Vorlesung Rechnerorganisation II

Prof. Dr. Harald Richter

Vorwort

Es ist zu hoffen, dass dieses Skript einen guten Zugang zu diesem Fach bietet. Allerdings ist es kein Buch; es soll nur die Zuhörer davon befreien, von Hand mitschreiben zu müssen, so dass sie sich besser auf den Stoff konzentrieren können. Ein ausführliches Buch zum Abschnitt über Parallelrechner ist in der Unibibliothek oder beim Verfasser erhältlich. Bilder in diesem Skript sind aus diesem Buch.
Jeder Student hat unterschiedliche Stärken und Schwächen. Deswegen werden sie persönliche Bemerkungen und Erklärungen an unterschiedlichen Stellen des Skripts einfügen, je nach Bedarf. Das vorliegende Dokument wird dabei helfen, dies in Ruhe zu tun.
Weder das Skript noch das web können den Besuch der Vorlesungen ersetzen. Es hat sich viele Male gezeigt, dass ein autodidaktisches Lernen umfangreichen Stoffs nur Hilfe von Schriften, dem web oder per Video viel mehr Zeit und persönliches Engagement erfordert als die Vorlesungen regelmäßig zu besuchen; insbesonders wenn ein persönlicher Kontakt während der Vorlesungszeit und den Übungen möglich ist.

1 Die Organisation von RISC-Prozessoren

- **Gegenstand dieser Vorlesung ist die Darstellung aktueller Organisationsformen von RISC-Prozessoren und Parallelrechnern**
- **Die verschiedenen Rechner-Organisationsformen haben alle das Ziel, Prozessoren bzw. Rechner schneller zu machen**
- **Typische Maßnahmen zur Beschleunigung von Prozessoren bzw. deren Cores sind:**

1.) Pipelining

2.) Sprungvorhersage

3.) Superskalarität

4.) Dynamische Befehlsausführung

5.) Spekulative Befehlsausführung

Hinweis: In der Vorlesung Rechnerorganisation I wurden bereits Pipelining und Sprungvorhersage behandelt.

Hinweis: Im ersten Teil dieser Vorlesung werden Superskalarität, Dynamische und Spekulative Befehlsausführung erläutert

Hinweis: Im zweiten Teil der Vorlesung werden Parallelrechner und Parallelverarbeitung behandelt. Diese dienen zur zusätzlichen Beschleunigung der Programmausführung

1.1 Superskalarität

Def.: Superskalarität: In einem Prozessor existieren n>1 Rechenwerke (ALUs) zur gleichzeitigen Bearbeitung mehrerer Befehle.

- **Superskalarität wurde erstmals Anfang der 1960er-Jahre in Mainframes eingesetzt**
- **Seit längerem wird sie in allen RISC-Prozessoren verwendet**
- **Superskalarität kann die Rechenleistung erheblich steigern, da sie es erlaubt, zu einem Zeitpunkt nicht nur einen einzelnen Zahlenwert, d.h. Skalar zu verarbeiten, sondern n>1 Skalare gleichzeitig**
- **Die n Skalare können z.B. die Elemente eines Arrays (Vektors) sein oder n voneinander unabhängige Zahlenwerte**
- **In beiden Fällen beschleunigen so die Bearbeitung des Arrays bzw. der Zahlenwerte um den Faktor n**
- **Allgemein wird die gleichzeitige Bearbeitung mehrerer Befehle als Parallelverarbeitung bezeichnet**

⇒ **Superskalarität bewirkt Parallelverarbeitung innerhalb des Cores, der wiederum Teil des Prozessors ist**

- **Aufgrund der Vorlesung von Rechnerorganisation I wissen wir, dass zusätzlich jedes Rechenwerk als Pipeline organisiert ist**

⇒ **Superskalarität bedeutet die Existenz von n>1 Befehls-Pipelines im Core, die im besten Fall alle gleichzeitig, d.h. parallel betrieben werden**

Beispiel: Es gebe in einem Core zwei Pipelines für je ein Addier- und Multiplizierwerk zur beschleunigten Berechnung von Skalarprodukten => n=2

- **In einem heutigen superskalaren Core können bis ca. 10 Befehle in ebensovielen Pipeline-Rechenwerken gleichzeitig bearbeitet werden (Bsp. Intel Itanium)**
- **In Graphikprozessoren für Spielekonsolen können ca. 128-256 Befehle gleichzeitig bearbeitet werden**
- **Dies ist aufgrund der auf Computergraphik eingeschränkten Problemklasse möglich**

Beispiel: Bereits beim historischen Sony/IBM Cell-Prozessor für die Sony Playstation 2 gab es bis zu 16 unabhängige Befehlsströme die jeweils 16 Daten gleichzeitig bearbeiten konnten => n = 256.

1.2 Parallelverarbeitung

- **Insgesamt existieren drei grundsätzlich verschiedene, organisatorische Möglichkeiten, einen Core schneller zu machen**
- **Alle drei beruhen auf der Parallelverarbeitung von Operanden und Befehlen**

1.) Die erste Art der Parallelverarbeitung sind das Befehls- sowie Arithmetik-Pipelining

2.) Die zweite Art der Parallelverarbeitung ist die Superskalarität

3.) Die dritte Art der Parallelverarbeitung ist die sog. explizite Parallelität

- **Letztere tritt bei Parallelrechnern und bei sog. Multi Core bzw. Many Core CPUs auf**
- **Als Multi Core bzw. Many Core CPUs werden Mehrkernrechner bezeichnet, deren Cores alle auf einem einzigen Silizium-Chip untergebracht sind**

Def.: Die Begriffe Prozessor und Core werden synonym zueinander verwendet. Eine Multi Core CPU besteht aus mehreren bzw. vielen Cores bzw. Prozessoren.

- **Bei expliziter Parallelverarbeitung werden n>1 Cores in einer CPU, oder n>1 CPUs oder n>1 Rechner auf Anweisung des Programmierers, d.h. explizit dazu eingesetzt, Befehle und Operanden parallel zu bearbeiten**
- **n kann dabei sehr groß werden: bis ca. 10^6**
- **Der Programmierer muss dabei jedesmal genau angeben, wann welcher Core/Prozessor/Rechner welchen Befehl bearbeitet**
- **Bei einer großen Zahl von Cores/Prozessoren oder Rechnern kann das sehr aufwendig werden**
- **Explizite Parallelverarbeitung erfolgt mit Hilfe von sog. Kommunikationsbibliotheken, die zur Benutzeranwendung hinzugebunden werden und die die sog. Interprozessorkommunikation abwickeln**
- **Als Interprozessorkommunikation wird der Informationsaustausch zwischen Prozessoren bezeichnet, der bei Parallelverarbeitung meistens anfällt**
- **Weit verbreitete Kommunikationsbibliotheken zur Interprozessorkommunikation sind:**

1.) MPI (Message Passing Interface)

2.) OpenMP (Open Source Multiprozessor)

- **Explizite Parallelverarbeitung kann auch über spezielle parallele Programmiersprachen erfolgen, wie z.B.:**

1.) High Performance Fortran (HPF) = historisch

2.) Occam = historisch

3.) Parallel C = historisch, SystemC

Hinweis: Von allen parallelen Programmiersprachen ist nur noch SystemC relevant, allerdings nur zur FPGA-Programmierung und zur Systemsimulation, bei der parallele Programmierung mit enthalten ist.

