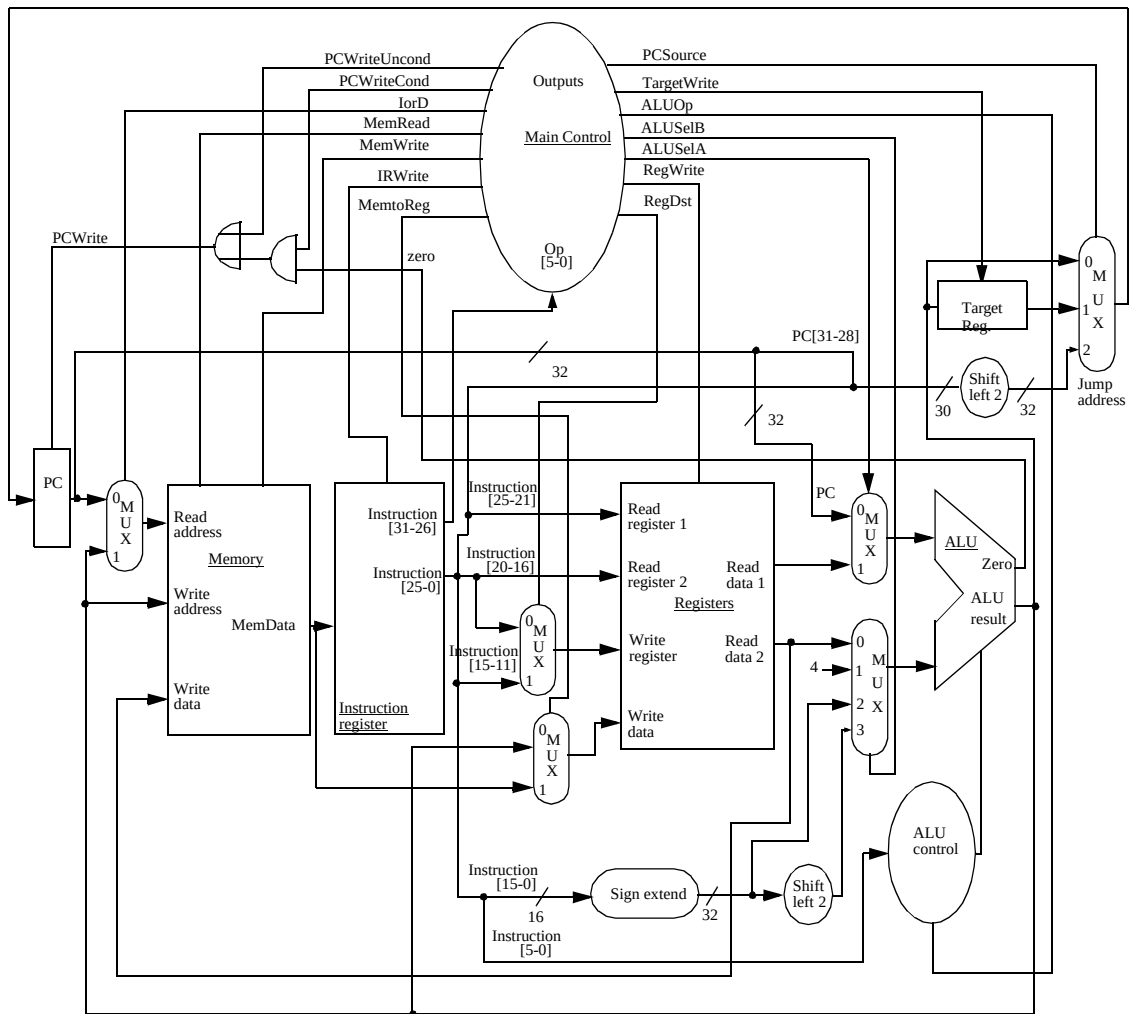
Der PC für den nächsten I-Fetch hat 3 mögliche Quellen (PC Source Selector):

- (0) ALU result im sequentiellen Fall: $PC \leftarrow PC + 4$
- (1) Target Register enthält die neue Befehlsadresse im Fall eines genommenen Sprungs (Branch Target Address BTA).
- (2) Jump Address = $(PC[31-28] @ Instr[25-0]) \ll 2$; 26 bit-Konstante aus dem Befehl wird mit den 4 MSBs von PC konkateniert (PC-relativer Sprung) und um 2 bit nach links geschoben (Wortadresse).

- PC Write-Signal = $PCWriteUncond \vee (PCWriteCond \wedge Zero)$

PCWriteUncond = 1 für jump

PCWriteCond = 1 für branch



(s 323)

Bild 2.39. Mehr-Takt-Datenpfad mit Main Control Unit und Branch/Jump-Logik.

1 Main Control

Die Main Control Unit erzeugt aus dem Opcode ($Op[5-0] = \text{Instruction}[31-26]$) die Steuersignale für jeden Takt. Die Techniken zur Implementierung der CU (Mikroprogrammierung, endlicher Automat) wurden in der Vorlesung RS behandelt. Die Implementierungen für MIPS können in Patterson, Hennessy: Computer Organization & Design, The Hardware Software Interface, S. 322-343, nachgelesen werden.

1 Performance des Mehr-Takt-Datenpfades

	CPI	Häufigkeit
Loads	5	22%
Stores	4	11%
R-type	4	49%
Branch	3	16%
Jump	3	2%

=> $CPI_{\text{mittel}} = 4.04$.

Performance-Verbesserung gegenüber Ein-Takt-Datenpfad = $\frac{5}{4.04} = 1.24$

2.3. Pipeline-Datenpfad

Vorgehen:

1. Aufteilung des Befehls in Einzelschritte.
2. Umformung des Ein-Takt-Datenpfades so, daß HW-Einheiten den Einzelschritten zugeordnet werden können. Wir gehen vom Ein-Takt-Datenpfad und nicht vom Mehr-Takt-Datenpfad aus, da der Ein-Takt-Datenpfad eine klarere Entkoppelung zwischen den Stufen aufweist (z.B. IMem und DMem separat).
3. Einführung von PL-Registern.
4. Durchspielen von charakteristischen Befehlen, ggf. Modifikation.
5. Identifikation von Steuersignalen.
6. Entwurf der Main Control Unit.
7. Maßnahmen bei PL-Konflikten.

2.3.1. Befehlszyklus

Es wird hier nur der einfache 5-stufige Befehlszyklus ohne Mehrfachzyklen betrachtet. Der Zeitbedarf wird wie folgt angenommen:

IF	10ns	(IMem)
ID	5ns	(Register Read)
EX	10ns	
MEM	10ns	(DMem)
WB	5ns	(Register Write)

Der Worst Case - Zeitbedarf (10ns) bestimmt den PL-Takt.

2.3.2. Umformung

Zuordnung der Registertransferbefehle zu Befehlsphasen:

- IF: IMem, PC, PC-Inkrement-Logik
- ID: Register lesen, Sign-Extension
- EX: ALU, Adreßrechnung (Adder)
- MEM: DMem
- WB: Register schreiben

Datenflußrichtung: links --> rechts

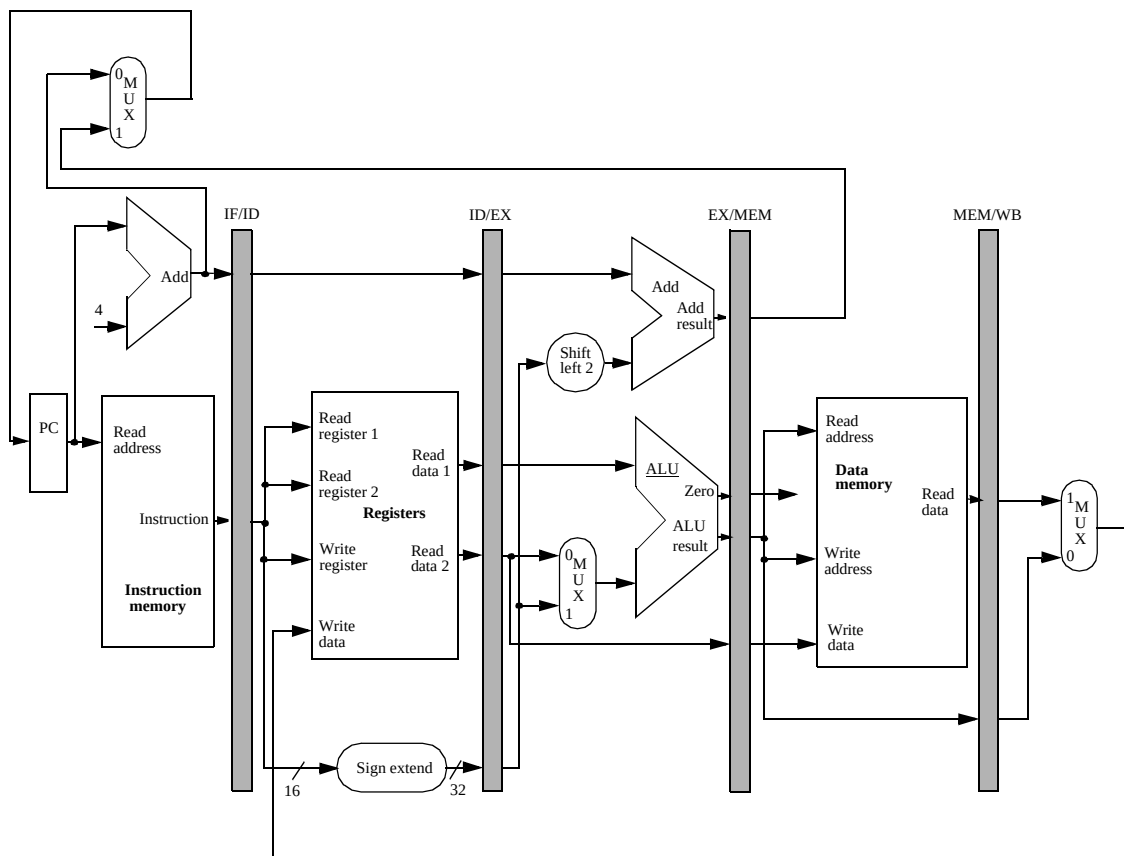


Bild 2.41. PL-Datenpfad mit PL-Registern.

(370)

2.3.4. Ablauf des Befehls Load Word (lw)

	31 - 26	25 - 21	20 - 16	15	-	0
lw	op	rs	rt	adr. immediate		
	6	5	5	16		

lw rt, imm(rs) /* rt <-- M[imm + rs]

Beispiel: lw \$1,100(\$2) /* \$1 <-- M[100 + \$2]

Animationsartige Darstellung durch Hervorheben der jeweils aktiven HW-Einheiten. Bei zu schreibenden Registern ist die linke Hälfte hervorgehoben (grau), bei zu lesenden die rechte Hälfte.

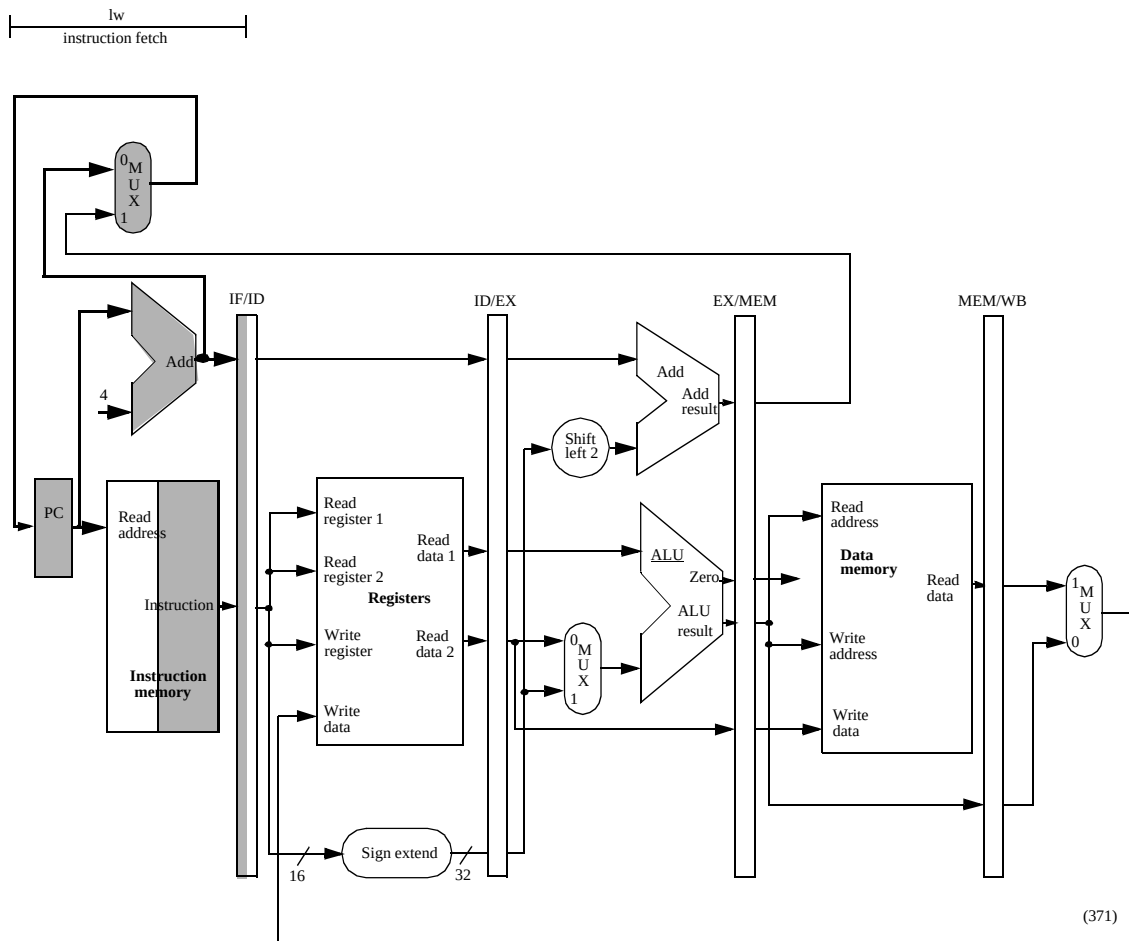


Bild 2.42. IF: PC wird gelesen, IMem wird gelesen, IF/ID wird geschrieben. Der Addierer berechnet die lineare Folgeadresse $PC + 4$ und gibt sie an PC und an IF/ID (für die evtl. Benutzung in späteren Stufen).

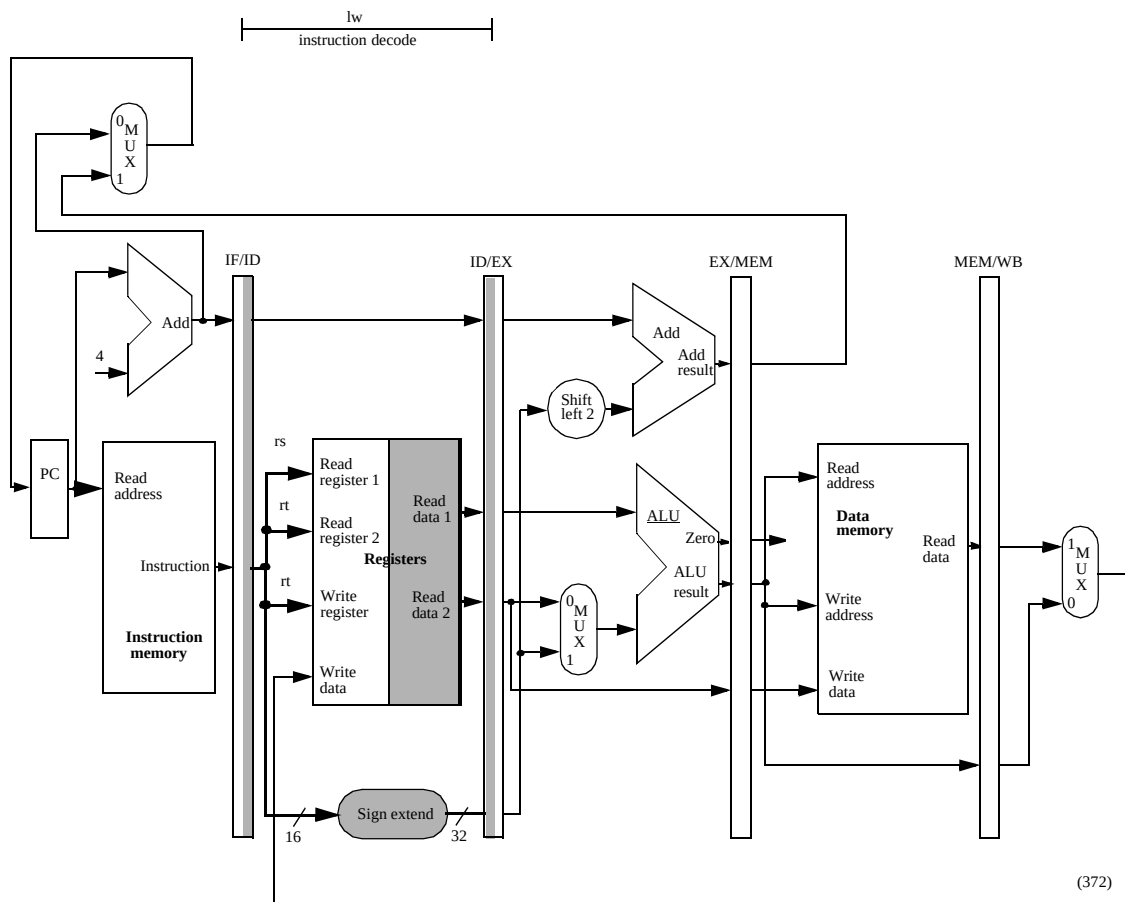


Bild 2.43. ID: IF/ID liefert 16 bit-Immediate an Sign-Extension, IF/ID liefert 2 Registeradressen (*rs* und *rt*). ID/EX speichert die beiden Registerinhalte, die 32 bit-Immediate-Konstante und den inkrementierten PC. *Read data 2* (= *rt*) wird nicht benötigt. *rt* wird während WB benutzt, um in das Zielregister zu schreiben (s. Modifikation des Datenpfades).

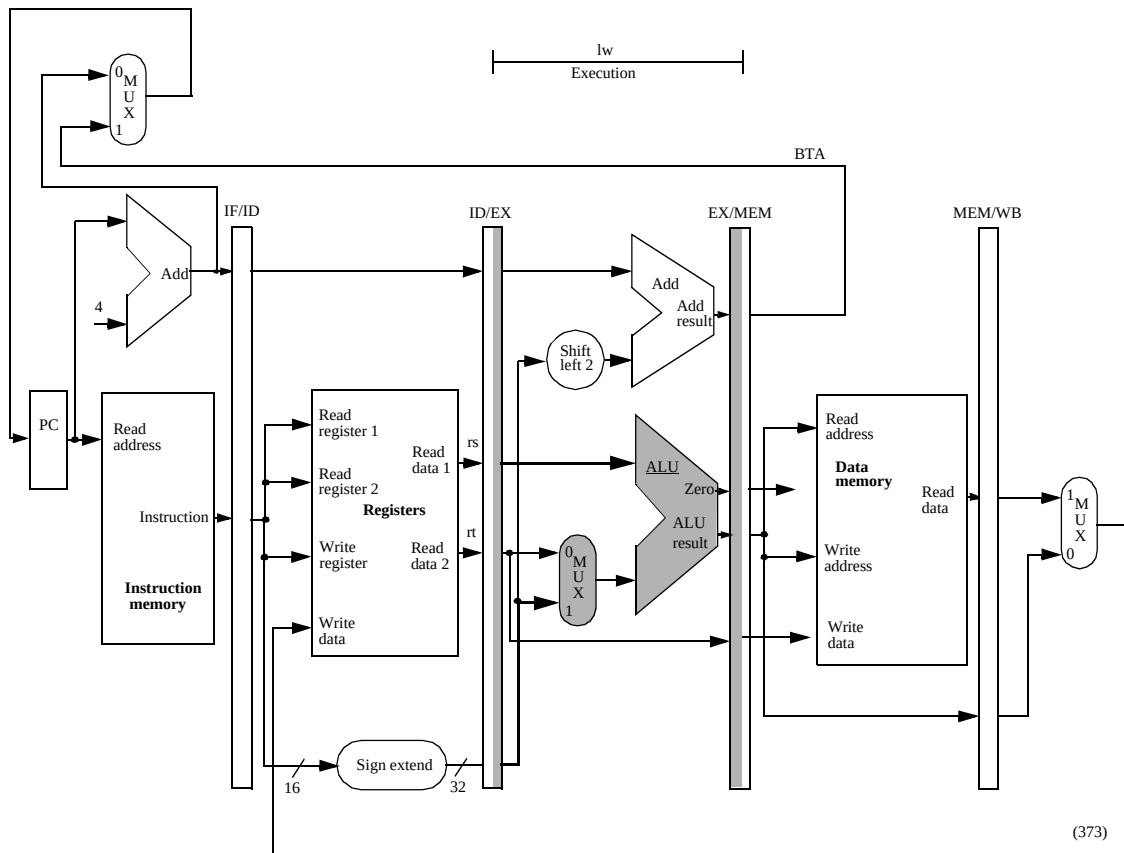


Bild 2.44. EX: ID/EX liefert rs und rt, 32 bit-Immediate und PC + 4. EX/MEM speichert Adder Result (BTA hier nicht nötig), ALU Result (= eff. Adresse), Zero-Flag und Register 2 (für Store-Befehl, hier nicht nötig). Der MUX für ALU-Input 2 wählt nicht rt, sondern 32 bit-Immediate aus (Steuersignal = 1).

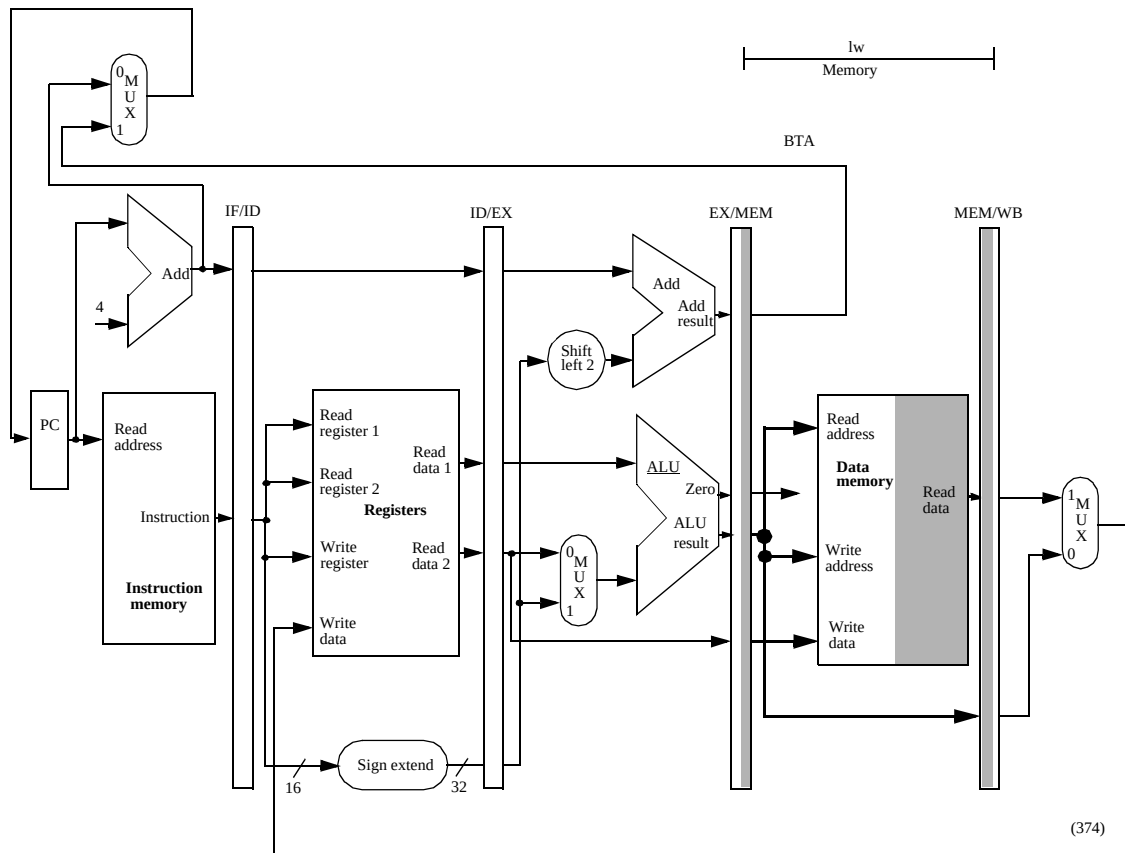


Bild 2.45. MEM: EX/MEM liefert die Leseadresse (bzw. Schreibadresse) für Datenspeicher, BTA (hier nicht nötig), Write data (hier nicht nötig). Das gelesene Datum (Read data) wird in MEM/WB gespeichert.

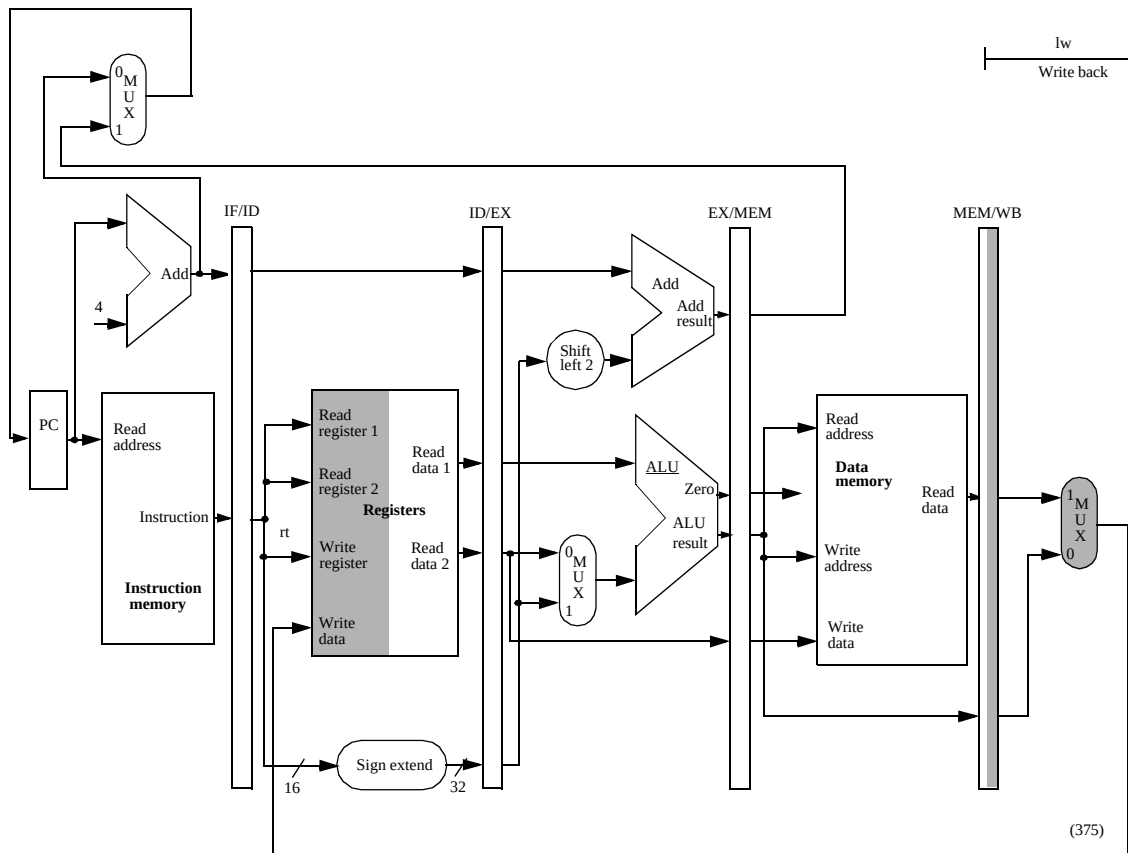


Bild 2.46. WB: Das aus dem Speicher (DMem) gelesene Datum wird aus MEM/WB in das Register geschrieben, welches durch Write register adressiert wird.

Achtung: Hier noch fehlerhafte Write register Adresse, s. Modifikation.

2.3.5. Datenpfad-Modifikation für lw

Der bis hier entwickelte PL-DP enthält noch einen Fehler: Während WB wird die Registeradresse rt des Folgebefehls $n + 3$ benutzt (rt_{n+3}). Um rt_n zu retten, muß es in den PL-Registern gespeichert und weitergereicht werden. Somit wird die Write register-Adresse rt_n während WB durch MEM/WB geliefert.

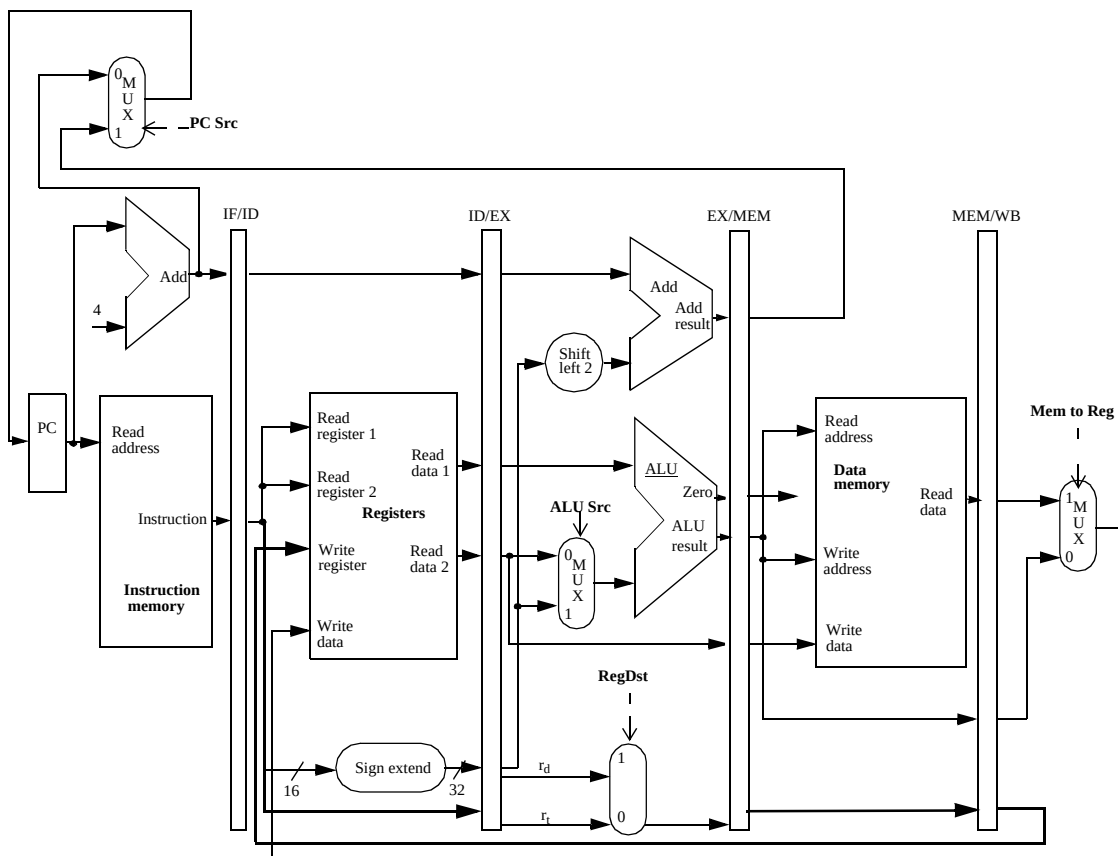


Bild 2.47. Modifizierter DP für korrektes WB beim Befehl lw.

2.3.6. Graphische PL-Darstellung

Der PL-Entwurf verlangt eine konfliktfreie Zuordnung von Datenpfad-Ressourcen an die Befehle. Eine Überprüfung der Konfliktfreiheit muß für alle denkbaren Befehlssequenzen erfolgen.

Hilfsmittel: graphische Darstellung, später auch Simulation.

3 Varianten

- Folge von Ein-Takt-Schnappschüssen (wie lw-Befehl in Kap. 2.3.4).
- Belegungsdiagramm 1: Ressourcennutzung (t)
- Belegungsdiagramm 2: Befehle (t)

Sichtweise: Befehle fließen durch die Ressourcen

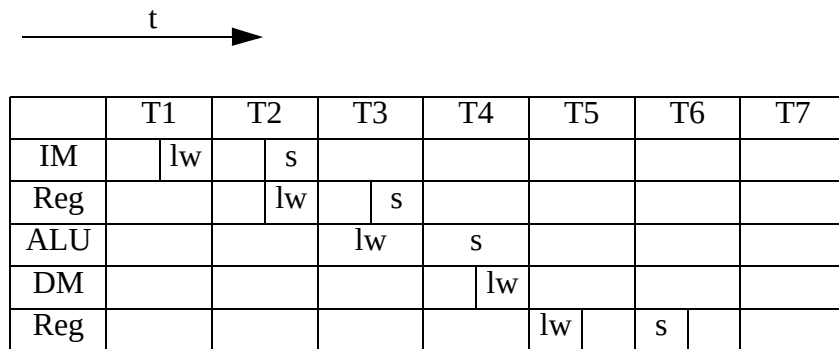


Bild 2.48. Belegungsdiagramm 1: Ressourcennutzung (t).

Eingezeichnet sind die beiden Befehle LoadWord (lw) und sub (s). Ist die rechte (linke) Hälfte eines Taktes belegt, so wird das entsprechende Speicherelement gelesen (geschrieben).

c) Belegungsdiagramm 2: Befehle (t)

Sichtweise: Ressourcen fließen an den Befehlen vorbei.

Andere Sichtweise: Dem Befehl i steht ein gesamter Datenpfad zur Verfügung.

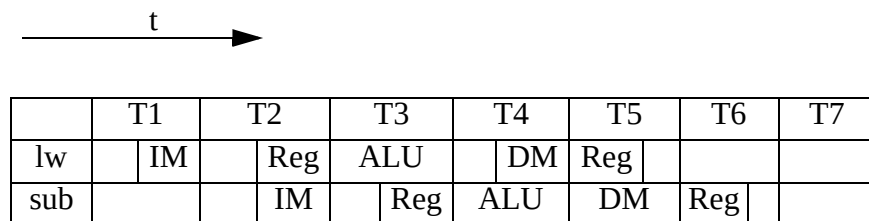


Bild 2.49. Belegungsdiagramm 2: Befehl (t). Jedem Befehl ist ein gesamter Datenpfad zugeordnet.

Das Belegungsdiagramm 2 kann beliebig verfeinert werden, indem weitere Ressourcen (z.B. die PL-Register) hinzugenommen werden.

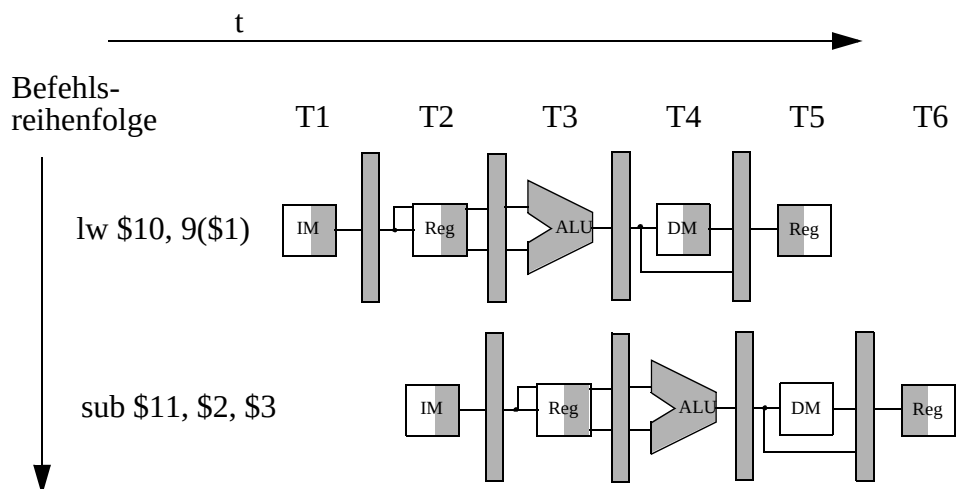


Bild 2.50. Verfeinertes Belegungsdiagramm 2 mit Darstellung der PL-Register.

2.3.7. PL-Steuerung

Zunächst wird ein einfacher PL-DP ohne Sprünge betrachtet (s. Bild 2.47). Er wird um die Steuersignale des Ein-Takt-DP (s. Bild 2.37) erweitert.

1 Erweiterungen:

1. IF: Steuersignal WritePC und ReadIM sind immer aktiv.
2. ID: Es werden in jedem Takt die Register gelesen (kein Steuersignal). Falls geschrieben werden soll, ist RegWrite zu aktivieren.
3. EX: (1) Auswahl der ALU-Quelle 2 (Register 2 oder 32 bit-Immediate), Steuersignal ALUSrc.
(2) ALU-Steuerung durch ALU Control (gesteuert durch ALUop und Instruction [5-0] (funct)).
(3) Auswahl Zielregister-Adresse (rt oder rd). Steuersignal RegDst. Der Multiplexer wird erst während EX angesteuert, da RegDst während ID berechnet wird.
4. MEM: Ein Lese- bzw. ein Schreibzugriff auf DM wird mittels MemRead bzw. MemWrite angesteuert. Im Fall eines Branch-Befehls wird das Zero-Signal der ALU zur Auswahl des nächsten PC verwendet (Stereusignal PCSrc).
5. WB: Auswahl von ALUresult oder DM-Daten zum Schreiben ins Register (Stereusignal MemtoReg).

Die Steuersignale sind somit identisch mit denen des Ein-Takt-Datenpfades. Sie werden aber hier den einzelnen PL-Stufen zugeordnet:

	IF	ID	EX				MEM			WB	
	-	-	RegDst	ALUop 1	ALUop 0	ALU Src	Branch	Mem- Write	Mem- Read	Reg- Write	Memto Reg
R-format			1	1	0	0	0	0	0	1	0
lw			0	0	0	1	0	0	1	1	1
sw			x	0	0	1	0	1	0	0	x
beq			x	0	1	0	1	0	0	0	x

1 Einordnung der Steuerung in die PL-Stufen:

Die Main Control Unit berechnet alle Steuersignale während ID. Sie werden dann stufenweise "verbraucht", d.h. die Steuersignale müssen über die PL-Register weitergereicht werden.

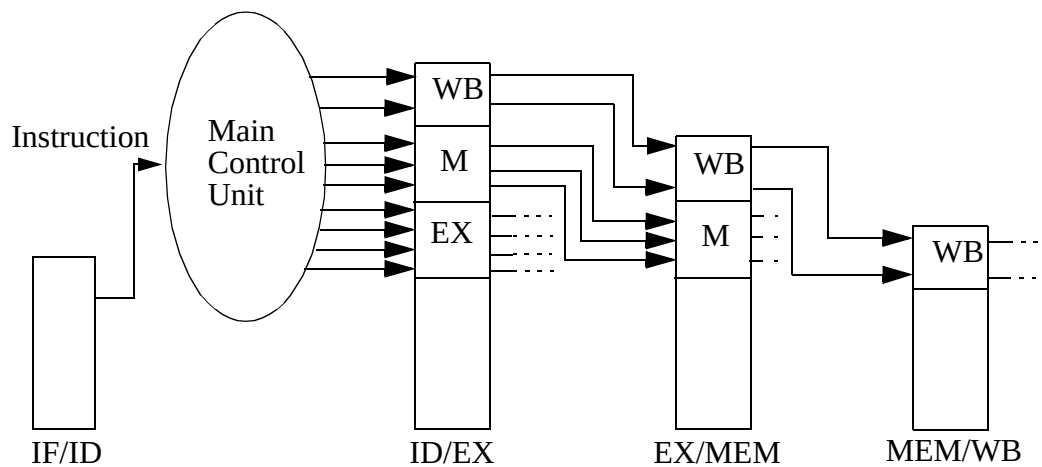


Bild 2.51. Weiterreichen der Steuersignale.

2.3.8. Datenkonflikte

Es werden nur Read after Write (RAW) Konflikte betrachtet.

sub \$2,\$1,\$3	IM	Reg	ALU	DM	Reg					
and \$12,\$2,\$5		IM	Reg	ALU	DM	Reg				
or \$13,\$6,\$2			IM	Reg	ALU	DM	Reg			
add \$14,\$2,\$2				IM	Reg	ALU	DM	Reg		
sw \$15,100(\$2)					IM	Reg	ALU	DM	Reg	

Bild 2.52. RAW-Konflikt über Register \$2. Zeitlich rückwärts gerichtete Abhängigkeitspfeile zeigen einen Konflikt an.

Die Konfliktbehandlung umfasst die 2 Schritte

- Konflikterkennung
- Konfliktbehandlung

1 Konflikterkennung

Die Registeradressen eines neu (während IF) eingelesenen Befehls stehen im PL-Register IF/ID zur Verfügung. Ein Konflikt besteht, wenn eines der zu lesenden Register in IF/ID gleich ist mit einem zu schreibenden Register in ID/EX oder EX/MEM oder MEM/WB.

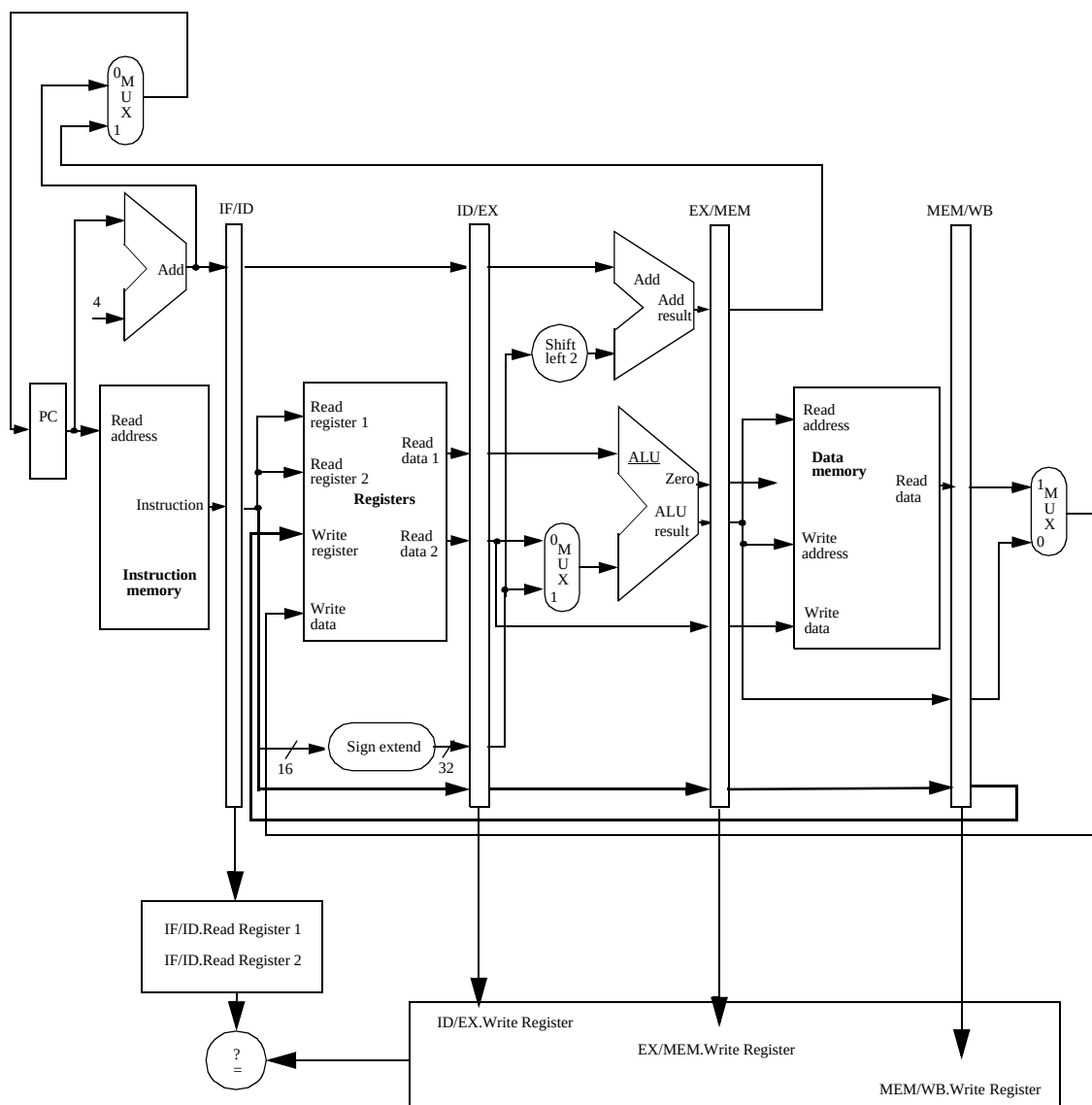


Bild 2.53. RAW-Konflikterkennung

Da nicht alle Befehle ein Register beschreiben, kann ein Konflikt nur dann auftreten, wenn das Signal RegWrite aktiviert ist.

ID/EX.RegWrite = 1?			
ja		nein	
ID/EX.RegDst =		kein Konflikt	
1 (R-format)			
ID/EX.RegDst =		0 (lw)	
IF/ID.ReadReg 1/2 = ID/EX.WriteReg Rd		IF/ID.ReadReg 1/2 = ID/EX.WriteReg Rt	
ja	nein	ja	nein
Konflikt	kein Konflikt	Konflikt	kein Konflikt

Bild 2.54. Konflikterkennung zwischen der EX- und der ID-Stufe. Da das Zielregister für R-format-Befehle und Load-Befehle an unterschiedlichen Stellen im Befehl steht (Rd bzw. Rt), muß noch eine entsprechende Unterscheidung mit Hilfe von ID/EX RegDst getroffen werden. Die Konflikterkennungen für MEM-ID und WB-ID sind ähnlich aber einfacher.