- **Beim bereits in RO 1 behandelten Befehls- bzw. Arithmetik-Pipelining sowie bei Superskalarität liegt implizite Parallelität vor, da der Prozessor ein rein sequentielles Computerprogramm als Eingabe erhält**
- **Implizite Parallelität bewirkt Prozessor-interne Parallelverarbeitung ohne explizite Anweisungen des Programmierers, wann welcher Befehl bearbeitet werden muss**
- **Bei impliziter Parallelität muss der Core selbst herauszufinden, welche Befehle er gleichzeitig mit welchen anderen ausführen kann**
- **Der Programmierer kann den Core dabei durch zusätzliche Hinweise im Programmkode unterstützen (sog. Pragmas oder Direktiven)**

- Diese Direktiven sind nicht zahlreich im Code vertreten und unterscheiden sich im Programmierstil fundamental davon, eine parallele Programmiersprache verwenden oder eine Kommunikationsbibliothek aufzurufen (= explizite Parallelität)
- Bei impliziter Parallelität muss der Core die interne Parallelverarbeitung im wesentlichen selbst leisten
- Dies ist eine heikle Aufgabe, da die Befehle eines Programms voneinander abhängen
- Die sequentiell aufeinanderfolgenden Befehle eines Programms bilden mathematisch gesprochen eine sog. totale Ordnung
- Die totale Ordnung bildet die Relation „ist Nachfolgebefehl von“
- Sie wird durch die Aufschreibreihenfolge der Befehle im Programmkode festgelegt und gibt an, welcher Befehl unmittelbar nach welchem anderen ausgeführt werden soll
- Die implizite Parallelität erfordert jedoch eine partielle Ordnung der Befehle, in der einzelne Befehlsstränge existieren, die unabhängig voneinander sind
- Denn nur voneinander unabhängige Befehlsstränge können parallel bearbeitet werden

⇒ Bei der impliziten Parallelität geht es um die automatische Konversion eines sequentiellen Programms von einer totalen Ordnung in mehrere bzw. viele partielle Ordnungen, die unabhängig voneinander sind

- Innerhalb einer partiellen Ordnung bildet jede geordnete Teilmenge die neue Relation „Befehl hängt ab von“

- Die neue Relation „Befehl hängt ab von “ hat mehr Freiheitsgrade als „ist Nachfolgebefehl von“, da zwischen zwei abhängigen Befehlen beliebig viele unabhängige Befehle eingefügt werden können
- Die neue Relation kann entweder in Echtzeit während der Programmausführung vom Prozessor herausgefunden werden, oder alternativ auch vor der Programmausführung, d.h. nicht in Echtzeit, vom Compiler ermittelt werden
- Die neue Relation beinhaltet alle Arten, wie Befehle voneinander abhängen können
- Diese Arten von Befehlsabhängigkeiten sind:

1.) Die logische Abhängigkeiten zwischen Befehlen, wie z.B. die sog. Read-After-Write-Abhängigkeit
   - „Zuerst muss das eine gemacht werden, bevor das andere gemacht werden kann“

2.) Die organisatorischen Abhängigkeiten von Befehlen, die aufgrund von Sachzwängen im Core existieren

- Solche Core-internen Abhängigkeiten äußern sich in Form von sog. Konflikten
- Die auftretenden Konflikte sind gemäß Rechnerorganisation I:

1.) Steuerflusskonflikte aufgrund von Sprungbefehlen

2.) Datenflusskonflikte aufgrund von Pipelining

3.) Ressourcenkonflikte aufgrund von einer begrenzten Zahl von Rechenwerken bzw. Speicherzugriffen pro Sekunde

- Für eine korrekte Programmausführung muss der Prozessor die logischen Abhängigkeiten sowie alle drei obigen Arten von Konflikten erkennen und beherrschen
- Bei Taktraten von 2-4 GHz steht dafür nur sehr wenig Zeit zur Verfügung (250-500 ps)

Hinweis: 1 Picosekunde = 10^{-12} s, d.h. der Tausendste Teil einer Milliardstel Sekunde!

⇒ **Core-interne Parallelverarbeitung ist schwierig zu realisieren. Sie macht Prozessoren sehr kompliziert.**

- Deshalb wird alternativ auch der Compiler zur Lösung dieser Aufgabe herangezogen
- Daraus ist eine neue Klasse von CPUs entstanden, die weder CISC noch RISC sind
- Diese werden als VLIW-CPUs bezeichnet

Hinweis: VLIW = Very Large Instruction Word Computer

- Das einzige kommerziell erfolgreiche Beispiel für VLIW-CPUs ist die Intel „Itanium"-Serie
- Bei einigen RISC CPUs wie z.B. Intel Pentium sind allerdings Konzepte von VLIW mit integriert
- Hat der Core oder der Compiler die Umwandlung des betrachteten Programmcodes in eine partielle Ordnung vorgenommen, sind n>1 Teilordnungen innerhalb des betrachteten Codes entstanden, die voneinander unabhängig sind
- Diese Teilordnungen können im besten Fall alle parallel ausgeführt werden

- In der Praxis hat die Umwandlung des betrachteten Programmcodes in eine partielle Ordnung eine hohe Rechenkomplexität bzw. ist bei ungünstigem Programmierstil manchmal sogar ganz unmöglich
- Zur Vereinfachung des Problems betrachtet der Prozessor deshalb während der Programmausführung nur einen kleinen Abschnitt von je 10-50 Maschinenbefehlen
- Dieser Programmabschnitt heißt „Instruction Window"

Def.: Die Menge der gleichzeitig betrachteten Befehle bilden das Instruction Window, innerhalb dessen die Zuordnung von Befehlen zu Befehls-Pipelines erfolgt, um implizite Parallelität bei der Befehlsausführung zu erzielen.

- Das Instruction Window wird nach der Ausführung eines Befehls um einen Befehl im Kode nach vorne geschoben, so dass stets nur kleine Ausschnitte umgewandelt werden
- Der Core versucht, im jeweils betrachteten Programmabschnitt voneinander unabhängige Maschinenbefehle oder Folgen von Maschinenbefehlen zu identifizieren
- Werden solche Befehle gefunden, kann der Core sie parallel ausführen, um Zeit zu sparen, sofern die Ressourcen dazu vorhanden sind
- Umgekehrt kann es vorkommen, dass im Core mehr Ressourcen existieren als partielle Ordnungen im Instruction Window vorhanden sind
- Dann wird der Core nicht effizient genutzt
- Man spricht insgesamt von Parallelität auf Instruktionsebene = „Instruction Level Parallelism"

- Die Core-interne Parallelisierung des Programms wird von einer sehr aufwendigen Zusatz-Hardware bewerkstelligt, die bis ca. 20% der Core-Fläche auf dem Chip verbrauchen kann und deren Entwurf sehr schwierig ist
- Der Entwurf dieser Zusatz-Hardware im Core ist seit ca. 10-15 Jahren so komplex geworden, dass die RISC-Architektur gewisse Grenzen erreicht hat

⇒ **Die Komplexität von einzelnen RISC-CPUs kann bei vertretbarem unternehmerischem Risiko kaum weiter gesteigert werden**

- Als einfacher Ausweg gelten Multi Core- oder Many Core CPUs, die es eben aus diesem Grunde seit 10-15 Jahren gibt
- Die Komplexität wird dadurch vom CPU-Hersteller auf den Programmierer verlagert

1.2.1 Multi Core- und Many Core-CPUs

- Multi Core- und Many Core CPUs unterscheiden sich voneinander in der Zahl der Prozessoren pro CPU-Chip und in der Art der Kopplung der Cores
- Multi Core CPUs haben bis ca. 16 Cores pro CPU auf dem Chip
- Many Core CPUs liegen z.T. deutlich darüber und haben aufgrund ihrer hohen Core-Zahl auch eine andere Art der Prozessor-Kopplung
- Multi Core CPUs beinhalten komplexe RISC-Prozessoren, mit allem, was bei vertretbarem Aufwand möglich ist

- Many Core CPUs bestehen nur noch aus rel. einfach organisierten RISC-Prozessoren und benötigen pro Core nur wenig Chipfläche

⇒ **Viele, einfach organisierte RISC-Prozessoren passen auf einen einzigen Siliziumchip, auch wenn das Moorsche Gesetz seit einigen Jahren nicht mehr gilt.**

Hinweis: Das Moorsche Gesetz galt mehr als 5 Jahrzehnte lang und sagte, dass sich die Zahl der Transistoren pro Chip alle 18 Monate verdoppelte.

1.2.2 Automatische Parallelisierung im Instruction Window

- Die Core-interne Code-Konversion im betrachteten Ausschnitt von Maschinenbefehlen löst näherungsweise das Problem, wie man ein sequentielles Computerprogramm automatisch in ein paralleles Programm überführt
- Näherungsweise deshalb, weil nie das ganze Programm am Stück betrachtet wird
- Die dazu im Core erforderlich Zusatz-Hardware neben dem Instruction Window sind die **Tomasulo-Reservierungsregister** und das **Scoreboard der vollen Ausbaustufe**

Hinweis: die Tomasulo-Reservierungsregister und das Scoreboard der einfachen Ausbaustufe wurden in RO I behandelt

- Das Scoreboard der vollen Ausbaustufe ist das bereits erwähnte, hochkomplexe Bestandteil jedes Cores
- Es besteht aus diversen Listen und gekoppelten endlichen Automaten

Hinweis: Der Aufbau des Scoreboards der vollen Ausbaustufe wird in einem extra Kapitel erläutert.

- Die Aufgabe des Scoreboards der vollen Ausbaustufe ist die folgende:
  - Es entscheidet während der Programmausführung, wann welcher Maschinenbefehl von welchem Rechenwerk bearbeitet wird, unter Beachtung sämtlicher Abhängigkeiten und der Resourcenbelegung des Cores
- Die Reservierungsregister sammeln währenddessen Zwischenergebnisse als Input für einen nachfolgenden Befehl, der diese Zwischenergebnisse benötigt

Beispiel: Die Auflösung einer Read-after-Write-Abhängigkeit (RAW) geschieht im Falle von Superskalarität dadurch, dass der betroffene Befehl aufgrund der Intervention Scoreboard solange nicht von einem Rechenwerk ausgeführt wird, bis seine Eingabeoperanden in einem Reservierungsregister zur Verfügung stehen.

- Währenddessen werden andere Befehle in der Bearbeitung vorgezogen, um keine Zeit zu verlieren

⇒ Befehle werden anders als vom Programmierer vorgesehen ausgeführt, d.h. außerhalb der geplanten Reihenfolge

- Man spricht deshalb auch von sog. dynamischer Befehlsausführung oder Out-of-order Execution

1.3 Dynamische Befehlsausführung

- Dynamische Befehlsausführung (Out-of-Order Execution) bewirkt, dass Befehle i.d.R. in einer anderen Reihenfolge ausgeführt werden, als vom Programmierer angegeben

Hinweis: i.d.R. = in der Regel.

- Die Assembler- bzw. Maschinenbefehle, in die ein Hochsprachenprogramm übersetzt wird, werden in verwürfelter Reihenfolge abgearbeitet um Parallelität zu erzielen

⇒ Die dynamische Befehlsausführung (Out-of-order Execution) bewirkt i.d.R eine andere, nicht-sequentielle Reihenfolge der Befehlsabarbeitung

- Sie wird bei Befehls-Pipelines und bei Superskalarität eingesetzt, um durch Umordnen der Befehle Datenfluss-, Steuerfluss- und Ressourcenkonflikte im Core zu vermeiden

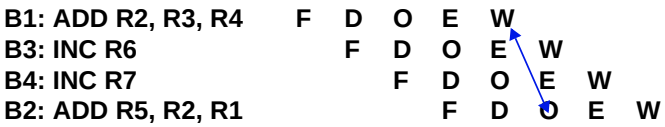
Hinweis: Datenfluss-, Steuerflusskonflikte und Ressourcenkonflikte wurden in RO I beschrieben.

Beispiel: Datenflusskonflikt, der Art RAW (Read-After-Write)

- Zwischen B1 und B2 existiert ein RAW-Konflikt, da R2 gelesen wird, bevor es beschrieben wurde
- B3 und B4 sind hingegen unabhängig von B1 und B2

⇒ Durch Vorziehen von B3 und B4 vor B2 wird der RAW-Konflikt vermieden und trotzdem dasselbe Ergebnis berechnet

- Nach einer out-of-order execution:



- Durch Vermeiden des RAW-Konflikts mit Hilfe von Befehlsumordnung, ist es oft unnötig, die Pipeline anzuhalten => Pipeline bleibt schnell
- Aber: bei n Rechenwerken müssen immer wieder n unabhängige Befehle gefunden werden, damit die ALUs ununterbrochen beschäftigt sind
- Dies ist sehr schwierig und gelingt auch bei dynamischer Befehlsausführung nur teilweise

⇒ Durch eine größere Core-interne Parallelität treten bei Superskalarität im Vergleich zum Befehls-Pipelining mit n=1 verstärkt Konflikte auf

- Für eine Befehlsumordnung ist entweder ein optimierender Compiler erforderlich oder ein Echtzeit-Scheduler in Hardware im Core
- Der Echtzeit-Scheduler ist das Scoreboard der vollen Ausbaustufe
- Ein optimierender Compiler erhöht allerdings Komplexität und Dauer des Übersetzungsvorgangs
- Umgekehrt trägt der Echtzeit-Scheduler wesentlich zur Komplexität des Cores bei

- Ein optimierender Compiler vermeidet Chip-Komplexität, kann aber im Gegensatz zum Scheduler nur eine statische Umordnung der Befehle zur Übersetzungszeit vornehmen, was ihn weniger effizient macht
- Der Scheduler versucht die Code-Umordnung zur Laufzeit zu erreichen (= dynamische Programmausführung) und ist dadurch flexibler, da bei manchen Programmen Konflikte erst während der Laufzeit auftreten
- In jedem Fall muss der Prozessor als Ergebnis genau das berechnen, was der Programmierer für den Fall einer sequentiellen Ausführung vorgegeben hat

⇒ Die „Logik“ des Programms wird eingehalten, obwohl die Befehle, sofern möglich, vom Core in einer anderen Reihenfolge und ganz oder teilweise parallel ausgeführt werden

1.4 Die Fetch Unit

- Für Superskalarität ist es außerdem erforderlich, dass in der Fetch- und Decode-Phase von der Befehls-Pipeline n>1 Befehle geholt und dekodiert werden

Hinweis: Fetch- und Decode-Phase wurde in RO I erklärt.

- Andernfalls könnten die n Rechenwerke nicht mit Input versorgt werden
- Das Holen und Dekodieren der Befehle wird von zwei Einheiten im Prozessor erledigt, die als Fetch Unit und als Decode Unit bezeichnet werden

- Die Fetch Unit sorgt dafür dass, im Core stets ein kleiner Vorrat von Befehlen im voraus gehalten wird

⇒ **Für Superskalarität werden neben dem Scoreboard der vollen Ausbaustufe, den Tomasulo-Reservierungsregistern auch eine Fetch Unit und eine Decode Unit benötigt**

Hinweis: Tomasulo-Reservierungsregister wurden in RO I erklärt.

- Der Befehlsvorrat wird in einem kleinen, Fetch-Unit-internen Speicher, dem sog. **Prefetch Buffer** zwischengespeichert
- Der Befehlsvorrat dient als Reservoir für die parallele Bearbeitung durch mehrere ALUs
- Aufgrund der Core-internen Parallelverarbeitung von Befehlen können diese zu jedem Zeitpunkt in unterschiedlichen Stadien, den sog. **Ausführungszuständen** sein
- Die aus der Vorlesung Rechnerorganisation I bekannten vier Ausführungszustände von Befehlen in RISC CPUs sind:

1.) **Fetch** = Befehl holen

2.) **Decode/Operand** (D/O) = Befehl dekodieren und Operanden aus den Registern holen

3.) **Execute** = Befehl ausführen

4.) **Write Back** = Resultat in ein Register zurückschreiben

- Aufgrund von Rechnerorganisation I wissen wir ferner, dass die vier Ausführungszustände einer Befehlsfolge eine eigene Pipeline bilden, die sog. **Befehls-Pipeline**
- Aufgrund der Superskalarität gibt es so viele Befehls-Pipelines wie es Rechenwerke gibt
- Neben den Befehls-Pipelines besteht auch jedes Rechenwerk intern aus einer Pipeline
- Diese ALU-Pipelines werden als **arithmetische Pipelines** bezeichnet
- Außerdem ist bei Superskalarität jede Befehls-Pipeline mit einer Arithmetik-Pipeline zu einer besonders langen Pipeline zusammengeschaltet
- Darüberhinaus bestehen auch die beiden anderen Phasen „Befehl/Operanden holen“ und „Ergebnis rückschreiben“ jeder Befehls-Pipeline aus einer Pipeline, der sog. **Speicherpipeline**, so dass sich insgesamt eine „Superpipeline“ ergibt

⇒ **Superskalarität heißt, dass es n>1 Superpipelines gibt, die im besten Fall alle parallel laufen**

1.4.1 Core-interne Parallelität durch die Fetch Unit

- Gleichzeitig zur Befehlsausführung in den Superpipelines liest die Fetch Unit neue Befehle aus dem Hauptspeicher bzw. aus dem sog. **1st Level-Befehlscache**

Hinweis: Jeder Core hat mehrere Befehls- und Datencaches zum schnelleren Zugriff auf Befehle und Daten.