Die logischen Gleichungen für alle 3 Konflikterkennungen lauten:

EX-ID-Konflikt:

$$\text{ID/EX.RegWrite and}$$

$$((\text{ID/EX.RegDst} = 0 \text{ and ID/EX.WriteRegisterRt} = \text{IF/ID.ReadRegister1}) \text{ or}$$

$$(\text{ID/EX.RegDst} = 1 \text{ and ID/EX.WriteRegisterRd} = \text{IF/ID.ReadRegister1}) \text{ or}$$

$$(\text{ID/EX.RegDst} = 0 \text{ and ID/EX.WriteRegisterRt} = \text{IF/ID.ReadRegister2}) \text{ or}$$

$$(\text{ID/EX.RegDst} = 1 \text{ and ID/EX.WriteRegisterRd} = \text{IF/ID.ReadRegister2}))$$

MEM-ID-Konflikt:

$$\text{EX/MEM.RegWrite and}$$

$$((\text{EX/MEM.WriteRegister} = \text{IF/ID.ReadRegister1}) \text{ or}$$

$$(\text{EX/MEM.WriteRegister} = \text{IF/ID.ReadRegister2}))$$

WB-ID-Konflikt:

$$\text{MEM/WB.RegWrite and}$$

$$((\text{MEM/WB.WriteRegister} = \text{IF/ID.ReadRegister1}) \text{ or}$$

$$(\text{MEM/WB.WriteRegister} = \text{IF/ID.ReadRegister2}))$$

Diese letzte Konflikterkennung kann entfallen, falls im selben Takt geschrieben und gelesen werden kann (Zyklenteilung).

1 Konfliktbehandlung

Aktionen:

1. PC bleibt gleich, wird nicht überschrieben: PCWrite = 0.
2. Befehl im IF/ID-PL-Register bleibt gleich: IF/IDWrite = 0.
3. Der Befehl in der ID-Stufe wird nicht weitergeschoben. Stattdessen:
Einfügen von 0...0 im ID/EX-PL-Register. Entspricht der Erkennung einer "Bubble". Zur Auswahl von 0...0 oder den von Main Control erzeugten Steuersignalen wird ein neuer MUX eingefügt.

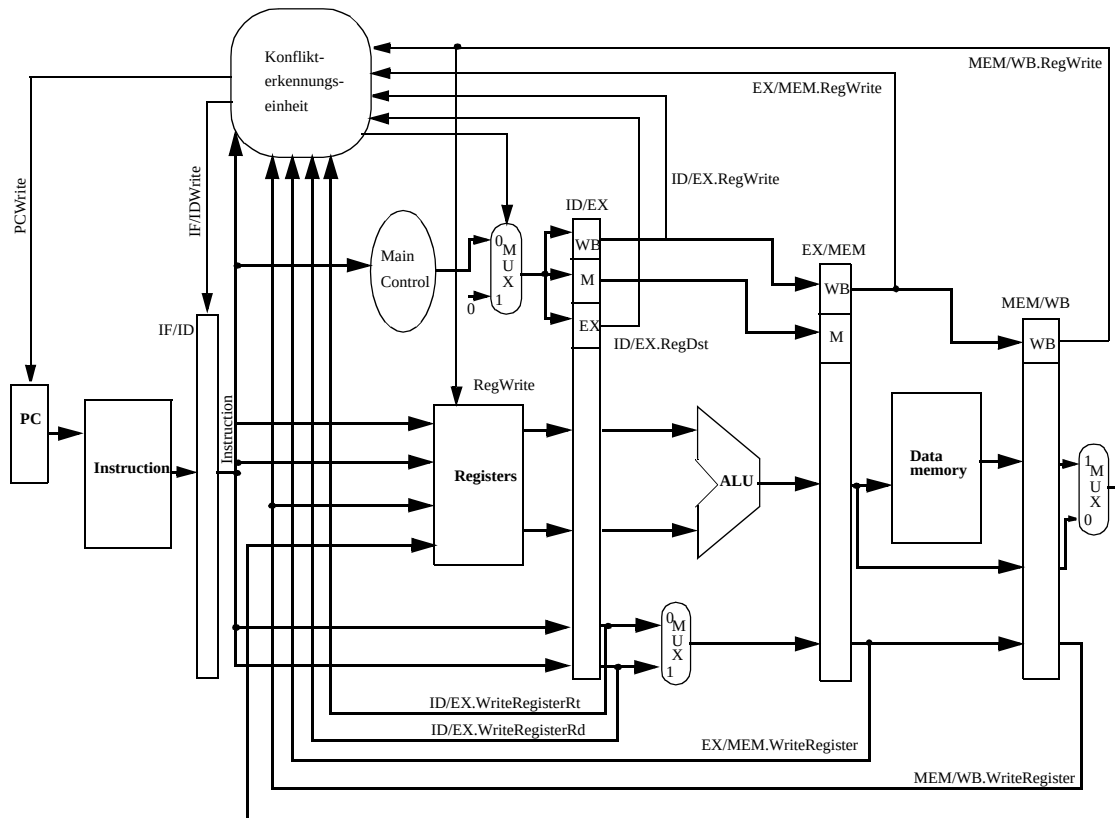


Bild 2.55. Erweiterung des PL-Datenpfades um eine Konflikterkennungseinheit.

2.3.9. Forwarding

Das Einfügen von "Bubbles" oder Stalls durch die Konflikterkennungseinheit kostet Performance. Besser: Register-Inhalte, welche bereits von der ALU berechnet wurden (vorheriger Befehl: aus EX/MEM-PL-Register; vor-voriger Befehl: aus MEM/WB-PL-Register), werden über zusätzliche MUXe an den beiden ALU-Eingängen ausgewählt: Forwarding oder Bypassing.

Die beiden MUXe mit je 3 Eingängen werden durch die Signale ALUSelA bzw. ALUSelB angesteuert.

Quelle für ALU-Eingang	normales Register	Vorgänger-Befehl (EX/MEM)	Vor-Vorgänger-Befehl (MEM/WB)
ALUSelA bzw. ALUSelB	00	01	10

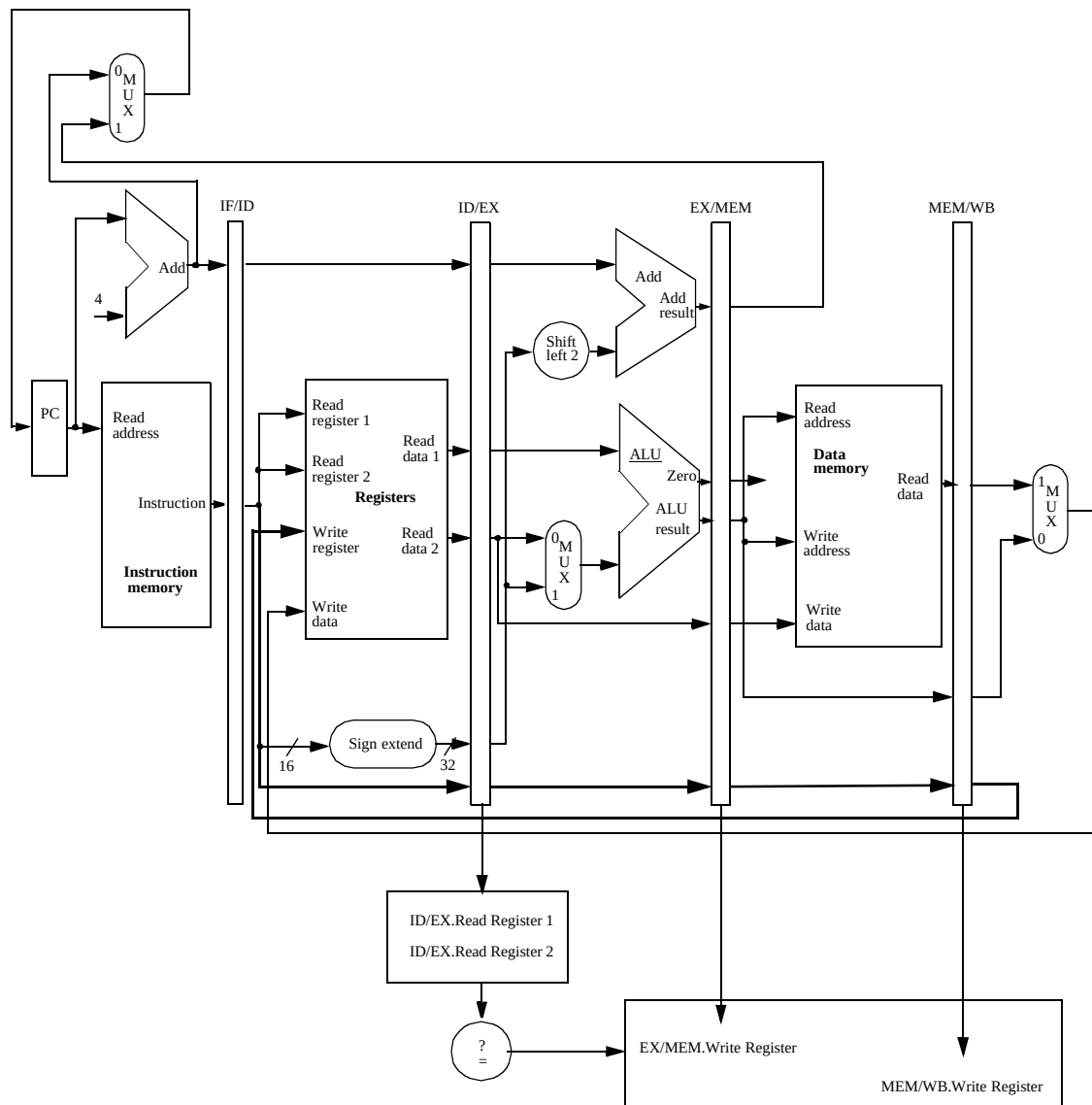


Bild 2.56. Forwarding-Erkennung

EX/MEM.RegWrite and (EX/MEM.WriteReg = ID/EX.ReadReg 1)	false	true	false	true
MEM/WB.RegWrite and (MEM/WB.WriteReg = ID/EX.ReadReg 1)	false	false	true	true
ALUSelA	00	01	10	01

Bild 2.57. Forwarding-Logik zur Ansteuerung von ALUSelA. ALUSelB analog.

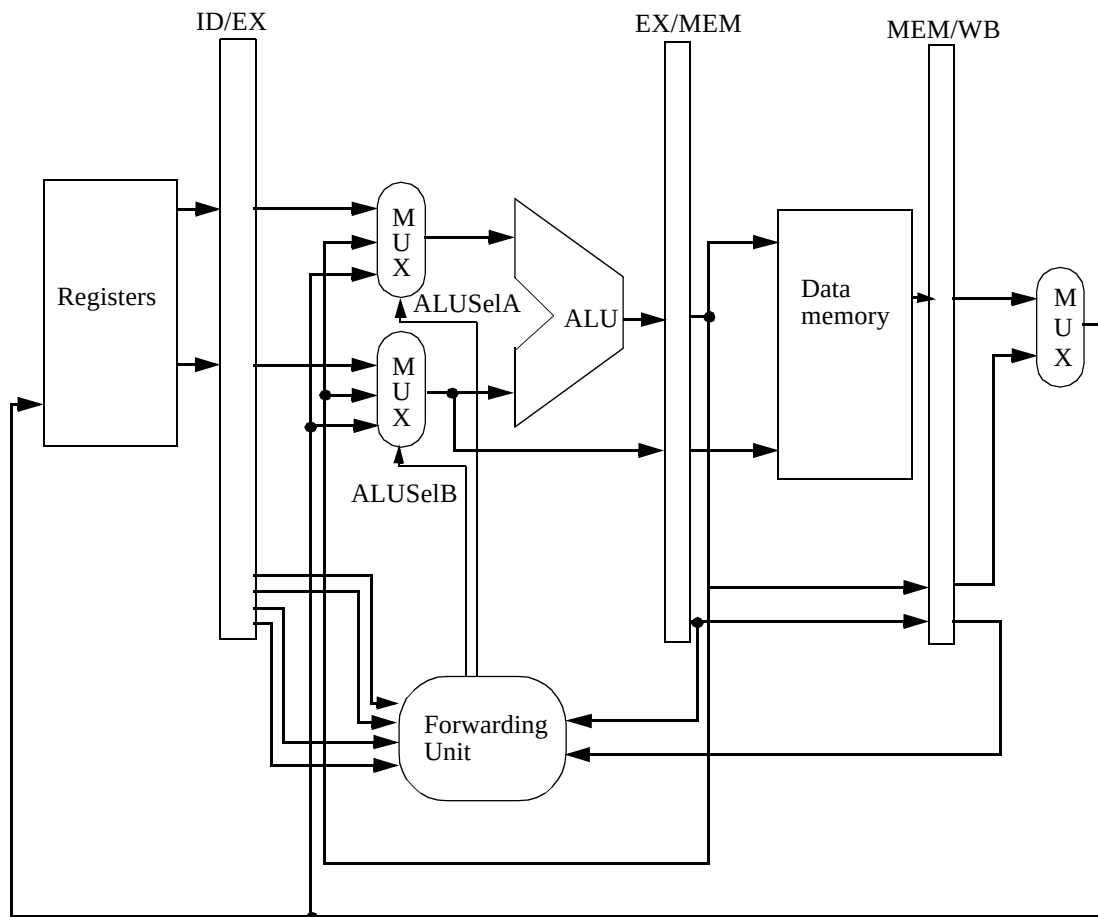


Bild 2.58. Erweiterter Datenpfad mit Forwarding Unit und 2 Multiplexern an den ALU-Eingängen.

Da die Forwarding Unit eine bestimmte Gruppe von Datenkonflikten ausschaltet, kann eine Konflikterkennungseinheit entfallen.

Ausnahme: Load-Befehl, dessen Datum im Folgebefehl benötigt wird. In diesem Fall muß die Forwarding Unit 1 Stall einfügen (hier nicht im Detail behandelt).

2.3.10. Steuerkonflikte

- zu ergänzen -

2.3.11. Exceptions

- zu ergänzen -

2.4. Superskalarität

Die in den bisherigen Kapiteln genannten Restriktionen machen PLs leichter realisierbar, aber sie kosten Performance. Die Restriktionen werden nun schrittweise aufgegeben.

1. 1 Takt für EX
2. genau 1 Ausführungseinheit
3. lineare Einspeisung
4. max. 1 Befehlseinspeisung/Takt

2.4.1. Mehrzyklenbefehle

l Probleme:

- Es gibt spezialisierte Ausführungseinheiten für Multiplikation, Division, Addition, Integeroperationen.
- Es sollen möglichst alle Ausführungseinheiten voll ausgelastet werden.
- Die EUs benötigen unterschiedliche Zyklenzahl, z.B. Integer-Unit 1 Takt, Division-Unit 50 Takte.

--> Restriktionen 1 und 2 werden aufgehoben.

l Erweiterte PL:

- Integer-Unit: Loads/Stores (Integer-Register und FP-Register), Integer-ALU-Operationen (außer Multiplikation, Division), Branches
- Multiplikationseinheit: FP und Integer (ca. 10 Takte)
- Divisionseinheit: FP und Integer (20-50 Takte)
- Addierer: FP (ca. 5 Takte)

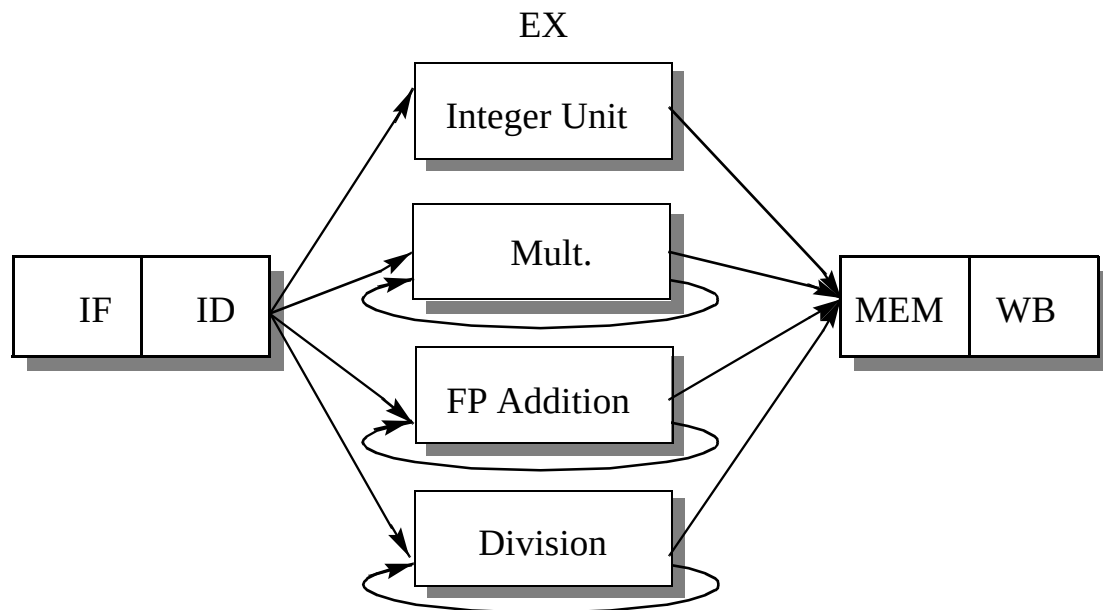


Bild 2.59. Erweiterte DLX-PL: die drei Arithmetik-Einheiten werden u.U. mehrfach durchlaufen

Die EUs sind zunächst nicht gepipelined. Folge: Ein neuer Befehl kann erst dann in eine EU eingespeist werden, wenn der Vorgängerbefehl abgearbeitet ist. Folge: Rückstau in ID, IF; Überholen ist vorläufig nicht möglich.

1 Komplikationen:

- Konfliktfenster kann bis 50 Takte groß sein: neue Lösung für dynamische Konflikterkennung nötig.
- zusätzliche Ressourcenkonflikte möglich in WB: IU schreibt in FP-Reg. (Load), FP-Einheit schreibt Ergebnis in FP-Reg.
- FP-Einheiten benötigen variable Taktzahl, können deshalb gleichzeitig oder in falscher Reihenfolge fertig werden: WAW.

Beispiel 2-12:

WAW-Konflikt

DIV	F0,F2,F4	40	Takte
SUB	F0,F8,F80	1	Takt

- kein WAR-Konflikt möglich, da lineare Einspeisung!

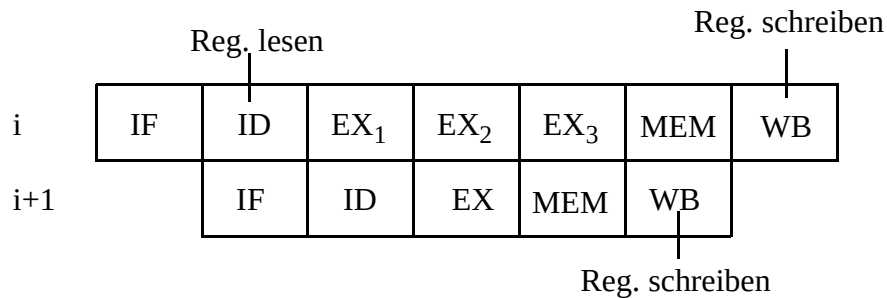


Bild 2.60. Wird linear eingespeist (i+1 nach i), dann kann Bef. i+1 nie schreiben bevor i gelesen hat: kein WAR. Aber: Bef. i kann Ergebnis von i+1 überschreiben: WAW!

1 Performance

Konfliktbehebung durch Einfügen von Stall-Zyklen ist teuer, wenn sich lange FP-Operationen in der PL befinden. Messung am Simulationsprogramm Spice auf MIPS R2010/3010 FPU: 35% aller Zyklen sind Stalls.

Abhilfe: Überholen wird zugelassen, d.h. nicht-lineare Einspeisung (Restriktion 3 wird gestrichen.)

2.4.2. Nichtlineare Einspeisung

Lineare Einspeisung erzeugt Rückstau, obwohl EUs frei sind.

Beispiel 2-13:

i	DIVF F0,	F2,	F4
i+1	ADDF F10,	F0,	F8
i+2	SUBF F8,	F8,	F14
i+1	-> i	:	RAW
i+2	-> i	:	keine Abhängigkeit
i+2	-> i+1	:	WAR

Vorgehen für nicht-lineare (*out-of-order*) Einspeisung:

1.	IF	Holen	Befehl holen und decodieren, linear
2.	II	Zuordnen	Befehl einer EU zuordnen (<i>issue</i>), dazu: Überprüfung auf Ressourcenkonflikt (ist die EU frei ?) und WAW-Konfliktprüfung.
3.	IS	Einspeisen	RAW-Konfliktprüfung, Operanden lesen, in der EU zur Ausführung bringen (<i>schedule</i>)
4.	EX	Ausführen	WAR-Konfliktprüfung, Register schreiben
5.	MEM	Speicherzugriff	
6.	WB	Register schreiben	

IF, II :linear

IS :Wartezyklen wegen Datenabhängigkeiten
-> Überholen möglich

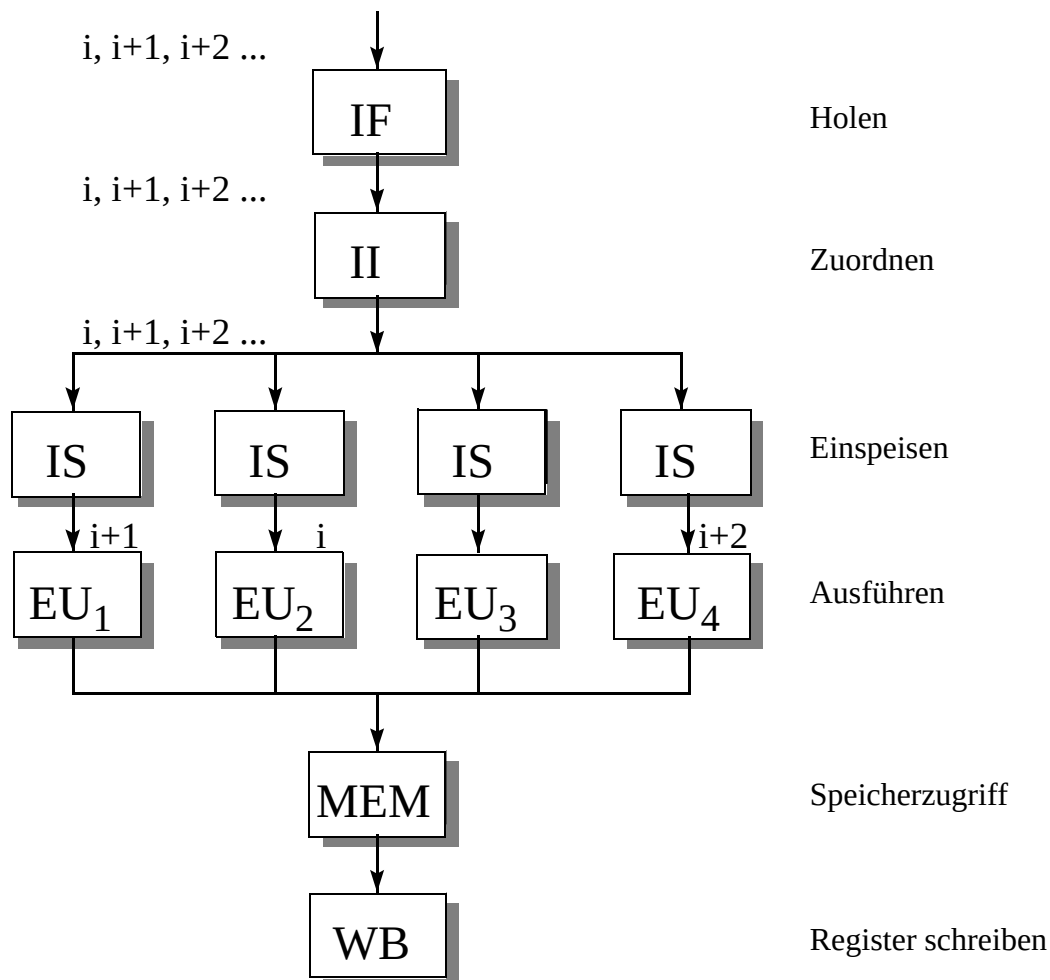


Bild 2.61. PL mit nicht-linearer Einspeisung

2.4.3. Scoreboard

Die 4 nötigen Konfliktprüfungen (dyn.) sind möglich, wenn der vollständige Prozessorzustand bekannt ist:

- Welche Befehle befinden sich in der PL (Befehlsstatus)
- Womit sind die EUs beschäftigt (EU-Status)
- Sind die Zielregister verfügbar (Register-Result-Status)?

Das Scoreboard faßt diese Information zusammen (Datenstruktur), der Scoreboard-Manager entscheidet auf Grund dieser Information über Zuordnung, Einspeisung und eventuelle Wartezyklen.

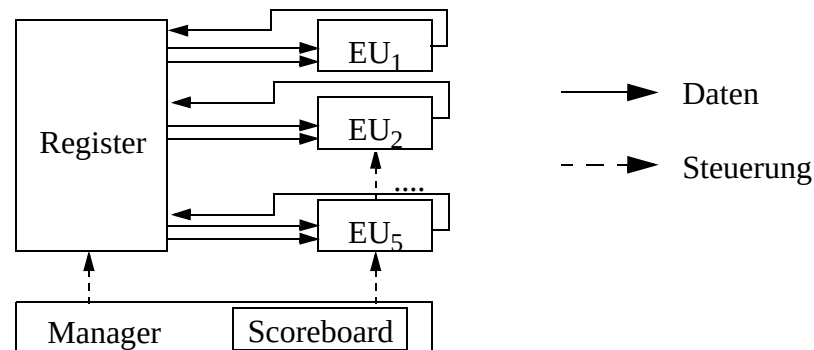


Bild 2.62. Ablaufsteuerung durch Scoreboard(-Manager)

Ablauf einer Befehlsabarbeitung mittels Scoreboard:

	Scoreboard - Aktion
IF	
II	(1) Eintrag des Befehls in Befehlsstatustabelle; (2) Ressourcenprüfung: ist die benötigte EU frei? (Zugriff auf EU-Status) (3) WAW-Prüfung: ist das Zielregister frei? (Zugriff auf Register-Result-Status) wenn ja: Belegen des Registers (4) Falls Konflikt in Schritt (2) oder (3): stall, kein IF mehr! (5) Falls kein Konflikt: markiere II-Stufe für Befehl i als erledigt, gib Befehl i weiter an IS.
IS	(1) Aktualisiere EU-Status (Befehl, Ziel, Quelle 1, Quelle 2) (2) RAW-Prüfung: ist ein Quellregister zum Schreiben belegt? (Zugriff Register-Result-Status) Wenn ja: markiere Quelle als nicht vorhanden, notiere zuständige Quell-EU, stall. (3) Falls kein Konflikt: lies Quellregister, markiere IS-Stufe für Befehl i als erledigt, gib Befehl i weiter an EU.
EX	nach Ende der Ausführung: markiere EX-Stufe für Befehl i als erledigt
MEM	leer, kein Speicherzugriff
WB	(1) WAR-Prüfung: ist das Zielregister gleich dem Quellregister eines älteren Befehls? Wenn ja: stall (2) Falls kein Konflikt: schreibe Ergebnis ins Zielregister; markiere WB-Stufe für Befehl i als erledigt; entferne Befehl i aus dem Scoreboard.

Befehlsstatus								
Befehl			II	IS	EX	WB		
LF	F6, 34 (R2)		x	x	x	x		
LF	F2, 45 (R3)		x	x	x			
MULTF	F0, F2, F4		x					
SUBF	F8, F6, F2		x					
DIVF	F10, F0, F6		x					
ADDF	F6, F8, F2							

EU-Status								
EU Nr.	EU-Name	busy	Operation	Ziel	Q1	Q2	EU_{Q1}	EU_{Q2}
1	INTEGER	j	LF	F2	R3			
2	MULT1	j	MULTF	F0	(F2)	F4	1	
3	MULT2	n						
4	ADD	j	SUBF	F6	F8	(F2)		1
5	DIVIDE	j	DIVF	F10	(F0)	F6	2	

Register Result -Status								
Register	F0	F2	F4	F6	F8	F10	...	F30

Bild 2.63. Momentaufnahme des Scoreboards (1). Ein eingeklammelter Registername bedeutet, daß das Register noch nicht geschrieben wurde. Der Quelloperand muß dann von der entsprechenden EU geholt werden.

Beispiel 2-14:

Ausgehend vom Scoreboard (1) ist der jeweilige Status darzustellen für die Zeitpunkte:

Scoreboard (2): MULTF befindet sich in WB

Scoreboard (3): DIVF befindet sich in WB

Ausführungszeiten: ADDF, SUBF: 2 Takte

MULTF 20 Takte

DIVF 40 Takte

Scoreboard (2):

Befehlsstatus					
Befehl		II	IS	EX	WB
LF	F6, 34 (R2)	x	x	x	x
LF	F2, 45 (R3)	x	x	x	x
MULTF	F0, F2, F4	x	x	x	
SUBF	F8, F6, F2	x	x	x	x
DIVF	F10, F0, F6	x			
ADDF	F6, F8, F2	x	x	x	

EU-Status								
EU Nr.	EU-Name	busy	Operation	Ziel	Q1	Q2	EU_{Q1}	EU_{Q2}
1	INTEGER	n						
2	MULT1	j	MULTF	F0	F2	F4		
3	MULT2	n						
4	ADD	j	ADDF	F6	F8	F2		
5	DIVIDE	j	DIVF	F10	(F0)	F6	2	

Register Result -Status								
Register	F0	F2	F4	F6	F8	F10	...	F30

Beide Ladebefehle sind beendet. Damit stand F2 bereit, Einspeisung von MULTIF und SUBF. SUBF nach 2 Takten beendet, dadurch Lösung des Ressourcenkonflikts für ADDF.

Bestehende Konflikte: WAR von DIVF nach ADDF
RAW von MULTF nach DIVF

Scoreboard (3):**Befehlsstatus**

Befehl		II	IS	EX	WB
LF	F6, 34 (R2)	x	x	x	x
LF	F2, 45 (R3)	x	x	x	x
MULTF	F0, F2, F4	x	x	x	x
SUBF	F8, F6, F2	x	x	x	x
DIVF	F10, F0, F6	x	x	x	
ADDF	F6, F8, F2	x	x	x	x

EU-Status

EU Nr.	EU-Name	busy	Operation	Ziel	Q1	Q2	EU_{Q1}	EU_{Q2}
1	INTEGER	n						
2	MULT1	n						
3	MULT2	n						
4	ADD	n						
5	DIVIDE	j	DIVF	F10	F0	F6		

Register Result -Status

Register	F0	F2	F4	F6	F8	F10	...	F30
EU-Nr.	5							

Beendigung von MULTF löst RAW-Konflikt, DIVF gestartet. Nach Lesen der Quelloperanden von DIVF ist auch der WAR-Konflikt gelöst, ADDF wurde gestartet und nach 2 Takten beendet.

2.4.4. Entkoppelte Architektur

Echter Konflikt: Der Ablauf eines Befehls ist nicht möglich, weil die Quelldaten nicht verfügbar sind.

Unechter Konflikt: Das Schreiben in ein Register ist nicht möglich, weil das Register noch anderweitig belegt ist.

Unechte Konflikte können vermieden werden z.B. durch andere Registerzuweisung durch den Compiler.

Alternative: Auf Register wird ganz verzichtet, wenn sie nur zur Zwischenspeicherung (Scratchregister) verwendet werden. Es ergibt sich damit eine Art von Datenflußarchitektur.

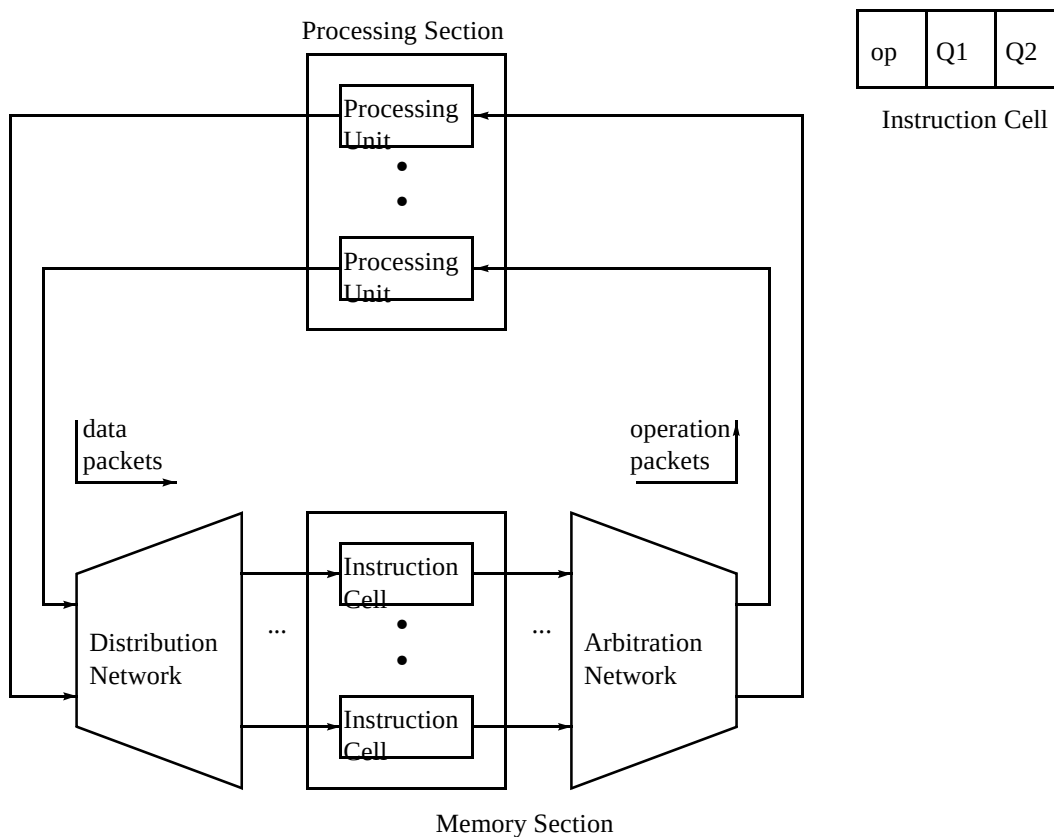


Bild 2.64. Prinzip eines Datenflußrechners: Ergebnisse von Befehlen (data packets) werden über das Distribution Network in die jeweiligen Instruction Cells eingespeist. Sobald alle Quelloperanden verfügbar sind, kann die Zelle ausgeführt werden und wird dazu über das Arbitration Network einem Prozessor zugewiesen.

Idee der entkoppelten Architektur von R. Tomasulo, benutzt in der IBM 360 / 91 FPU. Jede EU erhält eine "Reservierungsstation", sie entspricht der Instruction Cell in der Datenflußarchitektur. Registernamen werden eliminiert und durch Datenflußpfeile ersetzt:

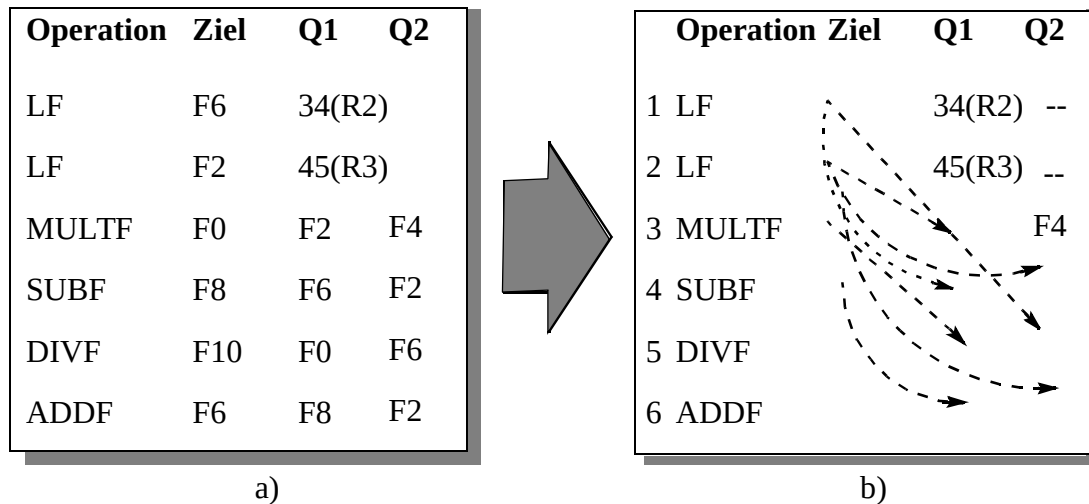


Bild 2.65. Äquivalente Darstellungen von Datenbeziehungen zwischen Befehlen eines Programmstücks:

a) durch Registernamen, b) durch Datenflußpfeile

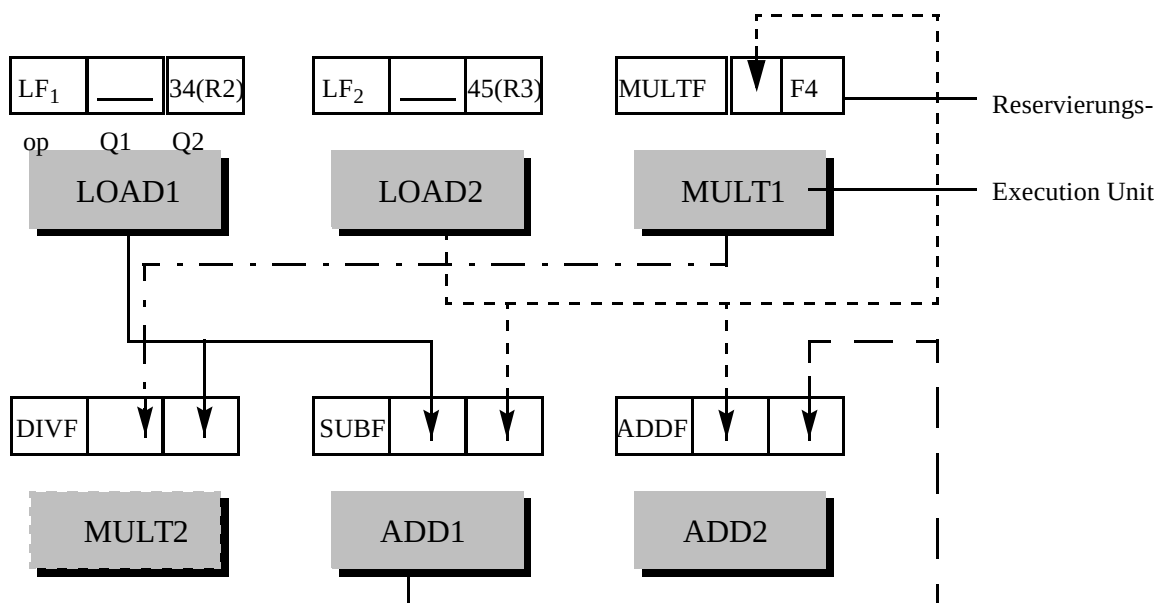


Bild 2.66. Übertragung der Datenflußpfeile in eine Architektur mit 6 EUs.

Realisierung: Zum Transport der Ergebnisse zu den Reservierungsstationen wird ein Medium gebraucht, das

- volle Erreichbarkeit garantiert
- schnell genug ist, um keinen Engpaß darzustellen.

IBM 360-Lösung: Common Data Bus CDB.

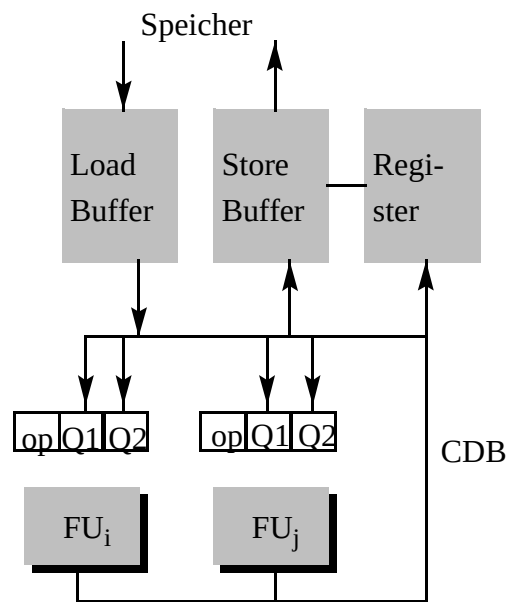


Bild 2.67. Realisierung der Tomasulo-Architektur. CDB sequentialisiert die Abarbeitung tlw.

Befehlsabarbeitung:

1. Issue (Zuordnung): Hole und decodiere Befehl, ordne ihn einer freien EU zu, hole Quellope-
randen a) aus einem Register, oder b) aus dem Load Buffer, (Res-
ourcen Konflikt?)
2. Execute (Ausführung): Hole (restliche) Operanden von der zuständigen EU. Wenn RAW-
Konflikt: überwache CDB, bis Operand verfügbar.
3. Write result: Schreibe das Ergebnis auf CDB. Es wird von da automatisch übernommen in
alle Reservierungsstationen oder Register, welche darauf warten.

Vorteil: Keine WAR-, WAW-Konflikte!

Implementierung der Datenflußpfeile durch Tag-Felder (4 bit): Tag in der Reservierungsstation
bezeichnet die EU, auf deren Ergebnis gewartet wird.

Ablauf kann dargestellt werden ähnlich wie für Scoreboard. Befehlsstatustabelle ist hier nur
zum Verständnis angegeben, ist aber nicht in der HW realisiert.

Beispiel 2-15:

Gleiches Programm wie im Scoreboard-Fall.

Start: Der zweite Ladebefehl befindet sich in der Ausführung, die anderen Befehle sind bereits jeweils einer Reservierungsstation zugeordnet:

Tomasulo(1):

Befehlsstatus				
Befehl		Issue	Execute	Write result
LF	F6, 34 (R2)	x	x	x
LF	F2, 45 (R3)	x	x	
MULTF	F0, F2, F4	x		
SUBF	F8, F6, F2	x		
DIVF	F10, F0, F6	x		
ADDF	F6, F8, F2	x		

EU-Status							
EU Nr.	EU-Name	busy	Operation	Q1	Q2	EU_{Q1}	EU_{Q2}
1	ADD1	j	SUBF	LOAD1			(LOAD2)
2	ADD2	j	ADDF				(ADD1) (LOAD2)
3	ADD3	n					
4	MULT1	j	MULTF		F4		(LOAD2)
5	MULT2	j	DIVF		LOAD1		(MULT1)

Register Result -Status								
Register	F0	F2	F4	F6	F8	F10	...	F30
Quelle	MULT1	LOAD2		ADD2	ADD1	MULT2		

EU-Status: Jeder Quelloperand ist entweder bereits vorhanden (verzeichnet unter Q1 oder Q2) oder wird von einer anderen EU erwartet (verzeichnet unter EU_{Q1} bzw. EU_{Q2}, in Klammern)

Register-Result-Status: Jedem Register ist hier seine Quell-EU zugeordnet.

Tomasulo(2):

Befehlsstatus				
Befehl		Issue	Ececute	Write result
LF	F6, 34 (R2)	x	x	x
LF	F2, 45 (R3)	x	x	x
MULTF	F0, F2, F4	x	x	
SUBF	F8, F6, F2	x	x	x
DIVF	F10, F0, F6	x		
ADDF	F6, F8, F2	x	x	x

EU-Status							
EU Nr.	EU-Name	busy	Operation	Q1	Q2	EU_{Q1}	EU_{Q2}
1	ADD1	n					
2	ADD2	n					
3	ADD3	n					
4	MULT1	j	MULTF	LOAD2 F4			
5	MULT2	j	DIVF		LOAD1 (MULT1)		

Register Result -Status								
Register	F0	F2	F4	F6	F8	F10	...	F30
Quelle	MULT1					MULT2		

Zeitpunkt: Vor dem Schreiben des MULTF-Ergebnisses.
 SUBF ist abgeschlossen,
 ebenso ADDF: hier kein Ressourcenkonflikt! DIVF wartet auf F0.

2.5. Ein/Ausgabe

E/A (oder I/O) ist ein von der Forschung häufig vernachlässigtes Gebiet.

Grund: wenige einheitliche Mechanismen und Konzepte, große Anzahl unterschiedlicher Endgeräte, starke Marktabhängigkeit (z.B. bei Busstandards).

Hier: Behandlung der wichtigsten Konzepte, nicht von Einzelgeräten.

I/O beschäftigt sich mit dem Anschluß von I/O-Geräten an den Basis-Computer (Prozessor + Cache + Hauptspeicher).

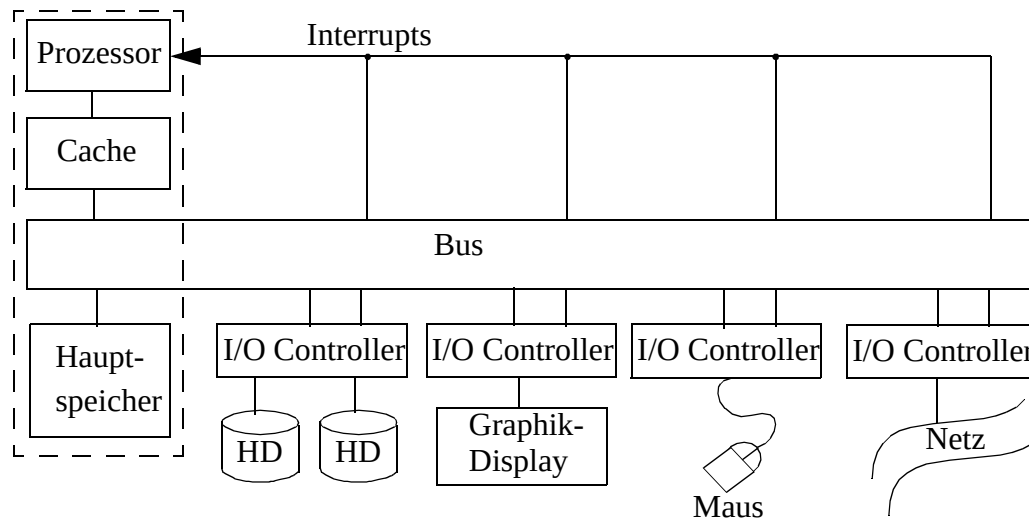


Bild 2.68. Mögliche I/O-Architektur

Fragen:

- Wie messe ich I/O-Performance?
- Welche I/O-Geräte sollen angeschlossen werden? Wie können I/O-Geräte charakterisiert/ klassifiziert werden?
- Welche Kommunikationsstrukturen, d.h. hier: Busstrukturen, werden verwendet? Wie werden Busse realisiert?
- Wie erfolgt die Kommunikation zwischen I/O-Geräten und Prozessor?