- Der Prefetch Buffer zur Befehls- und Datenzwischenspeicherung ist noch schneller als der 1st Level- Befehls-/Datencache, aus dem er seine Informationen holt, da er gleichzeitig n>1 Rechenwerke versorgen muss
- Der Prefetch Buffer arbeitet zwar mit derselben Taktrate wie die Superpipelines d.h. mit 2-4 GHz, hat aber eine kleinere Zugriffszeit als der 1st Level- Befehls-/Datencache
- Idealerweise ist der Prefetch Buffer nie leer, so dass den ALUs immer genügend Befehle zur parallelen Abarbeitung zur Verfügung stehen
- Zusätzlich arbeitet die Fetch Unit im Falle von Sprungbefehlen eng mit der sog. **Branch Prediction Unit** zusammen, die aus Rechnerorganisation I bekannt ist, um zu wissen, von welcher Programmadresse der nächste Instruction Fetch am besten erfolgen soll
- Um Zeitverluste bei falsch vorhergesagten Sprüngen auszugleichen, ist der Prefetch Buffer ausreichend groß, um auch die Zeit für eine ab und zu notwendig werdende Invalidierung der Befehls-Pipeline überbrücken zu können, währenddessen die ALUs sonst keine Befehle und Daten hätten
- Die Befehlspufferung im Prefetch Buffer dient somit auch zum temporären Ausgleich von „Befehlsmangel“, der bei falsch vorhergesagten, bedingten Sprungbefehlen in der Befehls-Pipeline auftritt

Hinweis: das Wechselspiel der Core-Komponenten, die alle parallel arbeiten, ist kompliziert, so dass das Scoreboard der vollen Ausbaustufe als oberster Aufseher und Befehlsverteiler agiert.

1.5 Der Prefetch Buffer

- Der Prefetch Buffer ist Teil der Fetch Unit und enthält Kopien aufeinanderfolgender Befehle aus dem Befehls-Cache, der ebenfalls nur Kopien enthält
- Die Befehls-Originale stehen im Hauptspeicher bzw. auf der Festplatte in der Benutzerdatei
- Typisch sind **10-50 Befehle**, die im Prefetch Buffer vorgehalten werden
- Diese Befehle bilden das Instruction Window, innerhalb dessen die Konversion von der totalen Ordnung in eine partielle Ordnung erfolgt
- Der Prefetch Buffer ist ein **Multiport Memory**, an den der 1st Level Befehls-/Datencache, und die Decode Unit angeschlossen sind

Hinweis: Ein Multiport Memory ist eine Speicherkomponente, die zwei oder mehr Schnittstellen hat, um damit gleichzeitig und unabhängig voneinander auf unterschiedliche Speicheradressen zugreifen zu können.

- Das Multiport Memory hat pro Speicherplatz eine Wortbreite, die es erlaubt, gleichzeitig n>1 Befehle in ein Puffer-Wort einzuspeichern und auszulesen

⇒ **Mit jedem Auslesen können bis zu n Superpipelines gleichzeitig mit Befehlen versorgt werden**

1.6 Die Decode Unit

- Die Decode-Unit ist dem Prefetch Buffer nachgeschaltet und liest aus dem Prefetch Buffer alles, was sie braucht, d.h. Befehle und Operanden
- Bei Superskalarität muss die Dekodiereinheit gleichzeitig n>1 Befehle dekodieren und an die verschiedenen Rechenwerke verteilen
- Dies geschieht mit Unterstützung durch das Scoreboard der vollen Ausbaustufe
- Deswegen werden die Decode Unit und das Scoreboard zusammen als Dispatcher bezeichnet

Hinweis: to dispatch = einplanen, absenden, abfertigen

- Das Verteilen der Befehle an die einzelnen Rechenwerke heißt auch „Befehle ausgeben“ (Instruction Issue) oder „Befehle schedulen“

1.7 Der 1st Level-Befehlscache

- Der 1st Level-Befehlscache ist ähnlich wie der Prefetch-Buffer ein kleines Depot für Befehle, in dem Kopien von Hauptspeicherbefehlen gehalten werden
- Allerdings ist er etwas langsamer als der Prefetch Buffer, dafür aber größer
- Im Gegensatz zum Prefetch-Buffer enthält er nicht notwendigerweise ganze Befehlsfolgen, sondern *nur die am dringendsten benötigten und die am häufigsten ausgeführten Befehle* (siehe Rechnerorganisation I)

- Der Cache sorgt dafür, dass für die dringendsten und häufigsten Befehle der langsame Hauptspeicherzugriff meistens unnötig wird
- In der Regel können >90% der momentan am dringendsten benötigten und der insgesamt am häufigsten ausgeführten Befehle im Befehls-Cache gespeichert werden

⇒ Der jeweils nächste auszuführende Befehl steht im Idealfall bereits im Cache, weil es von allen Befehlen der dringendste ist

⇒ Die Befehle einer Schleife stehen oft komplett im Befehls-Cache, da Schleifen i.d.R. häufig durchlaufen werden

- Da Caches wesentlich kleiner als der Hauptspeicher sind, können Speicherzugriffe nur mit einer Wahrscheinlichkeit <1 durch den Cache befriedigt werden
- Ist ein gewünschter Befehl nicht im Cache enthalten, spricht man von einem Cache Miss
- Der Cache ist zwar viel kleiner als der Hauptspeicher, aber dafür auch viel schneller
- Im Cache gibt es einen „Verdrängungsmechanismus“ und ein „Cache-Line Prefetch“

Hinweis: Für Verdrängungsmechanismus und Cache-Line Prefetch siehe RO I

- Der Verdrängungsmechanismus bewirkt, dass selten benötigte Befehle aus dem Cache entfernt werden, so dass nur die häufig benötigten übrig bleiben
- Der Cache-Line Prefetch sorgt dafür, dass die nächsten auszuführenden Befehle, d.h. die dringendsten Befehle im voraus geholt werden

- Da ein Cache seine Daten nicht in aufsteigender Reihenfolge speichert, sondern gemäß der Kriterien „dringend“ und „wichtig“, ist es erforderlich, den Cache nicht adressmäßig sondern assoziativ zu organisieren, um die Daten im Cache wieder zu finden
- Assoziativ heißt, dass eine Information allein dadurch im Cache wiedergefunden wird, dass ein Teil der gesuchten Information bereits bekannt ist
- Dies ist ähnlich einer Datenbank, die über zuvor bekannte Zugriffsschlüssel auf Einträge zugreift

⇒ **Beim Cache dient ein Teil einer Cache-Zeile dazu, den ganzen Zeileninhalt automatisch wiederaufzufinden**

- Ein Cache unterscheidet sich somit in Aufbau und Funktionsweise fundamental vom Hauptspeicher oder vom Prefetch-Buffer
- Der innere Aufbau von Caches ist aufgrund von Verdrängungsmechanismus, Cache-Line Prefetch und Assoziativität wesentlich komplexer als bei anderen Speichern

⇒ **Caches können keine große Speicherkapazität haben, weil sie viel Zusatzlogik enthalten**

- Caches bewirken andererseits eine wesentliche Beschleunigung der Programmausführung, da es ihnen oft gelingt, langsame Hauptspeicherzugriffe durch schnelle Cache-Zugriffe zu ersetzen

⇒ **Der von-Neumannsche Flaschenhals wird abgemildert**

- Das für Befehls-Caches Gesagte läßt sich analog auf Daten-Caches übertragen

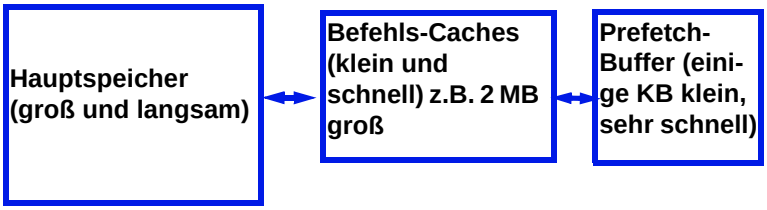
1.8 Der 1st Level-Datencache

- Ein Daten-Cache speichert die am dringendsten und häufigsten benötigten Variablen eines Computerprogramms
- Cores haben getrennte 1st Level-Befehls- und Daten-Caches (sog. split caches gemäß der Harvard-Architektur von Rechnerorganisation I)
- Bei einigen Multi Core CPUs sind auchin den Cores die 2nd Level Caches getrennt und nur der 3rd Level Cache ist gemeinsam

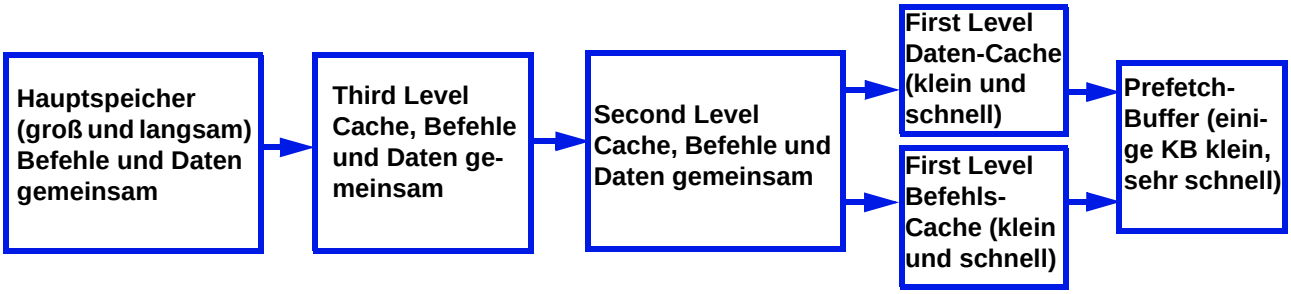
1.9 Die 2nd, 3rd Level Caches

- Die Idee des Cache läßt sich rekursiv erweitern: „einen Cache für den Cache“
- Daraus ensteht eine 2-3-stufige Hierarchie unterschiedlich großer und schneller Caches (1st, 2nd, und optional 3rd-Level-Cache)
- 2nd und 3rd-Level-Cache speichern i.d.R. Befehle und Daten gemeinsam (Ausnahme einige Multi Core CPUs)
- Hauptspeicher und Caches bilden insgesamt eine Speicherhierarchie, die als Speicherpipeline betrieben wird
- Die Speicherhierarchie ist nach zunehmender Geschwindigkeit und nach abnehmender Kapazität organisiert

- Nachfolgend sind zuerst eine vereinfachte und dann eine erweiterte Speicherpipeline für Befehle dargestellt



- Erweiterte Speicherhierarchie/-pipeline für Befehle und Daten



- Derzeit typische Werte für Caches sind:

1st-Level Caches: ca. 128 KB bei <0,5 ns Zugriffszeit
2nd-Level Cache: ca. 2 MB bei 1-10ns Zugriffszeit
3rd-Level Cache: ca. 10-20 MB bei 10-50 ns Zugriffszeit

1.10 Die Load- und Store Buffer

- Der langsame Hauptspeicherzugriff macht sich bei RISC CPUs nur bei zwei Befehlen bemerkbar: bei Load & Store für Daten vom bzw. zum Hauptspeicher
- Um Load & Store schneller zu machen, und um parallel je ein Load und ein Store zu ermöglichen, werden zwei FIFOs als „Auftragspuffer“ eingeführt, die laufende Load/ Stores aufnehmen und autonom ausführen
- Diese Auftragspuffer heißen Load- bzw. Store Buffer
- Dann, wenn der Hauptspeicher frei ist, werden die Einträge im Load- und Store Buffer durch autonome Lese-/ bzw. Schreibzugriffe ohne die CPU ausgeführt
- Für Datentransfers mittels Load/Store Buffer gibt es die folgenden Datenpfade:

1.) Für Load: Hauptsp.->Caches->Load Buffer -> Register

2.) Für Store: Register -> Store Buffer -> Caches -> Hauptsp.

- Die Pfade 1.) und 2.) sind Teil der Speicher-Pipeline

3.) Für alle anderen Befehle gibt es nur den Pfad „Register -> ALU Execute -> ALU Write Back-> Register“ für nach aussen sichtbare Register bzw. den Pfad „Reservierungs-register -> ALU -> Reservierungsregister“ für nach aussen nicht sichtbare Register

- Die Speicher-Pipeline ist darüberhinaus mit der Befehls-Pipeline gekoppelt
- Außerdem gilt ein Store für die CPU bereits dann als abgeschlossen, wenn der Store den zu speichernden Registerinhalt nur in den Store-Buffer geschrieben hat
- Zu diesem Zeitpunkt steht der Registerinhalt noch nicht im Cache oder Hauptspeicher!
- Deshalb inspiziert der Load Buffer zuerst den Store Buffer, ob dort nicht das steht, was gelesen werden soll
- Falls ja, geschieht je nach Komplexität der Rechnerorganisation unterschiedliches:

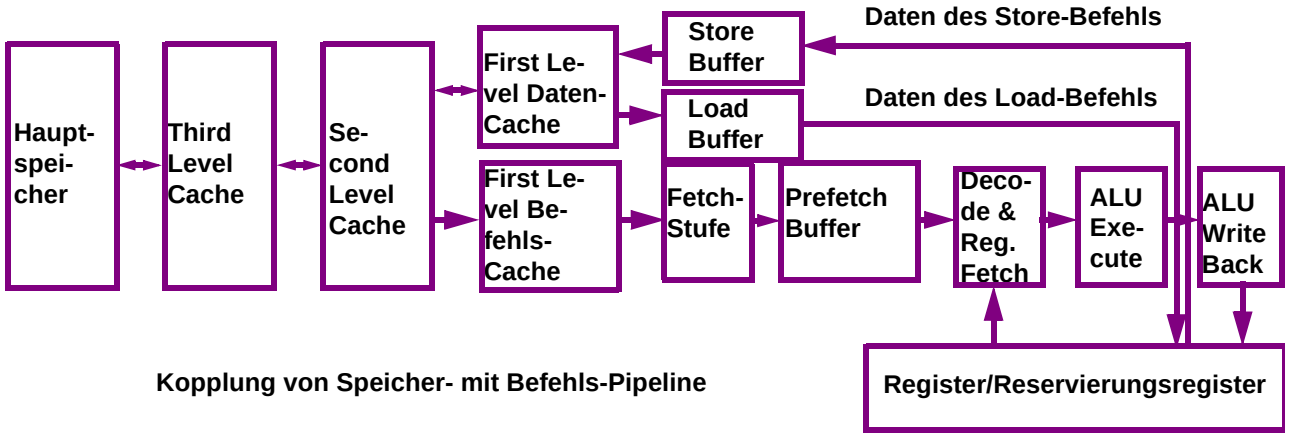
1.) Bei einer einfacheren Rechnerorganisation wartet der Load Buffer ab, bis der Store Buffer das Gesuchte in den Hauptspeicher geschrieben hat, denn der Load Buffer kann hier nur aus dem Hauptspeicher lesen

2.) Bei einer komplexeren Rechnerorganisation liest der Load Buffer direkt aus dem Store Buffer, entfernt das Gesuchte aber nicht daraus, damit es anschließend noch in den Cache bzw. in den Hauptspeicher geschrieben werden kann

1.11 Die Kopplung von Speicher- mit Befehls-Pipeline

- Fügt man die Register und die Befehls-Pipeline zur Speicherpipeline hinzu, erhält man die Kopplung aller wichtigen organisatorischen Einheiten im Prozessor

- Diese Kopplung von Speicher- mit Befehls-Pipeline erfolgt gemäß nachfolgendem Blockdiagramm

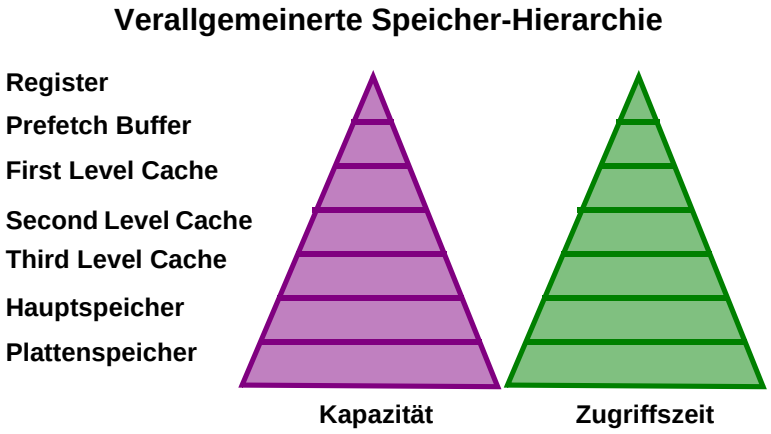


Kopplung von Speicher- mit Befehls-Pipeline

- Beide Pipelines zusammen stellen einen wesentlichen Teil der Organisation von RISC CPUs dar