2.5.1. I/O-Performance

Die Bestimmung der I/O-Performance erfolgt mit unterschiedlichen Metriken, je nach den Anforderungen.

Supercomputer: Relativ seltene Ausgabe großer Datenmengen vom Prozessor/Hauptspeicher auf die Platte. Maß: MB/sec

Datenbanken: Typische Datenbankzugriffe in kommerziellen Systemen umfassen kleine Datenmengen (Kontoauszug, Buchung einer Reise). Maßgeblich: Anzahl der Transaktionen pro Zeiteinheit, die die Datenbank abwickeln kann: Transactions per second (TPS).

Interaktiver Betrieb: Hier ist die Wartezeit von der Erteilung bis zur Erfüllung des Auftrags (= Transaktion) maßgeblich: Antwortzeit (Responsetime).

Ein hoher Durchsatz (in TPS) bedeutet nicht notwendigerweise kurze Antwortzeiten!

Problematik: Antwortzeit hängt ab von

- konstantem Anteil (Setup-Zeit, z.B. für SW-Anteil zur Aufbereitung des Auftrags),
- datenabhängigem Anteil, bestimmt durch die (Peak-) Datenrate $d(\text{MB/s})$ und die Datenmenge $D(\text{MB})$,
- der Anzahl n von nacheinander ausgeführten Zugriffen.

$$t_{\text{Antwort}} = n \cdot \left(t_{\text{const}} + \frac{D}{d} \right)$$

Je nach Größe der Parameter kann die Antwortzeit dominiert werden durch die konstanten Zeiteile oder durch die datenabhängigen.

I/O-Benchmarks werden verwendet zur Messung einzelner Parameter:

- Supercomputer-I/O: Zeitmessung für die Übertragung sehr großer ($\rightarrow \infty$) Blöcke, Bestimmung von d .
- Transaktionssysteme: Benchmark TPC-B simuliert ein Netz von Bankautomaten. Messung der erfolgreichen (d.h. innerhalb vorgegebener Antwortzeit) Transaktionen/sec (Service rate). Umfang einer Transaktion: 2 bis 10 Plattenzugriffe, 5.000 bis 20.000 CPU-Instruktionen pro Plattenzugriff. Mit steigender Größe des Benchmarks (Anzahl der Bankautomaten) steigt auch die Größe der Datenbank (verhindert HS-residente Datenbank!).

- File-System-Benchmarks

Nachbildung typischer Zugriffsmuster (UNIX):

- 80% der Zugriffe < 10 KB
- 90% der Zugriffe auf sequentielle Blöcke
- 67% Lesezugriffe
- 27% Schreibzugriffe
- 6% Lese-/Schreibzugriffe

Abfolge typischer Befehle: MakeDir, Copy, ScanDir, ReadAll, Make.

2.5.2. I/O-Geräte (Maus, HD, Netz)

I/O-Geräte können charakterisiert werden gemäß:

- Verhalten: Input (read once), Output (write only), Speicher (mehrfaches Lesen, mehrfaches Schreiben).
- Partner: Mensch oder Maschine, lesend oder schreibend.
- Datenrate: Peakrate zwischen I/O-Gerät und HS.

Charakterisierung wichtiger I/O-Geräte:

Gerät	Verhalten	Partner	Datenrate d (KB/sec)
Tastatur	Input	Mensch	0.01
Maus	Input	Mensch	0.02
Spracheingabe	Input	Mensch	0.02
Scanner	Input	Mensch	200.00
Sprachausgabe	Output	Mensch	0.60
Line Printer	Output	Mensch	1.00
Laser Printer	Output	Mensch	100.00
Graphic Display	Output	Mensch	30 000.00
Netzwerk-Terminal	Input oder Output	Maschine	0.05
Netzwerk-LAN	Input oder Output	Maschine	200.00
Floppy Disk	Speicher	Maschine	50.00
Optical Disk	Speicher	Maschine	500.00
Magnet-Band	Speicher	Maschine	2000.00
Magnet-Disk	Speicher	Maschine	2000.00

2.5.3. Busse

Busse sind Leitungsbündel zur Übertragung von Informationen zwischen Prozessor, Speicher und I/O-Geräten.

1 Ein Bus enthält Steuerleitungen und Datenleitungen.

Steuerleitungen: Request, Acknowledge, Übertragungsrichtung, Informationstyp usw.

Datenleitungen: Daten, Adressen Kommandos.

Busse können gemultiplext sein, d.h. Daten und Adressen werden in Zeitmultiplex über dieselben (Daten-)Leitungen übertragen.

1 Die Busgeschwindigkeit ist von einer Vielzahl von Faktoren abhängig:

- Schaltungstechnik der Treiber
- Buslänge
- Anzahl angeschlossener Geräte
- Busbreite
- Protokoll, spez. Arbitrierungsverfahren
- Kollisionen

1 Bus-Typen:

Man unterscheidet Prozessor-Speicher-Busse, I/O-Busse und Backplane-Busse.

Prozessor-Speicher-Busse: schnell, kurz, spezialisiert für bestimmten Prozessor und Speicher (proprietary), synchron.

I/O-Bus: länger, viele unterschiedliche I/O-Geräte, unterschiedliche Datenraten, Standards!

Backplane-Bus: vereinigt Prozessor-Speicher-Bus und I/O-Bus, u.U. Kompromißlösung; Name kommt von der Verbindungsplatine, in die Prozessor-, Speicher-, und I/O-Boards eingesteckt werden. Standards!

1 Bus-Strukturen:

Systeme werden je nach Leistungs- und Kostenzielen mit einer Kombination dieser Bus-Typen aufgebaut.

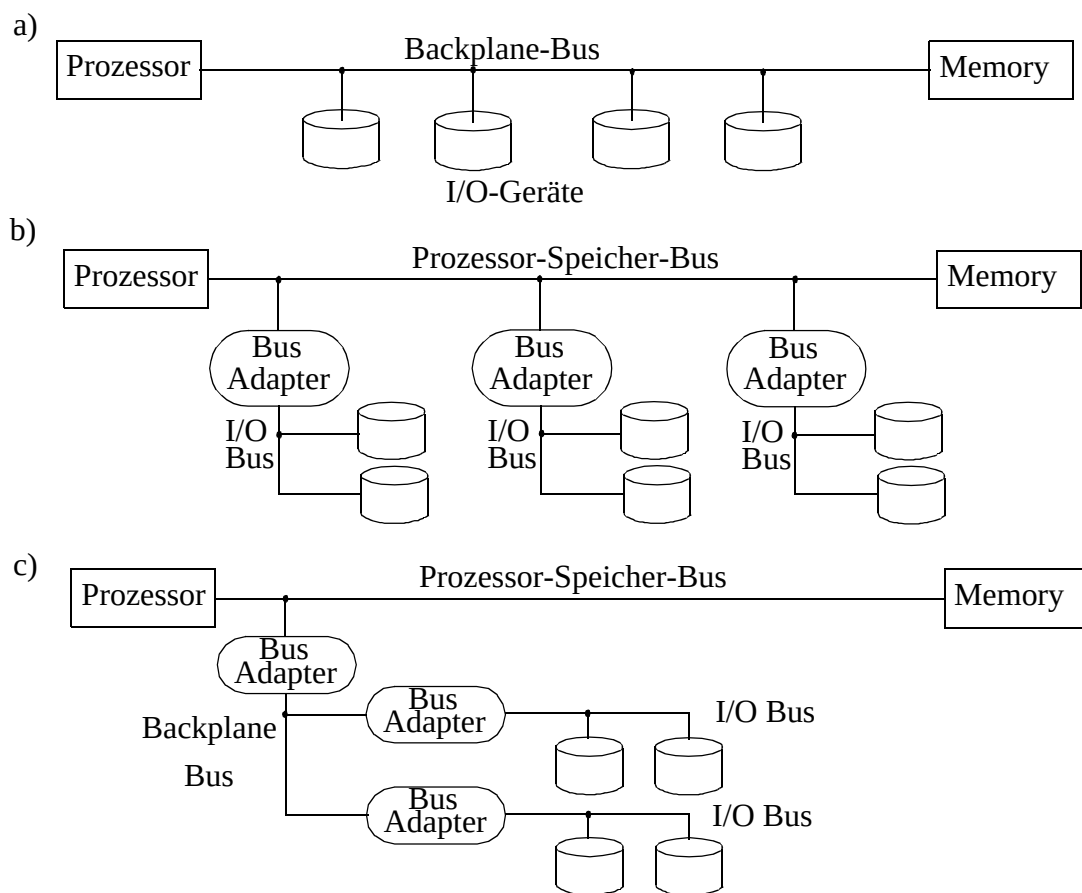


Bild 2.69. Bus-Strukturen:

a) Single Bus. Setzt voraus, daß der Backplane-Bus de facto-Standard ist.

Beispiel: IBM PC: AT-Bus

b) Spezieller High-Speed Prozessor-Memory-Bus (kann ein Backplane-Bus sein) sowie Adapter für unterschiedliche I/O-Bus-Systeme.

Beispiel: Apple McIntosh; P-Mem_Bus = Nubus, I/O-Bus = SCSI.

c) P-Speicher-Bus wird nur durch 1 Bus-Adapter belastet (z.B. SCSI Single-Chip-Controller).

1 Kommunikationsprotokolle regeln den Ablauf eines Dialogs zwischen Teilnehmern. Auf unterer Ebene wird unterschieden zwischen synchroner und asynchroner Kommunikation.

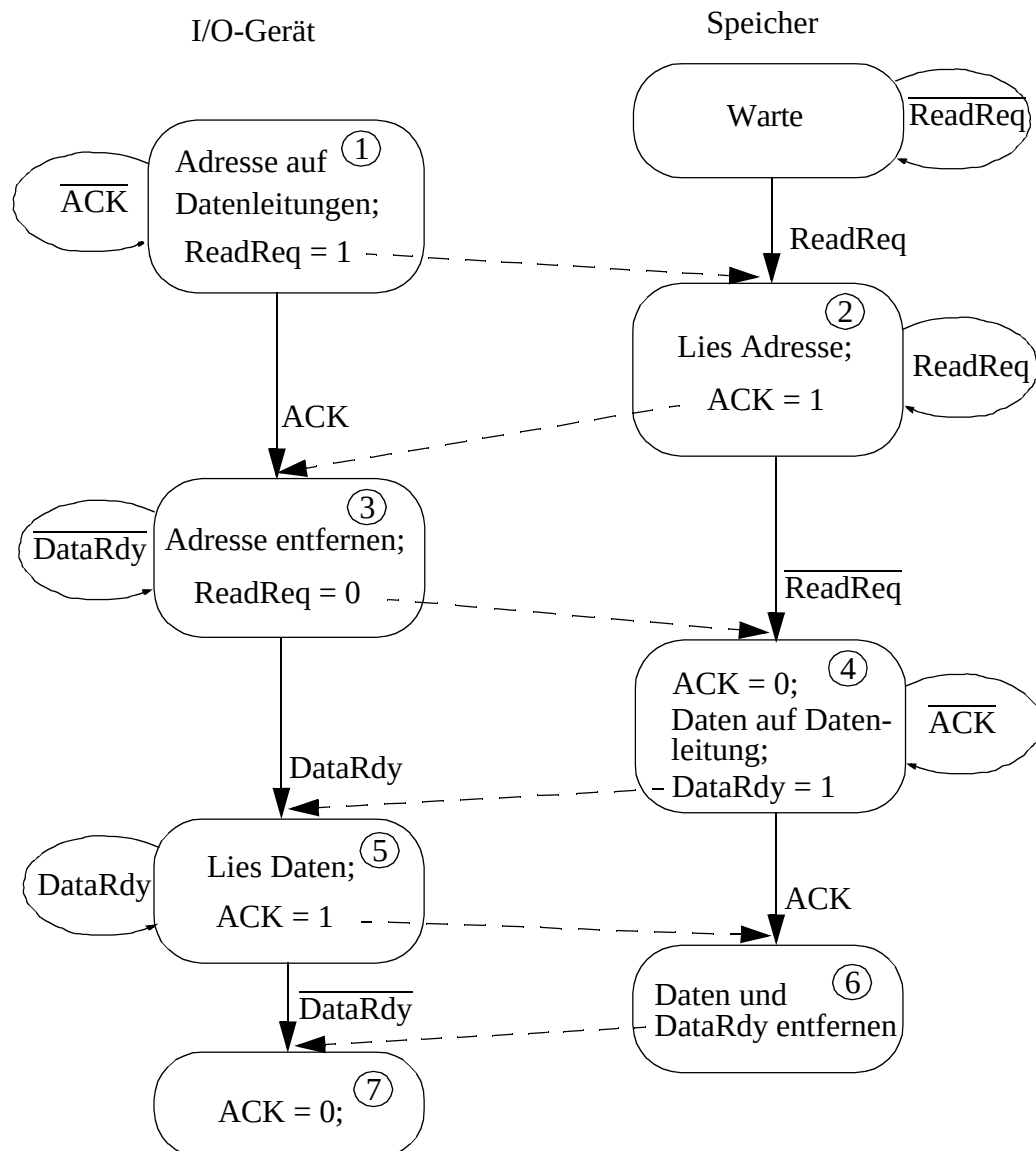
- Synchron: gemeinsamer Takt zwischen Kommunikationspartnern, fester zeitlicher Ablauf, einfach zu realisieren (FSM), schnell, unflexibel (alle Geräte mit gleicher Geschwindigkeit); kurze Distanzen wegen Clock-Skew.
- Asynchron: kein Takt, unterschiedliche Geschwindigkeit, flexibel, Handshake-Protokoll (mittels spez. Steuerleitungen), langsamer.

Ablauf einer Transaktion mittels Handshake:

3 Steuerleitungen:

- ReadReq: I/O-Gerät signalisiert Leseanforderung.
- DataRdy: Schreiben: I/O-Gerät signalisiert Datenverfügbarkeit; Lesen: Speicher signalisiert Datenverfügbarkeit.
- Ack: Bestätigung des ReadReq oder DataRdy Signals.

Ablauf einer Transaktion wird durch zwei endliche Automaten gesteuert.



Vergleich der maximalen Datenraten für asynchrone und synchrone Kommunikation (1 Wort Lesezugriff, $t_{AC} = 200 \text{ ns}$):

- Synchron

Taktperiode 50ns

1 Takt/Sendeschritt

Schritte:

1. Adresse --> Speicher: 50 ns

2. Speicherzugriff: 200 ns

3. Daten --> I/O-Gerät: 50 ns

$$\text{Datenrate: } d_s = \frac{4B}{300\text{ns}} = 13.3 \text{ MB/s}$$

- Asynchron

Zeit für ein Handshake: 40 ns

1. Schritte "I 40 ns

2. Schritte AID 200 ns

3. Schritte fYy 3 x 40 ns

$$\text{Datenrate: } d_a = \frac{4B}{360\text{ns}} = 11.1 \text{ MB/s}$$

l Erhöhung der Datenraten

- Erhöhung der Busbreite (spez. für kurze Proz-Mem-Busse) $\geq 128 \text{ bit}$
- Erhöhung der Taktfrequenz
- dedizierter Daten- und Adreßbus
- Blocktransfer (vgl. Page Mode-Speicher)
- Speicherzugriff = Auftrag; I/O-Gerät schickt mehrere Aufträge an Speicher, Speicher antwortet darauf. Vorteil: ungenutzte Buszyklen (während t_{AC}) können genutzt werden für weitere Aufträge.

l Arbitrierung

Falls mehrere I/O-Geräte und Prozessor: Wer erhält den Bus?

- Daisy Chain: HW-Lösung, feste Prioritäten, nicht fair
- Zentraler Arbiter, parallele Requestleitungen: Arbiter teilt Bus zu. Arbiter = Bottleneck
- Verteilte Arbitrierung: Anforderung über parallele Request-Leitungen und Anforderungscode auf dem Datenbus. I/O-Geräte werten dezentral die Codes aus und entscheiden lokal.
- Verteilte Arbitrierung mit Collision Detection: wie Ethernet

1 Bus Standards

Die folgende Tabelle zeigt die wichtigsten Eigenschaften von häufig benutzten Standard-Bus-Systemen

Merkmale von Standard-Bus-Systemen:

Characteristic	VME Bus	NuBus	FutureBus	IPI	SCSI
Bus type	Backplane	Backplane	Backplane	I/O	I/O
Bus width (signals)	128	96	96	16	8
Address/data multiplexed?	Not multiplexed	Multiplexed	Multiplexed	N/A	N/A
Data width (primary)	16-32 bits	32 bits	32 bits	16 bits	8 bits
Number of bus masters	Multiple	Multiple	Multiple	Single	Multiple
Arbitration	Multiple daisy chain	Distributed self-selection	Distributed self-selection	NA	Self-selection
Clocking	Asynchronous	Synchronous	Asynchronous	Asynchronous	Either
Bandwidth, 150 ns memory, single word	12,9 MB/sec	13,2 MB/sec	15,5 MB/sec	25,0 MB/sec	5,0 MB/sec or 1,5 MB/sec
Bandwidth, 150 ns memory, multiple words (infinite length)	13,6 MB/sec	26,4 MB/sec	20,8 MB/sec	25,0 MB/sec	5,0 MB/sec or 1,5 MB/sec
Maximum number of devices	21	16	20	8	7
Maximum bus length	0,5 meter	0,5 meter	0,5 meter	50 meters	25 meters
Standard name	IEEE 1014	pending	IEEE 896.1	ANSI X3.129	ANSI X3.131

2.5.4. Kommunikation

Externe Geräte arbeiten asynchron, d.h. ihre Aktivitäten müssen synchronisiert werden mit denen des Prozessors. Dabei ist zu unterscheiden zwischen Polling (Initiative beim Prozessor) und Interrupts (Initiative beim I/O-Gerät). Die Durchführung des eigentlichen Datentransfers kann entweder durch den Prozessor selbst oder durch spezielle HW-Einheiten (z.T. spezielle Prozessoren) durchgeführt werden, um den Hauptprozessor zu entlasten. Man spricht dann von Direct Memory Access (DMA).

1 Polling

Kommunikation über spezielle Register (memory mapped oder per I/O-Befehl). I/O-Gerät schreibt Status-Information (z.B. Maus: x- und y-Zähler) in Register, welche vom Prozessor abgeholt werden müssen.

Problem: Häufiges Pollen ist nötig für hohe Zeitauflösung, Verschwendung von CPU-Zeit.

Beispiel: Polling für ein langsames (Maus), mittleres (FD) und schnelles (HD) Gerät. Es wird jeweils die CPU-Belastung durch Polling berechnet:

1. Maus

Annahme: 30 Polls/sec
100 Instruktionen/Poll
 $f_c = 50 \text{ MHz}$

Polling-Zyklen/s = $30 \cdot 100$

$$\text{Gesamt-Zyklen/s} = 50 \cdot 10^6$$

$$\text{Polling-Belastung} = \frac{3000}{50 \cdot 10^6} = 0.006\%$$

2. FD

Annahme: $d = 50 \text{ KB/s}$

Blockgröße: 16 bit/Poll

$$\text{Pollingrate} = \frac{50 \frac{\text{KB}}{\text{s}}}{2 \frac{\text{B}}{\text{Poll}}} = 25 \cdot 2^{10} \frac{\text{Polls}}{\text{s}}$$

$$\text{Pollingzyklen/s} = 25 \cdot 2^{10} \cdot 100 = 2.56 \cdot 10^6$$

$$\text{Polling-Belastung} = \frac{2.56 \cdot 10^6}{50 \cdot 10^6} = 5 \%$$

3. HD

Annahme: $d = 2 \text{ MB/s}$

Blockgröße: 4 B/Poll

$$\text{Pollingrate} = \frac{2 \frac{\text{MB}}{\text{s}}}{4 \frac{\text{B}}{\text{Poll}}} = 5.2 \cdot 10^5 \frac{\text{Polls}}{\text{s}}$$

$$\text{Pollingzyklen/s} = 5.2 \cdot 10^7$$

$$\text{Polling-Belastung: } \frac{5.2 \cdot 10^7}{50 \cdot 10^6} = 100 \%$$

=> Ergebnis: Polling nicht tragbar für schnelle I/O-Geräte.

1 Interrupts

Reduzierung der CPU-Belastung, da CPU nur mehr im Bedarfsfall eingreifen muß.

Obriges Beispiel für FD:

FD aktiv: CPU-Belastung = 5%

FD nicht aktiv: CPU-Belastung = 0%.

Die einzelnen Methoden zur Meldung und Erkennung von Interrupts werden im Skript Betriebssysteme ausführlich behandelt.

1 DMA

Prozessor-Belastung entsteht bei Ereignissen. Im obigen Beispiel FD: 1 Ereignis pro 2 B-Transfer. Sind größere Datenmengen zu übertragen, so genügt es, wenn der Prozessor zu Beginn und am Ende aktiv wird. Die Steuerung der 2 B-Transfers übernimmt dann eine Spezial-HW (DMA-Controller).

Achtung: DMA-Controller wirkt wie ein zusätzlicher Bus-Master! Konflikte mit CPU sind möglich.

Beispiel HD:

Annahme: DMA-Initialisierung: 1000 CPU-Zyklen
 DMA-Beendung: 500 CPU-Zyklen
 Datenmenge/Transfer: 4 KB
 f_c : 50 MHz
 d : 2 MB/s

$$\text{Zeit/Transfer} = \frac{4\text{KB}}{2 \frac{\text{MB}}{\text{s}}} = 2 \cdot 10^{-3} \text{ s/Transfer}$$

$$\text{CPU-Belastung} = \frac{(1000 + 500) \frac{\text{Zyklen}}{\text{Transfer}}}{2 \cdot 10^{-3} \frac{\text{s}}{\text{Transfer}}} = 750 \cdot 10^3 \frac{\text{Zyklen}}{\text{s}}$$

$$\text{rel. CPU-Belastung} = \frac{750 \cdot 10^3}{50 \cdot 10^6} = 1.5 \%$$

Wird die "Intelligenz" des DMA-Controller weiter gesteigert, dann kann die CPU ihm ganze I/O-Programme zur selbstständigen Abarbeitung übergeben. DMA-Controller wird so zum I/O-Prozessor, CPU noch weiter entlastet.

2.6. Fallstudie: Mikroprozessor Alpha 21064

1 Überblick

Hersteller: Digital Equipment Corp.
 Erste Ankündigung: 1992
 Befehlslänge: 32 bit
 Architektur: Load/Store (RISC)
 Virtuelle Adressen: 43-64 bit unsegmentiert
 Byte/Halbwort load/store: nein
 Condition Code: nein
 Conditional Moves: ja
 Integer Register: 32
 Integer Register Breite: 64 bit
 FP Register: 32
 FP Register Breite: 64 bit
 FP Format: IEEE, (VAX 32 bit, 64 bit)
 Cache: split data/instr.

1 Entwurfsziel: höchste Taktfrequenz, einfache Architektur

1 Befehlsformate

Immediate-Befehle:

0	6	11	19	20	27	31
opcode	src A	immed	1	ext. op.	dst	

immed = pos. Integer

Reg-Reg-Befehle (Integer):

0	6	11	16	19	20	27	31
opcode	src A	src B	-	0	ext. op.	dst	

Reg-Reg-Befehl (FP):

0	6	11	16	27	31
opcode	src A	src B	ext. op.	dst	

Load/Store-Befehle:

0	6	11	16	31
opcode	dest	src A	displacement	

1 Adressierungsmodus: (nur 1 Modus)

Adresse = Register + Displacement

kein Reg + Reg - Modus!

Folge: Registerfile einfach, aber Array-Adressierung erfordert explizite Inkrementierung.

1 Alignment:

(Adresse für b Byte-Datum = $n \cdot b$, $n \in \mathbb{Z}$, $b = 2, 4, 8$)

2 Versionen:

a) aligned: Trap, falls Adresse nicht aligned, d.h. Befehl nimmt an, daß die Daten im Normalfall ausgerichtet sind.

b) häufig nicht aligned: Zugriff unaligned, Korrektur mittels Befehlssequenzen (insert, mask, extract)

1 Byte-Operationen:

Speicherzugriff nur in 32 bit- und 64 bit-Einheiten. Dann Byte-Manipulation (extract).

Folge: einfachere Speicherarchitektur.

1 Sprünge

Format (bedingt und unbedingt):

0	6	11	31
opcode	src A	displacement	

1 Festpunkt- und Load/Store-PL

IF (I-Fetch): Zugriff auf I-Cache, einlesen von 2 Befehlen.

S (Swap): Zuordnung der Instruktion zur FP-PL oder zur Integer- bzw. Load/Store-PL.

Vertauschen, falls nötig.

Branch Prediction, falls nötig.

D (Decode): Decodierung, Feststellung der nötigen Register und Ressourcen. Noch kein Auslesen der Register.

I (Issue): Lesen der Quellregister. Überprüfung auf Abhängigkeiten (Daten und Ressourcen), evtl. wird der Befehl zurückgehalten. Nach Issue: keine weiteren Wartezyklen.

A (ALU Stufe 1): Ausführung von Integer-Additionen, logischen Operationen, kurzen Shifts. Evtl. Bypass zurück nach I.

Load/Store: Berechnung der eff. Adresse.

B (ALU Stufe 2): Beendigung langer Shifts, evtl. Bypass zurück nach I.

Load/Store: Lesen der Datencache-Tags, Lesen der Cache-Daten (load).

W (Write Register): Schreiben der Ergebnisse (von ALU oder load) in das Zielregister. Feststellung von Cache Hit/Miss. Cache Hits zum Schreiben werden gepuffert, Übertragung zum Cache im nächsten Takt.

1 FP-PL

IF, S, D, I - Stufen wie bei Integer-PL.

F, G, H, J, K -Stufen für FP-Addition und FP-Multiplikation. Für Division mehrere Durchläufe (31 Takte SP, 61 Takte DP).

1 Assemblerprogramm-Beispiel und Ausführung auf der Alpha-PL

Der zeitliche Ablauf wird bestimmt durch

- Ressourcenkonflikte (s. Issue-Regeln)
- Bypasses
- Datenabhängigkeiten

Issue-Regeln für Alpha 21064. Ein x bedeutet, daß die gleichzeitige Herausgabe zweier Befehle (d.h. Auslesen der Quellregister und Einspeisung in die Ausführungsstufen) erlaubt ist.

	Integer:			Floating:			Branch	
	Load	Store	Operate	Load	Store	Operate	Integer	Floating
Integer load			x			x		
Integer store			x					
Integer operate	x	x		x		x	x	
Floating load			x			x		
Floating store						x		
Floating operate	x		x	x	x			x
Integer branch			x					
Floating branch						x		

(a) C-Code:

```
for (k = 0; k < 512; k++)
    x[k] = r * x[k] + t * y[k];
```

(b) Assembler-Code:

LOOP: ldt	fp3 = y(r2,0)	#r1 points to x.
ldt	fp1 = x(r1,0)	#r2 points to y.
mult	fp3 = fp3,fp2	#r6 points to the end of y.
addq	r2 = r2,8	#fp2 contains t.
mult	fp1 = fp1,fp4	#fp4 contains r.
subq	r4 = r2,r6	#r5 contains the constant 1.
addt	fp1 = fp3,fp1	#load floating double.
stt	x(r1,0) = fp1	#load floating double.
addq	r1 = r1,8	#floating multiply double t * y.
bne	r4,Loop	#bump y pointer.
		#floating multiply double r * x.
		#subtract y end from current pointer.
		#floating add double, r * x + t * y.
		#store floating double to x[k].
		#bump x pointer.
		#branch on r4 ≠ 0

Bild 2.71. (a) C und (b) Assembler-Code Beispiel

Takte:		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	ldt	fp3 = y(r2,0)	IF	S	D	I	A	B	W													
2	ldt	fp1 = x(r1,0)	IF	.	S	D	I	A	B	W												
3	mult	fp3 = fp3,fp2		IF	S	D	.	I	F	G	H	J	K	W								
4	addq	r2 = r2,8		IF	S	D	.	I	A	B	W											
5	mult	fp1 = fp1,fp4			IF	S	.	D	I	F	G	H	J	K	W							
6	subq	r4 = r2,r6			IF	S	.	D	I	A	B	W										
7	addt	fp1 = fp3,fp1				IF	.	S	D	.	.	.	.	I	F	G	H	J	K	W		
8	stt	x(r1,0) = fp1				IF	.	S	D	.	.	.	.	.	.	.	.	I	A	B	W	
9	addq	r1 = r1,8						IF	S	.	.	.	.	.	.	.	.	D	I	A	B	
10	bne	r4,LOOP						IF	S	.	.	.	.	.	.	.	.	D	I	A	.	
11	ldt	fp3 = y(r2,0)								IF	.	.	.	.	.	.	.	S	D	I	A	
12	ldt	fp1 = x(r1,0)								IF	.	.	.	.	.	.	.	.	S	D	I	

Bild 2.72. Ablauf des Beispielprogramms auf der Alpha-PL

Anmerkungen:

- 1.) Befehl 2, Takt2: S verzögert um 1 Takt, da 2 FP-Loads nicht gleichzeitig herausgegeben werden dürfen.
- 2.) Befehle 3 und 4: Double Issue (FP operate und Integer operate).
Takt 6: Datenabhängigkeit fp3
- 3.) Befehle 5 und 6, Takt 6: Datenabhängigkeit r2
- 4.) Befehl 7, Takte 9-13: Datenabh. fp1
- 5.) Befehl 8 Takte 9-17: Datenabh. fp1, stt wartet 4 Takte, I in 18 so, daß das FP-Add-Ergebnis gerade über Bypass übernommen werden kann.
- 6.) Befehle 11 und 12, Takt 8: Branch Delay.
- 7.) Takte 10-17 bzw. 10-18: Datenabh. r1 und Issue-Verbot FP-Load/FP-Load.

l Implementierungsdaten 21164 (Stand Okt. 96)

fc	500 MHz
Cache primär	8 KB Daten, 8KB Befehle
Cache sekundär	96 KB
Issue	4
PL Stufen	7
Out of Order	6 Loads
TLB-Einträge	48 I, 64 D
Speicherbandbreite	400 MB/s
Gehäuse	CPGA 499
Prozeß	0.35 μ 4 Metalllagen
Die-Größe	209 mm ²
Transistoren	9.3 Mio
Herst. Kosten (gesch.)	\$ 150
Verlust-Leistung (max)	25 W
SPEC 95 (int/fp)	12.6/18.3
Listenpreis (1000)	\$ 3000

l Performance-Vergleich (Stand Okt. 96)

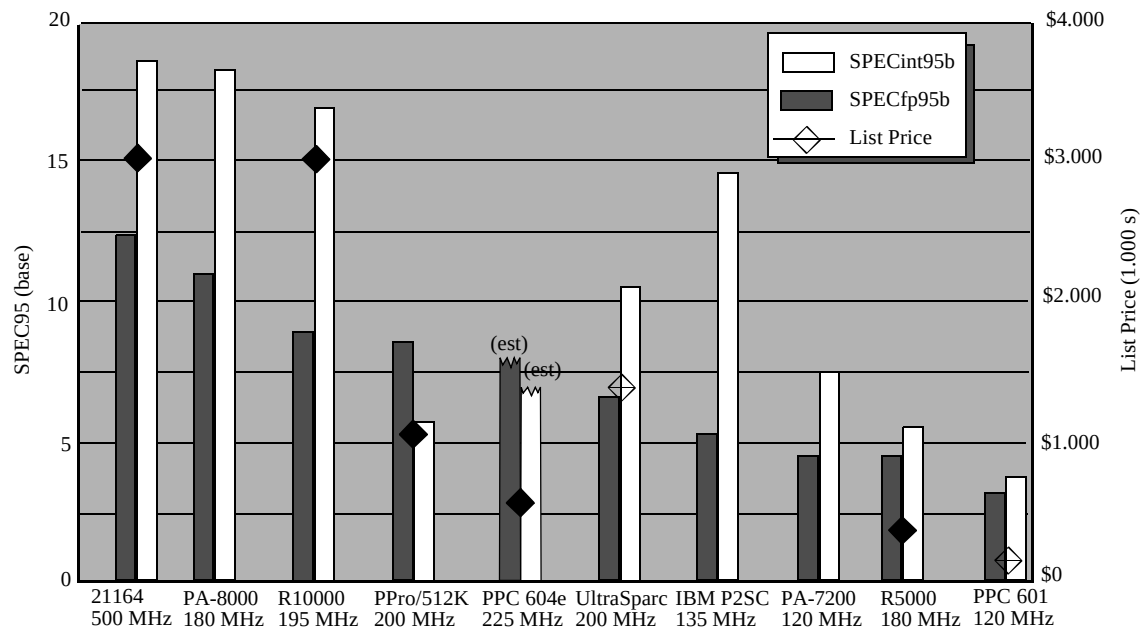


Bild 2.73. Performance-Vergleich (Stand Okt. 96)

3 Parallelrechner

3.1. Anwendungen

Wo werden höchste Rechenleistungen gebraucht?

Welche Anwendungsklassen gibt es?

Welche Leistungen und Speichergrößen werden benötigt?

In welchen Anwendungen steckt genügend "natürliche" Parallelität?

Wie läßt sich Parallelarbeit organisieren?

3.1.1. Überblick

Häufige Aufgabenstellung: Lösung drei-dimensionaler PDGL-Systeme,

10^9 Gitterpunkte : ~ GB Speicher

~ GFLOPS Leistung

Zukünftiger Bedarf:

~ 10 GWorte Speicher

~ 1 TFLOPS (10^{12} FLOPS)

Sequentieller Rechner: 1ps Zykluszeit (?)

4 Anwendungsgebiete:

- Naturwissenschaften (Simulation natürlicher Systeme)
- Technik (Simulation künstlicher Systeme)
- Logik (Simulation des Gehirns)
- Kommerzielle Anwendungen

Naturwissenschaften

Astrophysik	Entstehung von Galaxien, schwarze Löcher
Biologie	Gentechnik
Chemie	Molekülsynthese
Geophysik	Exploration
Hochenergiephysik	Stoßprozesse
Meteorologie	Vorhersage
Kernphysik	Quantenchromodynamik
Plasmaphysik	Stoßprozesse

Technik

Aerodynamik	Tragflächenentwicklung
Mechanik	Festigkeitsberechnung
Motortechnik	Verbrennungsmodellierung
Werkstoffkunde	Materialzusammensetzung
VLSI-Entwurf	Logiksimulation, Circuitsimulation, Placement und Routing, Fehlersimulation, ATPG

Logik (Computertechnik, künstliche Intelligenz)

Gehirnforschung	Neuronale Netze, semantische Netzwerke, Lernen und Adaptivität
Mustererkennung	Bild, Sprache
Graphik	Bildverarbeitung, Ray Tracing, Fraktale, virtuelle Realität
Suche	lineare Optimierung, genetische Algorithmen, Datenbanken, Hypermedia

Grand Challenges

Parallelrechner können Aufgaben, an denen heutige Computer viele Stunden oder Tage rechnen, in wenigen Minuten bearbeiten (gleiche Arbeit). Oder: Parallelrechner können Aufgaben, die heutige Computer überfordern, in akzeptabler Zeit lösen: Grand Challenges (Justin Rattner, Intel)

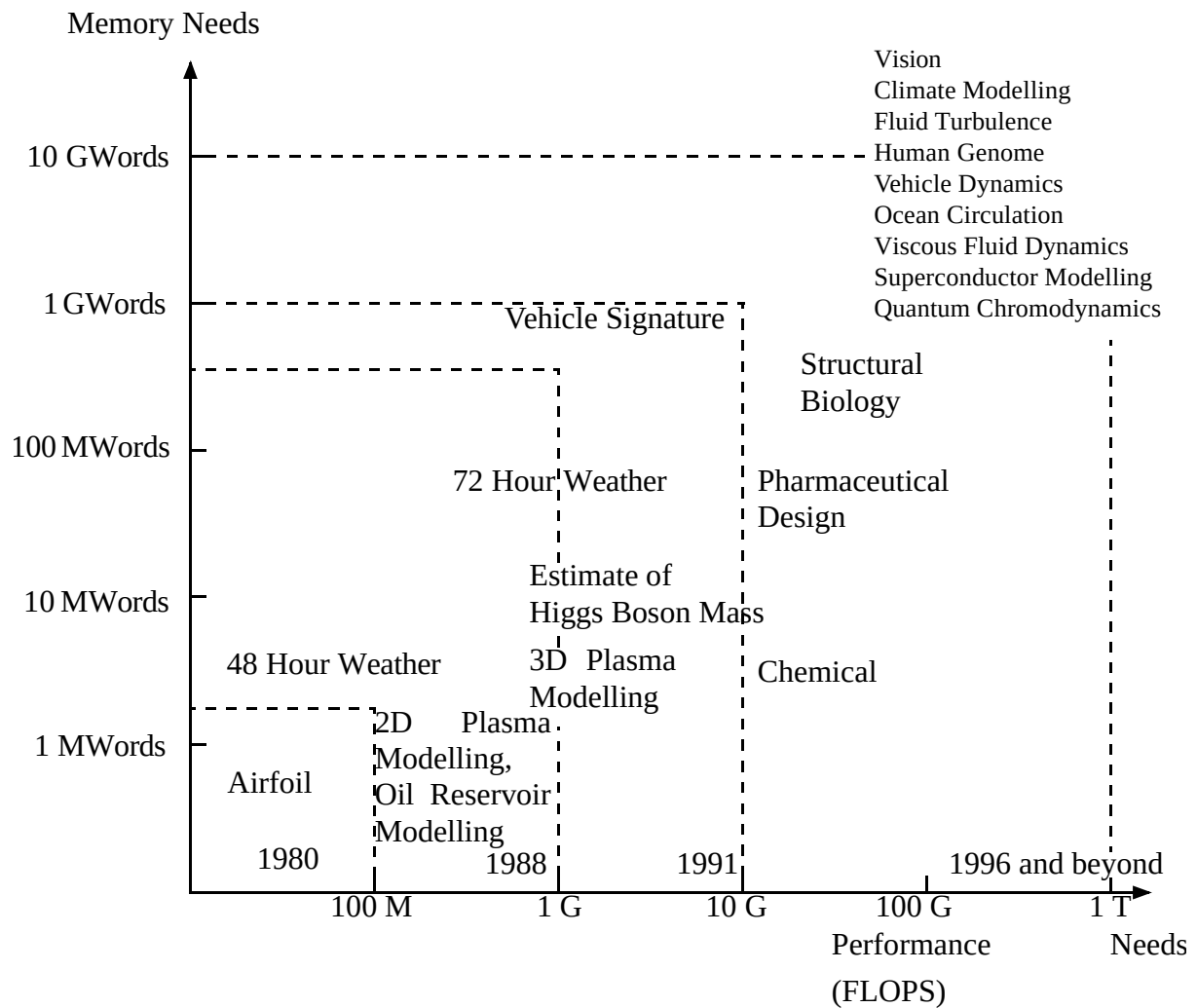


Bild 3.1. Speicherplatzbedarf und Performance-Anforderungen großer Parallelrechner-Anwendungen (Grand Challenges)

Maschinen	Moderate Probleme		Grand Challenge Probleme	
TFlop Machine	2	seconds	10	hours
CM-2 64K	30	minutes	1	year
CRAY Y-MP/8	4	hours	10	years
Alliant FX/80	5	days	250	years
SUN 4/60	4	month	1,5K	years
VAX 11/780	9	month	14K	years
IBM PC/8087	9	years	170K	years
Apple Mac	23	years	450K	years

Bild 3.2. Problemgröße und Bearbeitungszeiten auf unterschiedlichen Rechnern

3.1.2. Anwendung: Transaktionssysteme

- Definition: Transaktion =

Anfrage an Datenbank	(<i>query</i>)
Modifikation	(I/O)
Auslesen	(I/O)

Beispiele: Buchung (Bank)
 Reservierung (Reiseveranstalter)
 Kalkulation (Vertrieb, Außendienst)

- Quellen für Parallelität:
 - a) viele Transaktionen gleichzeitig
 - b) parallele Abarbeitung innerhalb einer Transaktion
 - zu a: Benutzer weitgehend unabhängig voneinander.
 Parallelitätsgrad: 100 ... 1000
 Voraussetzung: Plattenzugriff konfliktfrei
 - zu b: Parallelitätsgrade innerhalb einer Anfragebearbeitung: 10 ... 100

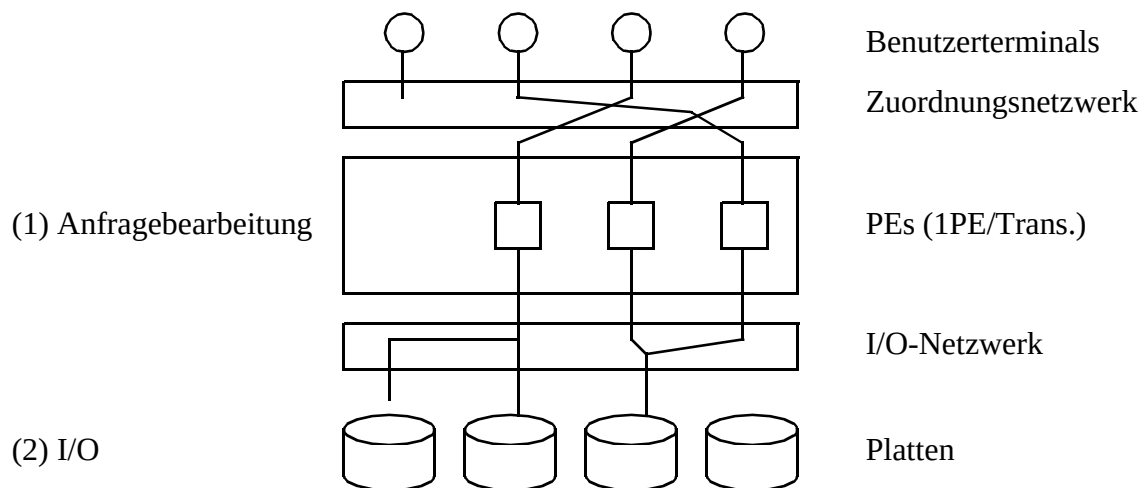


Bild 3.3. Prinzip einer parallelen Datenbankmaschine. Beide Bearbeitungszeiten (Anfrage und I/O) müssen reduziert werden, um eine Geschwindigkeitserhöhung zu erreichen.

- I/O-Situation

$t(\text{Anfragebearbeitung}) = 10 \text{ ms}$

$t(\text{I/O}) = 20 \text{ ms} + \text{Transfer} + \text{Konflikte}$

• I/O-Zeiten sind zu reduzieren.

Maßnahmen: Disk-Striping, Platten-Caches, große Speicher in den PEs, technologische Verbesserungen (Zugriffszeiten)

- Parallelität innerhalb einer Anfragebearbeitung am Beispiel einer Relationalen Datenbank

Relation = Tabelle

Reihe = Tupel

Spalte = Attribut

Anfrage = query

3 Basisoperationen

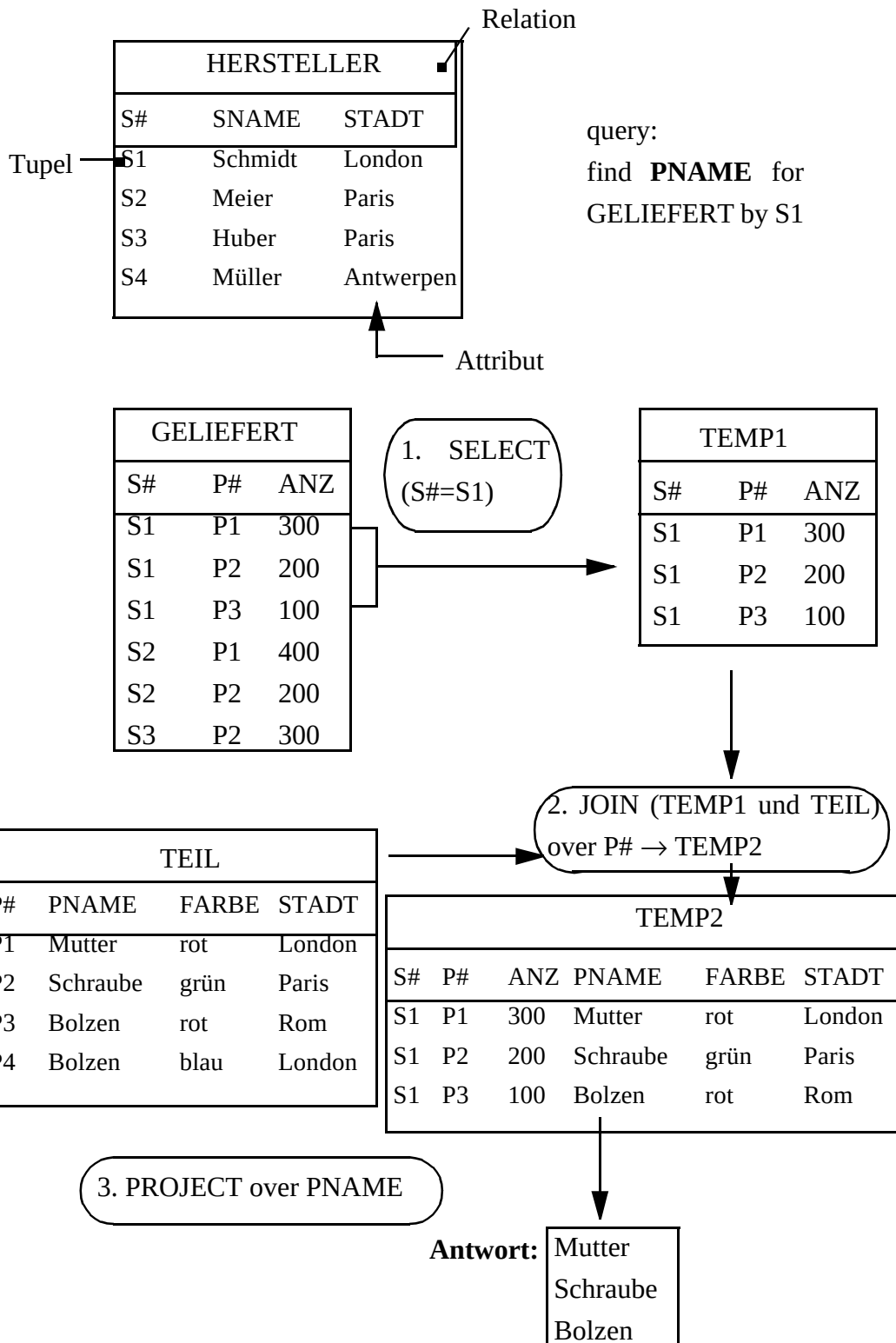
-SELECT: bilde neue Relation durch Auswahl aller Tupels, die eine bestimmte Bedingung erfüllen (z.B. $S\# = S1$).