1.12 Die Verallgemeinerte Speicherhierarchie

- Die Speicher-Pipeline läßt sich auch auf die Festplatte erweitern, sofern das Betriebssystem das sog. Seitentauschen (Paging) unterstützt
- Die Festplatte wirkt bei Paging wie ein virtueller Hauptspeicher, der groß aber langsam ist (siehe RO I)
- Dazu werden jeweils gleich große Speicherabschnitte (Seiten) auf Platte ausgelagert und bei Bedarf wieder eingelesen
- Man erhält als Graphik für Speicherkapazität und -zugriffszeit eine Doppel-Pyramide

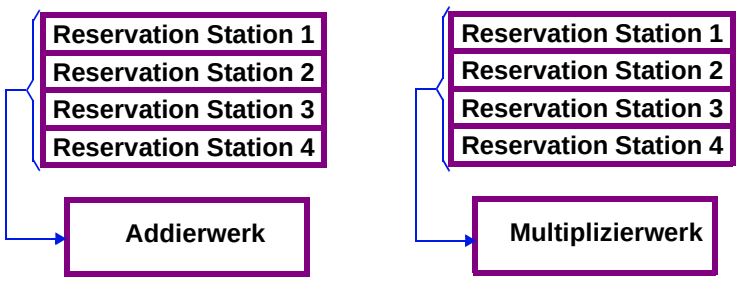


1.13 Die Reservierungsregister

- Bei dynamischer Befehlsausführung und Superskalarität hat jedes Rechenwerk seine eigenen „Reservierungsregister“ direkt an der ALU zum temporären Speichern von Input-Operanden und von Zwischenergebnissen
- Reservierungsregister sind integraler Bestandeil der jeweiligen ALU
- Es existieren m>1 Reservierungsregister in jeder ALU

⇒ Es gibt bei Superskalarität insgesamt m*n>1 Reservierungsregister, wobei m die Zahl der Res.reg. pro ALU und n die Zahl der ALUs pro Core ist

- Beispiel: Addierwerk- und Multiplizierwerk (n=2) mit je m=4 Reservierungsregistern



- Sobald für einen Befehl alle benötigten Operanden in einem Reservierungsregister „eingesammelt“ worden sind, kann dieser Befehl von der ALU ausgeführt werden
- Dies ist unabhängig davon, ob für einen vorangegangenen Befehl alle Operanden bereits vorhanden sind oder nicht, vorausgesetzt, es entstehen dadurch keine Steuerfluss-, Datenfluss oder Ressourcenkonflikte

⇒ **Der Ausführungszeitpunkt von Befehlen ist so innerhalb des Instruction Windows von dem Ausführungszeitpunkt, den der Programmierer vorgesehen hat, losgelöst**

- Das einzige, was berücksichtigt werden muss, sind mögliche Konflikte zwischen den Befehlen
- Das, was mit Hilfe der Reservierungsregistern gewonnen wird, ist eine sog. **Datenflussarchitektur**
- Die Datenflussarchitektur steht im Gegensatz zur **Steuerflussarchitektur**, die durch die Hauptbefehlsschleife der von-Neumann-Maschine definiert ist

⇒ **RISC-Rechner folgen nicht mehr der Hauptbefehlsschleife einer Steuerflussarchitektur**

- Bei der Datenflussarchitektur bestimmt die Fertigstellung von Operanden und deren logische Abhängigkeiten, sowie potentielle Konflikte, wann Operanden als ALU Input verwendet werden

1.14 Das Scoreboard der vollen Ausbaustufe

- Durch das Scoreboard der vollen Ausbaustufe werden alle Datenfluss- und Ressourcenkonflikte erkannt und aufgelöst
- Ressourcenkonflikte entstehen, wenn nicht genügend Betriebsmittel wie z.B. Rechenwerke oder Reservierungsregister zur Verfügung stehen, oder wenn die Befehls-Pipelines gleichzeitig mehrere Load/Stores ausführen sollen
- Datenflusskonflikte entstehen, wenn es logische Abhängigkeiten zwischen den Registerinhalten gibt
- Datenflusskonflikte haben die Unterkategorien **Read-after-Write (RAW), Write-after-Write (WAW) und Write-after-Read (WAR)**
- WAR-, RAW- und WAW-Konflikte werden durch das Scoreboard der vollen Ausbaustufe erkannt und können oft durch reines Umordnen der Befehle behoben werden
- Die Umordnung bewirkt ein **verzögertes instruction issue and dispatch** des betroffenen Befehls
- Wenn das nicht hilft, verlängert das Scoreboard der vollen Ausbaustufe eine Phase der betroffenen Befehls-Pipeline solange, bis der Konflikt aufgelöst ist
- Meistens wird die Operandenholphase verlängert, bis eine andere Befehls-Pipeline den benötigten Operanden fertig berechnet hat
- Bei Ressourcenkonflikten wird sichergestellt, dass ein Betriebsmittel wie Register, Hauptspeicher, ALU, das gleichzeitig mehrfach benötigt wird, der Reihe nach benützt wird

- Volles Scoreboarding schließt deshalb den Tomasulo-Mechanismus mit ein

Hinweis: für Tomasulo-Mechanismus siehe Rechnerorganisation I.

- Der Tomasulo-Mechanismus bewirkt ein sog. Operand Forwarding über den Common Data Bus (CDB)
- Operand Forwarding heißt, dass berechnete Zwischenergebnisse gleich wieder in diejenige ALU bzw. Reservation Station zurückgespeist werden, in der sie gebraucht werden
- Parallel zur Berechnung neuer Werte werden die alten Zwischenergebnisse in die echten Register geschrieben, da diese nach aussen für den Programmierer über einen Debugger sichtbar sind
- Zusammen mit der Decode Unit werden die Befehle des Instruction Windows out-of-order ge-scheduled und zur Ausführung auf die ALUs verteilt (= instruction issue and dispatch)
- Volles Scoreboarding ermöglicht die gefahrlose Kopplung der Befehlspipelines mit den Arithmetik-Pipelines und mit der Speicherpipeline
- Alle Prozessoren für PCs und Workstations, die nicht Many Core sind, verwenden standardmäßig volles Scoreboarding
- Many Core CPUs sind dagegen eher einfach aufgebaut, um die Chipfläche pro Prozessor zu reduzieren

1.14.1 Der Aufbau des Scoreboard der vollen Ausbaustufe

- Das Scoreboard der vollen Ausbaustufe besteht aus drei CPU-internen Listen, die mit jedem Prozessortakt auf den neuesten Stand gebracht werden
- Diese Listen existieren in Hardware und werden von Hardware verwaltet
- Sie tragen wesentlich zur Komplexität des CPU-Chips bei, sind aber für die Realisierung der Datenflussarchitektur zusammen mit den Reservierungsregistern unverzichtbar
- Die Listen enthalten Zustände von Befehlen und von Betriebsmitteln wie ALUs und Reservierungsregister und speichern folgende Informationen:

1.) Die Zustände der Befehle im Instruction Window

2.) Die Zustände der ALUs

3.) Die Zustände der Ergebnisregister

- Die Liste der ALU-Zustände und die Liste der Ergebnisregister sind Hilfen für die Hauptliste, die Buch über die Zustände der Befehle im Instruction Window führt
- Die drei Zustandslisten haben die folgenden Aufgaben:

1.14.1.1 Die Liste der Befehlszustände

- Die Liste der Befehlszustände vermerkt für jeden betrachteten Befehl im Instruction Window, welchen momentanen Zustand er hat

- **Die Befehlszustände sind:**

1.) Befehl da? = instruction in prefetch Buffer present? ja/nein

2.) Befehl dekodiert? = Instruction Decode completed? ja/nein

3.) Operanden vollzählig da? = Operand fetch completed? ja/nein

4.) Befehl an ein Rechenwerk verteilt? = Instruction dispatch completed? ja/nein

5.) Ausführungsphase beendet? = ALU Execute completed? ja/nein

6.) Rückschreibephase beendet? = Write Back in Register completed? ja/nein

- **Die Befehlszustände werden über einen endlichen Automaten mit Ein-/Ausgabe verwaltet**
- **Die Übergänge zwischen den Befehlszuständen gehorchen bestimmten Regeln**
- **Beispielsweise wird nur dann ein neuer Befehl an eine ALU ausgegeben (issued), wenn:**

1.) kein Ressourcenkonflikt durch diesen Befehl entsteht, d.h.:
  - **eine ALU muss frei sein oder der Hauptspeicher muss frei sein, sofern benötigt**
  - **das Ergebnisregister des Befehls darf nicht bereits anderweitig als Ergebnisregister verwendet werden**
  - **dadurch werden WAW/WAR-Konflikte vermieden**

2.) alle Eingabeoperanden fertig berechnet wurden und in einem Reservierungsregister stehen
  - **Dadurch werden RAW-Konflikte vermieden**

- **Die Operanden eines Befehls sind dann berechnet, wenn in den Reservierungsregistern, in denen sie stehen, das Valid-Bit auf TRUE steht**
- **Bei einem TRUE-Bit können die Eingabeoperanden von einer ALU verarbeitet werden**
- **Dies entspricht dem Konzept der Datenflussarchitektur**
- **Ein Reservierungsregister ist solange noch nicht gültig, wie es zwar als ALU-Ausgaberegister markiert aber noch nicht von dieser ALU mit dem Ergebnis beschrieben wurde**
- **Damit die Regeln für die Zustandsübergänge der Befehle eingehalten werden können, existieren zwei Hilfslisten, die ALUs und Ergebnisregister verwalten:**

1.14.1.2 Die Liste der ALU-Zustände

- **Die Liste der ALU-Zustände vermerkt für jedes Rechenwerk und den Hauptspeicher, ob sie beschäftigt sind oder nicht**
- **Dadurch können Ressourcenkonflikte vermieden werden, da ein belegtes Rechenwerk oder ein beschäftigter Hauptspeicher für andere Operationen nicht zur Verfügung stehen**
- **Zusätzlich wird in dieser Liste vermerkt, welches Reservierungsregister für die Eingabeoperanden einer ALU zuständig ist**

- Dies löst für jede ALU die Frage, von welchem Reservierungsregister die nächsten Eingabeoperanden kommen werden

1.14.1.3 Die Liste der Zustände der Ergebnisregister

- Die Liste der Zustände der Ergebnisregister sagt für jedes Reservierungsregister, ob es als Ziel für ein ALU-Ergebnis markiert wurde = „als Ergebnisregister reserviert? ja/nein“
- Falls ja, wird noch vermerkt, von welcher ALU das Ergebnis angeliefert werden wird
- Zusammen mit den Ready Bits in den Reservierungsregistern werden dadurch RAW-Konflikte vermieden
- Außerdem wird vom Scoreboard der vollen Ausbaustufe das Schreiben in ein Reservierungsregister und das Kopieren in echte Prozessorregister, die nach außen sichtbar sind, solange verzögert, bis diese Register frei sind, also nicht mehr als ALU-Eingabeoperand- oder als Ergebnisregister für anderweitige ALU-Operationen markiert sind
- Dadurch werden WAR/WAW-Konflikte vermieden, da kein vorzeitiges Überschreiben des Inhalts eines Reservierungsregisters stattfindet

1.15 Spekulative Befehlsausführung

- Um die Geschwindigkeit eines Prozessors weiter zu steigern, kann man Befehls-Pipelining mit Superskalarität kombinieren
- Dann erhält man die sog. spekulative Befehlsausführung (= speculative execution)

- Alle RISC-Prozessoren, die nicht Many Core sind, verwenden spekulative Befehlsausführung und sind dadurch so schnell, aber auch so komplex wie nie zuvor
- Denn daraus ergibt sich leider aber auch die Addition zweier bekannter Schwierigkeiten
- Bei Superskalarität können Befehle eine andere Ausführungsreihenfolge haben als vom Programmierer vorgesehen und werden außerdem, so gut es geht, parallel ausgeführt
- Bei Befehls-Pipelining mit Sprungvorhersage werden Befehle auf Verdacht ausgeführt und können sich hinterher als falsch ausgeführt herausstellen
- Bei reiner Superskalarität sorgt das Scoreboard der vollen Ausbaustufe durch out-of-order execution oder durch Phasenverlängerung in der Befehls-Pipeline dafür, dass keine Datenfluss- und Ressourcenkonflikte auftreten
- Beim reinem Befehls-Pipelining sorgt die Sprung-/Sprungzielvorhersage dafür, dass bei IF... THEN ... ELSE der richtige Programmzweig meistens korrekt vorhergesagt wird
- Bei falscher Sprungvorhersage wird einfach die ganze Befehls-Pipeline gelöscht, bevor die Ergebnisse in die nach aussen sichtbaren Register zurückgeschrieben werden
- Diese Lösung versagt jedoch, sobald n>1 Befehls-Pipelines parallel mehrere Befehle in verwürfelter Reihenfolge ausführen, denn:
  - Bei Superskalarität sind die Rechenwerke ständig damit beschäftigt, irgendwelche Befehle von der Fetch Unit zu berechnen und die Ergebnisse in die Register zurückzuschreiben
  - Stellt sich aber heraus, dass der vorausgesagte Programmzweig falsch war (= falscher Fetch), würden falsche Werte in den Ergebnisregistern stehen

- Dies ist nicht mehr zu beheben, da die falschen Werte in den Ergebnisregistern nicht einfach gelöscht werden können, vielmehr haben sie die alten, d.h. richtigen Registerwerte bereits überschrieben

- Um dieses Problem zu lösen, ist es nötig, Zusatzmaßnahmen zu ergreifen
- Diese Zusatzmaßnahmen sind:

1.) Die Rückschreibephase der Befehls-Pipeline (Write Back) wird in zwei Teilphasen aufgespalten

2.) Eine neue Hardware-Einheit wird in die Write Back-Stufe der Befehls-Pipeline eingebaut

- Die beiden Teilphasen der Rückschreibephase heissen „Ergebnisbereitstellungsphase“ und „Ergebnisübergabephase“
- Die extra eingebaute Hardware-Einheit heißt „Reorder Buffer“

1.15.1 Der Reorder Buffer

- In der Ergebnisbereitstellungsphase innerhalb von Write Back werden Ergebnisse solange im Reorder Buffer zwischengespeichert, bis von einer ALU berechnet wurde, ob der IF- oder der ELSE-Zweig im Programm ausgeführt werden muss
- In der Ergebnisübergabephase von Write Back wird der Inhalt des Reorder Buffers in die nach außen sichtbaren CPU-Register geschrieben, sofern der Reorder Buffer gültig ist

- Zusätzlich wird in der Ergebnisbereitstellungsphase die Reihenfolge der berechneten Ergebnisse wieder in die ursprüngliche Aufschreibreihenfolge des Programms gebracht (in-order)
- Daher rührt auch die Namensgebung „Reorder Buffer“, da in ihn „in-order“ zurückschrieben wird
- Zur Wiederherstellung der Ergebnisse in der Kode-Aufschreibreihenfolge bekommt jedes zukünftige Ergebnis denjenigen Speicherplatz im Reorder Buffer zugewiesen, der die sequentiellen Kode-Aufschreibreihenfolge wiederherstellt

⇒ Die Zuweisung der Speicherplätze im Reorder Buffer erfolgt in der Reihenfolge, wie die Sequenz der Befehle, die der Programmierer kodiert hat

- Der Reorder Buffer ist außerdem so groß gewählt, dass er ALU-Ergebnisse ausreichend lange zwischenspeichern kann, bis feststeht, ob IF- oder ELSE richtig ist
- Des weiteren hat der Reorder Buffer mindestens die Größe des Instruction Windows, d.h. mindestens die Größe des jeweils betrachteten Programmausschnitts

⇒ Es könnten theoretisch alle Befehle des Instruction Windows gleichzeitig berechnet und die Resultate im Reorder Buffer in der Kode-Aufschreibreihenfolge zwischengespeichert werden. So viele ALUs gibt es allerdings in keiner CPU. Das würde i.A. auch an den zahlreichen Abhängigkeiten und Konflikten zwischen Befehlen scheitern.

- Spätestens wenn der Reorder Buffer voll ist, muss auch die Auswertung der Sprungbedingung fertig sein, sonst läuft der Reorder Buffer über

- Sobald eine ALU die Sprungbedingung hinter dem IF ... THEN .... ELSE ausgewertet hat, kann der Reorder Buffer in die Register geschrieben oder gelöscht werden
- Der Reorder-Buffer ist damit das Gegenstück des Prefetch Buffers, da er die Kode-Aufschreibreihenfolge wiederherstellt, die im Prefetch Buffer aufgelöst wurde

1.15.1.1 Zusammenfassung der Aufgaben des Reorder Buffers

1.) In der Ergebnisbereitstellungsphase werden von den ALUs Resultate berechnet, aber nicht sofort in nach außen sichtbare CPU-Register zurückgeschrieben, sondern im Reorder-Buffer zwischengespeichert

2.) Sobald eine der ALUs die Sprungbedingung von IF ... THEN .... ELSE ausgewertet hat, ist klar, ob der Reorder Buffer einen gültigen Inhalt hat oder nicht

3.) Ist sein Inhalt ungültig, wird er zusammen mit den Befehls-Pipelines gelöscht, und die richtige Sequenz von Nachfolgebefehlen wird aus dem 1st Level-Befehlscache geholt

4.) Da der Reorder Buffer nach außen, d.h. zum Programmierer hin, nicht sichtbar ist, kann er problemlos gelöscht werden, ohne wichtige Daten zu verlieren

5.) Ist der Reorder Buffer hingegen gültig, wird sein Inhalt Eintrag für Eintrag in die CPU-Register kopiert. Dies ist die Ergebnisübergabephase (Instruction Commit).