-JOIN: bilde neue Relation mit allen Attributen der beiden Ausgangsrelationen, für alle Tupel, die eine bestimmte Bedingung erfüllen.

-PROJECT: bilde vertikale Untermengen durch Auswahl einzelner Attribute (Spalten), eliminiere Duplikate.

Beispiel 3-1:

Anfrage an Relationale Datenbank



Parallelisierungspotentiale:

1. SELECT: Aufteilung der Relation in N Teiltabellen (N = Anzahl der PEs), unabhängige Überprüfung der Bedingung (S# = S1?)
2. PROJECT: Aktion A: Spaltenauswahl (Entfernen der nicht benötigten Spalten)
 Aktion B: Duplikatelimination
 Schritt 1: Aktionen A und B für Teilrelationen (parallel, t_{par})
 Schritt 2: globale Duplikatelimination erfordert Kooperation zwischen allen PEs (t_{ser}).
3. JOIN: 2 Relationen als Quellen
 1 Relation als Ziel

Am Beispiel:

```

Für alle Tupel aus TEMP1
  inspiziere alle Tupel aus TEIL
    überprüfe Bedingung (P# gleich)
    schreibe nach TEMP2
  END
END

```

$t_{\text{exec}} \sim \text{Tupelanzahl (TEMP1)} \times \text{Tupelanzahl (TEIL)}$

Parallelisierung 1: teile Quellrelation 1 in N Teile mit je $\frac{1}{N}$ Tupeln

$$\text{Beschleunigung} \approx \frac{1}{N}$$

Parallelisierung 2: (Annahme: Relationen geordnet, z.B. alphabetisch)
 bilde disjunkte Subsets in beiden Quellrelationen, teile jeweils 2 gleiche Subsets jeweils einem PE zu.

$$\text{Beschleunigung} \approx \frac{1}{N^2}$$

Beispiel 3-2:

JOIN (HERSTELLER, TEIL) over STADT → TEMP3

TEMP3					
S#	SNAME	STADT	P#	PNAME	FARBE
S1	Schmidt	London	P1	Mutter	rot
S1	Schmidt	London	P4	Bolzen	blau
S2	Meier	Paris	P2	Schraube	grün
S3	Huber	Paris	P2	Schraube	grün

Sequentiell: 16 Zeiteinheiten

Parallelisierung 1: (N=2) 8

Parallelisierung 2: (N=2) 4

PE.1: Antwerpen, London

PE.2: Paris, Rom

(Annahme: gleich große Subsets)

• **Definitionen:**

$$allg.: t_{exec} = t_{par} + t_{comm} + t_{seriell}$$

Aufteilung in diese 3 Zeiten abhängig von der Granularität des Problems:

hohe Granularität (feingranular):

viele PEs - kleine Teilaufgaben - kleine Korngröße - viel Kommunikation

kleine Granularität (grobgranular):

wenige PEs- große Teilaufgaben - große Korngröße - wenig Kommunikation

Dekomposition (Zerlegung) heißt die Verteilung einer großen Aufgabe in parallel abzuarbeitende Teilaufgaben.

Bei der Daten-Dekomposition geht man von der Annahme aus, daß eine Gleichverteilung der Daten über die PEs auch gleiche Auslastung zur Folge hat.

3.1.3. Anwendung AI**3.1.3.1 Regelbasierte Systeme**

- Prinzip: Fakten, Regeln, Steuerung

Steuerung führt Fakten mithilfe von Regeln in andere Fakten über; entspricht Durchsuchen des Zustandsraums, Aufbau von AND/OR-Bäumen.

Beispiel 3-3:

Buchstaben-Substitution

Fakten: C, B, Z (Anfangsbelegung der Datenbank)

Regeln: R1: C $\rightarrow$ D,LR2: C $\rightarrow$ B,MR3: B $\rightarrow$ M,MR4: Z $\rightarrow$ B,B,M

Ziel: Datenbank enthält nur Ms

AND/OR-Graph enthält alle Möglichkeiten der Regelanwendung:

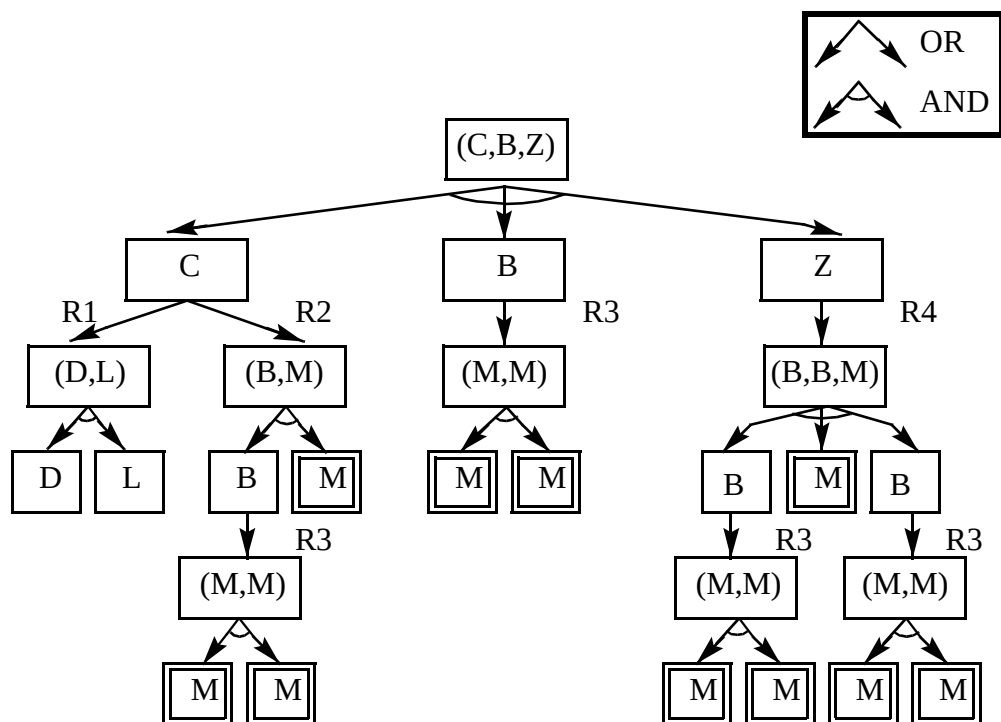


Bild 3.4. AND/OR-Graph für die Buchstaben-Substitution

- Mögliche Parallelisierung: 1 PE pro Regel, PEs versuchen permanent, ihre Regel auf den Inhalt der DB anzuwenden

Nachteil: PE mit nicht-benutzbarer Regel leistet keinen Beitrag.

U. U. existieren sehr viele (X1000) Regeln, und sehr wenige werden benutzt.

Realisierung: DADO-Projekt, Columbia University (1K PEs).

DADO-Topologie ist der Problemstruktur angepaßt.

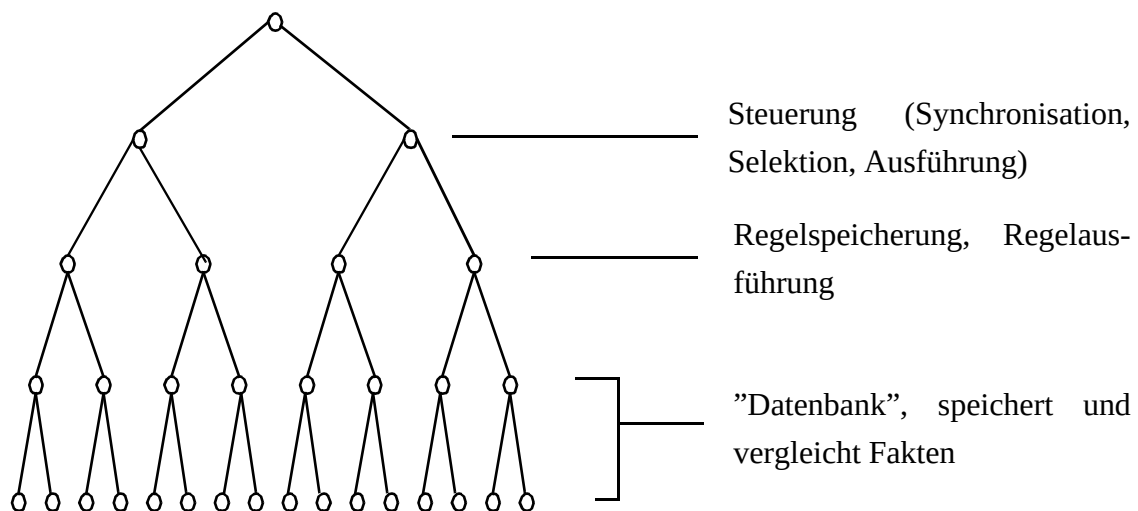


Bild 3.5. Aufbau des DADO-Baums (1K PEs, Intel 8751)

- AND-Parallelismus:
verfolge alle (möglichst viele) Teilziele parallel
- OR-Parallelismus:
versuche, zunächst ein Teilziel möglichst schnell zu erreichen durch gleichzeitige Verfolgung möglichst vieler OR-Äste.
- superlinearer Speed-up möglich vor allem bei OR-Parallelismus.

3.1.3.2 Semantische Netzwerke

Grundüberlegung: wie arbeitet das Gehirn?

Modellierung durch Graph, Knoten sind Konzepte, Kanten sind Relationen.

Fahlman: (1) Clyde ist ein Tier.
(2) Clyde ist ein Elefant

Satz 2 baut "Clyde" in ein Netzwerk ein, welches das Wissen über Elefanten repräsentiert:

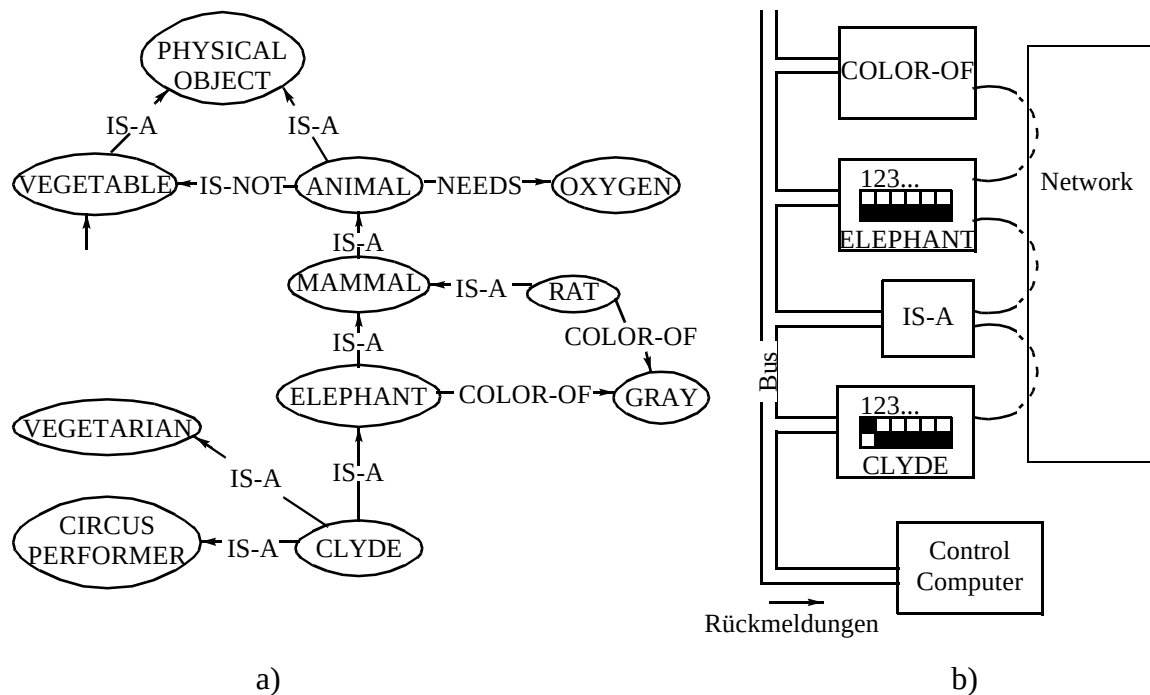


Bild 3.6. a) Kleines Semantisches Netzwerk b) HW-Realisierung (Zustand: M1 in Knoten CLYDE gesetzt, noch nicht an ELEPHANT weitergegeben)

Bearbeitung der Frage: "Welche Farbe hat Clyde?"

1. Durchgang:

1. markiere "Clyde" mit einem Marker M1
2. alle IS-A-Links, deren Startpunkt mit M1 markiert ist, geben die Markierung an ihren Zielpunkt weiter (an ihre Klasse)
3. wiederhole Schritt 2, solange nötig

2. Durchgang:

1. alle COLOR-Of-Links mit M1-Markierung am Eingang markieren ihren Zielpunkt mit M2
2. alle Knoten (hier: genau 1) mit M2 meldet sich bei der globalen Steuerung

Semantische Netzwerke sind ein extremes Beispiel für anwendungsspezifische Architekturen.

- **Parallelisierungsstrategie ist auch abhängig vom Ziel (1 Lösung, alle Lösungen?)**
- **Architektur- und Problem-Topologie sollten übereinstimmen**

3.1.4. Grundbegriffe der Parallelarbeit

Parallelarbeit ist eine übliche Arbeitstechnik, die Problematik ist unabhängig vom Spezialfall "Parallelarbeit in Computern".

Einführendes Beispiel:

Hadrianswall, $L = 120 \text{ km}$, $H = 2 \text{ m}$, gebaut zwischen England und Schottland

Aufgabe:

Bau des Hadrianswalls durch N Maurer in kürzestmöglicher Zeit. Maurer = PE

Problemdomäne:

Mauer der Länge L und Höhe H

Problemaufteilung (*domain decomposition*):

N gleichlange vertikale Segmente der Länge l , 1 Maurer pro Segment

Beschleunigung (*speed-up*):

$S=N$ mit der Annahme, daß alle Maurer gleich schnell (langsam!) arbeiten und kein Kommunikationsoverhead auftritt.

$$\text{Speed-up} = \frac{\text{Zeit}(1 \text{ Maurer})}{\text{Zeit}(N \text{ Maurer})}$$

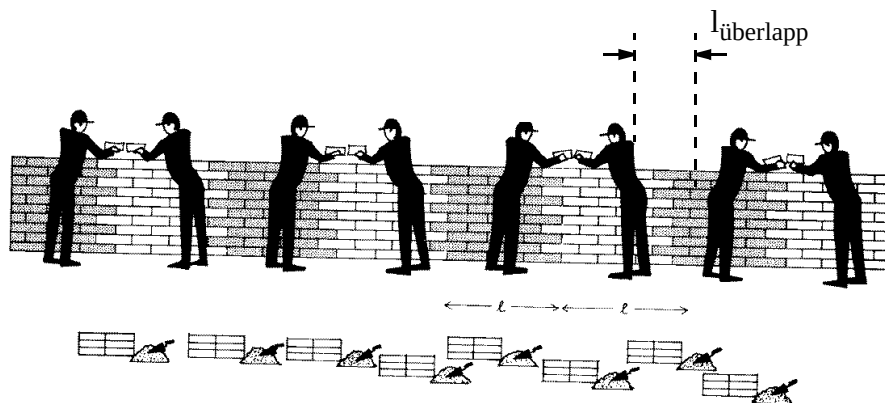


Bild 3.7. Problemaufteilung in $N=8$ vertikale Segmente

Die vertikale Aufteilung ermöglicht Nebenläufigkeit (*concurrency*), da die Teilaufgaben unabhängig voneinander sind. Ausnahme: Überlappbereich zwischen Maurer i und $i+1$, (Länge: $l_{\text{überlapp}}$) Zusammenarbeit erforderlich, d.h. Serialisierung, Verluste

Andere Problemaufteilung: horizontale Schichten, führt zu einer Pipelineorganisation.

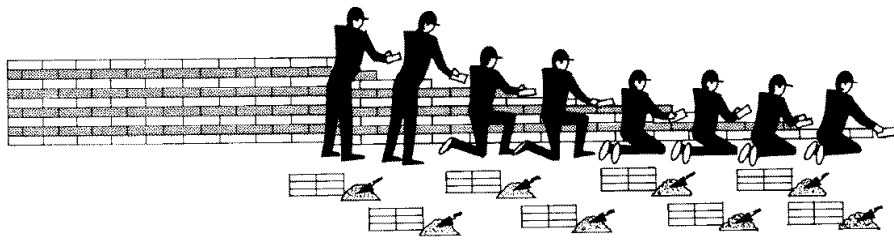


Bild 3.8. Horizontale Aufteilung. Parallelarbeit nur im Pipelining

Effizienz (tatsächlicher Speed-up S bezogen auf Ideal-Speed-up N) $\epsilon = \frac{S}{N}$; ($\epsilon \leq 1$; $S \leq N$)

Korngröße (*grain size*): Umfang der einem Maurer zugeordneten Arbeit.

Korngröße darf nicht zu klein werden im Vergleich zum Überlapp:

$$S = \frac{N \cdot l}{l + l_{\text{ueberlapp}}}; \quad \epsilon = \frac{l}{l + l_{\text{ueberlapp}}}; \quad \epsilon = \frac{1}{1 + \frac{l_{\text{ueberlapp}}}{l}}$$

- relative Verlustarbeit oder Overhead μ :

$$\mu = \frac{l_{\text{ueberlapp}}}{l}; \quad \epsilon = \frac{1}{1 + \mu}$$

Für $\epsilon > 90\%$: $\mu < 10\%$ $l_{\text{überlapp}} < 0,1 \cdot l$

- Problemgröße und Überlapp bestimmen den maximalen Parallelitätsgrad. Parallelitätsgrad ist die Anzahl von in einem Zeitpunkt produktiv arbeitenden Maurern (PEs).
- Teilaufgabe (pro Maurer) ist ähnlich wie im sequentiellen Fall ($N=1$), aber: Randbereiche anders (hier Verzahnung).

Die Arbeitsorganisation (Architektur) bestimmt das Zusammenwirken der Maurer. Z.B. 1 Vorarbeiter, Kommandos gleichzeitig an alle Maurer

- Die Arbeitsorganisation ist der Problemstruktur anzupassen. Hier eindimensionales Feld von Arbeitern. Andere Organisationen: 2D-Feld, n-dimensionaler Würfel, Ring.

Irreguläre Probleme (hier: Mauersegmente unterschiedlicher Höhe) erfordern angepasste Arbeitsorganisation (unterschiedliche Segmentlängen).

- Ziel ist die gleichmäßige Auslastung aller Arbeiter: (Lastausgleich, *load balancing*) oder u. U. die Beendigung der Arbeit in der kürzestmöglichen Zeit.



Bild 3.9. Irreguläres Problem. Gleichmäßige Auslastung erfordert unterschiedliche Segmentlängen.

Bei statisch irregulären Problemen ist die Arbeitsorganisation komplizierter, aber voraussagbar.

Bei dynamisch irregulären Problemen kann sich die Arbeitsverteilung dynamisch ändern: dynamischer Lastausgleich wird nötig.

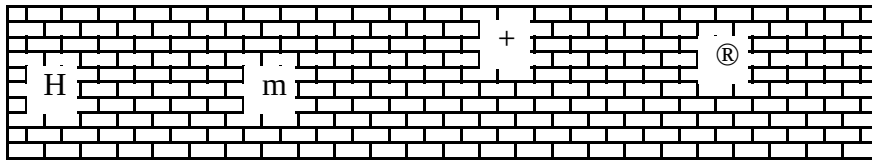


Bild 3.10. Irreguläres Problem

Ebenen der Parallelarbeit:

Globale Parallelarbeit: N Gruppen

Lokale Parallelarbeit: n Arbeiter pro Gruppe (Steine holen, Mörtel rühren, Ziegel legen, verfugen)

Entsprechung: lokal: Pipelining im Prozessor
global: Zusammenarbeit vieler Prozessoren

- Globale und lokale Parallelarbeit sind weitgehend unabhängig voneinander, werden somit auch zusammen eingesetzt. (Parallelarbeit kann auf mehr als zwei Ebenen erfolgen.)

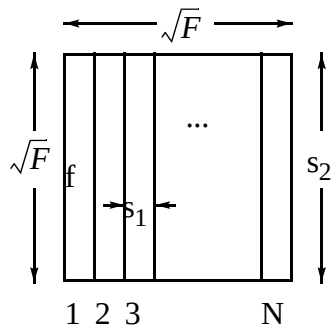
Dimensionalität

Hadrianswall: eindimensionales Problem

Feldbearbeitung: zweidimensionales Problem

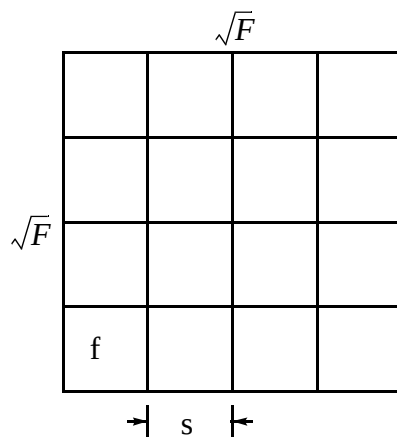
Frage: günstigste Art der Zerlegung?

Günstige Zerlegung, wenn geringer Überlapp, kurze Grenzen.



$$\sum \text{Grenzen} \approx N \cdot \sqrt{F} = N \cdot \sqrt{N \cdot f}$$

a) eindimensionaler Fall



$$F = N \cdot f$$

$$s = \sqrt{f}$$

$$\sum \text{Grenzen} = N \cdot 4 \cdot \sqrt{f} \cdot \frac{1}{2} = 2 \cdot N \cdot \sqrt{f}$$

b) zweidimensionaler Fall

Bild 3.11. Ein- und zweidimensionale Zerlegung

$$q = \frac{\text{Grenzlaenge}(1 - \text{dim})}{\text{Grenzlaenge}(2 - \text{dim})} = \frac{N \cdot \sqrt{N \cdot f}}{2 \cdot N \cdot \sqrt{f}} = \frac{\sqrt{N}}{2} \quad \text{für } N \geq 4 ; q \geq 1$$

- Grenzlängen sind zu minimieren, aber: Die Dimensionalität der Bearbeitung soll der der Problemstellung angepaßt sein.

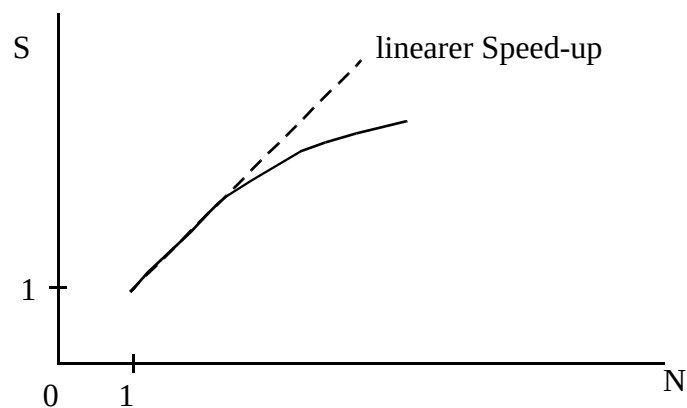
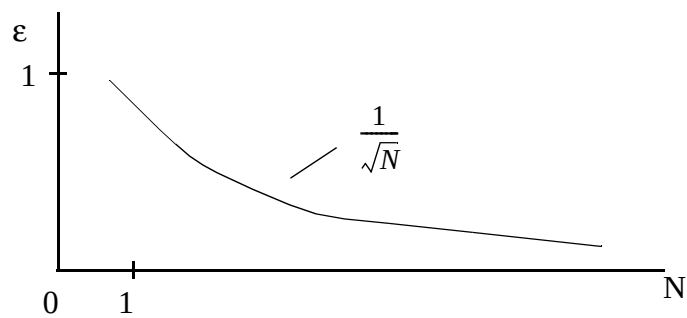
Hohe Effizienz und hoher Speed-up?

$$t_{comm} \sim 4 \cdot \frac{1}{2} \cdot \sqrt{\frac{F}{N}} ; \quad (N \gg 1)$$

$$t_{calc} \sim \frac{F}{N} ; \quad t_{seriell} \sim F$$

$$S = \frac{F}{2 \cdot \sqrt{\frac{F}{N}} + \frac{F}{N}}$$

$$\varepsilon = \frac{1}{1 + 2 \cdot \sqrt{\frac{N}{F}}}$$



- Je kleiner die Prozessorzahl N , desto größer ist die Effizienz: Monoprozessor hat $\varepsilon = 1$
- Probleme mit geringem Kommunikationsanteil eignen sich besser für hohe Parallelisierung (N groß, Korngröße gering).

Ein/Ausgabe (I/O)

I/O muß der Arbeitsorganisation angepaßt sein.

N Maurer: N Versorgungswege für Ziegel, jeweilige Bandbreite ausreichend für 1 Maurer
 oder: 1 Versorgungsweg mit N -facher Bandbreite

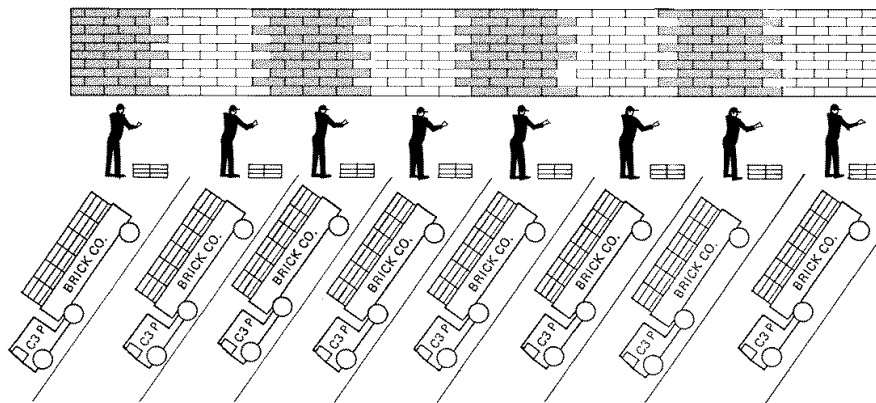


Bild 3.12. Ein/Ausgabe: N Versorgungswege (Eingabe)

Entsprechung: Eingabedaten, Ergebnisdaten (z.B. Bildschirmausgabe zur Visualisierung)

- Ein/Ausgabebandbreite muß der Verarbeitungsleistung angepaßt sein.

3.1.5. Anwendung Numerik: Wellengleichung

$\psi(x,t)$ beschreibt die Auslenkung einer schwingenden Saite, eindimensional, als Funktion von Ort x und Zeit t .

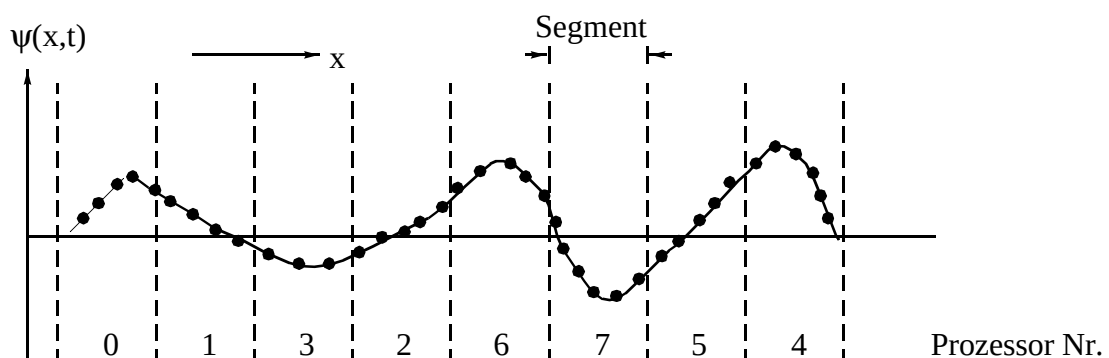
Hyperbolische PDGL:

$$\frac{1}{c^2} \frac{\partial^2 \Psi}{\partial t^2} - \frac{\partial^2 \Psi}{\partial x^2} = 0$$

Anfangsbedingung

$\psi(0,t) = \psi(L,t) = 0$ (Saite fest eingespannt)

$\psi(x,t=0) = f(x)$

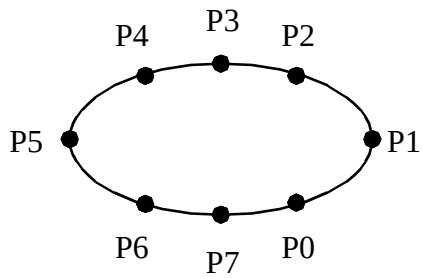
Bild 3.13. Auslenkung $\psi(x,t)$, Verteilung auf die Prozessoren Nr. 0 bis 7

1. Schritt: Festlegung der Gitterpunkte. Die gesuchte Größe ψ ist auf genau 48 Gitterpunkten (hier: x-Positionen) definiert.
2. Schritt: Diskretisierung der PDGL
 Gleichförmige Diskretisierung:
 $\partial x \rightarrow \Delta x$
 $\partial t \rightarrow \Delta t$
 $\psi_i(t)$ = Näherungswert für $x = i \cdot \Delta x$ zur Zeit t
 Finite Differenzengleichung zweiter Ordnung:

$$\frac{\Psi_i(t - \Delta t) - 2\Psi_i(t) + \Psi_i(t + \Delta t)}{c^2 \Delta t^2} - \frac{\Psi_{i-1}(t) - 2\Psi_i(t) + \Psi_{i+1}(t)}{\Delta x^2} = 0$$

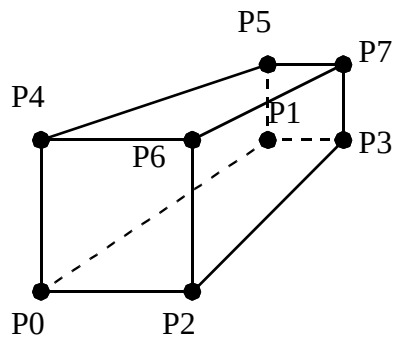
$$\Psi_i(t + \Delta t) = 2\Psi_i(t) - \Psi_i(t - \Delta t) + \tau^2 [\Psi_{i-1}(t) - 2\Psi_i(t) + \Psi_{i+1}(t)]$$
 mit $\tau = c \frac{\Delta t}{\Delta x}$
3. Schritt: Betrachtung des Kommunikations-Overheads
 2 Randpunkte pro Segment, jeweils Austausch eines Nachbarwertes nötig:
 $\Psi_{i-1}(t)$ und $\Psi_{i+1}(t)$
 Hohe Effizienz, wenn Anzahl lokaler Gitterpunkte $\gg$ Anzahl der Randpunkte.
 Folge: möglichst große Korngröße
 Aber: Problemgröße begrenzt
4. Schritt: Zerlegung entlang der Raumkoordinate (entspricht Hadrianswall-Beispiel)
 $\rightarrow$ 8 Segmente mit je 6 Gitterpunkten
 Korngröße = 6 Gitterpunkte
5. Schritt: Zuordnung der Segmente zu Prozessoren unter Berücksichtigung der Kommunikationskosten
 Hier: Prozessor i kommuniziert mit $i+1$ und $i-1$.

Für Ringtopologie:



Segment $i \rightarrow P_i$

Für Hypercube-Topologie (3-dim):



Segment	0	1	2	3	4	5	6	7
Prozessor	P0	P1	P3	P2	P6	P7	P5	P4

6. Schritt: Algorithmus für Segment i

1. Initialisiere Startwerte
2. Bestimme Kanäle für Kommunikationspartner
3. Austausch der Randwerte
4. Aktualisiere $\psi_i(t) \rightarrow \psi_i(t+\Delta t)$ für alle lokalen Gitterpunkte
5. Goto 3 solange vorgegebene Schrittzahl k nicht erreicht

7. Schritt: Codierung

- Annahme: Programmiermodell bietet Control Processor/Knotenprozessor-Organisation

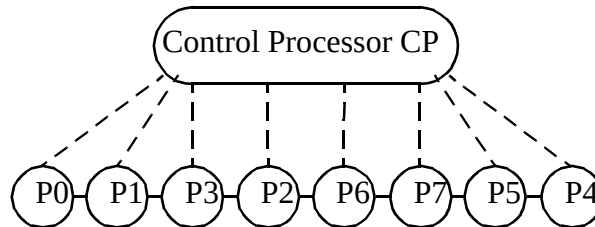


Bild 3.14. Bild Control Processor CP steuert die Ausführung der Knotenprogramme

Folge: 2 Programme nötig: CPNODE- und NODE-Programm

- Pseudocode CPNODE.WAVE ("Ensemble" = Menge der Prozessoren)

```

proc_begin main (run the control process program)
  for_begin (interactive command loop)
    [read command from terminal or standard input]
    [vm_bcastcp-broadcast command to ensemble]
    if_begin (command is INITIALIZE) then
      proc_call initialize_problem (signal
                                   ensemble to set starting values)
    else_if (command is UPDATE) then
      [read number of steps, k]
      [vm_bcastcp-broadcast k to ensemble]
    else_if (command is OUTPUT) then
      proc_call output_results (receive results from ensemble using
                                   vm_dumpcp and direct it to output stream)
    else_if (command is STOP) then
      [terminate program]
    if_end
  for_end
proc_end
  
```

- Pseudocode NODE.WAVE

```

proc_begin main (node process main program)
  [gridmap-assign processors to 1-d lattice; assign nearest neighbors]
  for_begin (interactive command loop)
    [vm_bcastelt-get command from control process]
    if_begin (command is INITIALIZE) then
      proc_call initialize_problem (assign starting values)
    else_if (command is UPDATE) then
      [vm_bcastelt-get number of steps, k, from control process]
      proc_call do_steps (perform k update steps)
    else_if (command is OUTPUT) then
      proc_call output_results (send results to control
                                process using vm_dumpelt)
    else_if (command is STOP) then
      [terminate program]
    if_end
  for_end
proc_end

```

- Pseudocode NODE.WAVE.SOLVER

```

proc_begin do_steps (perform k update steps)
  for_begin (a total of k loops)
    [load end values into l_neighbor and r_neighbor]
    [vm_shift-send r_neighbor to processor on right while
    receiving into l_point from processor on left]
    [vm_shift-send l_neighbor to processor on left while
    receiving into r_point from processor on right]
    for_begin (number of points in subdomain, i)
      if_begin (the point is a global endpoint) then
        [the value remains zero]
      else_if (the point is a left boundary) then
        [use l_point in place of  $\psi_{i+1}$  to do update]
      else_if (the point is a right boundary) then
        [use r_point in place of  $\psi_{i+1}$  to do update]
      else_if (otherwise) then
        [perform the usual update]
      if_end
    for_end
    [replace previous and current step values for next step]
  for_end
proc_end

```

Anmerkung: Die hier gezeigte Steuerung durch einen CP ist umständlich. Sie kann eleganter bewerkstelligt werden.

3.1.6. Zusammenfassung wichtiger Begriffe

Domäne (Problemdomäne, Problemraum, *domain*)

Dekomposition (Zerlegung, *decomposition*)

Segment ("Korn", *granule, grain*)

Segmentgröße, Granularität (*granularity*)

Domäne enthält M Elemente (atomare Teile, *members*) und N Segmente.

$$\text{Korngröße} = \frac{M}{N}$$

reguläre Domäne: einfache Geometrie, alle Segmente benötigen gleichen Rechenaufwand

statisch irreguläre Domäne: unterschiedlicher Aufwand pro Segment, aber ohne Änderung zur Laufzeit

dynamisch irreguläre Domäne: Aufwand pro Segment ändert sich zur Laufzeit.

Rechenelement, Knoten, Processing Element, PE

Effizienz ϵ

Speed-up S

Overhead μ

Parallelitätsgrad P

Ensemble: Menge aller an einem Problem arbeitenden PEs

3.2. Merkmale und Klassifikation

3.2.1. Vorgehen

Gesamtheit der Merkmale spannt vollständigen Entwurfsraum auf.

Verkleidungsproblem: Parallelrechner stellen sich häufig anders dar als sie in Wirklichkeit sind. (Bsp.: KSR1 = Maschine mit verteiltem Speicher, Sicht: shared memory)
Wie zu klassifizieren?

3 Beschreibungsansätze

1. Merkmale der Parallelarbeit
2. SW-Sicht (Benutzer)
3. HW-Sicht (Architekt)

3.2.2. Allgemeine Merkmale der Parallelarbeit

- Ebenen der Parallelarbeit

“ Wortebene: bit-parallele Verarbeitung (32 bit-Operanden)

| Instruktionsebene: mehrere Befehle gleichzeitig (VLIW, superskalar, PL)

Æ Fadenebene: Faden = Menge von Maschineninstruktionen innerhalb eines Adreßraums.

Mehrere Fäden in einem Adreßraum.

Beispiele für Fäden:

Schleifeniterationen

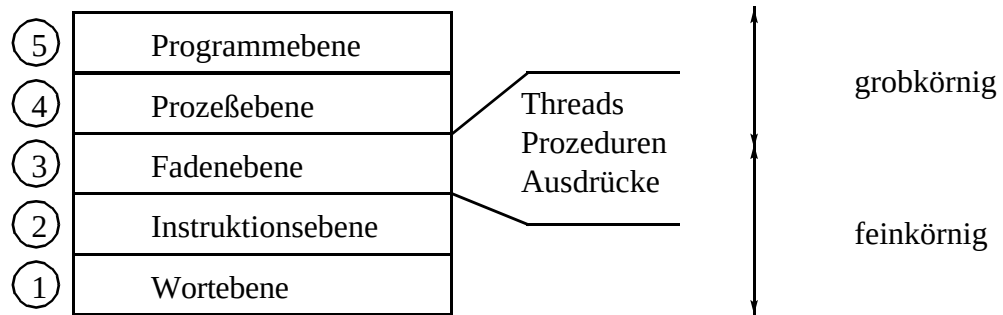
Hochsprachenausdrücke

Prozeduraufrufe

Threads (Leichtgewichtige Prozesse)

∅ Prozeßebe, mehrere Adreßräume, ”kooperierende Prozesse”

f Programmebene (multiprogramming), mehrere unabhängige Programme



- Abarbeitungsreihenfolge

Serienverarbeitung: Zu keinem Zeitpunkt, bezogen auf eine bestimmte Abstraktionsebene, befindet sich mehr als eine Aktion in Ausführung.

Parallelverarbeitung: zu einem bestimmten Zeitpunkt, auf einer bestimmten Abstraktionsebene, befindet sich mehr als eine Aktion in Ausführung.

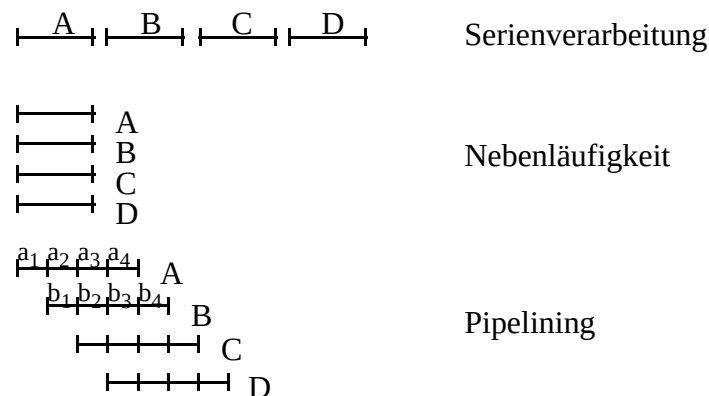


Bild 3.15. Drei mögliche Abarbeitungsreihenfolgen

Parallelverarbeitung spaltet sich auf in Nebenläufigkeit und Pipelineverarbeitung.

Nebenläufigkeit: Aktionen A, B, C... laufen gleichzeitig und unabhängig voneinander ab.

Pipelining: Aktionen A, B, C... jeweils unterteilt in n Schritte (a₁, a₂, ..., a_n), Abarbeitung in fester Reihenfolge, takt synchron. Maximaler Parallelitätsgrad: n

Veranschaulichung der 3 Abarbeitungsreihenfolgen durch ein Belegungsschema

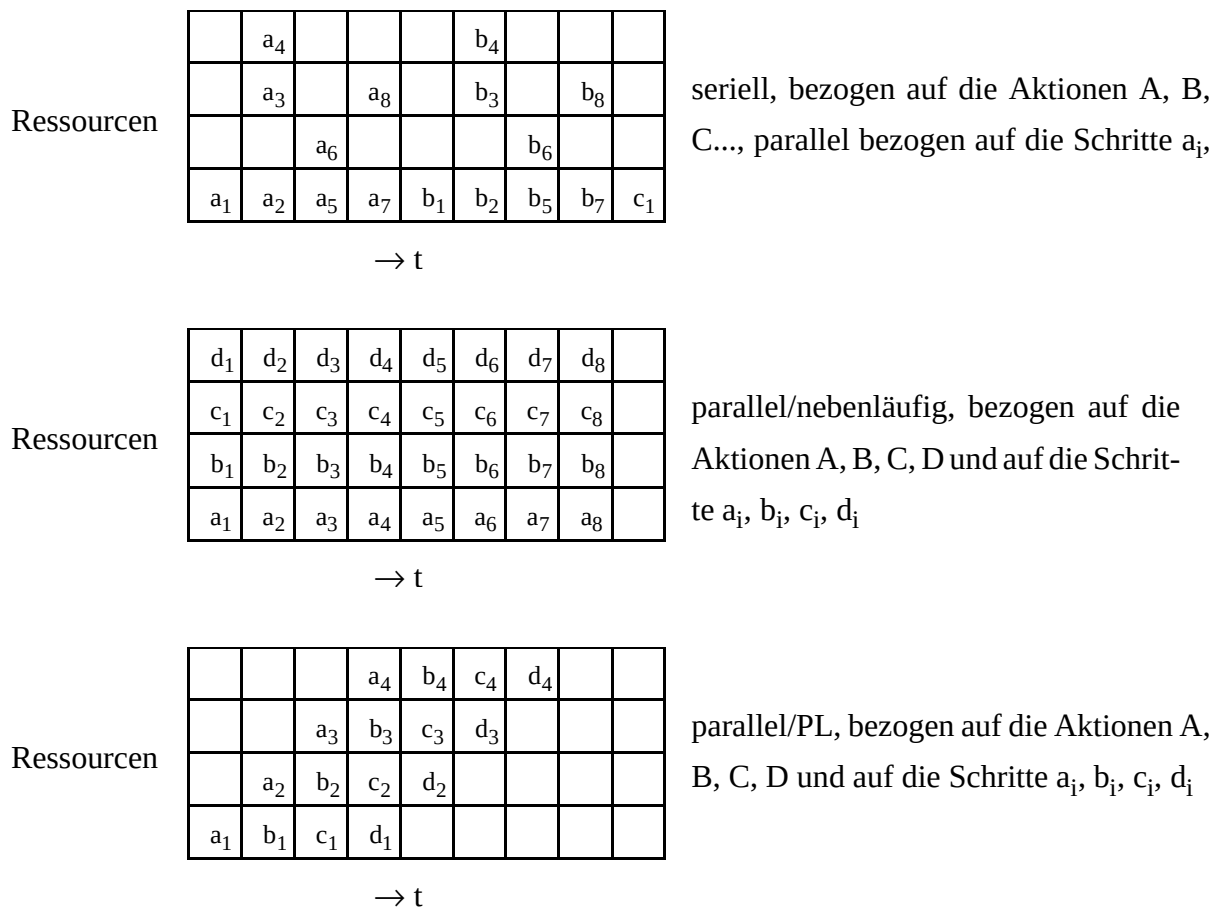


Bild 3.16. Abarbeitungsreihenfolgen im Belegungsschema

- Typen von Parallelrechnern (nach Bode)

Abarbeitungsreihenfolge → HW-Ebene ↓	Nebenläufigkeit	PL
mehrere Prozessoren pro Maschine (CU + EU)	Multicomputer	Makropipeline
mehrere Ausführungseinheiten EU pro Prozessor	Feldrechner Superskalar-Prozessor	Funktions-PL (Vektorrechner)
mehrere Elementarschaltungen pro EU	Parallelwortrechner	-

- Sichtbarkeit der Parallelarbeit

1. implizite Parallelität

Parallelität ist unsichtbar, verborgen hinter einem sequentiellen Programmiermodell (typ. FORTRAN). Vorhandene Alt-SW (*dusty decks*) soll automatisch parallelisiert werden.

2. explizite Parallelität

Herkömmliche Programmiersprachen mit Kommunikationserweiterungen. Programmierer sieht die Architektur.

3. natürliche Parallelität

Parallelität wird nicht heraussequentialisiert. Vermeidung von Seiteneffekten erleichtert Parallelisierung. Denkweise beim Erstellen des Algorithmus ist bereits nicht-sequentiell. Neue Sprachen, Paradigmen: funktional (applikativ), reduktional, logisch, objektorientiert.

In dieser Vorlesung: Schwerpunkt auf expliziter Parallelität.

Automatische Parallelisierung ist üblich auf den unteren (feineren) Ebenen (z.B. Instruktionsebene: PL), explizite Parallelität auf den oberen.

langsame VN ← grobkörnig → explizite Parallelität

schnelle VN ← feinkörnig → automatische Parallelisierung

3.2.3. Merkmale aus Benutzersicht

(SW-Sicht, Top-down-Vorgehen von der Sprache zur Architektur)

- Programmiermodell

Der Benutzer sieht unterschiedliche Abstraktionen der Maschine. Seine Sicht (view) wird als Programmiermodell bezeichnet. Es besteht in einer abstrakten Architekturbeschreibung. Diese muß nicht unbedingt identisch mit der realen Architektur sein. Die reale Maschine wird durch das Verarbeitungsmodell beschrieben.

Definition: Ein Programmiermodell ist ein abstrakter Mechanismus, der angibt, wie ein Anwender einer Maschine sein Problem formulieren muß, damit es von der Maschine abgearbeitet werden kann.

Definition: Ein Verarbeitungsmodell gibt an, wie eine bestimmte Maschine eine (programmierte) Arbeitslast wirklich abarbeitet.

Beispiel 3-4:

Verarbeitungsmodell:	Prozessoren, Bus, gemeinsamer Speicher
Programmiermodell:	Message Passing
Realisierung:	Message-Puffer im gemeinsamen Speicher, <u>keine</u> Benutzung gemeinsamer Variabler

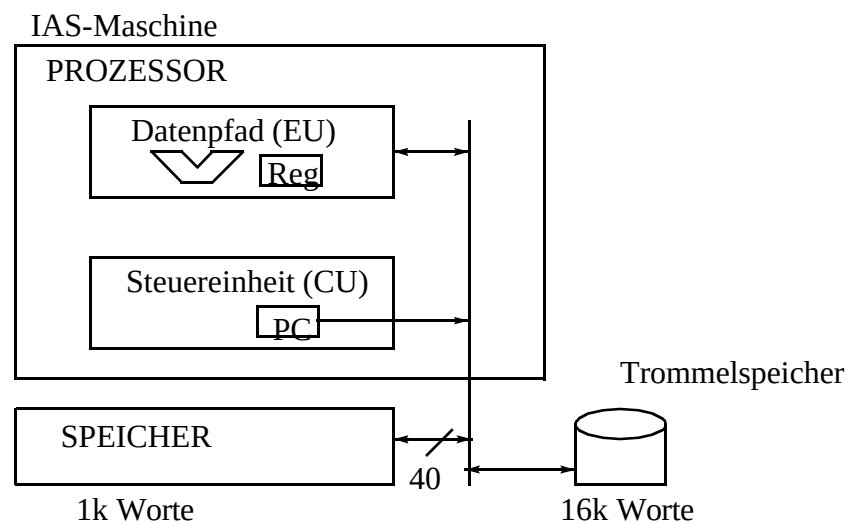
Ist Programmiermodell $\neq$ Verarbeitungsmodell, dann ist eine Übersetzung (P.Modell $\rightarrow$ V.Modell) oder eine Virtualisierung der Maschine (V.Modell $\rightarrow$ P.Modell) nötig. Im folgenden wird der Begriff "Programmiermodell" für beide Aspekte verwendet, in der Annahme, daß der Anwender so programmiert, daß sein Programm auf dem Verarbeitungsmodell einer unter Umständen höheren virtuellen Maschine ablauffähig ist.

Ein Programmiermodell (PM) beschreibt das Operationsprinzip (Funktion) und die HW-Struktur einer virtuellen Maschine, im einzelnen:

1. **Grundfunktionen der Maschine (Befehle)**
2. **Speichermechanismen (Daten)**
3. **Ablaufmechanismus, d. h. Ablaufsteuerung für die Reihenfolge bestimmter Ausführungseinheiten (Befehle, Prozeduren, Prozesse)**
4. **Kommunikationsmechanismen zwischen diesen Ausführungseinheiten**
5. **Synchronisationsmechanismen**

Beispiel 3-5:

von Neumann-Programmiermodell



1. Grundfunktionen: Befehlssatz, z. B. ADD, JMP, LOAD...
2. Speicher für Befehle und Daten in Zellen mit eindeutigen Adressen (HS, Platte)
3. Ablaufmechanismus IF, ID, EX, WB (sequentiell), dabei Datentransport zwischen ACCU, Register und Speicher. Befehlsreihenfolge festgelegt durch Befehlszeiger (PC). Implizite (Inkrement) oder explizite (Sprünge) Manipulation des PC.
4. -
5. -

Programmiermodelle können auf mehreren Ebenen existieren:

PM1	Problem
PM2	Algorithmus
PM3	Hochsprache
PM4	Betriebssystem
	HW

Bild 3.17. Vier unterschiedliche Programmiermodelle entsprechend vier unterschiedlichen Sichten der Maschine.

Programmiermodelle sind in Form von (parallelen) Programmiersprachen oder Spracherweiterungen konkretisiert. Diese werden in Abschnitt 6 behandelt.

- Programmiersprache: stellt Sprachkonstrukte gemäß Programmiermodell zur Verfügung (implizit, explizit, natürlich)
- Programmierparadigma: Denkweise, die der Formulierung von Algorithmen zugrunde liegt.
prozedural, imperativ (von Neumann)
funktional, applikativ
logisch
objektorientiert
- Kommunikationsmodell: (= Teil des Programmiermodells)
Kommunikation über gemeinsame Variable oder durch Nachrichtenaustausch
- Virtualisierung: Dem Programmiermodell kann eine virtuelle Architektur zugrunde liegen, d.h. die Realisierung (HW, Architektur) wird verborgen, verkleidet, stellt sich dem Benutzer anders (einfacher) dar als sie ist.
”Unter *Virtualisierung* versteht man die Verdeckung von architektonischen Eigenschaften. Durch Virtualisierung wird von spezifischen Parallelrechner-Eigenschaften wie Prozessoranzahl, Verbindungsstruktur und Speicherausbau abstrahiert, insbesondere wird die Verteiltheit dieser physikalischen Betriebsmittel verdeckt. Die Virtualisierung kann auf unterschiedlichen Ebenen abstrakter Maschinen erfolgen. Die Einführung virtueller Maschinen erleichtert die Erstellung und Portierbarkeit paralleler Anwendungen

gen sowie die Realisierung von dynamischem Lastausgleich.”
 (Th. Bemerl: Programmierung skalierbarer Multiprozessoren,
 BI-Verlag 1992)

Beispiel 3-6:

Realisierung → Kommunikationsmodell↓	gemeinsamer Speicher	verteilter Speicher
gemeinsame Variable	Sequent Symmetry Challenge	KSR1
Nachrichtenaustausch	Challenge	nCube

3.2.4. Merkmale aus HW-Sicht

Die Attribute des Programmiermodells beschreiben auch die HW-Verarbeitungsmodelle: Die HW muß realisieren können, was dem Programmierer in Form des Programmiermodells angeboten wird.

1. Funktionen:

Befehle, Programmstücke, Fäden

2. Datenmechanismus:

gemeinsamer/ verteilter (privater) Speicher

3. Steuer- und Ausführungsmechanismus:

1 CU, 1 EU

1 CU, n EUs

n (CU, EU)

(CU verarbeitet Befehlsstrom, EU Datenstrom)

4. Kommunikationsmechanismus/Synchronisation:

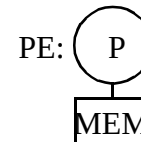
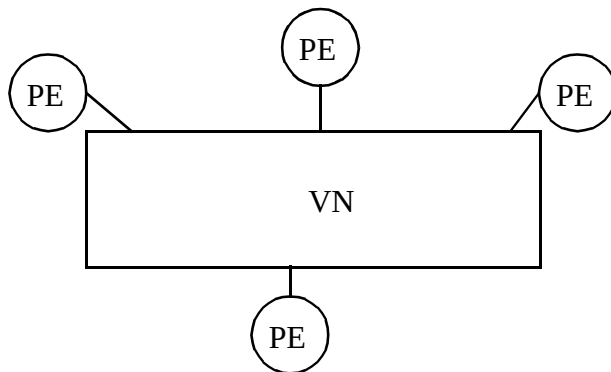
Verbindungsnetzwerke (VN), Topologien

Typ der Parallelarbeit:

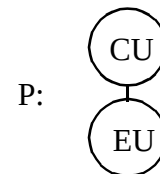
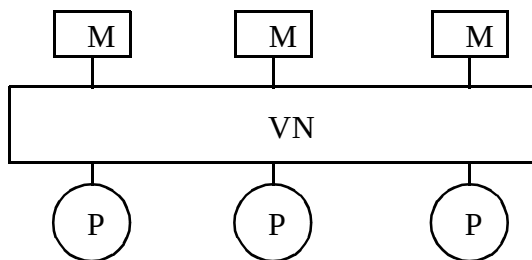
vollständige/ versetzte Nebenläufigkeit

HW-Modelle:

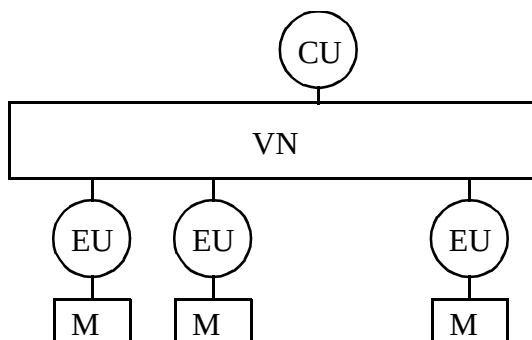
HW-Modell 1 (verteilter Speicher): Prozessorelemente (PE) und VN



HW-Modell 2 (gemeinsamer Speicher):



HW-Modell 3 (gemeinsame Steuerung):



3.2.5. Wichtige Klassifizierungssysteme

3.2.5.1 Flynn

Verwendetes Merkmal: Anzahl der Befehls- und Datenströme

		Datenströme (data)	
		single	multiple
Befehlsströme (instructions)	single	SISD	SIMD
	multiple	MISD	MIMD

- SISD: single instruction, single data stream
klass. von Neumann-Rechner (1CU, 1 EU)

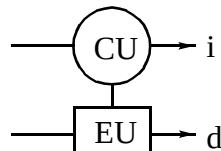


Bild 3.18. SISD-Rechner

- SIMD: single instruction, multiple data stream

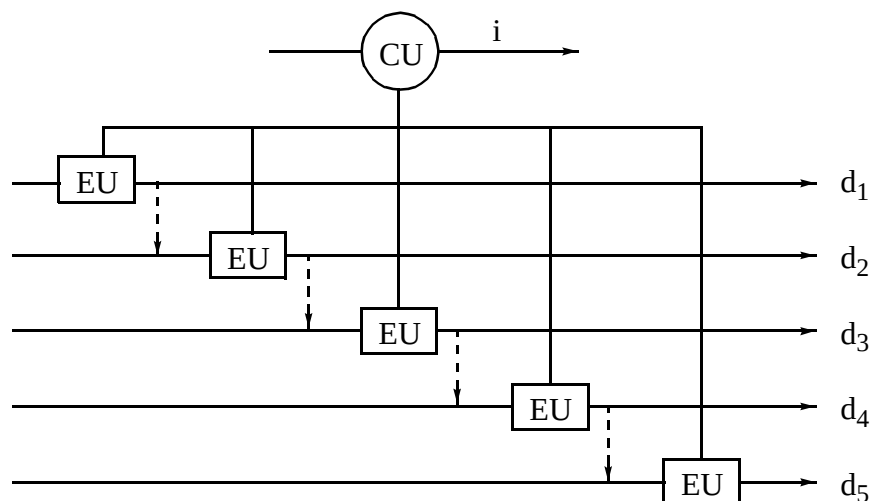


Bild 3.19. SIMD-Rechner, 1 i-Strom, 5 d-Ströme.

Gestrichelte Pfeile: Datenstrom bei PL-Rechner

SIMD-Rechner werden ausgeführt als Feldrechner oder als Pipeline-Rechner. Alle EUs führen dieselbe Operation aus, sie arbeiten takt synchron. Beispiele: ILLIAC IV (64 x 64-bit Prozessor), DAP, Cray, Connection Machine 2 (65536 x 1 bit Prozessor), Systolic Array

- MISD: nicht besetzt
(Alternativmeinung: MISD = PL-Rechner)
- MIMD: multiple instruction, multiple data stream
Je eine CU und eine EU bearbeiten einen i- bzw. d-Strom. Die Knoten arbeiten nicht notwendigerweise taktsynchron. (MIMD = mehrfache Mikroprozessoren)

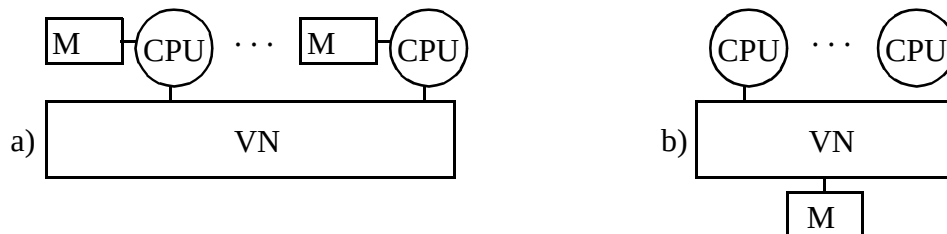


Bild 3.20. Varianten von MIMD-Rechnern

a) verteilter Speicher, b) gemeinsamer Speicher

Beispiele: iPSC/2, iPSC/860, SUPRENUM, Sequent Symmetry, Challenge, nCube

Kritik am Flynn'schen Klassifikationssystem:

- PL nicht eindeutig zuzuordnen (SIMD, MISD)
- Die weitaus meisten der heute interessanten Rechner fallen in die Klasse MIMD.

3.2.5.2 Snyder

Flynn-Schema zu grob. Verfeinerung durch weitere Unterteilung der Befehls- und Datenströme:

i-Strom	adr	instruction
d-Strom	adr	datum

Snyder-Schema gibt Anzahl (single oder multiple) der Adressen und Anzahl der Werte (i bzw. d) an, wobei der 1. Index für Adressen, der 2. für Daten steht:

$$I_{ad} D_{ad}$$

(Zum Beispiel: $I_{ss}D_{sm}$ = Befehle: single address, single instruction, Daten: single address, multiple data)

$I_{ss} D_{sm}$: Feldrechner oder Systolic Arrays (SIMD nach Flynn): mit 1 Adresse wird ein Datenfeld adressiert.

$I_{sm} D_{mm}$: VLIW-Rechner

$I_{mm} D_{mm}$: MIMD-Rechner

Kritik: Verfeinerung, aber die beiden obigen Kritikpunkte am Flynnschema gelten auch hier.

3.2.5.3 ECS

Erlangen Classification System (ECS):

Drei logische Ebenen:

- CU k
- EU d
- Wortparallelität w

Das Tripel (k, d, w) gibt jeweils den Grad der Nebenläufigkeit an.

Zusätzlich kann jede Ebene als Pipeline aufgebaut sein, wobei (k', d', w') den jeweiligen Grad der Pipeline angibt.

Ein Rechnertyp ist charakterisiert durch

$$t_{\text{RECHNER}} = (k * k', d * d', w * w')$$

* = Trennzeichen.

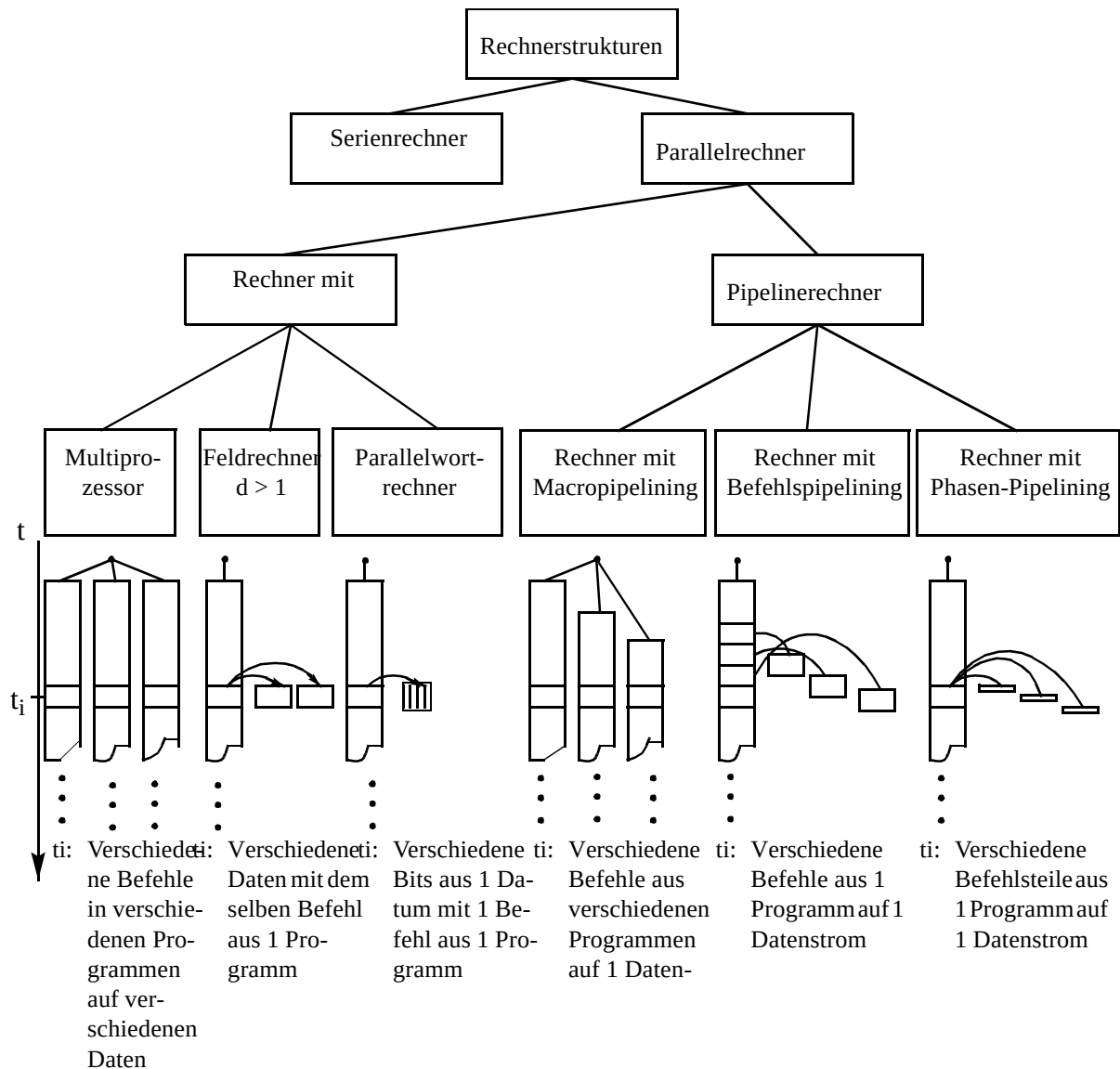


Bild 3.21. Rechnerstrukturen und ihre Charakterisierung in ECS

Kritik:

- + quantitative Beschreibung möglich
- + Typen der Parallelität werden zur Unterscheidung verwendet
- für breite Akzeptanz zu kompliziert
- zusätzliche Merkmale sind nötig: Speicherstruktur, VN...

ECS ist nützlich, aber hat sich praktisch nicht durchgesetzt.

3.2.5.4 Hockney

Hockney verwendet den Typ des VN sowie die Speicherstruktur zur Unterscheidung:

VN-Typ:

- geschaltet (*switched*)

Leitungsvermittlung: Phase 1 = Schaltung des Kommunikationspfades, Phase 2 = Datenübertragung. Beispiele für solche VN: Bus, Kreuzschienen-Verteiler, mehrstufiges Netzwerk.

Knoten: P, MEM oder PE

- statische Punkt-zu-Punkt-Verbindungen:

Knoten: PE

Beispiele für solche VN: 2-D-Feld, Hypercube, Baum sowie rekonfigurierbare Netzwerke.

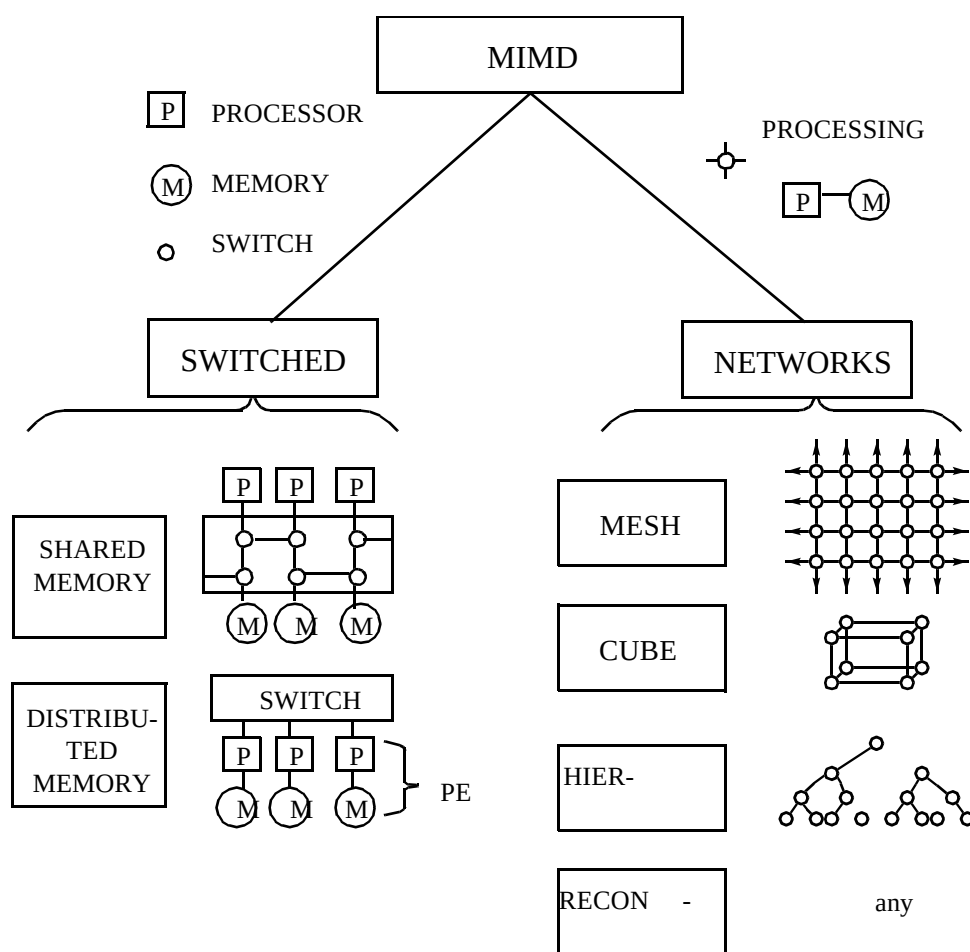


Bild 3.22. Klassifikation nach Hockney

Kritik:

- + Wesentliche Parallelrechner-Eigenschaft (VN) wird als Merkmal verwendet.
- Zuordnung nicht eindeutig: geschaltete Verbindung über Baum oder Hypercube ??
Grund für Mehrdeutigkeit: es ist unklar, ob die wirkliche oder die virtuelle Topologie als Merkmal verwendet wird.

3.2.5.5 Treleaven

Verwendete Merkmale: 2 Aspekte des Programmiermodells, nämlich Abarbeitungsmechanismus und Speichermechanismus

		Speichermechanismus	
abnehmend explizite Steuerung ↓	Abarbeitungsmechanismus	gemeinsamer Speicher	privater Speicher
	PC-Auslösung	von Neumann	kommunizierende
	Muster-(<i>pattern</i>)	Logische Programmierung	Aktoren
	Auslösung durch	Graphen-Reduktion	String-Reduktion
	Auslösung durch	Datenfluß (I-structures)	Datenfluß (Token)

Bild 3.23. Klassifikationsschema nach Treleaven

- PC-Auslösung: Die Befehlsfolge wird durch den Befehlszähler (program counter PC) gesteuert. Denkweise (*mind set*): von Neumann bzw. mehrfache von Neumann-Prozesse, welche explizit kommunizieren.

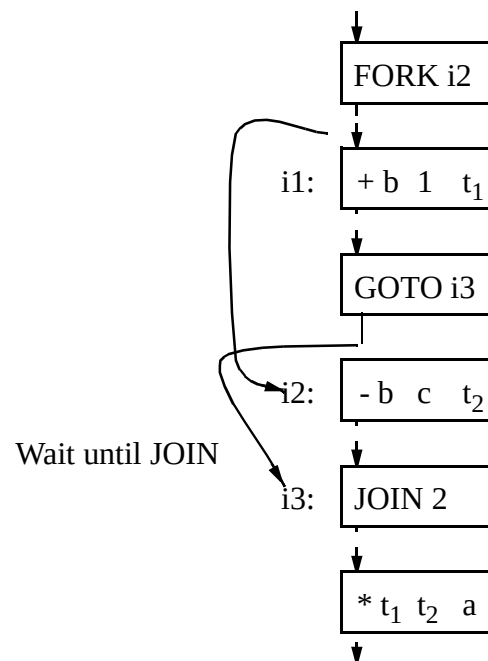
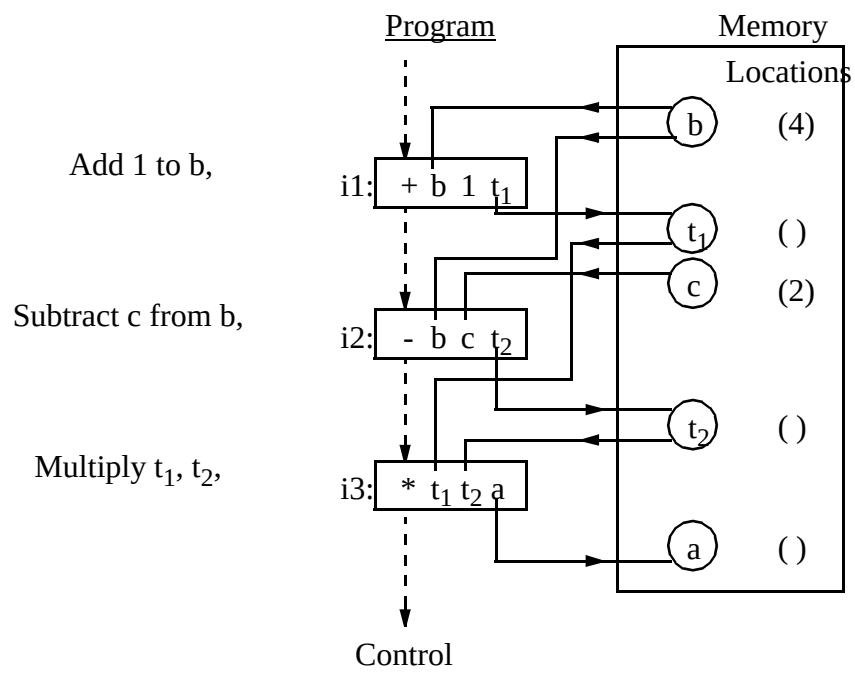
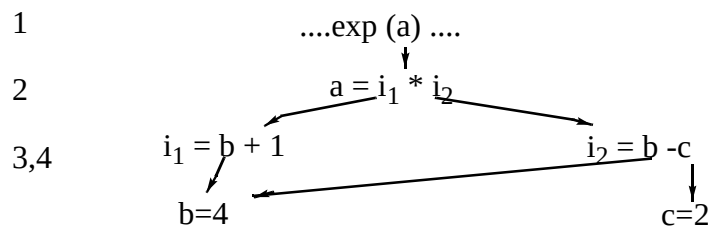


Bild 3.24. PC-Auslösung (Beispiel: $a=(b+1)*(b-c)$):

- Sequentielle Ausführung von Maschinenbefehlen: $PC \leftarrow PC + 1$.
Sprung: $PC \leftarrow f(PC, \text{const})$
- FORK-JOIN-Konstrukt: FORK wirkt wie Sprung, u. U. in einen anderen Prozessor. Alter PC bleibt erhalten.

- **Muster-Auslösung:** Der nächste auszuführende Befehl (Ausdruck, Regel, Prozeß) wird durch Mustervergleich ermittelt (vgl. PROLOG)
- **Auslösung durch Anforderung (*demand driven*, auch: *lazy evaluation*):** Ein (Teil-)Ausdruck wird ausgewertet, wenn das Auswertungsergebnis benötigt wird (vgl. funktionale, applikative Programmierung). Darstellung als Auswertungsgraph.

Anforderung: Wert von a wird in einem Ausdruck (Funktion) benötigt:



Auswertungsmethode 1: Anforderer von a erhält eine Kopie der Definition von a :
do it yourself (String-Reduktion)

Auswertungsmethode 2: a wird bei der ersten Anforderung ausgewertet, jeder weitere Anforderer erhält Zeiger auf das Ergebnis (Graphen-Reduktion)

Bild 3.25. Auslösung durch Anforderung: Reduktionsverfahren.

Beispiel: $a=(b+1)*(b-c)$

- **Auslösung durch Datenverfügbarkeit (*data driven*, auch: *greedy evaluation*):** Ein Ausdruck (Prozeß, Objekt) wartet an einer Auswertungseinheit, bis alle nötigen Daten verfügbar sind.

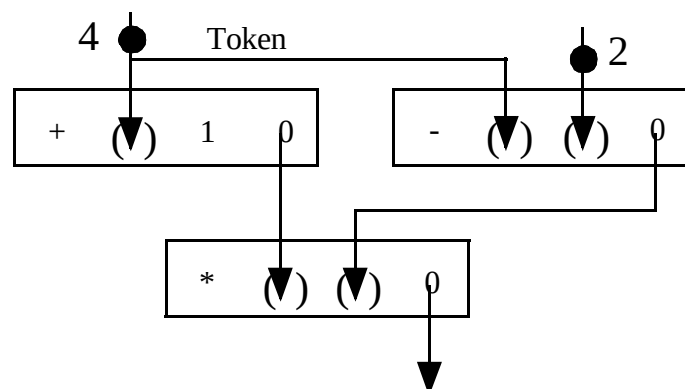


Bild 3.26. Auslösung durch Datenverfügbarkeit: Sobald alle Eingangsdaten einer Operation verfügbar, wird sie ausgeführt. Ergebnis, gekoppelt an Token (Marke), wandert zur nächsten Operation. Datenfluß-Prinzip, es gibt keine Variablen mehr.

Kritik:

- + Sowohl Benutzersicht (Abarbeitungsmechanismus) wie auch Architektur-Sicht wird berücksichtigt.
- + Zukünftige Programmiermodelle vorgezeichnet
- + Programmierparadigmen $\cong$ Abarbeitungsmechanismen
- + Schema erlaubt auch Einordnung von Programmiersprachen (vgl. Kapitel "Programmierung von Parallelrechnern")
- O-O? Akteure
- keine Berücksichtigung von VN

3.3. Verbindungsnetzwerke

3.3.1. Grundstrukturen und Klassifikation

Verbindungsnetzwerke (VN) dienen zur Informationsübertragung zwischen

Prozessor - Prozessor

Prozessor - Speicher.

Ihre logische Struktur heißt Topologie. Grundelemente von VN sind Leitungen und Schaltfunktionen.

Leitungen (Kanäle, Links):

- seriell / parallel
- elektrisch / optisch
- fest / schaltbar
- physikalisches Übertragungsverfahren
- Übertragungsprotokoll

Schaltfunktionen ermöglichen den gezielten Transport von Information:

- Adressierung (Speicheradressierung, Prozessoradressierung)
- Wegewahl (Routing)
- Vermittlungstyp (statisch / dynamisch, Durchschaltung / Paketvermittlung)
- Flußsteuerung (store-and-forward / wormhole)

VN lassen sich in 4 Klassen einteilen: Bus-VN, direkte VN, indirekte VN und virtuelle VN.

3.3.1.1 Bus-Verbindungsnetzwerke

Alle Teilnehmer (Prozessoren, Speicher) sind durch ein Leitungsbündel (Bus) verbunden; Nutzung im Zeitmultiplex; Zielangabe durch Anlegen einer (Speicher-) Adresse an die Adreßleitungen; Nutzinformation wird übertragen auf den Datenleitungen.

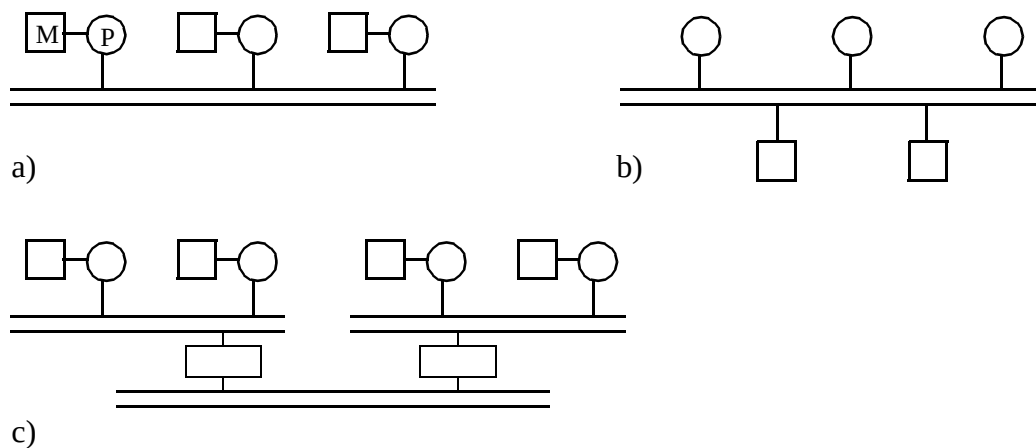


Bild 3.27. Typen von Bus-VN: a) PE-Verbindungsbus, b) Speicherbus, c) hierarchisches Bussystem

Busbandbreite = Datenwortbreite $\times f_c$

Bandbreite/PE = Busbandbreite/N

Anpassung zwischen PE-Leistung und Busbandbreite notwendig. Mit steigender Taktfrequenz f_c steigt sowohl PE-Leistung wie auch Busbandbreite, d.h. die maximale Anzahl von PEs pro Bus bleibt gleich (typisch 16 bis 32 PEs).

3.3.1.2 Direkte Verbindungsnetzwerke

Teilnehmer liegen "im Netz", sie können nur ihre direkten Nachbarn erreichen. Nachrichten für entfernte Empfänger werden von den Zwischenknoten weitergereicht (mit / ohne CPU-Beteiligung). Jeder Teilnehmer ist somit für Netzfunktionen mitverantwortlich, die nötigen Schaltelemente sind den Teilnehmern zugeordnet. Das VN besteht nur aus den Links. Charakteristisch: Nachbarschaftsbeziehungen.

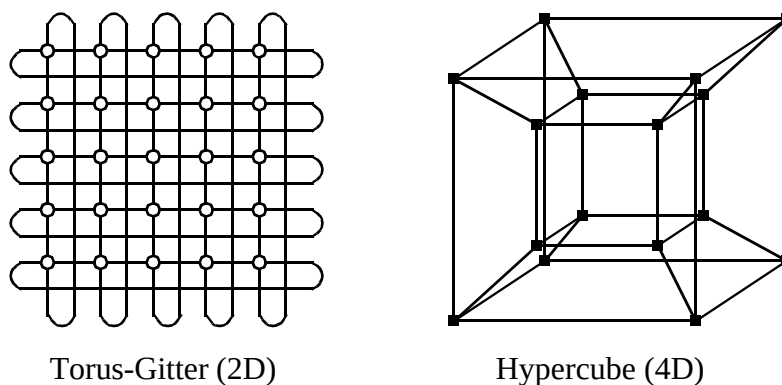


Bild 3.28. Beispiele direkter VN, Links bidirektional

3.3.1.3 Indirekte Verbindungsnetzwerke

Teilnehmer liegen außerhalb des Netzes, Schaltelemente gehören zum Netz. VN wirkt als Black Box, welche volle Erreichbarkeit aller Teilnehmer bietet. Damit funktionale Trennung von Verarbeitung (bzw. Speicherung) und Übertragung.

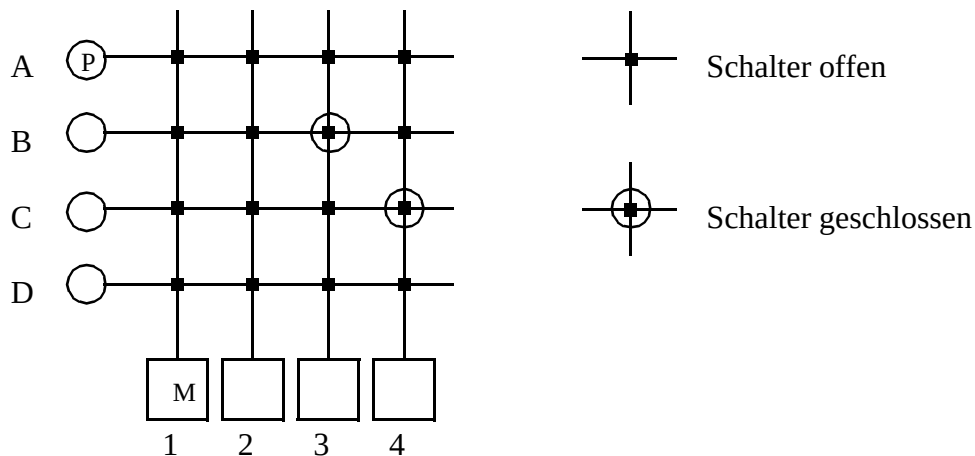


Bild 3.29. Kreuzschienenverteiler (Crossbar): An jedem Kreuzungspunkt kann eine Verbindung geschaltet sein. Im Beispiel kann Prozessor B auf Speicher 3, Prozessor C auf Speicher 4 zugreifen. Die Speicher sind dabei zu einem Zeitpunkt genau einem Prozessor zugeordnet.

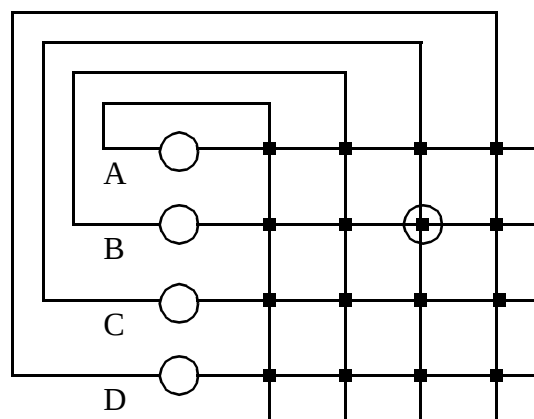


Bild 3.30. Ein Crossbar-VN kann auch verwendet werden, um Prozessoren miteinander zu verbinden, hier B-C.

Zur Klasse der indirekten VN gehören die Multistage-Netzwerke. Sie bestehen aus einer Anzahl von untereinander verbundenen und in mehreren Stufen angeordneten Schaltern.

Vertreter der Multistage-Netzwerke sind Banyan, Baseline, Benes, Delta und Omega-Netzwerk.

Zunächst wird hier das Omega-Netzwerk betrachtet.

Schalter: 2*2 mit 4 Schaltfunktionen

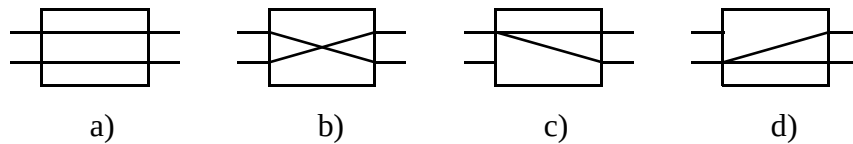


Bild 3.31. Omega-Schaltfunktionen: a) Durchschaltung, b) Überkreuzung, c) oberer Broadcast, d) unterer Broadcast

Anzahl der Schalter: $\frac{N}{2} \log_2 N$ (zum Vergleich: Schalterzahl für Crossbar: N^2)

Omega-VN bietet genau 1 Pfad von jedem Eingang zu jedem Ausgang. Verbindung der Schalter gemäß "perfect shuffle" (vgl. z.B. Harold Stone: High Performance Computer Architecture, Addison-Wesley, 1990).

Vermittlungsmethode: Packet Switching

Routing: verteilt auf die Schalter

Blockierung: hohe Wahrscheinlichkeit

Omega-Routing-Algorithmus: Führendes Bit (MSB) setzt Schalter

1	Nachricht	→	unterer Ausgang
0	Nachricht	→	oberer Ausgang

Beispiel 3-7:

Omega-VN

N = 8 Knoten

 $N/2 \log_2 N = 12$ Schalter

Wegesuche 010 → 110 und
 110 → 100 → Konflikt

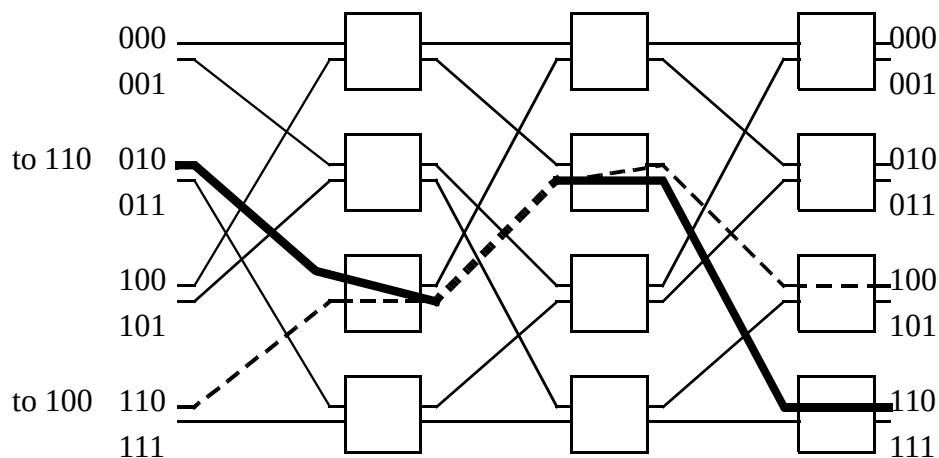


Bild 3.32. Bild Omega-Netzwerk mit N = 8 Knoten

Nachteil Omega-VN: Blockierungsmöglichkeit

Verbesserung: nicht-blockierende VN durch Mehrfach-Erreichbarkeit. Einschränkung: alle Verbindungswünsche müssen bekannt sein, dann erfolgt (evtl. dynamische) Neufestlegung (*rearrangement*).

Vertreter der nicht-blockierenden VN sind das Benes-Netzwerk und das Banyan-Netzwerk.

3.3.1.4 Virtuelle Verbindungsnetzwerke

Durch Trennung von Knotenfunktionen (Verarbeitung, Speicherung) von Übertragungsfunktionen läßt sich jedes VN als virtuelles indirektes VN auffassen. Jeder Teilnehmer besitzt eine Kommunikationseinheit KE (zum Netz gehörig). Sie übernimmt (in Kooperation mit anderen KE) alle Übertragungsaufgaben. Jeder Teilnehmer hat die Illusion voller Erreichbarkeit. Nachbarschaftsbeziehungen äußern sich in unterschiedlichen Verzögerungszeiten.

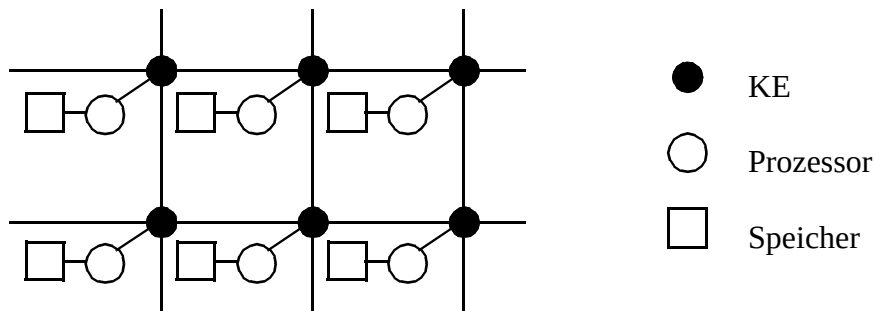


Bild 3.33. 2D-Gitter mit KEs für virtuell vollständige Erreichbarkeit

3.3.2. Dynamische Bewertungskriterien

Aus Teilnehmersicht besteht die Leistung des Netzes im möglichst schnellen Transport von Nachrichten vom Sender zum Empfänger. Man unterscheidet hierbei die Meßgrößen Durchsatz (globale Sicht) und Kommunikationszeit (lokale Sicht).

- Durchsatz

Maximaldurchsatz: $D_{\max} = n \cdot b_{\text{Link}}$

n = Anzahl unidirektionaler Sendepoints im Netz

b_{Link} = Linkbandbreite

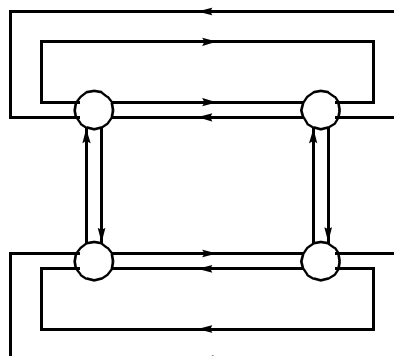
Beispiel 3-8:

Maximaldurchsatz eines 2D-Netzes mit Teiltorus, $b_{\text{Link}} = 2 \text{ MB/s}$

3 Sendepoints / Prozessor

$n = 12$

$D_{\max} = 24 \text{ MB/s}$



Der theoretische Maximaldurchsatz ist i.A. nicht erreichbar.

Gründe:

- äußere Blockierung durch Zielkonflikt (2 Teilnehmer senden an gleichen Empfänger)
- innere Blockierung durch Benutzung gemeinsamer Ressourcen (Schalter, Link)

Der tatsächlich erreichbare effektive Durchsatz $D_{\text{eff}} < D_{\text{max}}$ hängt ab vom Netzwerktyp und vom Lastmuster.

- Kommunikationszeit (*delay*, Latenz)

Für die Beurteilung des Zeitverhaltens paralleler Prozesse sind die Kommunikationsmechanismen genauer zu betrachten.

Der Kommunikationsvorgang beginnt mit dem Sendeauftrag des Sendeprozesses und endet mit dem Empfang des letzten Bytes der Nachricht beim Empfangsprozess (genauer: mit dem Return vom Receive-Aufruf des Empfangsprozesses).

Die Kommunikationszeit t_K setzt sich aus folgenden Komponenten zusammen:

Vorbereitungsszeit t_{VS} : Zeit vom Sendeauftrag bis zum Absenden des ersten Bytes

Ursachen: SW-Overhead, Prozeduraufruf, Prozeßwechsel,
Protokoll-Overhead, Umspeicherzeit (Puffer)

Durchlaufzeit t_D : Zeit vom Absenden bis zur Ankunft des ersten Bytes (ohne Blockierungen)

Ursachen: Leitungslaufzeiten, Zwischenspeicherung,
Routingverzögerungen (bei dynamischen Routing)

Blockierungszeit t_B : Wartezeit im Netz durch Ressourcenkonflikte

Ursachen: Netzstruktur, Last

Übertragungszeit $t_{\bar{u}}$: Zeit von der Ankunft des ersten Bytes bis zur Ankunft des letzten Bytes beim Empfänger

$t_{\bar{u}} = m / b_{\text{Link}}$ mit m = Nachrichtenlänge

Nachbereitungszeit t_{NE} : Zeit von der Ankunft des ersten Bytes bis zum Ende des Receive-Aufrufs im Empfangsprozess

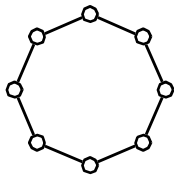
Die Kommunikationszeit t_K läßt sich darstellen als

$$t_K = t_S + \frac{m}{b_{\text{Link}}}.$$

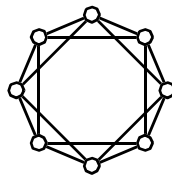
Hierbei sind alle von der Nachrichtenlänge unabhängigen Verzögerungsanteile in der Setup-Zeit t_S zusammengefaßt. t_S ist eine wichtige Kenngröße von Parallelrechnern. Sie wird ohne Blockierungen gemessen und liegt heute im Bereich von einigen μs .

Häufig verwendete Netztopologien sind:

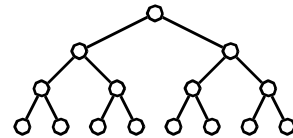
- Ring
- Chordaler Ring (Verbesserung der Fehlertoleranz)
- 2D-Gitter (mit / ohne Torus)
- D-dimensionaler Hypercube
- Cube Connected Cycles (entsteht aus D-dimensionalem Hypercube, indem jeder Knoten durch einen D-fachen Ring von Knoten ersetzt wird). Vorteil: lokale Komplexität bleibt konstant bei wachsendem D.



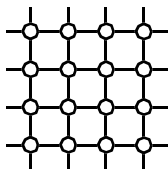
Ring



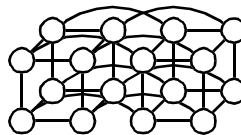
Chordaler Ring



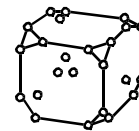
Binärbaum



2D-Gitter



4D-Hypercube



Cube-Connected Cycles

Bild 3.35. Beispiele häufiger direkter VN-Topologien

Kriterien:

- Ports pro Knoten (Knotenkomplexität): bestimmt die Kosten eines Knotens mit. Sollte konstant sein, wenn Knotenzahl wächst.
- Weglänge: Anzahl der nötigen Routingschritte
- Durchmesser: maximale Weglänge
- durchschnittliche Weglänge: Durchschnitt der Weglängen aller möglichen Zweierbeziehungen
- Verbindungsaufwand: Zahl aller Links im VN

In der folgenden Tabelle sind einige Topologien indirekter VN anhand dieser Kriterien bewertet.

Der Aufwand eines VN steigt mit der Anzahl der Ports pro Knoten. Höhere Knotenkomplexitäten haben i.A. kleinere durchschnittliche Weglängen zur Folge. Es wird daher als Figure of Merit das Produkt aus Knotenkomplexität und durchschnittlicher Weglänge angegeben (Kommunikationskomplexität).

Topologie	Vollverma- schung	Ring (bidi- rekt.)	Binärbaum (D Ebenen)	2dim. Git- ter (Torus)	D-dim. Hy- percube	D-dim. CCC
Knotenanzahl N	N	N	$2^D - 1$	W^2	2^D	$D \times 2^D$
Ports pro Knoten	N-1	2	≤ 3	4	D	3
Max. Weglänge	1	$\approx N/2$	$2(D-1)$	W	D	$\approx 5D/2 - 2$
Ø Weglänge	1	$\approx (N+1)/4$		$\approx W/2$	D/2	$\approx 7D/4 - 3$
Verbindungs- aufwand	$N \cdot (N-1)/2$	N	N-1	2N	D/2 N	3/2 N
Beispiel						
Knotenanzahl	2048	2048	2047	2025	2048	2048
Ports pro Knoten	2047	2	≤ 3	4	11	3
Max. Weglänge	1	1024	20	45	11	18
Ø Weglänge	1	≈ 512		$\approx 22,5$	5,5	11
Kommunikations- komplexität	2047	1024		90	60,5	33
Verbindungs- aufwand	$2 \cdot 10^6$	2048	2046	4050	11264	3072

Bild 3.36. Vergleich einiger häufiger Topologien

3.3.4. Wegesuche (Routing)

Routing-Verfahren legen den Weg einer Nachricht vom Sender zum Empfänger fest. Sie sind abhängig von der Topologie und der Knotenadressierung.

Die Knotenadressierung sollte so gewählt werden, daß die Adressen in einfacher Weise für Routingentscheidungen benutzt werden können.

Die Entscheidung über einen Weg kann zentral oder verteilt getroffen werden. Eine zentrale Steuerung kann den Zustand des VN berücksichtigen (Vermeidung von Blockierungen). Basis jeder Routing-Entscheidung sind Start- und Zieladresse. Sie werden meist der Nachricht vorangestellt.

Routing-Entscheidungen basieren auf

- Table-Lookup (Ausgangsport (Zieladresse))
- Adreßrechnung (z.B. Omega, Hypercube)

Werden dabei nacheinander die Adreßbits verbraucht, so spricht man von Adreß-Erosion.

- Das Intervall-Routing ist ein Spezialfall der Table-Lookup-Verfahren. Einsatz für beliebige Topologien.

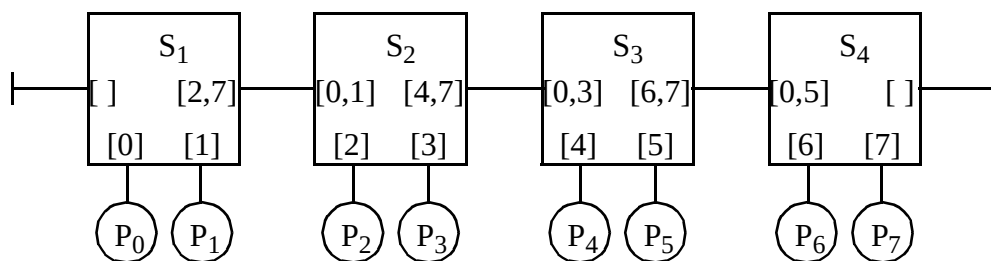
Idee: Statt einer starren Zuordnung (Ziel → Ausgangsport) wird eine Menge von Ausgangsports in Form eines Intervalls angegeben.

Folge: Tabelle wird kleiner, aber Routing-Entscheidung verlangt komplexere HW (Vergleiche für Intervalle)

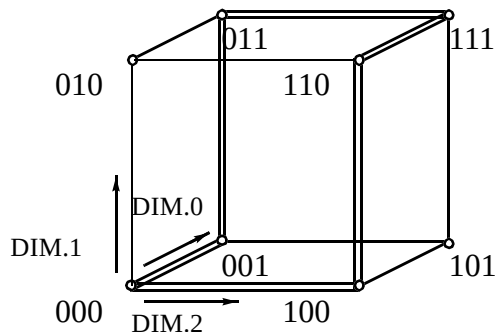
Verwendung im C104-Routing-Switch für den INMOS-Transputer T9000.

Beispiel 3-9:

Intervall-Routing für lineare Kette von vier 4-Wegeschaltern mit je zwei Teilnehmer-(PE-) Anschlüssen



- Das Hypercube-Routing beruht auf der regulären Adressierung eines D-dimensionalen Würfels. Eine Annäherung an das Ziel erfolgt durch stufenweise Verringerung der Hammingdistanz. Unterschiedliche Wahl des Ausgangsports ist möglich, wenn Hammingdistanz > 1. XOR-Verknüpfung von aktueller und Zieladresse ergibt die möglichen Ausgangsports ("1"-Stellen).

Beispiel 3-10:Hypercube-Routing, $D=3$, Start=000, Ziel=011

.. $\text{XOR}(000, 011) = 011$

Mögliche Wege: Dim. 0, Dim. 1

Entscheidung: Dim. 0, d.h. Knoten 001

| $\text{XOR}(001, 011) = 010$

Möglicher Weg: Dim. 1

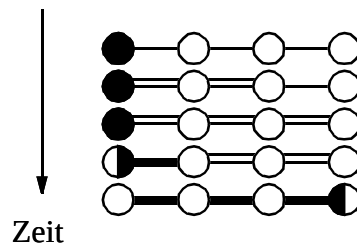
1. Alternativweg: $000 \rightarrow 010 \rightarrow 011$

2. Alternativweg (Umweg mit zeitweise steigender Hammingdistanz):

3.3.5. Vermittlungsprinzip

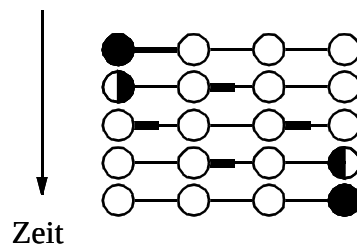
- Im Telefonnetz wird zunächst eine Verbindung aufgebaut (Durchschaltung), dann erfolgt die Sprachübertragung, schließlich der Verbindungsabbau. Dieses Prinzip wird als Leitungsvermittlung (*circuit switching*) bezeichnet. Übertragung auf Datennetze: Steuerpaket mit Absender- und Zieladresse wird dynamisch durch das Netz geroutet, es belegt dabei die nötigen Ressourcen (Leitungen, Puffer) und etabliert damit einen logischen Kanal. Die Datenübertragung erfolgt dann mit der vollen Kanalbandbreite. Blockierungen sind nur möglich beim Verbindungsaufbau.
- Bei der Paketvermittlung (*packet switching*) wird jede (längere) Nachricht in Blöcke (Pakete) fester Länge aufgeteilt. Jedes Paket erhält Absender- und Zieladresse im Kopf (Header) und sucht sich selbständig den Weg durch das VN. Resultat: bessere Auslastung des Netzes, aber Steuerinformation und -aufwand für jedes Paket. Pakete können verloren gehen oder in falscher Reihenfolge ankommen: Sequenzkontrolle auf höherer Protokollebene nötig.

Leitungsvermittlung (circuit switching)



- statische Zuteilung von Übertragungskapazität
- Routing-Overhead nur beim Verbindungsaufbau
- keine netzinterne Datenpufferung erforderlich (Blockierungen nur beim Verbindungsaufbau möglich)

Paketvermittlung (packet switching)



- dynamische Zuteilung von Übertragungskapazität
- Routing-Overhead für jedes Paket
- Paketpufferung in den Schaltern erforderlich

Bild 3.37. Vergleich von Leitungs- und Paketvermittlung

- Durch Sequenz- und Ausfallkontrolle sowie ggf. Wiederholung von Paketübertragungen kann dem Teilnehmer der Eindruck einer fest geschalteten Leitung gegeben werden, obwohl auf niedriger Ebene zunächst nur ein Strom ungeordneter Datenpakete (Datagramm-Service) ankommt. Der Teilnehmer nutzt damit einen virtuellen Kanal.

3.3.6. Flußmechanismen

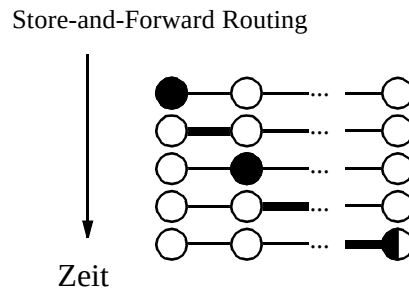
Im Fall der Paketvermittlung kann mit oder ohne Zwischenpufferung gearbeitet werden.

Beim Store-and-Forward-Verfahren wird das Paket in jedem Knoten zwischengespeichert, die Übertragungszeiten steigen proportional zur Anzahl von Zwischenstufen.

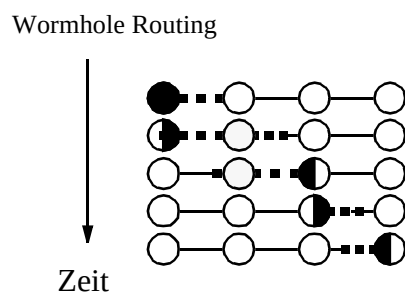
Beim Wormhole-Routing (auch: Virtual Cut Through) wird der Kopf der Nachricht ohne Pufferung sofort (d.h. nach der Routing-Entscheidung) zum nächsten Knoten weitergeleitet. Eine Zwischenpufferung erfolgt nur im Blockierungsfall. Die Pufferkapazität muß nicht die gesamte

Paketlänge umfassen, es genügt die Pufferung eines Paketteils, des sogenannten Flits (*flow control unit*). Bei Blockierungen wird das Nachfließen weiterer Flits verhindert. Verfahren wird der Leitungsvermittlung ähnlich (für lange Nachrichten).

Analogie: Wurm im Sandhaufen.



- Pakete werden in jedem Schalter zwischengepuffert
- Übertragungszeiten addieren sich mit jeder Schaltstufe



- Kopf des Daten"wurms" öffnet sich einen Kanal, der
- Wird der Kopf blockiert, stoppt der ganze Wurm, alle
- Ohne Blockierungen fällt die Übertragungszeit nur

Bild 3.38. Vergleich von Store-and-Forward- und Wormhole-Routing.

3.3.7. Lastverteilung und Hot Spots

- Blockierung und Deadlock

Bestehen keine Alternativwege, so kann es zu Verklemmungen kommen (zyklische Ressourcenbelegung). Maßnahmen:

- Deadlockerkennung durch Time-out, Auflösung durch Löschen von Puffern und Neuaufsetzen
- Deadlockvermeidung durch Einschränkungen beim Routing oder Partitionierung in Deadlock-freie Teilnetze mit Ressourcen-Replikation.

- Lastkonzentration und Hot Spot-Problem

Jede Anwendung kann in Verbindung mit bestimmten Topologien zu einer ungleichmäßigen Belastung von Links und Kommunikationsknoten führen. Gründe:

- ungünstiges Mapping
- dynamisch variable Lastverteilung

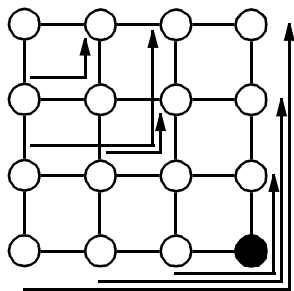
Rückstauereffekt im Netz: Kurzzeitige Überlastung eines Links führt zu einem Rückstau, welcher sich baumartig ausbreitet: Tree Saturation, ausgehend von einem Hot Spot.

Beispiel 3-11:

Matrixtransposition

2D-Gitter, festes Routingschema

Operation: Vertauschung jeder Matrixposition mit der an der Diagonalen gespiegelten Position.



Routing-Schema: 1. horizontal rechts
2. vertikal oben

Folge: Überlastung der Links rechts unten.

Rückstauereffekt beim Teilnehmer: Ausgangspunkt für eine Tree Saturation kann auch ein überlasteter Teilnehmer sein (*hot destination*): Alle Teilnehmer senden an einen Knoten nur leicht gegenüber der Gleichverteilung erhöhte Datenmengen: Pufferüberlauf beim Zielknoten, dann bei den Vorgängerknoten usw. Folge: auch freie Ports oder Links werden überlastet. Hot Destinations sind häufig globale Variablen im gemeinsamen Speicher, globale Systemtabellen, Auftragswarteschlangen oder Semaphore.

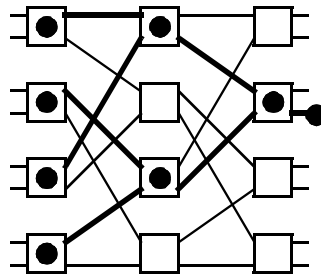


Bild 3.39. Tree Saturation in einem Multistage-Netzwerk

- Dynamische Lastverteilung durch nicht-deterministisches Routing

Hot Spot-Ursache: festes Routing

Abhilfe durch dynamisches Routing

-Zwei-Phasen-Random-Routing

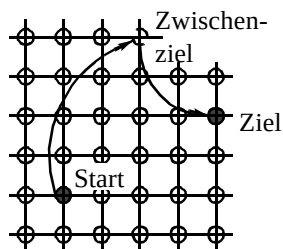
Phase 1: Zufällige Auswahl eines Zwischenziels

Phase 2: Weitersenden zum Ziel

Folge: gleichmäßige Lastverteilung

Nachteile: im Mittel doppelte Weglänge

keine Verbesserung bei Zielkonflikt



Effekt des Random Routing:

- verbessertes Worst-Case Verhalten zu Lasten des Best-Case Verhaltens
- vorhersagbare Kommunikationsdelays

Bild 3.40. Zwei-Phasen-Random-Routing

-Adaptives Routing

lokale Entscheidung für den weniger belasteten Weg (wenn Alternativen vorhanden)

3.3.8. Kombinierende Netzwerke

- Prinzip

Annahme: Hot Spot ist verursacht durch Speicherzugriffe auf globale Variablen, d.h. das Multistage-VN verbindet PEs mit MMs.

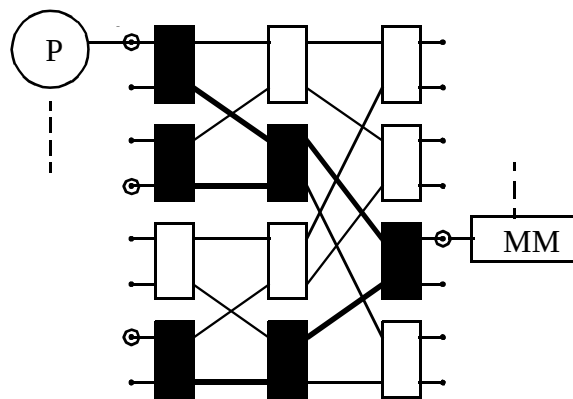


Bild 3.41. Hot Spot infolge mehrfachen Speicherzugriffs auf eine globale Variable im Memory Module MM.

Beobachtung: Es existieren "positive Konflikte", die zu einer Verkehrsreduktion eingesetzt werden können. Dazu werden zwei Speicherzugriffe durch geeignete Kombination auf einen reduziert. Der Schalter muß dazu zusätzliche HW besitzen.

Mögliche Kombinationen:

Load-Load (Lesen derselben Variablen):

Nur eine Load-Operation wird zum MM geleitet. Nach Eintreffen der Antwort wird diese dupliziert.

Store-Store (Schreiben auf dieselbe Variable):

Ein Store (das zuerst eingetroffene) wird ignoriert, nur das zweite wird an MM geschickt.

Load-Store:

Schicke Store-Operation an MM, beantworte die Lade-Operation mit dem neuen Wert.

Fetch & Add (atomare Operation zur Synchronisation und Warteschlangenverwaltung):

F&A (V,e) entspricht den beiden Operationen (im MM)

1. schicke V an den Prozessor
2. ersetze V durch V+e (im MM)

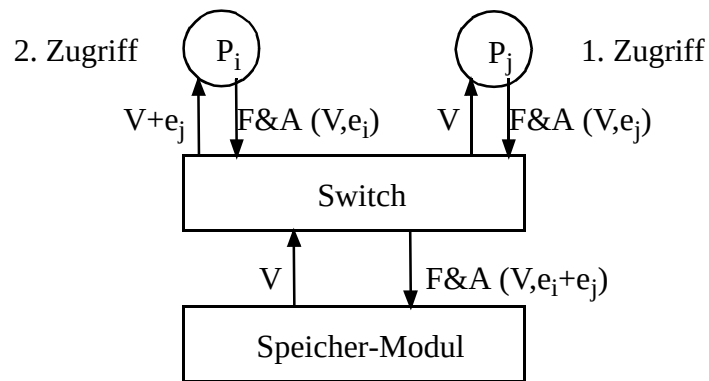


Bild 3.42. Kombination zweier F&A-Operationen

Beispiel 3-12:

Kombination zweier F&A

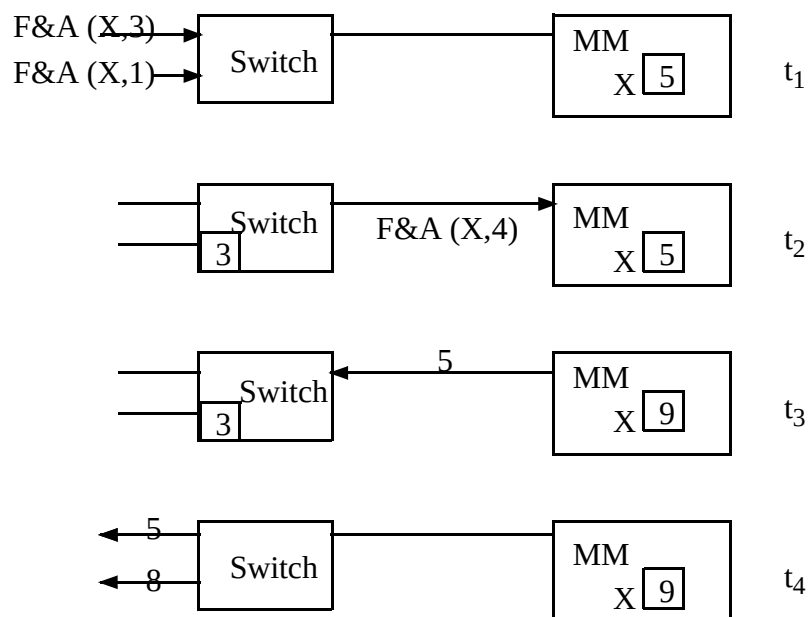


Bild 3.43. Schrittweiser Ablauf eines kombinierten F&A.

- Aufbau und Arbeitsweise eines Kombinationsschalters

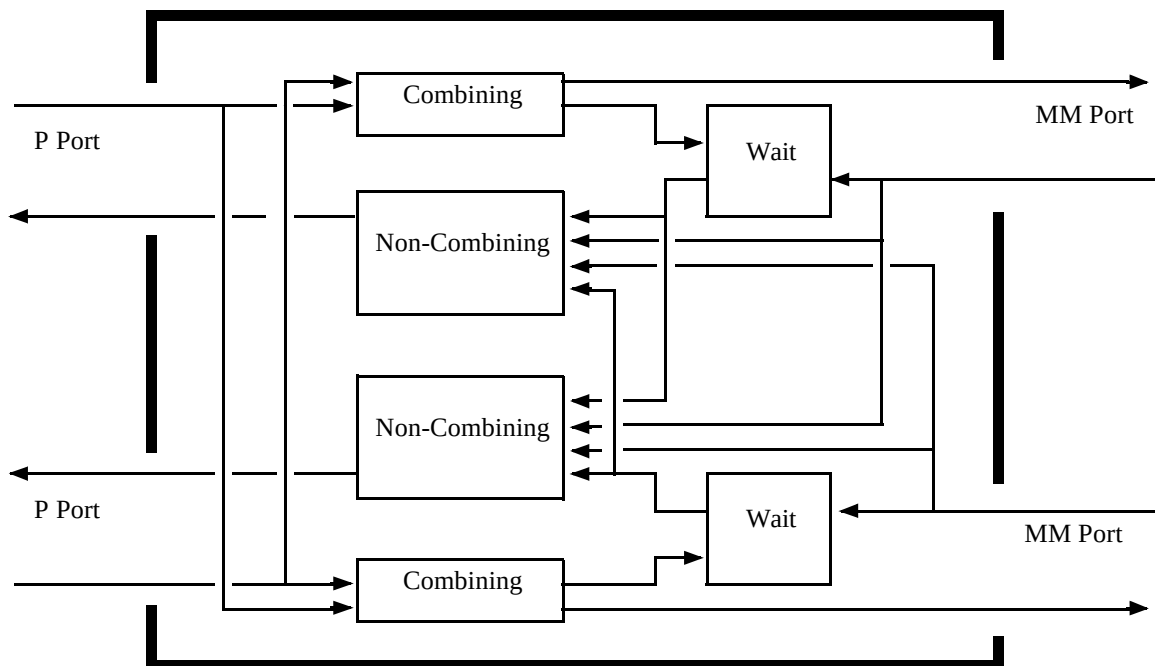


Bild 3.44. Kombinationsschalter eines Omega-Netzwerks

Speicherzugriffsanforderung = (Opcode, Adresse, Daten)

Adresse = (P-Adresse, MM-Adresse, Speicheradresse in MM)

Ablauf

1. Speicherzugriffsanforderung wird in "Combining Queue" gespeichert.
 2. Neue Anforderung wird mit allen alten (in der Queue) verglichen: gleicher MM? gleiche Speicheradresse? Operationen kombinierbar?
 - 3a. Vergleich negativ: neue Anforderung ans Ende der Queue. STOP
 - 3b. Vergleich positiv: beide Anforderungen werden im Wait Buffer notiert.
 4. Durchführung der Kombination (z.B. Load-Load): Neue Anforderung in der Combining Queue wird gelöscht.
 5. Antwort vom MM kommt an:
durchsuche Wait Buffer
 - a) negativ: Antwort → Non-Combining Queue
 - b) positiv: 2 Antworten → Non-Combining Queue
- Kosten eines Kombinationsschalters: 6 einfacher Schalter (Pfister, Norton, 1985)
 - Effekt der Kombination (Simulation 64 x 64 Omega-Netzwerk, 2 x 2 Switches, 2% der Speicherzugriffe auf 1 Variable)

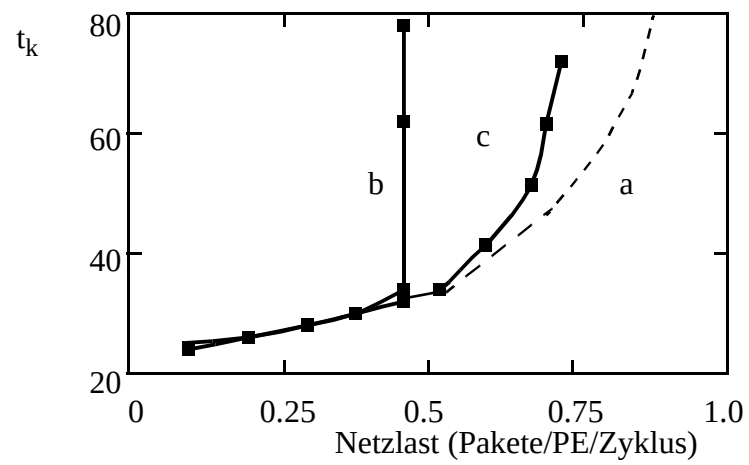


Bild 3.45. Durchschnittliche Kommunikationszeit t_k (gemessen in Zyklen) als Funktion der Netzlast (nach Pfister, Norton, 1985).
a) gleichförmige Lastverteilung (Modell)
b) Omega-Netz ohne Kombination
c) Omega-Netz mit Kombination

3.4. Quantitative Betrachtung der Parallelarbeit

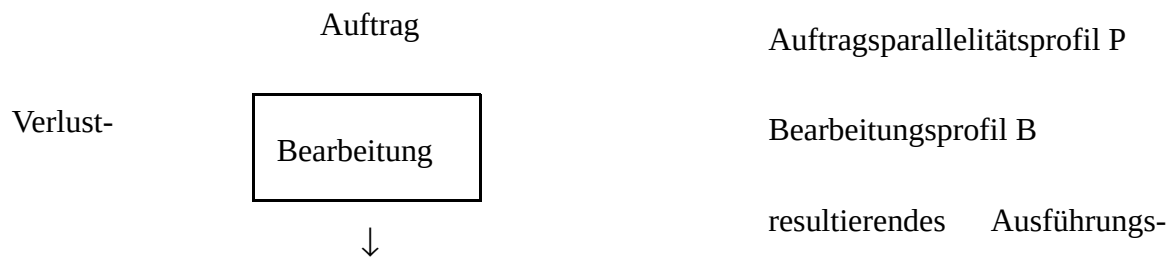
3.4.1. Verlustmechanismen

Soll eine Problemstellung auf einem Rechner behandelt werden, so durchläuft sie eine Folge von Formulierungsstufen. Jede dieser Stufen wird ein charakteristisches Parallelarbeitsverhalten aufweisen. Dabei wird in der Regel der mittlere Parallelitätsgrad von Stufe zu Stufe abnehmen. Beim Übergang von einer Stufe zu einer niedrigeren werden Verlustmechanismen wirksam, welche den Parallelitätsgrad einschränken.

<u>Formulierungsstufe</u>	<u>Verlustmechanismus</u>
Problem (Natur)	
natürlichsprachliche	Beschreibungsebene, gewählte
algorithmische Problembeschreibung	Art der Dekomposition,
HLL-Programm	Art der Kommunikationsprimitive,
auf Prozessoren verteiltes	Anzahl zugewiesener Prozessoren, Fehlanpassung zwischen Problem-
laufendes Programm	CPU-Konflikte, andere Ressourcen- konflikte, Kommunikationsverzögerungen (Netzüberlastung), Datenab-

Bild 3.46. Formulierungsstufen und Verlustmechanismen für parallele Probleme. Zusätzliche Stufen und Verluste sind möglich, z.B. wenn als Zwischenstufe eine virtuelle Parallelrechnerarchitektur eingeführt wird.

3.4.2. Kenngrößen

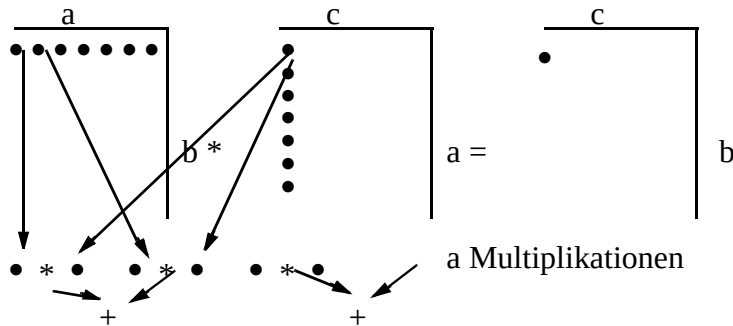


- Ein Auftrag wird beschrieben durch sein Auftragsparallelitätsprofil $P(i)$. $P(i)$ gibt an, wieviele Elementaroperationen (z.B. Maschinenbefehle) zum Zeitpunkt i gleichzeitig ausgeführt werden können. i ist die diskrete Zeit, gemessen in ganzzahligen Vielfachen der Taktzeit τ . $P(i)$ ist definiert für $i = 1, 2, \dots, i_m$. $P(i)$ gibt den Parallelitätsgrad P zur Zeit i an.

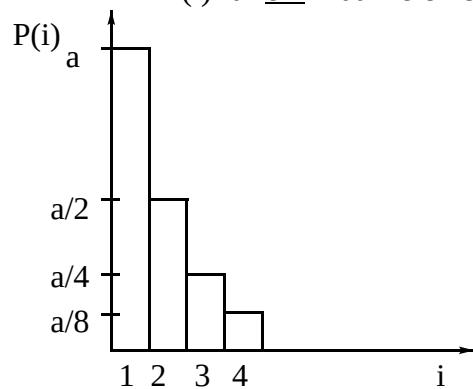
Messung von $P(i)$: prinzipiell unmöglich, da die beobachtbare Ausführung auf der parallelen Maschine bereits durch das Bearbeitungsprofil beeinflusst ist. Aber: die ideale Maschine (unbegrenzte Prozessorzahl, Latenzzeit 0) kann simuliert werden.

Beispiel 3-13:

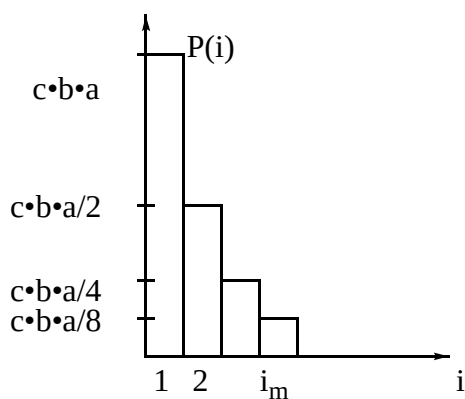
Parallelitätsprofil $P(i)$ einer Matrixmultiplikation



$P(i)$ für ein Matricelement:



Annahme: keine Speicherzugriffskonflikte $\rightarrow$ alle $c \cdot b$ Matricelemente können unabhängig voneinander (also gleichzeitig) berechnet werden.



mit $a = b = c = 16$:

Gesamtzahl der Elementaroperationen: $16 \cdot 16 (16 + 8 + 4 + 2 + 1) = 7936$

$i_m = 5$

Prozessorzahl (ideal) = 16^3

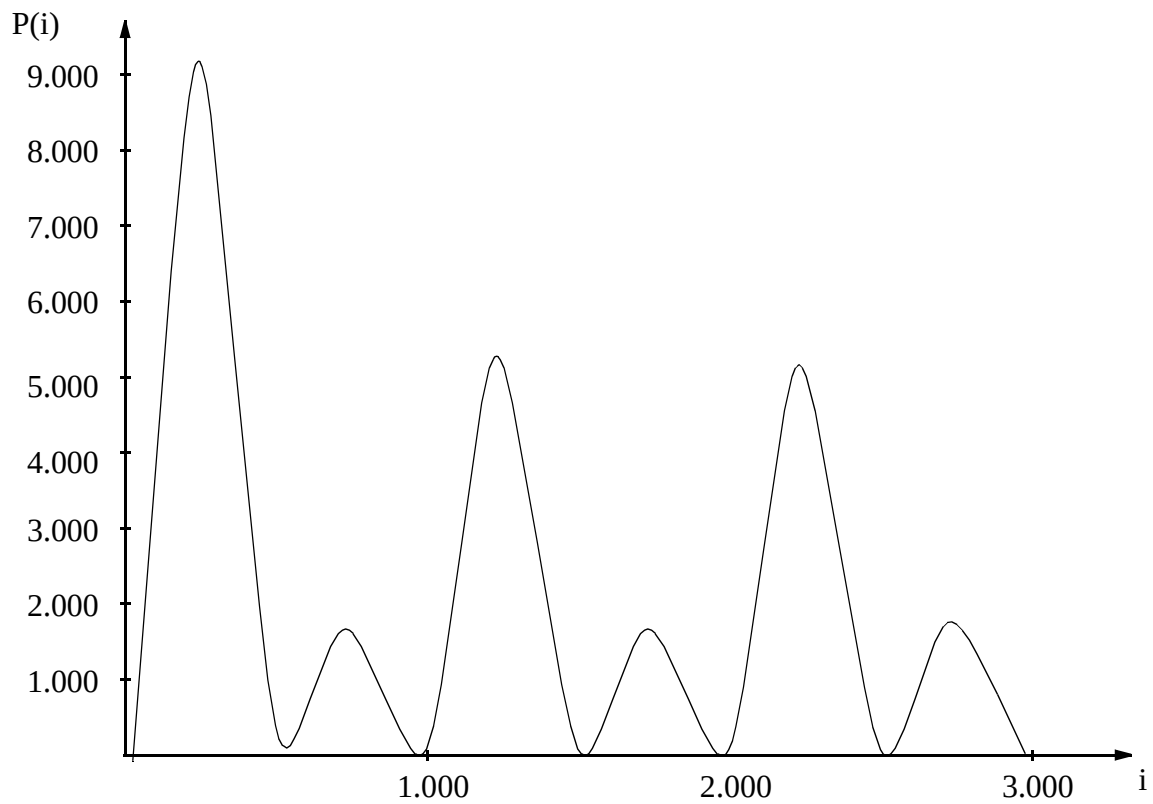
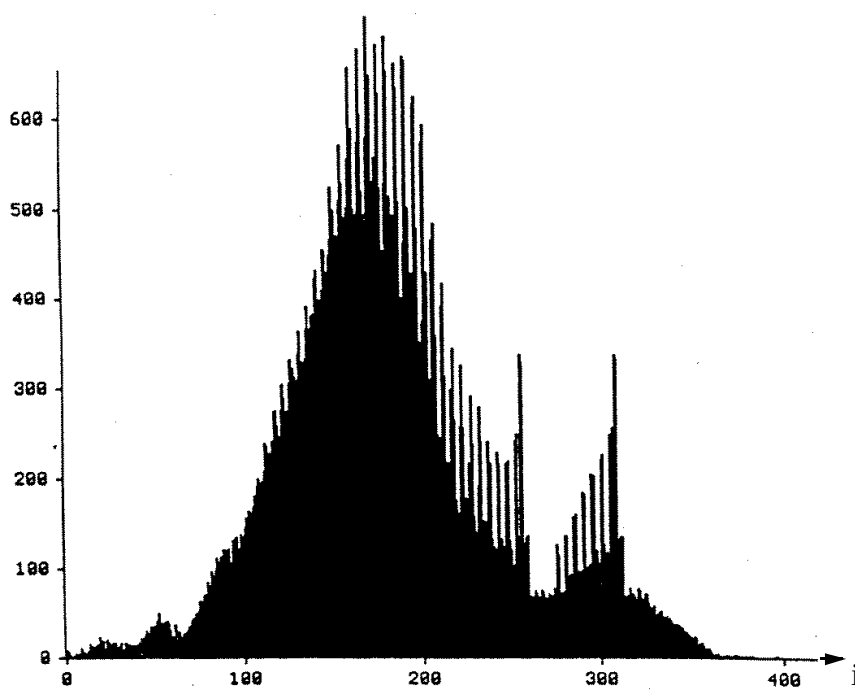


Bild 3.47. Parallelitätsprofil für Wärmeleitungsgleichung

Bild 3.48. Parallelitätsprofil für N-Körper-Problem ($N=10$)

- Die Gesamtzahl aller Elementaroperationen ist die Arbeit A:

$$A = \sum_{i=1}^{i_m} P(i)$$

Anmerkung: Bei der Ausführung eines Programms auf einem Parallelrechner sind zusätzliche Operationen zur Kommunikation und Synchronisation nötig. Außerdem können Prozessoren untätig sein (idle). Dies wird hier zunächst nicht berücksichtigt.

- Um ein gesamtes Parallelitätsprofil zu charakterisieren, wird der durchschnittliche Parallelitätsgrad p angegeben:

$$p = \frac{1}{i_m} \cdot \sum_{i=1}^{i_m} P(i) = \frac{A}{i_m}$$

p gibt somit den Parallelitätsgrad an für den Fall, daß alle Elementaroperationen über die Zeit i_m gleichverteilt werden könnten.

Beispiel 3-14:

Matrixmultiplikation mit $a = b = c = 16$

$$A = 16^2 (16 + 8 + 4 + 2 + 1) = 7936$$

$$i_m = 5$$

$$p = 1587,20$$

- Das Verhalten eines Programms läßt sich auch durch seine Parallelitätsverteilung $V(P)$ beschreiben. $V(P)$ gibt an, für welchen Bruchteil seiner Gesamtlaufzeit ein Programm (besser: ein Parallelitätsprofil $P(i)$) einen bestimmten Parallelitätsgrad P aufweist.

$$V(P) = \frac{\sum_{i=1}^{i_m} z(P(i), P')}{i_m} \quad , 0 \leq P \leq P_{\max}$$

$$\text{mit} \quad z(P(i), P') = \begin{cases} 1 & P(i) = P' \\ 0 & P(i) \neq P' \end{cases}$$

Wegen der Normierung gilt:

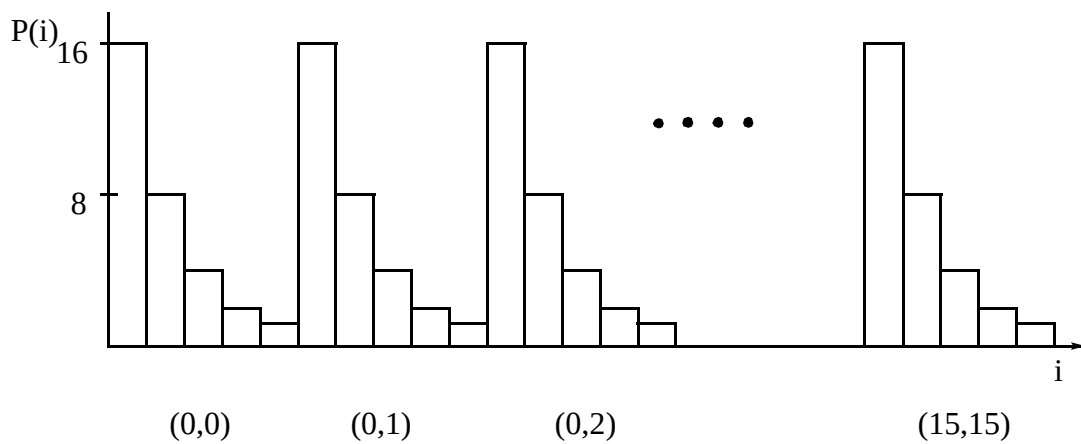
$$\sum_{P=0}^{P_{\max}} V(P) = 1$$

mit $P_{\max}$ = größter im Profil $P(i)$ vorkommender Parallelitätsgrad

Bei $P=0$ werden Zeiten berücksichtigt, in denen kein Prozessor arbeitet (z.B. wegen Kommunikationsverzögerungen).

Beispiel 3-15:

Matrixmultiplikation mit $a = b = c = 16$ wie vorher, aber ohne gleichzeitige Ausführung aller Matrixelementberechnungen



Die Arbeit berechnet sich aus $V(P)$:

$$A = i_m \cdot \sum_{P=0}^{P_{\max}} V(P) \cdot P$$

Wegen $p = \frac{A}{i_m}$ gilt:

$$p = \sum_{P=0}^{P_{\max}} V(P) \cdot P$$

- Das Bearbeitungsprofil $B(i)$ gibt an, wieviele Operationen des Auftragsprofils $P(i)$ maximal zum Zeitpunkt i ausgeführt werden können. $B(i)$ ist abhängig von den Architekturparametern. Wegen der Vielzahl unterschiedlicher Verlustmechanismen ist eine geschlossene Angabe von $B(i)$ i.A. nicht möglich. Zur Veranschaulichung wird hier $B(i)$ für den Fall einer Begrenzung der Prozessorzahl angegeben. Ohne Prozessorausfall ist $B(i) = \text{const} = N$ (N Prozessoren verfügbar).

Ausführbarkeit:

$\forall i: \quad P(i) \leq B(i) \quad \text{Ausführbarkeitsbedingung}$

Wenn Ausführbarkeitsbedingung nicht gegeben:

Umformung $P(i) \rightarrow P^*(i)$, so daß $P^*(i) \leq B(i)$

- Das resultierende Ausführungsparallelitätsprofil $P^*(i)$ liefert analoge Kenngrößen zu $P(i)$; sie werden mit $*$ gekennzeichnet.

$$A^* = \sum_{i=1}^{i_m^*} P^*(i)$$

Die Ausführungszeit verlängert sich von i_m auf i_m^* . Die Anzahl der auszuführenden Elementaroperationen verändert sich nicht, also ist

$$A^* = A.$$

Der durchschnittliche Parallelitätsgrad p^* sinkt mit wachsenden i_m^* :

$$p^* = \frac{A^*}{i_m^*} = \frac{A}{i_m^*}$$

Die Parallelitätsverteilung $V^*(P^*)$ läuft von $0 \leq P^* \leq P_{\max}^*$ mit $P_{\max}^* = N$.

- Die maximal ausgelastete Parallelmaschine könnte während i_m^*

$$A_{max} = \sum_{i=1}^{i_m^*} B(i) = N \cdot i_m^*$$

Elementaroperationen ausführen. A_{max} ist die Kapazität der Maschine.

- Die Effizienz ε ist definiert als

$$\varepsilon = \frac{\text{geleistete Arbeit}}{\text{Kapazität } A_{max}} \quad (\text{"Wirkungsgrad"})$$

$$\varepsilon = \frac{A}{A_{max}} = \frac{A}{N \cdot i_m^*} = \frac{p^*}{N}$$

- Der Speed-up S gibt an, um welchen Faktor die parallele Maschine schneller ist als eine sequentielle (für dasselbe Problem).

Notation: $T(m, N)$ = Zeit für Problemgröße m auf Maschine mit N Prozessoren.

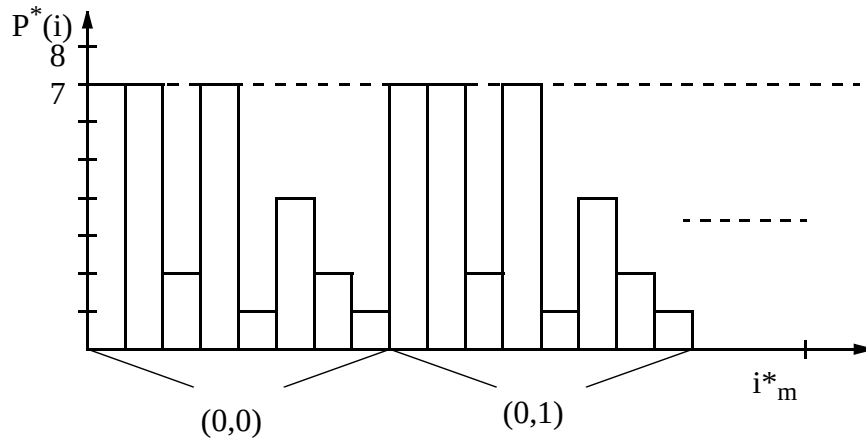
$$S = \frac{T(m, 1)}{T(m, N)} = \frac{A}{i_m^*} \quad (\equiv p^*)$$

Mit $\varepsilon = \frac{p^*}{N}$ folgt $\varepsilon = \frac{S}{N}$.

Beispiel 3-16:

Matrixmultiplikation ohne gleichzeitige Ausführung aller Matrixelementberechnungen mit $a = b = c = 16$

$$B(i) = N = 7$$



$$i^*_m = 256 \cdot 8 = 2^{11}$$

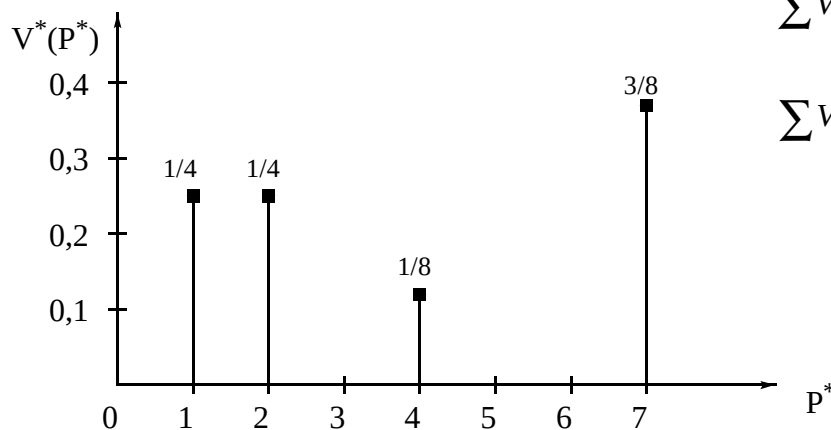
$$A = 256 \cdot 31 = 7936$$

$$p^* = \frac{7936}{256 \cdot 8} = 3,88; \quad S = p^* = 3.88$$

$$P^*_{\max} = 7$$

$$A_{\max} = N \cdot i^*_m = 7 \cdot 2^{11}$$

$$\varepsilon = \frac{3,88}{7} = 55\%$$



$$\sum V^* = 1$$

$$\sum V^* \cdot P^* = 3,88$$

Beispiel 3-17:

von Neumann-Maschine, belastet mit Arbeit A

$$N = 1$$

$$i_m^* = A$$

$$A_{\max} = i_m^* \cdot 1 = A$$

$$\varepsilon = 100\%$$

$$S = 1$$

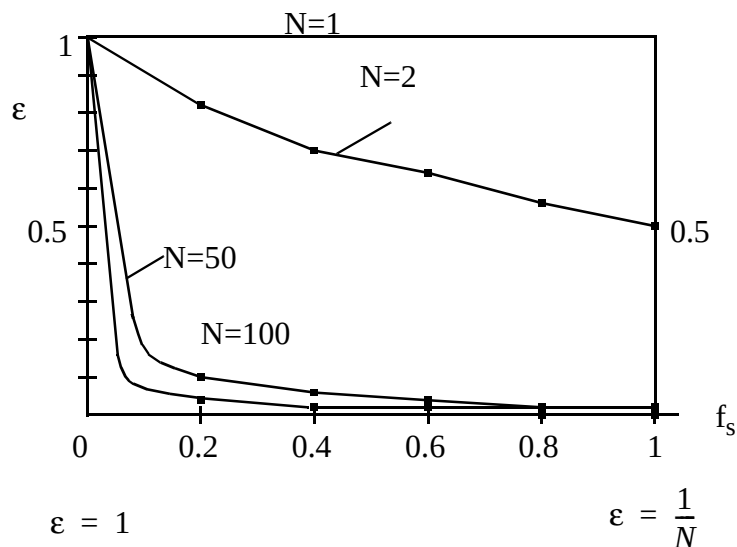
3.4.3. Gesetz von Amdahl

Annahme: Programme bestehen aus einem seriellen Anteil, der sich auf der parallelen Maschine nicht beschleunigen lässt, und einem parallelisierbaren Anteil, der mit N Prozessoren N mal schneller läuft.

$$S = \frac{T_s + T_p}{T_s + \frac{T_p}{N}} ;$$

mit $f_s = \frac{T_s}{T_s + T_p} ; \quad f_p = 1 - f_s$

$$\varepsilon = \frac{S}{N} = \frac{1}{1 + f_s(N-1)}$$



$$S = \frac{1}{f_s + \frac{f_p}{N}} ; \quad N \rightarrow \infty : \quad S \leq \frac{1}{f_s}$$

Beispiel 3-18:

n Cube-System $N=128$ Prozessoren

serieller Anteil von $f_S = 5/100$

$\varepsilon = 14\%$; $S = 18 < 100/5$

3.4.4. Speed-up vs. Scale-up**3.4.4.1 Idealisierte Annahmen**

Speed-up-Definition: Problemgröße bleibt gleich für sequentielle, kleine und große Parallelrechner. Aber: typischer ist die Anwendung großer Rechner für große Probleme, kleiner Rechner für kleine Probleme.

Beobachtung: f_S nimmt ab mit wachsender Problemgröße m (Gustafson, 1988).

Konsequenz: nicht das Zeitverhältnis für konstante Problemgröße, sondern das Verhältnis der Problemgrößen, die in konstanter Zeit gerechnet werden, ist der Vergleichsmaßstab.

Zeit für Monoprozessor und Problemgröße m_1 : $T(m_1, 1) = c_1 + a_1 \cdot m_1$

Zeit für N Prozessoren und Problemgröße m_N : $T(m_N, N) = c_N + a_N \cdot \frac{m_N}{N}$

Aus $T(m_1, 1) = T(m_N, N)$ folgt: $m_N = \frac{N}{a_N} \cdot [a_1 \cdot m_1 - (c_N - c_1)]$

Die in einer bestimmten Zeit bearbeitbare Problemgröße wächst proportional zur Prozessorzahl.

Der Scale-up $\frac{m_N}{m_1}$ ist linear in N .

Mit der Annahme $c_1 = c_N$ und $a_1 = a_N$ wird der Scale-up $\frac{m_N}{m_1} = N$.

3.4.4.2 Realistische Annahmen

Eine genauere Modellierung der Programmlaufzeit auf einem Parallelrechner ergibt drei Terme:

c_N : serielle Anteile, Vorbereitung, Nachbereitung

d_N : Zeitbedarf für die Verteilung von Programmen und Daten.

Baum, Hypercube: $d_N = b_N \cdot \log_2 N$

2-dim-Feld: $d_N = b_N \cdot \sqrt{N}$

Produktivteil: $\frac{a_N \cdot m_N}{N}$

Damit ist $T(m_N, N) = c_N + d_N + \frac{a_N \cdot m_N}{N}$

z.B. für Baum $T(m_N, N) = c_N + b_N \cdot \log_2 N + \frac{a_N \cdot m_N}{N}$

Der Scale-up ist dann mit gleichen seriellen Anteilen ($c_1 = c_N$) und gleicher Prozessorleistung

($a_1 = a_N = a$): $\frac{m_N}{m_1} = N \left(1 - \frac{b_N \cdot \log_2 N}{a \cdot m_1} \right)$

Der Scale-up wird negativ für $b_N \cdot \log_2 N > a \cdot m_1$, d.h. er ist nur sinnvoll, wenn die Verteilzeit für den Parallelrechner nicht größer ist als die Produktivzeit auf dem Monoprozessor.

Berücksichtigt man die Verteilzeiten T_V auf dem Parallelrechner auch beim Speed-up, so erhält man einen zusätzlichen Term im Nenner:

$$S = \frac{T_S + T_P}{T_S + T_{PV} + \frac{T_P}{N}},$$

d.h. auch ein vollständig parallelisierbares Programm mit $T_S = 0$ erreicht nur ein $S < N$.

3.4.5. Overhead

3.4.5.3 Definition

Das Verhalten eines Parallelrechners lässt sich darstellen als Belegungsschema.

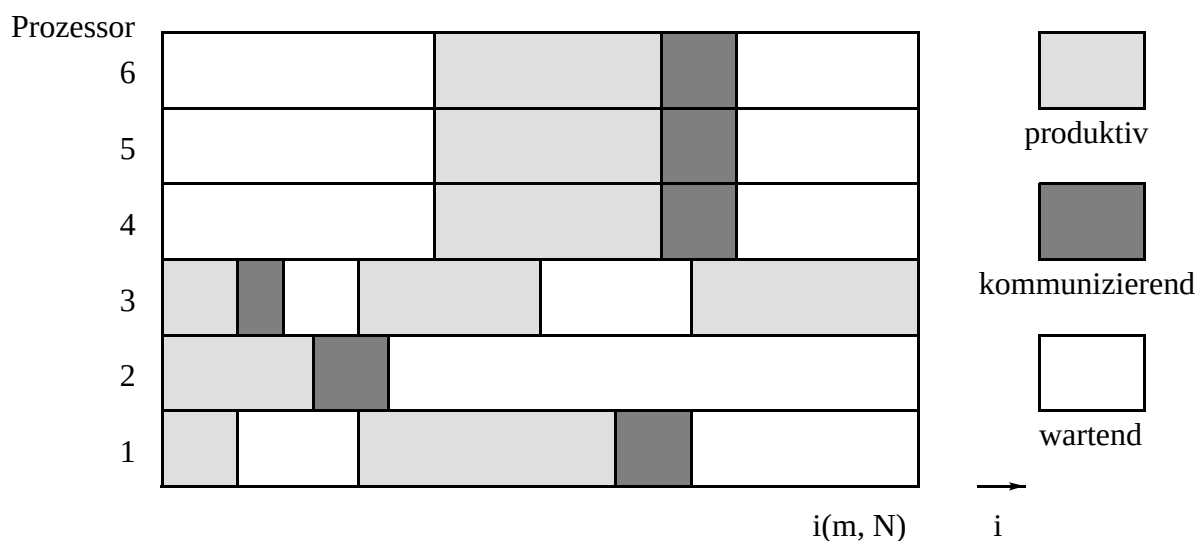


Bild 3.49. Belegungsschema für $N = 6$ Prozessoren

In das Belegungsschema wird für jeden Prozessor das Aktivitätsmuster eingetragen. Eine einfache Fallunterscheidung führt zu den Aktivitätstypen

- produktiv (p)
- kommunizierend (c)
- wartend (i = idle)

Jedem Aktivitätstyp kann die entsprechende Arbeit zugeordnet werden:

- Produktivarbeit A_p
- Kommunikationsarbeit A_c
- Warte-Arbeit A_i

Der Overhead ist dann die Summe aller Arbeitsanteile $\neq A_p$ (d.h. Kapazität - Produktivarbeit):

$$\text{Overhead} = i(m, N) \cdot N - A_p$$

wobei $i(m, N)$ = diskrete Bearbeitungszeit für ein Problem der Größe m auf N Prozessoren, gemessen in Vielfachen von Elementaroperationen.

Der relative Overhead μ ist der absolute Overhead in Bezug gesetzt zur Produktivarbeit A_p :

$$\mu = \frac{i(m, N) \cdot N - A_p}{A_p}$$

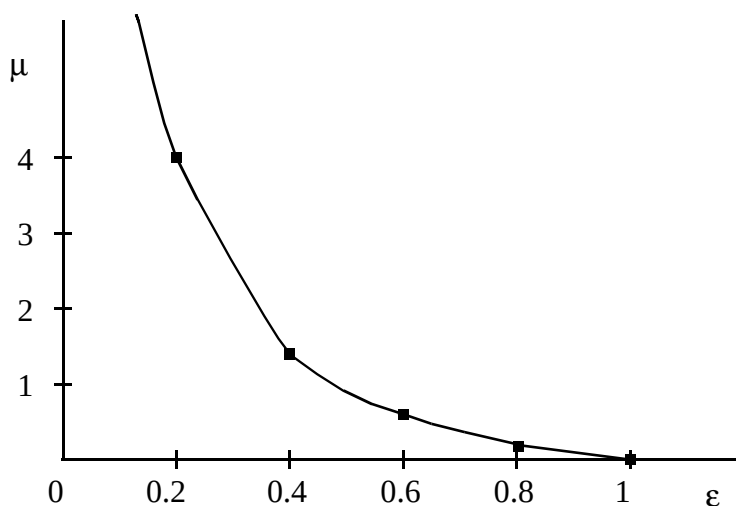
Verschiedene Verlustanteile addieren sich zum Gesamtoverhead.

$$A_p + A_c + A_i = i(m, N) \cdot N$$

$$\frac{A_c}{A_p} + \frac{A_i}{A_p} = \frac{i(m, N) \cdot N - A_p}{A_p}$$

$$\mu_c + \mu_i = \mu$$

$$\text{Wegen } \varepsilon = \frac{A_p}{A_{\max}} \text{ wird } \mu = \frac{1}{\varepsilon} - 1 .$$



3.4.5.4 Overhead-Berechnung

Beschränkung auf numerisches Beispiel, eindimensionale Wellengleichung.

N Prozessoren m Gitterpunkte

$\frac{m}{N}$ Korngröße

Algorithmus: Berechnung aller m Gitterpunkte, Kommunikation von 2 Randpunkten pro Korn

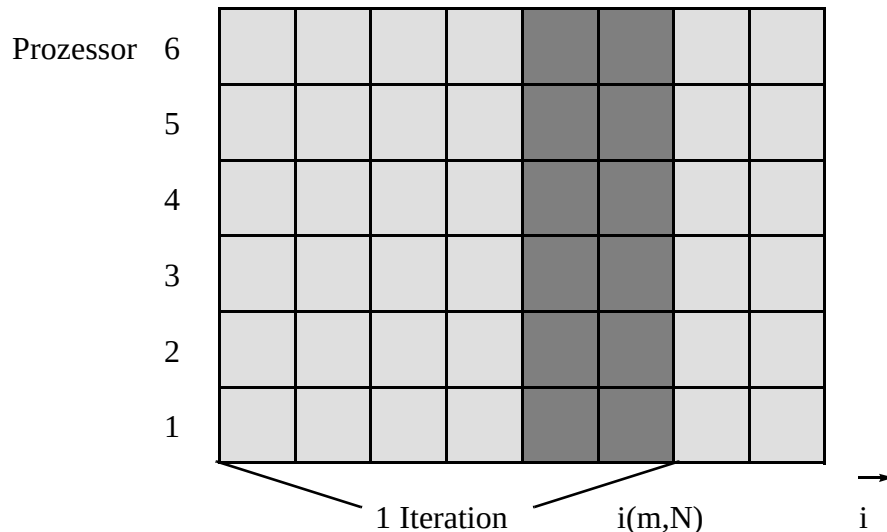


Bild 3.50. Belegungsschema für $N = 6$ und $m = 24$:

$$A_p = 24 \cdot t_{\text{calc}}; A_c = 12 \cdot t_{\text{com}}; \mu_c = 0.5 \cdot t_{\text{com}}/t_{\text{calc}}$$

- ohne Lastungleichgewicht

Die diskrete Zeit für die Berechnung eines Gitterpunkts sei t_{calc} , die Zeit für einen Kommunikationsvorgang t_{com} .

Im allgemeinen Fall ist $t_{\text{com}} \neq t_{\text{calc}}$.

$$\text{Damit wird } \mu = \mu_c = \frac{2 \cdot N \cdot t_{\text{com}}}{m \cdot t_{\text{calc}}} \quad \mu_i = 0$$

- mit Lastungleichgewicht

Ist $\frac{m}{N}$ nicht ganzzahlig, so entstehen Wartezeiten (A_i): weniger belastete Prozessoren müssen auf stärker belastete warten.

$$A_i = (N - (m \cdot \text{mod } N)) \left\lceil \frac{m \cdot \text{mod } N}{N} \right\rceil$$

(Die Ceiling-Funktion $\lceil x \rceil$ ergibt 0 für m/N ganzzahlig, sonst 1.)

$$\mu_i = \frac{N - m \cdot \text{mod } N}{m} \left\lceil \frac{m \cdot \text{mod } N}{N} \right\rceil$$

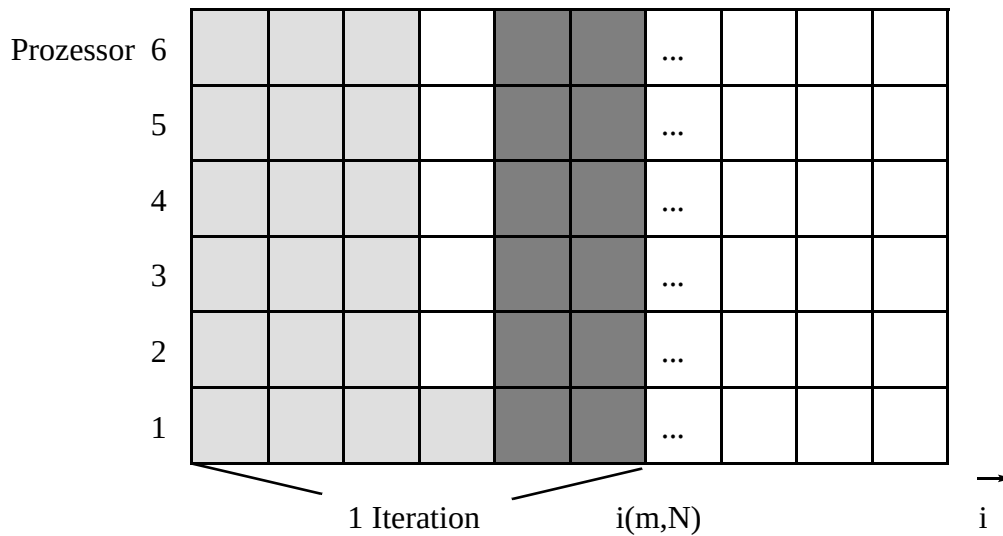


Bild 3.51. Belegungsschema für $N = 6$ und $m = 19$: $A_p = 19 \cdot t_{\text{calc}}$; $A_c = 12 \cdot t_{\text{com}}$;

$$A_i = 5 \cdot t_{\text{calc}}; \mu_i = 5/19; \mu_c = 12/19 \cdot t_{\text{com}}/t_{\text{calc}}; \text{für } t_{\text{calc}} = t_{\text{com}}: \varepsilon = 1/(\mu+1) = 19/36$$

- Effizienz als Funktion der Korngröße $\frac{m}{N}$
 Für die graphische Darstellung von $\varepsilon\left(\frac{m}{N}\right)$ werden 2 Grenzwerte für $\frac{t_{\text{com}}}{t_{\text{calc}}}$ angenommen
 (0,1 und 10). Typischerweise liegt dieses Verhältnis bei 1.

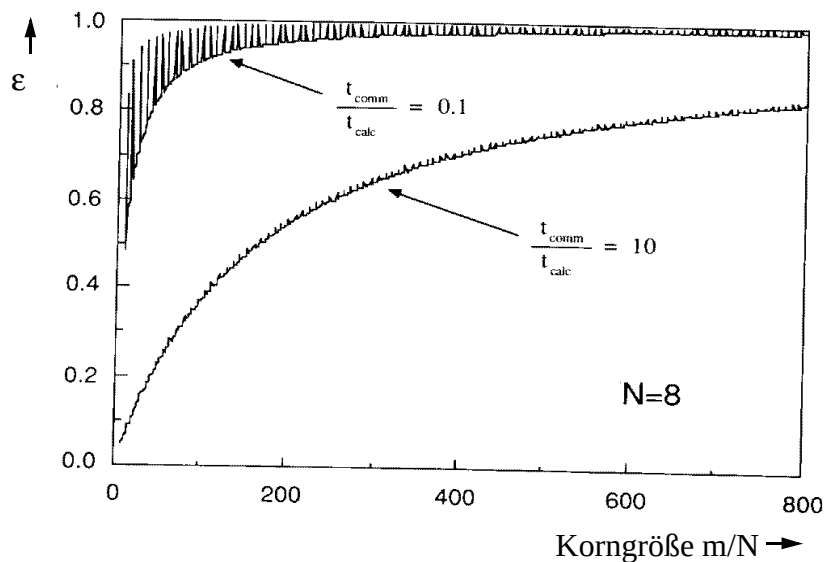


Bild 3.52. Effizienz ε als Funktion der Korngröße

- Messung von $\frac{t_{com}}{t_{calc}}$
- Bestimmung von μ ohne Lastungleichgewicht für verschiedene Werte von $\frac{m}{N}$

$$\mu = \frac{2 \cdot t_{com}}{t_{calc}} \cdot \frac{N}{m}$$

Steigung der Geraden $\mu \left(\frac{N}{m} \right) \rightarrow \frac{2 \cdot t_{com}}{t_{calc}}$

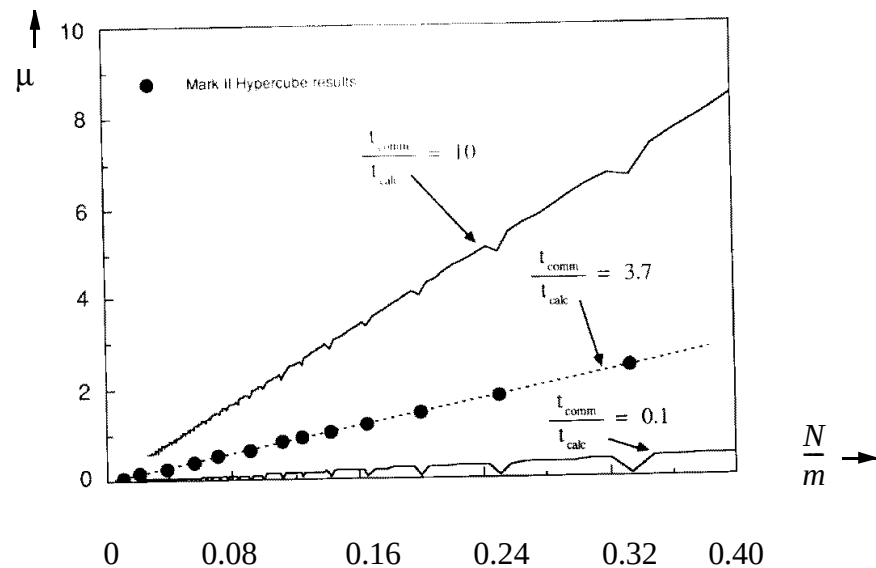


Bild 3.53. Berechnete (-) und gemessene (i) Werte des Overheads μ als Funktion der inversen Korngröße N/m .

Die bisherigen Betrachtungen gehen von einem Wechsel zwischen Produktivarbeit und Kommunikationsphasen aus. Dies entspricht aber dem Worst Case. In Wirklichkeit können Kommunikationszeiten und Wartezeiten zum Teil mit Produktivarbeit gefüllt werden. Eine Analyse des Latency Hiding wird im folgenden Kapitel gegeben.

3.4.6. Speed-up mit Latenzzeit

Die folgenden Abschätzung von unteren und oberen Schranken für Speed-up und Effizienz folgt einem Vorschlag von M. Dertouzos (MIT).

Die Abschätzung berücksichtigt

- nicht überlappende Kommunikationszeiten
- überlappende Kommunikationszeiten
- Lastungleichgewicht

Für die Kommunikationszeit t_k wird im folgenden die dimensionslose Größe L (= Latenzzeit, in Vielfachen einer Taktzeit t_0) verwendet. Ausgangspunkt ist ein gegebenes Parallelitätsprofil $P(i)$ sowie das zugeordnete ausführbare Parallelitätsprofil $P^*(i)$.

3.4.6.1 Untere Schranke (Worst Case)

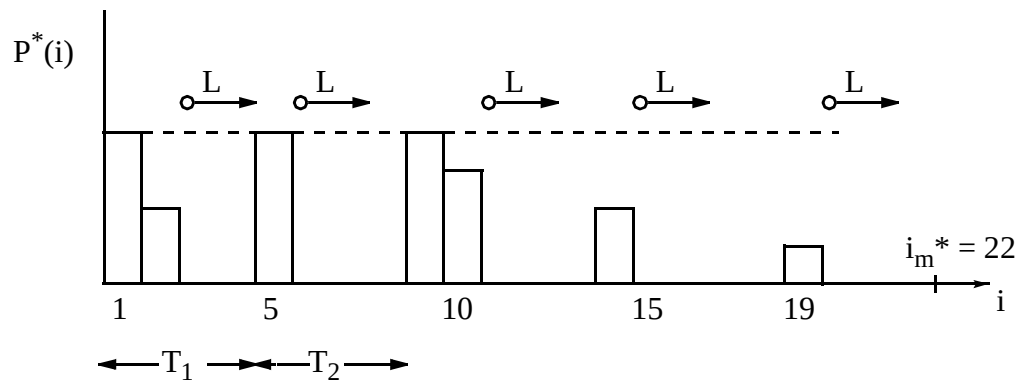
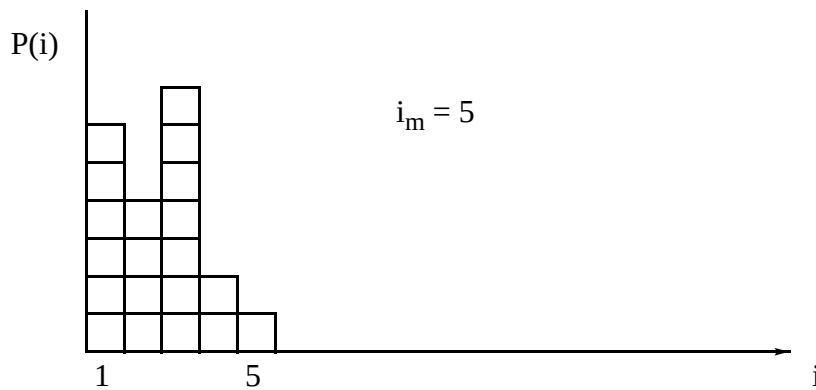


Bild 3.54. Umwandlung eines Parallelitätsprofils $P(i)$ in ein ausführbares ($N = 4$) Profil $P^*(i)$ mit Latenz $L = 3$

$$T_i \leq \left\lceil \frac{P(i)}{N} \right\rceil + L$$

$$i_m^* = \sum_{i=1}^{i_m} T_i$$

$$i_m^* \leq i_m \cdot L + \sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil$$

$$\Delta T_i \geq 0 : P(i) \leq l \cdot N$$

$$i_m^* < i_m \cdot (L + 1) + \frac{\sum_{i=1}^{i_m} P(i)}{N} \quad \left(da \quad \frac{a}{b} + 1 > \left\lceil \frac{a}{b} \right\rceil \right)$$

$$i_m^* < i_m \cdot (L + 1) + \frac{A}{N}$$

mit $S = \frac{A}{i_m^*}$; $p = \frac{A}{i_m}$; $l = L + 1$ wird

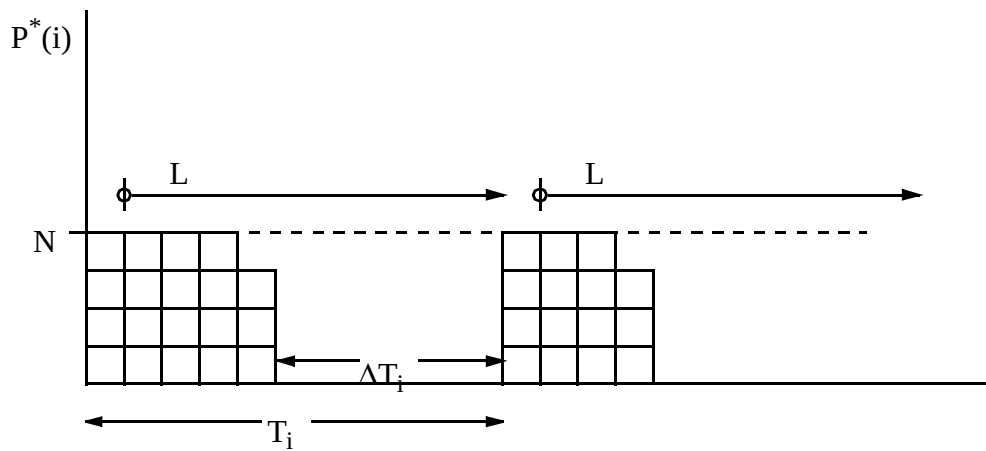
$$S > \frac{N}{1 + \frac{l}{p} \cdot N}$$

Gleichung 1

$$l \geq 1$$

3.4.6.2 Obere Schranke (Best Case)

Annahme: Nur die erste "Scheibe" von N parallelen Operationen führt zu Kommunikation (mit Latenzzeit L).



$$T_i = \max\left(L + 1, \left\lceil \frac{P(i)}{N} \right\rceil\right)$$

$$T_i = \left\lceil \frac{P(i)}{N} \right\rceil + \max(0, \Delta T_i) \quad \text{mit } \Delta T_i = l - \left\lceil \frac{P(i)}{N} \right\rceil$$

$$i_m^* = \sum_{i=1}^{i_m} T_i = \sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil + \sum_{i=1}^{i_m} (\max(0, \Delta T_i))$$

D.h. nur positive Werte von ΔT_i liefern einen Beitrag.

Häufigkeit $D(P)$ des Parallelitätsgrades $P(i)$ gegeben durch Parallelitätsverteilung $V(P)$ multipliziert mit der Laufzeit i_m :

$$D(P) = i_m \cdot V(P)$$

Summation über $D(P) \cdot \Delta T_i$ von $P = 1$ bis $P = lN$.

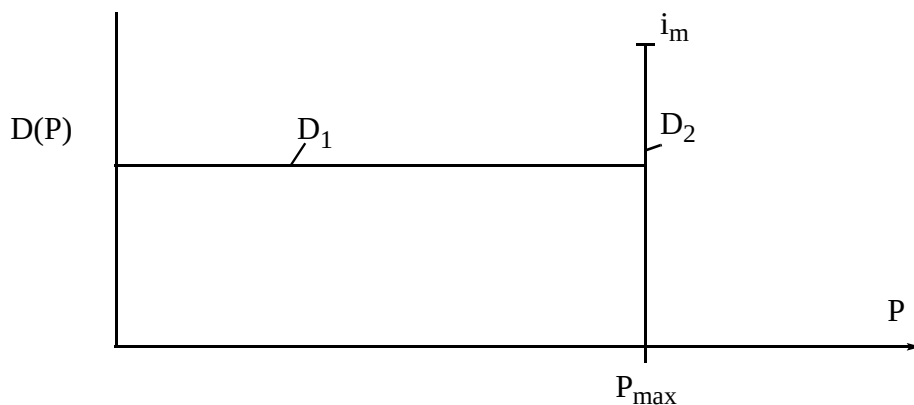
$$i_m^* = \sum_{i=1}^{i_m} \left\lceil \frac{P(i)}{N} \right\rceil + \sum_{P=1}^{lN} \left[D(P) \cdot \left(l - \left\lceil \frac{P(i)}{N} \right\rceil \right) \right]$$

$$i_m^* \geq \sum_{i=1}^{i_m} \frac{P(i)}{N} + l \sum_{P=1}^{lN} D(P) - \frac{1}{N} \sum_{P=1}^{lN} D(P) \cdot P \quad (\text{da } x \leq \lceil x \rceil)$$

$$i_m^* \geq \frac{A}{N} + l \sum_{P=1}^{lN} D(P) - \frac{1}{N} \sum_{P=1}^{lN} D(P) \cdot P$$

Gleichung 2

Einschub: Vergleich unterschiedlicher Verteilungsfunktionen



$$\sum_{P=1}^{P_{\max}} D(P) = i_m$$

$$\sum_{P=1}^{P_{\max}} P \cdot D(P) = A$$

- Verteilungsfunktion D_1 :

Jeder Parallelitätsgrad $P = 1, 2, \dots, P_{\max}$ kommt $\frac{i_m}{P_{\max}}$ mal vor: $D_1(P) = \frac{i_m}{P_{\max}}$

$$A_1 = \sum_{P=1}^{P_{\max}} P \cdot \frac{i_m}{P_{\max}} = (P_{\max} + 1) \cdot \frac{i_m}{2}$$

$$p_1 = \frac{P_{\max} + 1}{2}$$

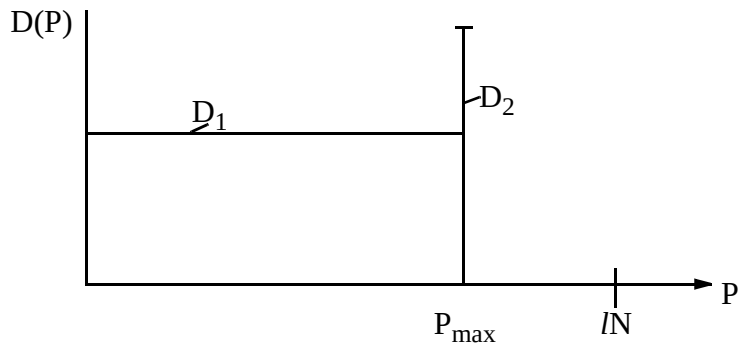
$$p_1 \approx \frac{P_{\max}}{2} \quad (P_{\max} \gg 1)$$

- Verteilungsfunktion D_2 :
Alle i_m Schritte haben den Parallelitätsgrad $P = P_{\max}$.
 $A_2 = P_{\max} \cdot i_m$; $p_2 = P_{\max}$
- Vergleich: $A_2 - A_1 = \frac{i_m}{2} \cdot (P_{\max} - 1) \geq 0$
Es gilt allgemein für festes i_m und $P_{\max} > 1$: $A_2 > A_1$ für beliebige $D_i(P)$.

Ende Einschub

Fallunterscheidung

Ⓐ $P_{\max} \leq lN$

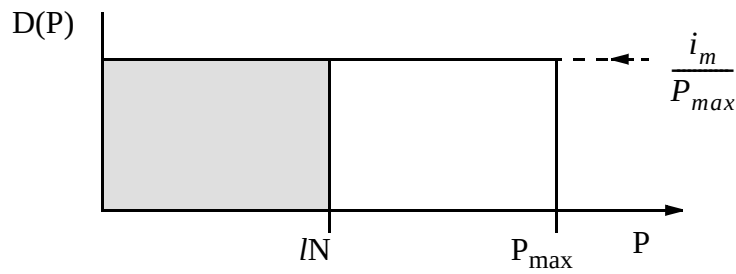


mit Gleichung 2.: $i_m^* \geq \frac{A}{N} + l \cdot i_m - \frac{A}{N}$

$$i_m^* \geq (l \cdot i_m)$$

$S_1 \leq \frac{p}{l}$	für $p \leq \frac{lN+1}{2}$ (Vert. 1)
2	für $p \leq lN$ (Vert. 2)

Ⓑ $P_{\max} > lN$ und Verteilung D_1



$$\sum_{P=1}^{lN} D(P) = \frac{i_m}{P_{\max}} \cdot lN$$

$$\sum_{P=1}^{lN} P \cdot D(P) = \frac{i_m}{P_{\max}} \cdot lN \cdot \frac{lN+1}{2}$$

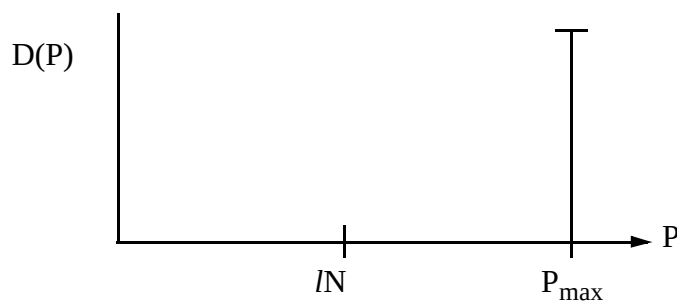
in Gleichung 2.: $i_m^* \geq \frac{A}{N} + l \cdot \frac{lN \cdot i_m}{P_{\max}} - \frac{1}{N} \cdot \frac{i_m}{P_{\max}} \cdot lN \cdot \frac{lN+1}{2}$

mit $P_{\max} \approx 2p$:
$$i_m^* \geq \frac{P \cdot i_m}{N} + \frac{l^2 \cdot N \cdot i_m}{2p} - \frac{i_m \cdot l \cdot (lN+1)}{4p}$$

$$S = \frac{p \cdot i_m}{i_m^*} : \quad S \leq \frac{N}{1 + \frac{lN(lN-1)}{4p^2}}$$

$S \leq \frac{N}{1 + \left(\frac{lN}{2p}\right)^2}$	für $p > \frac{lN}{2}$ für Verteilung 1 für $lN \gg 1$
---	--

Ⓒ $P_{\max} > lN$ und Verteilung D_2



$$\sum D(P) = 0$$

$$\sum P \cdot D(P) = 0$$

in Gleichung 2.: $i_m^* \geq \frac{A}{N}$

$$S_2 \leq N \quad \begin{array}{l} \text{für } p > lN \\ \text{für Verteilung 2} \end{array}$$

3.4.6.3 Diskussion

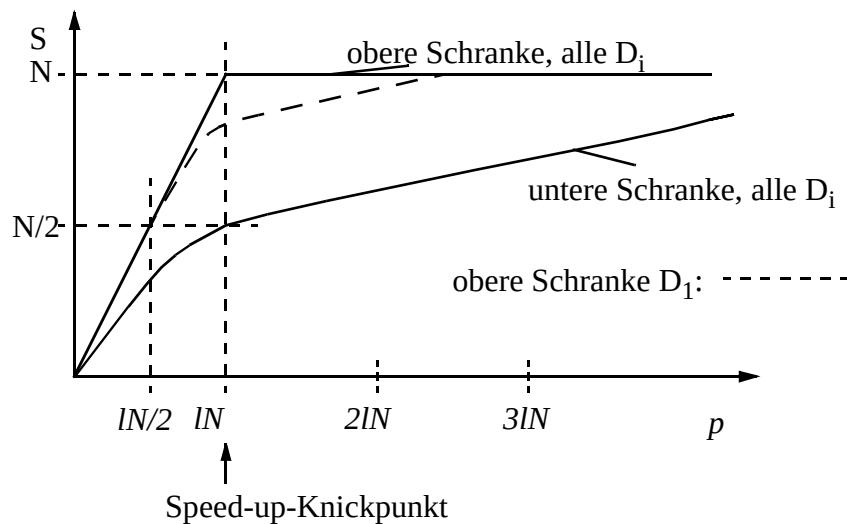
Zusammenfassung:

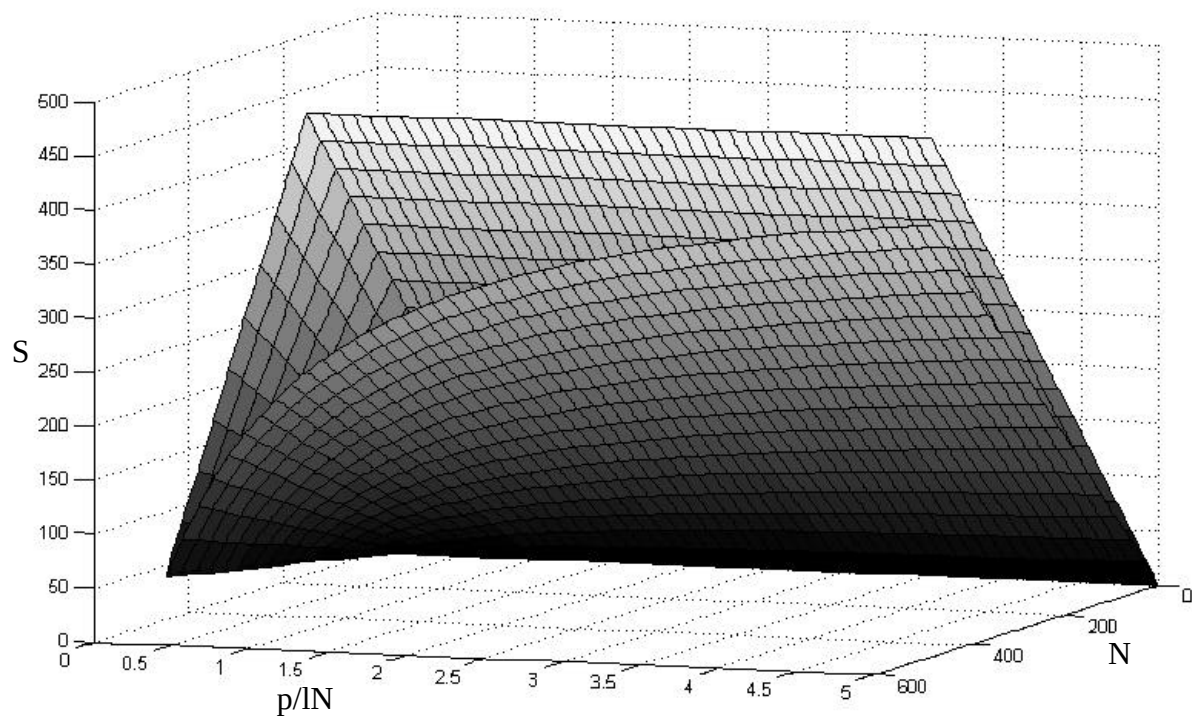
für Verteilung D_1 :

$$\frac{N}{1 + \frac{lN}{p}} < S_1 \leq \begin{cases} \frac{p}{l} & p \leq \frac{lN+1}{2} \\ \frac{N}{1 + \left(\frac{lN}{2p}\right)^2} & p > \frac{lN+1}{2} \end{cases}$$

für beliebige Verteilungen:

$$\frac{N}{1 + \frac{lN}{p}} < S_2 \leq \begin{cases} \frac{p}{l} & p \leq lN \\ N & p > lN \end{cases}$$





Der Punkt $p = lN$ ist charakteristisch für die Anpassung zwischen Workload (beschrieben durch p) und Architektur (beschrieben durch lN). $p = lN$ heißt Speed-up-Knickpunkt. Für $p > lN$ wird eine Effizienz $\varepsilon > 50\%$ erreicht.

Obere Schranke (best case): für $p > lN$ kann die Latenzzeit voll hinter den Produktivoperationen versteckt werden (*latency hiding*).

Schranken für den Overhead μ : Aus $\mu = \frac{1}{\varepsilon} - 1$ folgt (für beliebige Verteilungen):

$$\left. \begin{array}{l} (p \leq lN:) \\ (p > N:) \end{array} \right\} \begin{array}{l} \frac{lN}{p} - 1 \\ 0 \end{array} \leq \mu \leq \frac{lN}{P}$$

Interpretation:

$$\begin{aligned} A_p &= \frac{p}{N} \\ A_c &= l \end{aligned}$$

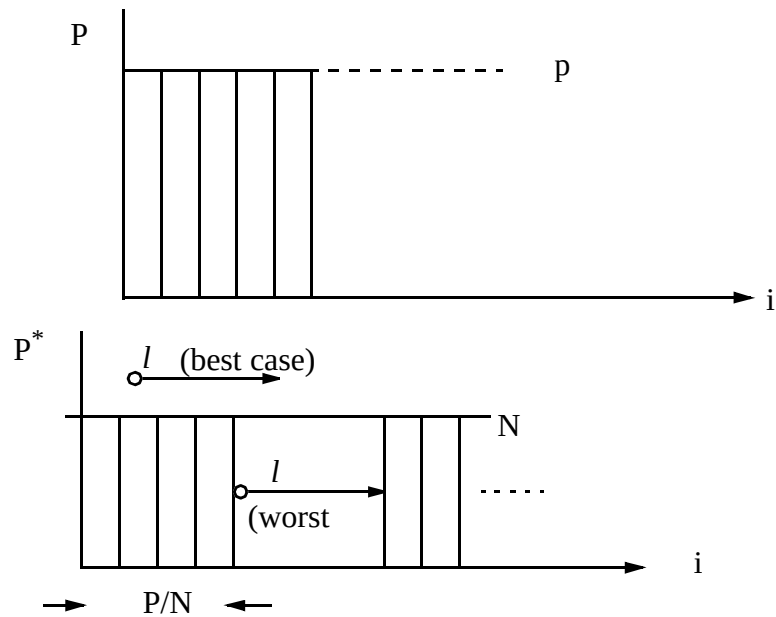


Bild 3.55. Produktivarbeit während p/N , Wartezeit während l .

Abschätzung:

nCube 64: Latenzzeit (HW) = $10 \mu\text{s}$

$f_c = 20 \text{ MHz}$

$l = 200$

$lN = 12800$

Latenzzeit (SW) = $150 \mu\text{s}$

$l = 3000$

$lN = 192000$

3.5. Parallelrechnerarchitekturen am Beispiel

3.5.1. SIMD Parallel

3.5.1.1 ILLIAC IV Projekt-Start 1966, University of Illinois

Anwendung: Matrixoperationen, PDGL (CFD, NASA)

→ HW- Struktur: Gitter

- Global-Architektur
 - 64 Processing Units (PUs)
 - $PU = PE + PM$
 - 1 Control Unit CU, 1 Host (Supervisory Computer System)
 - 2 Verbindungsnetzwerke:
 - $CU \leftrightarrow PUs$ (Stern)
 - $PU \leftrightarrow PU$ (2-dim. Gitter)

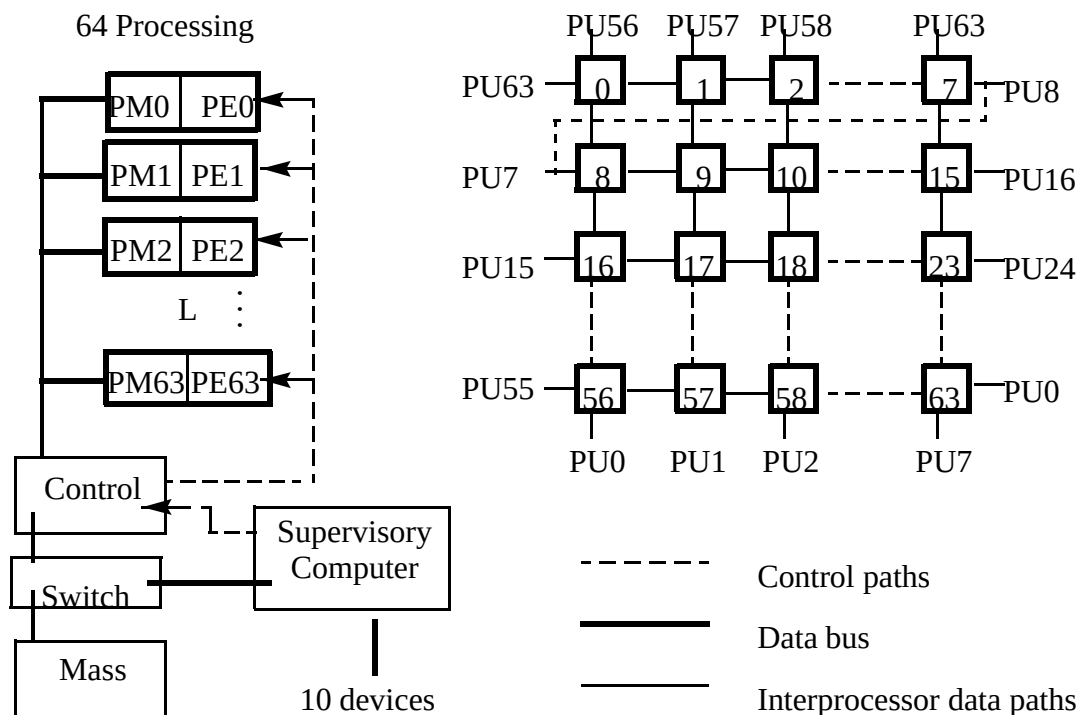


Bild 3.56. HW-Aufbau der ILLIAC IV

a) sternförmiges VN von der CU zu den PUs b) 2-dim Gitter

- Knoten-Architektur
 - PE: 64 bit-Verarbeitung
 - Multiplikation: 400 ns
 - Addition: 240 ns
 - Befehlssatz: wie üblich, zusätzlich
 - Kommunikation mit 4 Nachbarn
 - Mode-Register setzen (enable/disable)

Aufbau: 210 Leiterplatten pro PE, je 20 DIPs mit je 7 ECL-Gattern + diskrete Elemente.

PM: 2K x 64 bit-Worte (16 KB)
4 Leiterplatten pro PM,
je 128 x 256 bit-Speicher

- Massenspeicher:

Trommelspeicher, 128 Köpfe (par.) 10^9 bit
Transferrate: 62 MB /s
Zugriffszeit: 20 ms
→ 16 ms zum Laden aller PMs

Arbeitsweise: CU schickt Instruktionen an alle (broadcast). Ausblenden einzelner PUs über Mode-Register.

Performance: 64 bit-FP-Vektoroperationen: 4 MFLOPS (Peak: $64/240 \text{ ns} \approx 260 \text{ MFLOPS}$ Add.)

Software: rudimentäres BS (I/O-Monitoring, Transfer Massenspeicher $\leftrightarrow$ PM)

Hochsprachen:

1. GLYPNIR: ALGOL-ähnlich, zusätzlich 64bit-Worte, FORALL
2. CFD: Fortran - ähnlich, mehr Detailwissen über HW-Arch. nötig
3. IV TRAN: Vorläufer parallelisierender Compiler

Bedeutung:

- HW-first-Projekt: SW-Defizite
- Technologie-Treiber: erster Großeinsatz von ECL ICs, PCB-Design automatisch, erster Großeinsatz von Halbleiterspeichern, erster Einsatz von großen Multilayer (12 Ebenen, $40 \times 50 \text{ cm}^2$) - Boards
- keine Redundanz: anfällig
- Vollausbau (256 PUs) nicht erreicht.

3.5.1.2 Connection Machine (D.Hillis)

Thinking Machine Corporation, 1986 CM-1, 1988 CM-2

- Anwendung: AI, CFD, Datenbanksuche "Symbol Cruncher", nicht "Number Cruncher". Paralleler Mustervergleich, zunächst ohne FP-Einheit
- Global-Architektur

Sehr viele (2^{16}) sehr einfache (bit-serielle) PEs.

Ansichten:

- großer Speicher, darin viele PEs verteilt ($2^{16} \cdot 4 \text{ Kbit} = 32 \text{ MByte}$)
- höchstmögliche Parallelität, nur durch Technologie begrenzt

- Nachbildung des menschlichen Gehirns, Information gespeichert in den Verbindungen (*connections*), PE langsam.

PEs in Clustern à 16 (1 Chip), 1 Router pro Cluster

VN1: Verbindung der Router über 12-dim. Hypercube, packet-switched

VN2: 2-dim. Gitter (North, East, West, South: NEWS)

VN3: Daisy Chain zwischen PEs eines Clusters

Steuerung: Microcontroller übersetzt *macroinstructions* (vom Host) in *nanoinstructions* (ausführbar von den PEs). Übersetzung zur Laufzeit.

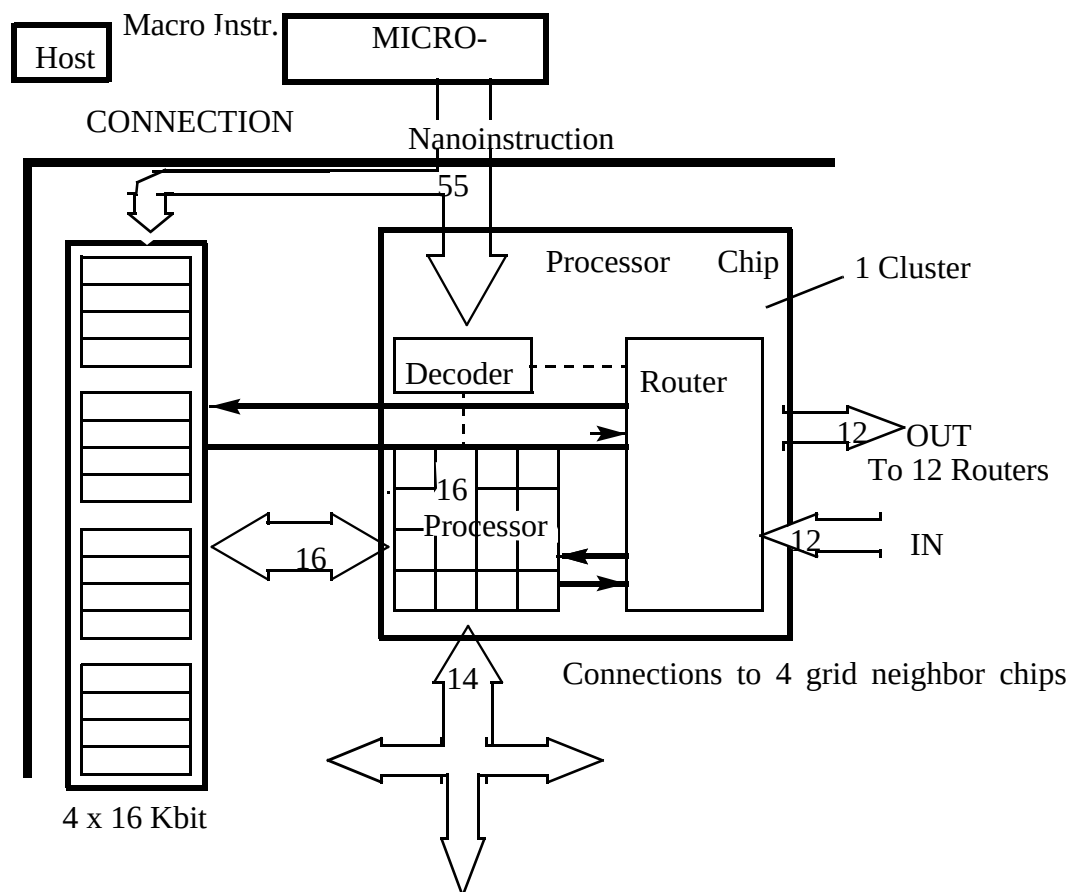


Bild 3.57. Global-Architektur der Connection Machine CM-1

- Knoten-Architektur

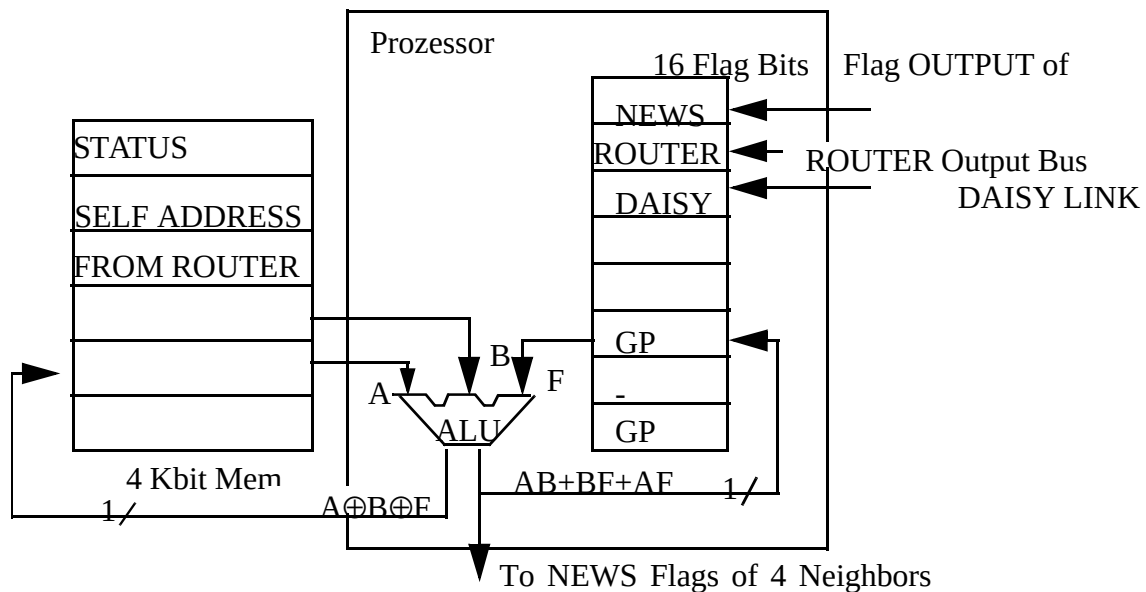


Bild 3.58. Aufbau eines PE der CM-1

PEs sind sehr einfach: 16 PEs auf 1 cm² CMOS-Chip (2µm), incl. Router und nanoinstruction-Decoder ("Cluster"). Pro Cluster: 4 Speicher-Chips à 16 Kbit (4 Kbit pro PE).

PE besteht aus: ALU (3-input, 2-output), 16 Flag-Bits (je 1 bit)

Komplexität der ALU: 2 x 1 bit-Volladdierer

ALU-Operation:

- für je 1 bit von Speicherplatz A und B sowie F (Flag-Bit).
- Verknüpfte A,B,F
- Schreibe Ergebnis nach A und Ziel-Flag (Fdest).

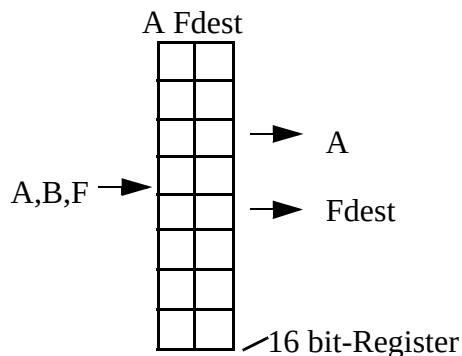
Adressen von A,B,F, Fdest in nanoinstruction.

Beispiele für ALU-Operationen:

	Memory Output:	Flag Output:
ADD	$A \leftarrow \text{XOR}(A,B,F)$	$F_{\text{dest}} \leftarrow \text{MAJORITY}(A,B,F)$
AND	$A \leftarrow \text{AND}(A,B,F)$	$F_{\text{dest}} \leftarrow \text{AND}(A,B,F)$
OR	$A \leftarrow \text{OR}(A,B,F)$	$F_{\text{dest}} \leftarrow \text{OR}(A,B,F)$
MOVE	$A \leftarrow B$	$F_{\text{dest}} \leftarrow \text{AND}(\text{XOR}(A,B),F)$
SWAP (CROSS)	$A \leftarrow F$	$F \leftarrow \text{dest } B$

Eingänge können invertiert werden

ALU-Implementierung durch Funktionstabelle:



Inhalt des ALU-Registers (= Opcode) definiert die auszuführende ALU-Operation, wird mit der nanoinstruction verschickt (2^{16} Kombinationen, aber die meisten unbenutzt).

Aufbau einer nanoinstruction (55 bit):

- ALU-Funktionstabelle (16 bit)
- A- und B-Adressen (2 x 12 bit)
- F-Adresse (read) (4 bit)
- Fdest-Adresse (write) (4 bit)
- Condition Flag (4 bit)

PEs können, abhängig von den lokalen Flags, abweichend reagieren, z.B. eine (globale) Operation nicht ausführen

- Condition Sense (1 bit)

Soll Condition auf "1" oder "0" abfragen?

- NEWS-Direction (2 bit)

Datentransferrichtung über 2-dim. Gitter

Nutzung der Flag-Bits:

- Carry für ADD
- NEWS-Flag: empfängt ALU-Fdest von Nachbar-PE (read only). Richtung spezifiziert durch NEWS-Direction.
- Router-Data-Flag: Datenübertragung zum/vom Router
- Router-Acknowledge-Flag: Router signalisiert dem PE, daß er ein Bit akzeptiert.
- Daisy Chain Flag: dient zur Auflösung von Konflikten beim Message Routing.
- Global Flag: NOR-Verknüpfung aller Global Flags eines Chips dient zur Kommunikation mit dem Host.

PE-Performance: 1 bit-Addition: 750 ns

→ Peak-Performance (32 bit-Additionen):

$$\frac{2^{16}}{32 \cdot 750ns} = 2.7 \cdot 10^{10} OPS$$

750 ns = 1 Instruktionszyklus

- Kommunikation

schnell: über 2-dim Gitter zu den direkten Nachbarn

langsam: über 12-dim. Hypercube

2 Modi für Hypercube-Routing:

1. Store- and- Forward: Übertragungszeit ~ Nachrichtenlänge * Anzahl der Übertragungsstufen

2. Wormhole: Übertragungszeit ~ Nachrichtenlänge + Anzahl der Übertragungsstufen

Nachteil des Wormhole-Routing auf CM-1: Router müssen umgangen werden, Anwender ist verantwortlich für Übertragung.

Ablauf einer Store- and Forward-Übertragung:

Message-Länge: ca. 65 bit (32 Datenbits + Zieladresse + Senderadresse)

Übertragung eines Bits: 1 Instruktionszyklus (750 ns)

Übertragung einer Message über s Stufen: $s \cdot 65 \cdot \text{Instruktionszyklusdauer}$ ($s = 12 : 585 \mu s$)

Evtl. Verzögerungen durch Routing-Konflikte zusätzlich, dadurch sind Verzögerungen im ms-Bereich möglich.

Die Message-Übertragung erfolgt synchron durch alle PEs (einige u.U. disabled durch Flags); während der Übertragungsphase ist keine Verarbeitung im PE möglich.

”Communication dominates computation”.

”Communication is computation.”

Komplikation bei Puffer-Überlauf: Messages werden dann zu Nachbar-Routern geschickt, selbst wenn ihre Zieldistanz dadurch größer wird (*desparation routing*) → Hot Spots?

- Software

Erweiterungen für C und Lisp für parallele Datenstrukturen, Operationen auf Xektoren (auf mehrere PEs verteilte Vektoren).

- CM-2

Erweiterung durch eine Fp-Einheit pro 32 PEs. Bemerkenswerte Abweichung vom Konzept der einfachen PEs!

- CM-5 → MIMD !

3.5.2. MIMD Shared Memory

3.5.2.3 CMU-Maschinen

- C.mmp (Carnegie Mellon multi mini processors)
Projekt-Beginn 1971 an der Carnegie Mellon University (CMU).

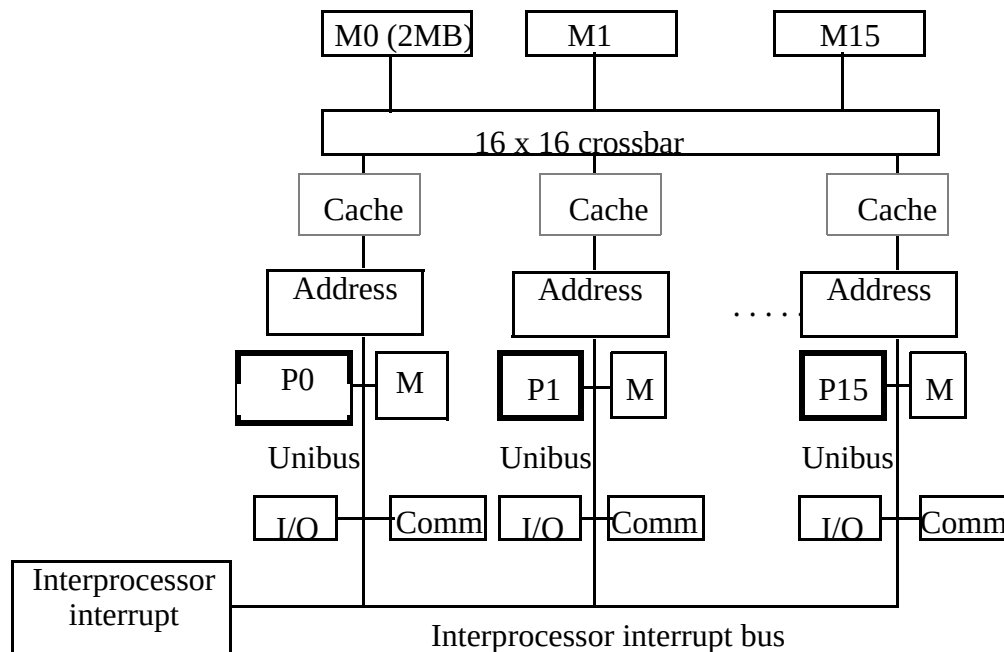


Bild 3.59. Globalarchitektur der C.mmp

Globalarchitektur:

16 DEC PDP-11 mit jeweils lokaler Peripherie (I/O, Kommunikation),

VN: 16 x 16 Crossbar

16 Speichermodule (je 2 MB) als shared Memory.

Lokaler Speicher: 8 KB pro PE.

Caches wurden entwickelt, aber nicht eingesetzt (Kohärenzprobleme). Adress translator nötig, da Adreßraum der PDP11 (64 KB) < 32 MB.

Crossbar erlaubt 16 simultane Speicherzugriffe zu unterschiedlichen Speichermodulen. Dauer eines Speicherzugriffs: 1 ms.

Zum Vergleich: Befehlsausführungszeit der PDP 11 : 2.5 ms.

Peripheriegeräte sind einzelnen PEs zugeordnet, können daher von anderen nicht (direkt) benutzt werden. Abhilfe durch Hydra-BS.

Bedeutung:

- Erfahrungen mit HW-Ausfällen führte zu Watchdog - System.
- Linearer Speedup erreicht für spezielle Anwendungen.

- Ernsthafte Einschränkung: Adreßraum der PDP 11 zu klein.
- Notwendigkeit für Debugging und Checkpointing erkannt.
- Cm* Nachfolger der C.mmp, 1978 mit 50 PEs realisiert.

Globalarchitektur

Grundbaustein: Prozessor-Speicher-Block (Computer Module Cm), physikalisch verteilter, logisch gemeinsamer Speicher

Bus-orientiertes VN mit 3 Stufen:

1. Lokaler Speicherbus PE - M/IO
2. Map Bus zur Verbindung von bis zu 14 Cm's, gesteuert durch einen K.map-Controller (1 Cluster).
3. Mehrere Inter-Cluster-Busse

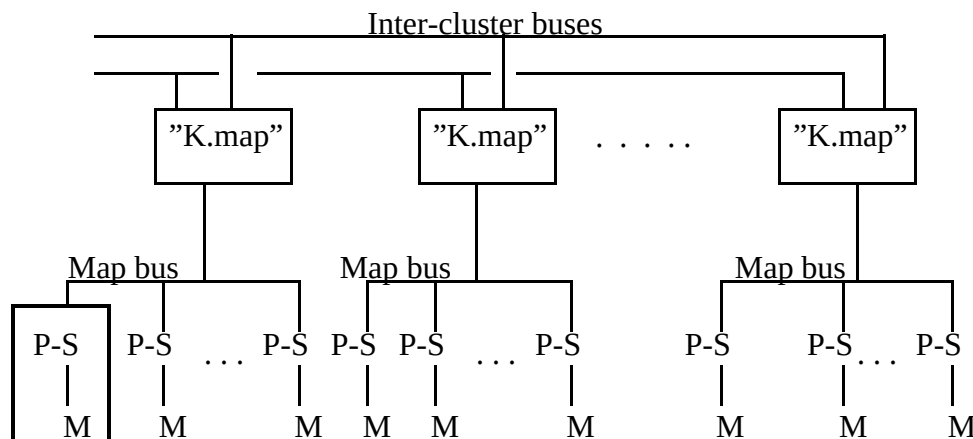


Bild 3.60. Globalarchitektur der Cm* . P = Prozessor, S = switch, M = memory (bis zu 256 KB)

Speicherzugriffszeiten abhängig von der Distanz: Zugriffszeit Lokaler Bus: Map Bus: Inter Cluster-Bus = 1 : 3 : 9

Rekonfigurierbarkeit: *milliday replugging*

Prozessor: DEC LSI-11

Speicherorganisation: logisch gemeinsamer Adreßraum 2^{28} Byte, direkter Zugriff durch jedes PE möglich.

Zusätzlich: Message Passing möglich.

Vorteil: Nutzung der Zugriffslokalität, dadurch Entlastung der höheren Bushierarchien.

K.map-Controller: nicht nur Adreßübersetzung, sondern auch Überwachung von wechselseitigem Ausschluß (*mutual exclusion*) beim Zugriff zu gemeinsamen Daten.

Performance: abhängig von der Verteilung von Code und Daten.

Code lokal, Daten entfernt: keine Sättigung durch K.map-Controller

Code und Daten entfernt: Sättigung bei 6 Cm's

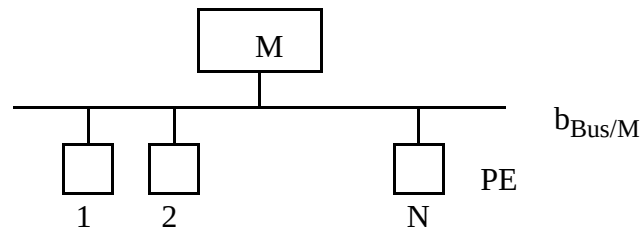
Software:

Sprache: Algol 68 mit *eventual values* für *lazy evaluation*, *fork-and-join*-Konstrukte

BS: Verteilter Scheduler weist Prozesse (*activities*) den verfügbaren Cm's zu; Prozesse bleiben dort bis zu Blockierung oder Ende; genau 1 Cm bleibt "Eigentümer" eines Prozesses. Benutzeroberfläche rudimentär. Nur wenige Benutzer.

3.5.2.4 NYU Ultracomputer

- Grenzen von Shared Memory Systemen (Abschätzung)



N PEs, Performance jeweils P, 1,3 Worte pro Befehl von M

nötige Übertragungsrate zur Versorgung eines PE: $b_{PE} = 1.3 \cdot 4 \cdot \frac{B}{instr} \cdot P$

gesamte Übertragungsrate: $N \cdot b_{PE} \leq b_{\frac{Bus}{M}}$

mit $b_{\frac{Bus}{M}} = 40 \frac{MB}{s}$ und $P = 20$ MIPS

$N \leq 0.4$ ohne Cache!

mit Cache ($r_m = 1\%$), $b_{Bus/M} = 80$ MB/s, $P = 50$ MIPS: $N \leq 30$

optimistische Annahme: die Speicherzugriffe aller PEs erfolgen ordentlich in Folge, keine Kollisionen!

Konsequenz: Shared Memory-Maschinen können (ohne weitere Maßnahmen) nur mit ca. 16 bis 32 Knoten ausgebaut werden.

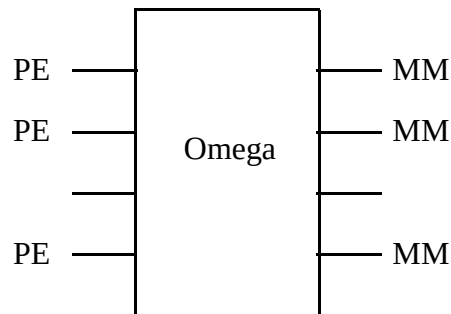
- Ansätze des NYU Ultracomputerprojekts
 - Vergrößerung der Übertragungsrate des VN: statt Bus: Omega-Netzwerk
 - Verminderung der Speicherbelastung

a) Interleaving

b) Kombination von Speicherzugriffen (Combining Network)

- HW-Unterstützung häufiger Koordinationsaktionen durch Fetch & ADD direkt im Speicher-Modul

Ziel: 4096 PEs (realisiert: 8 PEs!)



8 x PE: Motorola 68010 (10 MHz) + FPU (Weitek)

4 x MM: + 32 KB Cache je 2 MB

- Ergebnisse
 - gezielte Unterstützung kritischer Koordinationsaktionen, z.B. Verwaltung von Prozeßwarteschlangen mit parallelem Zugriff, dabei Verzögerungen (infolge Combining Network und Fetch & Add) mit $O(\log N)$.
 - Combining Network in der Realisierung sehr teuer, komplexe Schalter erhöhen Latenzzeit.
 - Weiterführung im Rahmen des IBM-Projekts RP³.

3.5.2.5 Sequent

Kommerziell erfolgreiche Shared Memory Multiprozessoren auf Basis National Semiconductor (B-Serie) und Intel (S-Serie). Sequent wurde 1984 von Intel-Mitarbeitern gegründet.

Konzept: Standard-Mikroprozessoren, VN = Bus, benutzbare SW, insbesondere BS

Global-Architektur:

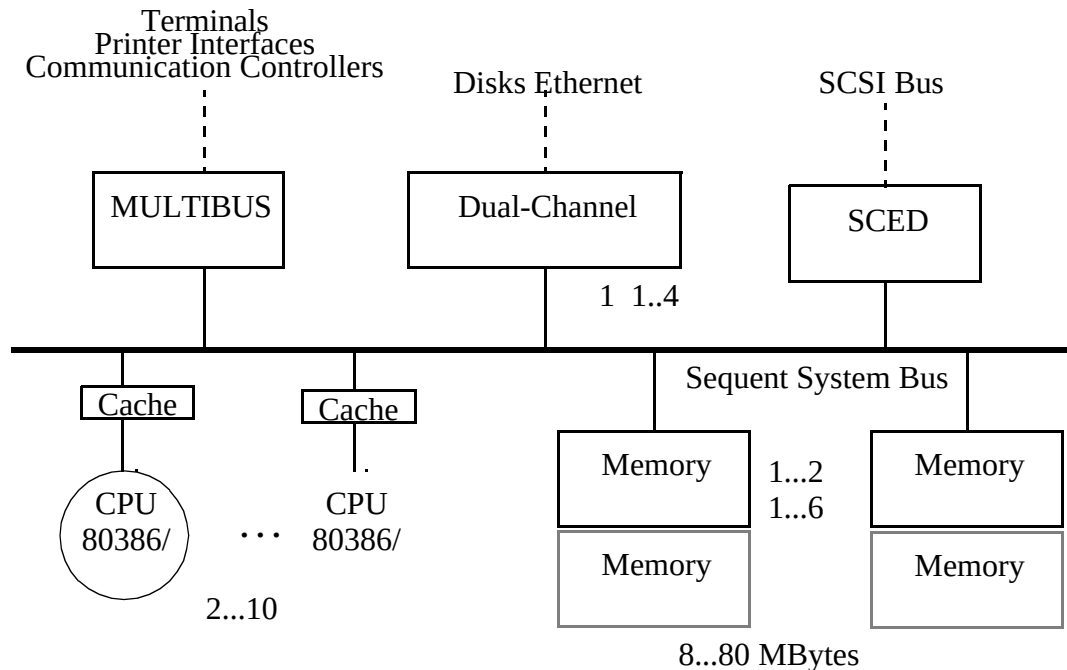


Bild 3.61. Architektur der Sequent Symmetry S 27 bzw. S 81

S27 2-10 PEs (80386)

S81 2-30 PEs (80386)

Globaler Speicher modular ausbaufähig (8...240 MB)

Sequent-Systembus: 10 MHz-Takt, 64 bit Daten, 32 bit Adressen (max.), Übertragungsrate 53.3 MB/s

PE: 80 386, 16 MHz, 80 387 Coprozessor + FP-Akzelerator, 64 KB Cache, SLIC (System Link and Interrupt Controller): steuert Unterbrechungen und Interprozessorkommunikation

Software: Multiprozessor-BS DYNIX, symmetrisch, d.h. es gibt kein ausgezeichnetes PE, welches das BS besitzt. Alle PEs sind indentisch und bedienen sich aus dem Pool der lauffähigen Prozesse.

Sprachen: C mit Erweiterungen "shared" und "private", Fortran 77, Pascal, Lisp, Ada, Cobol
Automatische Erkennung von Parallelität (z.B. aus DO-Schleifen) durch optimierenden Präprozessor.

Anwendungen: kommerzieller Bereich, Transaktionssysteme, Datenbanken

3.5.3. MIMD Message Passing

3.5.3.1 Hypercube-basierte Multiprozessoren

- Cosmic Cube (Caltech, 1983), 6-dim. Hypercube, 64 PEs

Globalarchitektur:

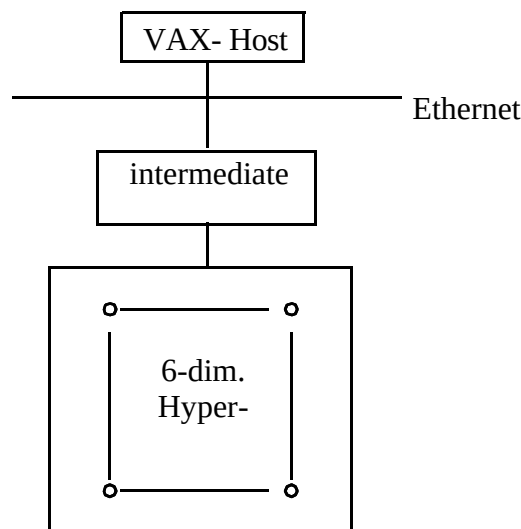


Bild 3.62. HW-Architektur des Cosmic Cube. Zugang zum Cosmic Cube über Vorrechner (VAX-Host: Compilation, BS) und Verbindungsrechner (intermediate Host).

Knoten: 8086/8087 @ 5 MHz, 128 KB RAM, 8 KB ROM, Knoten arbeiten asynchron zueinander, Synchronisation implizit durch Message Passing

VN: 6-dim. Hypercube, Übertragung Knoten-Knoten mit 2 Mbit/s (full duplex).

Protokoll: Store- and- Forward, d.h. jeder Knoten besitzt Puffer zur Aufnahme einer ganzen Message. Routing durch BS: Prozessor wird unterbrochen.

Software: Programmierung in C oder Fortran.

2 Programmiermodelle:

- Crystalline: 1 Prozeß/Knoten, Messages werden an andere Prozesse/Knoten direkt versandt: Anwender sieht und nutzt die Hypercube-Struktur (explizite Einbettung des Problemgraphen auf die Verbindungstopologie). Anwender: Physiker
- Virtualisiert: Sequentielle Prozesse kommunizieren über virtuelle Kanäle. BS ordnet Prozesse den Knoten (1 Prozeß/Knoten), virtuelle Kanäle den physikalischen Links zu. (Aber: Prozeßzuordnung auf Knoten ist statisch.) Zuordnung Prozeß-Knoten kann vom Anwender beeinflußt oder automa-

tisch vorgenommen werden. Dazu: Analyse der (dynamischen) Prozeßstruktur; geeignetes Zusammenfassen der Prozesse so, daß a) Lastgleichgewicht, b) Kommunikationsgleichgewicht.

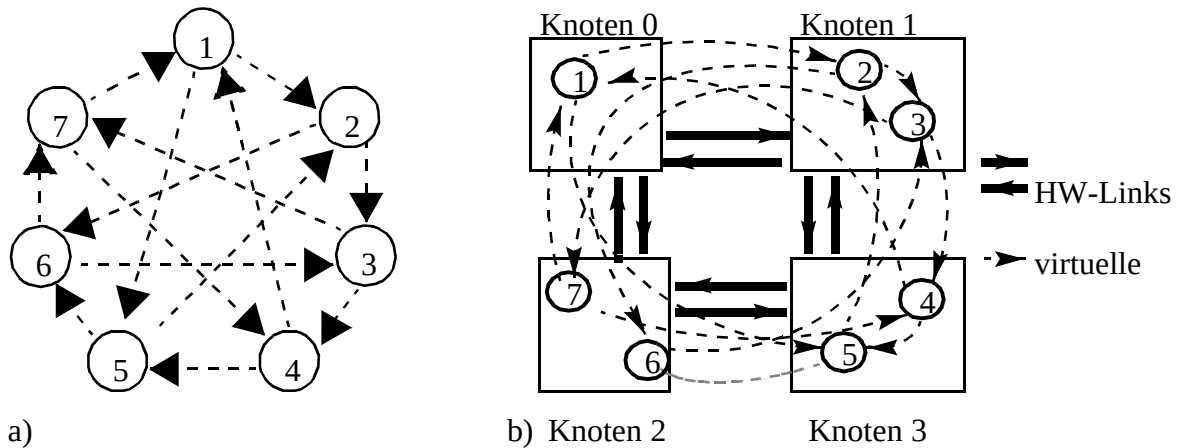


Bild 3.63. Abbildung der dynamischen Prozeßstruktur (a) auf die HW-Link-Struktur (b)

Anwendungen: Hochenergiephysik, Astrophysik, Chemie, Seismik

- nCube

Kommerzielle Version des Cosmic Cube.

Ziele:

- gesamter Knoten (außer Speicher) auf einem Chip (Custom) senkt Chip count, erhöht Zuverlässigkeit
- VN auf Backplane
- Skalierbarkeit

Firmengründung 1983, Ende 1985 nCube/10 (1024 PEs), 1 m³, 1989 nCube/2 (8192 PEs)

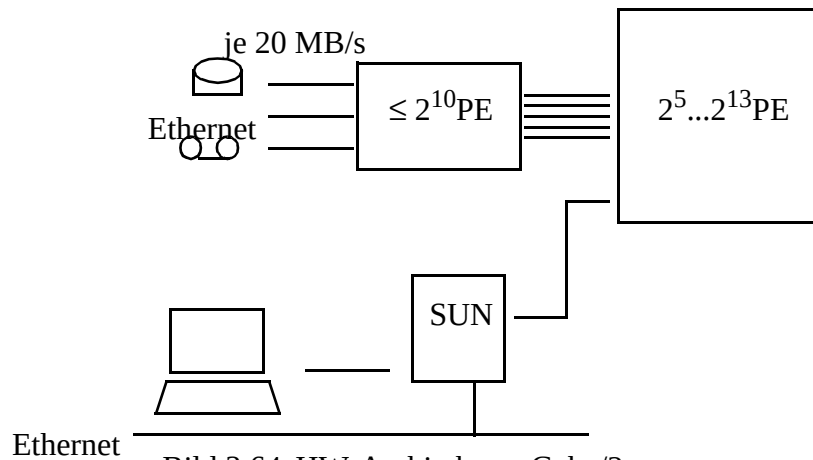
Globalarchitektur (nCUBE/2)

Bild 3.64. HW-Architektur nCube/2

Knoten

1 PE = 1 Full Custom Chip = (500.000 Transistoren, 1 μ m CMOS) + DRAM = "Computing Element"

64 bit-CPU, 64 bit IEEE FPU, Message Router (CPU wird nicht unterbrochen), 14 bidirektionale Links, 25 MHz, 10 MIPS, 4 MFLOPS (SP), 3 MFLOPS (DP), 4-stufige PL, Daten-cache, Befehlscache.

Speicher: 1....64 MB/PE

Links: 2.22 MB/s pro Richtung, 13 Links für 13-dim. Hypercube, 1 Link für I/O, HW-unterstütztes Wormhole-Routing, 64 PEs pro Board (6dim.-Hypercube), 16 Boards pro Motherboard

Peak-Performance:

	128	8192
GIPS	1.280	82
GFLOPS (SP)	0.512	32
GFLOPS (DP)	0.384	25

Software

- BS-Kern (VERTEX) auf jedem Knoten (Laden von Programmen, whoami, nwrite, nread)
- Subcube- Allocation: Multi-User-Fähigkeit durch Space-Sharing (Anwender fordern Subcube der Dimension dim an)
- Sprachen: C, Fortran 77
- Entwicklungsumgebung (Cross Compiler) auf SUN-Workstations
- Tools: FORGE, MIMDizer, Express, PICL

Anwendungen

- technisch-wissenschaftliche Anwendungen (wie Cosmic Cube)
- kommerzielle Anwendungen: ORACLE-Datenbank
- iPSC

Intel-Version des Cosmic Cube

iPSC/1 (1985):

Knoten: 80286/80287 (1 MIPS, 62 KFLOPS), 0.5...4.5 MB, 8 Ethernet (!)-Kanäle, davon 7 zum Aufbau des Hypercube (max. 128 Knoten), einer für I/O

VN: 7-dim. Hypercube, Ethernet-Controller (10 MBit/s pro Kanal): lange Setup-Zeiten (10 ms für 8 B-Nachricht)

iPSC/2 (1988):

wie iPSC/1, folgende Verbesserungen

- 80386, 80387 CPU (64 KB Cache, 4 MIPS, 200 KFLOPS)
- 1..16 MB/Knoten
- 8 bidirektionale Kanäle, Wormhole-Routing, 22 Mbit/s / Kanal, 1 Kanal für I/O
- zentrale Verwaltung (Programmverteilung, Subcube-Allocation) durch Systems Resource Manager: zusätzliche Rechner, verbunden mit einem der Knoten.
- Erweiterung durch Concurrent I/O Facility: jeder Knoten erhält über Kanal 8 einen I/O-Prozessor (80386/7, SCSI, bis 7 Platten)

iPSC/860 (1990):

wie iPSC/2, aber

- Knotenprozessor i80860 (40 MHz, 60 MFLOPS)
- heterogenes System (Mischung 386,860) ist möglich.
- großer Unterschied zwischen Peak (3.8 GFLOPS, 64 Knoten) und tatsächlicher Leistung (1GFLOPS): 26 % Effizienz

Software:

- kleines Knoten-BS NX/2 (Multiprocessing, Message Routing)
- Concurrent File System (CFS): Anwender sieht ein UNIX-artiges Filesystem, CFS verteilt Files transparent auf die Platten.

Konkurrierender Zugriff durch alle Prozessoren:

a) auf 1 File: jeder N-te Block wird einem Knoten zugewiesen

b) separate Files, unterschieden durch Knotennummer (Filename fred, Knoten 0 erhält fred 000, Knoten 127 erhält fred 127)

- Sprachen: Fortran 77, C (mit Unterprogrammaufrufen für Message Passing), Concurrent Common Lisp, Ada
- Tools

Debugger: source level (Lesen/Setzen von Variablen, Message-Warteschlangen, Setzen von Break-Points, Single Step)

Parallelisierung alter Fortran-Programme (dusty deck Fortran) durch FORGE, MIMDizer

3.5.3.2 Parsytec

Deutsche Firma, Sitz in Aachen. Parallelrechner auf Transputer-Basis. Gigacube GC-Serie: 64 ...16 384 Prozessoren

Knoten: Transputer T 9000 (32 bit, 64 bit FPU, 16 KB Cache, Kommunikationsprozessor, 200 MIPS peak, 25 MFLOPS peak), 4 Links (2 x 100 Mbit/s pro Link), Erweiterung auf beliebig viele virtuelle Kommunikationskanäle über Baustein C 104.

VN: 3-D-Kommunikationsnetz, hierarchische Gliederung in Building Blocks (16 Proz.), Giga Cubes (4 Building Blocks).

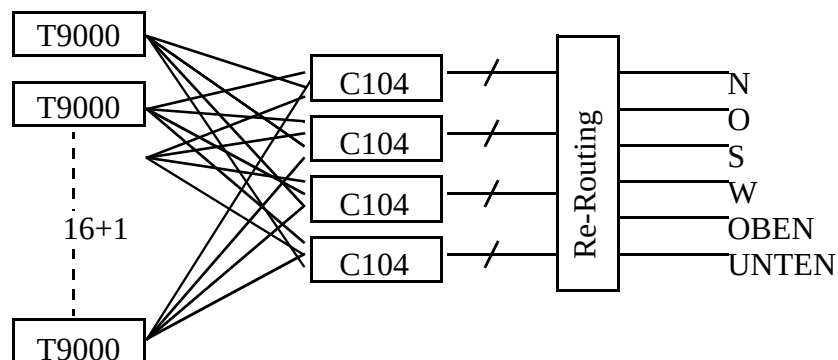


Bild 3.65. Kommunikation innerhalb eines "Building Blocks" des Parsytec GC sowie nach außen in 6 Raumrichtungen

Maximaler Ausbau: 8 x 8 x 16 Building Blocks (16 384 Prozessoren), Peak Leistung 400 GFLOPS.

Virtuelle verlustfreie Einbettung anderer Topologien (Bäume, Hypercube) in das 3-D-Netzwerk.

Latenzzeiten incl. SW-Oberhead (9 ms): benachbarte Knoten 10 ms

entfernteste Knoten 11 ms.....15 ms

Software:

C, Fortran 77, Pascal, Modula-2, Cross-Compiler, BS PARIX (send/receive für 3-D-Netz und Hypercube, Einrichten von Kommunikationskanälen, Laden), außerdem: Virtuelle Topologien (2D, Baum, Perfect Shuffle), vorgefertigte Routinen z.B. für Summe, Maximum, Broadcast, Punkt- zu-Punkt-Kommunikation

3.5.3.3 Paragon XP/S (Touchstone)

Neue Intel-Architektur (1991), max. 4096 i 860 XP, Peak Performance 5...300 GFLOPs (DP)

Knoten: i 860 XP (50 MHz, 75 MFLOPS DP, 42 MIPS), 16...128 MB, zweiter 860 für Kommunikation

VN: 2-D-Netz, Übertragungsrate zwischen beliebigen Knoten: 1.6 Gbit/s (full duplex).

Prototype Touchstone Delta: 576 Knoten (16 x 36)

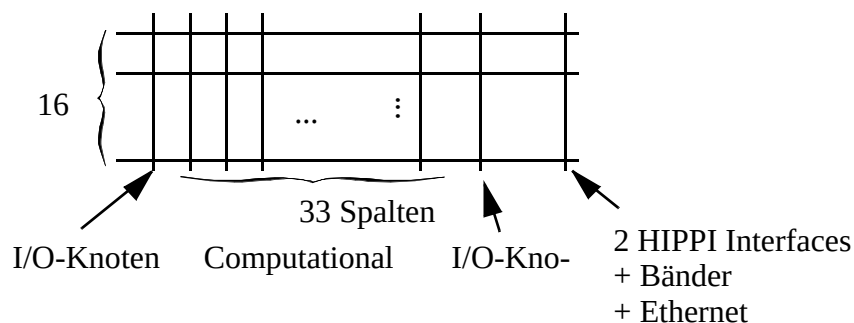


Bild 3.66. Globalarchitektur der Touchstone-Delta-Maschine

Peak Performance Delta: $528 \cdot 60 \text{ MFLOPS} \approx 30 \text{ GFLOP}$

erreichte Performance (JPL): 1.....15 GFLOPS

Zeitweise weltschnellster Supercomputer.

3.5.3.4 CM-5

Connection Machine, aber MIMD!

VN: Fat Tree: konstante Anzahl von Nachbarn, variable Bandbreite

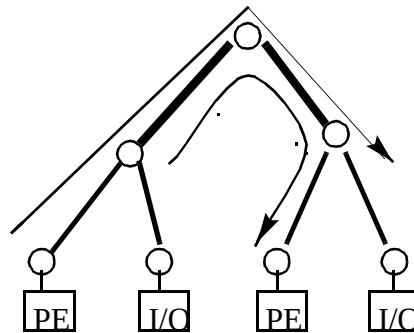


Bild 3.67. Fat Tree: Binärbaum mit höherer Linkbandbreite in den höheren Hierarchiestufen. Blätter: Processing Elements oder I/O-Geräte.

Fat Tree für Message Passing. 2 zusätzliche VNs:

- Control Network, u.a. für die PE-Synchronisation (globales OR) und für Combining Operationen (z.B. Addition, Max, OR, XOR)
- Diagnostic Network

Knoten (PE): SPARC, 64 KB Cache, bis 32 MB, optional: 4 Vektoreinheiten (32 MOPS, "operation" = integer oder floating point dp)

Performance: bis 128 MOPS/Knoten

max. Knotenzahl: 256 K

aktuelle Version: 16 K → > 2 TFLOPS peak , Kosten 16 K-Maschine: 500 Mio US \$.

3.5.4. Hybrid-Architektur

- KSR-1 (Siemens)

Kendall-Square Research (1992)

Shared Memory MIMD (Benutzersicht), nur private Speicher, als konsistente Caches organisiert (Realisierung)

Klassifizierung: gemeinsame Variable (shared variables), verteilter Speicher (distributed mem)
auch: Virtuell gemeinsamer Speicher

Knotenzahl: max. 1088

VN: Hierarchischer Aufbau:

Stufe 0: Selection Engine (SE: 0), bis zu 32 PEs, angeordnet im Ring.

Stufe 1: 34 SE: O-Ringe in einem SE: 1-Ring

Knoten: 2-skalar, 64 bit, 20 MHz: 40 MIPS peak, 40 MFLOPS peak

4 Chip-Satz, Eigenentwicklung

Verteilter Speicher organisiert als ALLCACHE Memory.

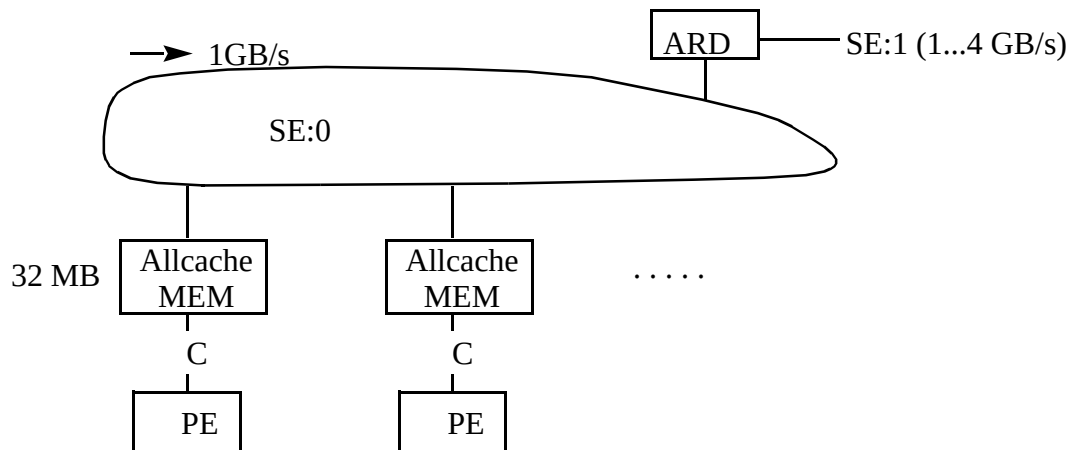


Bild 3.68. ALLCACHE-Organisation: Processing Element spricht Cache C (256 KB Instruktionen, 256 KB Daten) an. Im Miss-Fall: Datum wird geholt aus 2nd-Level-ALLCACHE. Im Miss-Fall: Datum wird geholt aus dem ALLCACHE-Memory eines anderen PE. Es gibt keinen zentralen Speicher. Cache-Block-Größe: 128 B. Latenz in SE:0 : 6.7 ms. Verbindung zur höheren Hierarchiestufe über ALLCACHE Routing and Directory Cell ARD.

Software:

BS: Erweiterung von OSF/1 (UNIX), dynamische Prozessor-Allokation erleichtert Lastausgleich.

Sprachen: C, Fortran, Cabol

ORACLE DBMS

Tuxedo transaction processing