Hinweis: Falls sog. Register Renaming vorliegt, kann sogar auf das Kopieren verzichtet werden.

6.) Punkt 5.) geschieht in der Aufschreibreihenfolge der Befehle im Programmkode, damit der Debugger, und somit der Programmierer, den Eindruck erhält, als ob er eine von-Neumann-Maschine mit Steuerflussarchitektur programmieren würde

Hinweis: Die nach aussen sichtbaren Register der CPU bleiben so stets in einem konsistenten Zustand.

⇒ Die Ausführung der Befehle erfolgt out-of-order, die Übergabe der Resultate an die sichtbaren CPU-Register jedoch in-order

1.15.1.2 Präzise Interrupts durch Reorder Buffer

- Durch out-of-order execution und in-order write back ergibt sich ein wichtiger Nebeneffekt:
- Die CPU kann jederzeit von einem Interrupt unterbrochen werden, ohne dass der nach außen sichtbare Zustand der CPU-Register inkonsistent ist
- D.h., es gibt nicht den Fall, dass beim Eintreffen des Interrupts manche Register mit Resultaten schon beschrieben sind, andere aber noch nicht, obwohl sie im Programm weiter vorne stehen
- Man spricht deshalb auch von präzisen Interrupts
- Ohne ein Reorder Buffer sind präzise Interrupts nicht zu realisieren
- Ohne präzise Interrupts wiederum sind Gerätetreiber, Betriebssysteme und Steuerungen und Regelungen nicht zu programmieren
- Sie sind deshalb wichtig

1.15.1.3 Sonderrolle von Load/Store und Sprüngen

- Im Reorder Buffer werden u.a. auch die Ergebnisse von indirekten Sprüngen (=„Springe nach Inhalt von Pointer“), von bedingten Sprüngen (=„Springe, wenn ... gilt) und von unbedingten Sprüngen (=„Springe nach ...“), sowie von Load/Store-Befehlen zwischengespeichert
- Jeder Sprung bestimmt die Nachfolgeadresse für den nächsten Instruction Fetch
- Load und Store enthalten die Adressen für das Lesen von Operanden aus dem langsamen Hauptspeicher bzw. für das Schreiben von Ergebnissen in den langsamen Hauptspeicher
- Da Load und Store sehr lange dauern, werden sie im Reorder-Buffer hoch priorisiert
- Um das zu ermöglichen, hat der Reorder Buffer eine dreigliedrigen Aufbau

1.15.1.4 Aufbau des Reorder Buffers

- Jedes Wort im Reorder Buffer, d.h. jede Zeile besteht aus drei Feldern (= Einträgen):

- Die Gründe für den dreigliedrigen Aufbau sind:

  - Die Nachfolgeadressen für den jeweils nächsten Fetch müssen so schnell wie möglich der Fetch-Unit mitgeteilt werden, damit diese Unit ohne Verzögerung den richtigen Nachfolgebefehl holen kann
  - Die Nachfolgeadressen müssen außerdem der Branch Prediction Unit so schnell wie möglich mitgeteilt werden, damit diese einen Update ihrer Branch History Table vornehmen kann und alsbald wieder für eine Sprungvorhersage zur Verfügung steht
  - Load und Store-Befehle müssen frühzeitig dem Load bzw. Store Buffer mitgeteilt werden, um unnötige Latenzen durch einen langsamen Hauptspeicherzugriff zu vermeiden
- Im Befehlstypfeld wird deshalb vermerkt, ob es sich um das Ergebnis eines normalen Befehls oder eines Sprungbefehls oder eines Load/Store-Befehls handelt (= 3 Befehlstypen)
- Je nach Befehlstyp haben die beiden anderen Felder unterschiedliche Inhalte
- Im Zieladressenfeld ist abhängig vom Befehlstyp folgende Information gespeichert:
  - im Falle eines Load/Store-Befehls steht darin die Hauptspeicheradresse für das Lesen bzw. das Schreiben
  - im Falle eines Sprungbefehls steht darin die Nachfolgeadresse im Programm für den nächsten Instruction Fetch
  - im Falle eines normalen Befehls steht darin die Registeradresse für Operanden-Lesen bzw. -Schreiben
- Das Resultat-Feld enthält abhängig vom Befehlstyp folgende Information:
  - im Falle eines Load-Befehls steht darin der Inhalt, der unter der Hauptspeicheradresse gelesen wurde

- im Falle eines Store-Befehls steht darin der Inhalt, der in die Hauptspeicheradresse zu schreiben ist
- im Falle eines normalen Befehls steht darin der Inhalt, der in die Registeradresse zu schreiben ist
- im Falle eines bedingten Sprungbefehls steht darin das Resultat der ausgewerteten Sprungbedingung für IF ... THEN .... ELSE, d.h. TRUE oder FALSE

- Zeigt das Befehlstypfeld an, dass ein Eintrag im Reorder Buffer zu einem Sprungbefehl oder zu einem Load/Store-Befehl gehört, teilt dies der Reorder Buffer sofort der Fetch Unit und auch der Branch Prediction Unit bzw. dem Load/Store Buffer mit
- Diese Einheiten werten dann die Zieladresse und das Resultatfeld noch in der Ergebnisbereitstellungsphase aus und reagieren entsprechend, um Zeit zu sparen
- Wird der Reorder Buffer gelöscht, wird dies ebenfalls der Fetch Unit und der Branch Prediction Unit bzw. dem Load/Store Buffer sofort mitgeteilt, damit diese darauf reagieren können, um Zeit zu sparen
- Nach einem Löschen des Reorder Buffers stellen Fetch Unit, Branch Prediction Unit bzw. Load/Store Buffer ihre gerade laufenden Aktionen, die sich auf Inhalte des Reorder Buffers beziehen, nicht fertig, so dass nichts Falsches geschieht

1.16 Eager Execution

Hinweis: eager: eifrig, dienstbeflissen

- Wenn sehr große Ressourcen an Transistoren auf dem Chip zur Verfügung stehen, dann kann man darauf verzichten, eine Sprungvorhersage durchzuführen
- Dadurch wird die CPU strukturell einfacher, weil der Reorder Buffer simpler wird und weil die Sprung-/Sprungzielvorhersage entfallen kann
- Ein weiterer Vorteil von Eager Execution, ist, dass eine Invalidierung einer Befehls-Pipeline aufgrund einer falschen Sprungvorhersage nicht mehr nötig ist, weil es die Sprungvorhersage nicht mehr gibt
- Die Zahl der Transistoren wird dennoch größer, weil bei Eager Execution sehr viel mehr Register benötigt werden
- Register sind allerdings im Vergleich einfach aufgebaut, so dass sie ohne besondere Designschwierigkeiten auf dem Chip implementiert werden können
- Bei Eager Execution werden bei einem bedingten Sprungbefehl gleichzeitig die ersten 5-25 Statements beider Programmzweige (IF und ELSE) in die CPU geladen, dekodiert und ausgeführt
- Eager Execution erfordert in der Fetch-Phase doppeltes Befehlsholen und in der Decode-Phase doppeltes Befehldekodieren
- Danach wird damit begonnen, die Eingabeoperanden für beide Zweige in einem doppelten Satz von Reservierungsregistern zu sammeln
- Dazu hat jedes Rechenwerk einen zweiten Satz von Reservierungsregistern in Form von sog. Schattenreservierungsregistern RR'_0, RR'_1, RR'_2, ...

- Freie ALUs werden herangezogen, um die Eingabeoperanden der Schattenreservierungsregister zu bearbeiten und ihre Ergebnisse in die Schattenreservierungsregister zurückzuschreiben
- Ansonsten läuft der Betrieb bei den normalen Reservierungsregistern weiter wie bisher, außer dass nur noch halb so viele ALUs zur Bearbeitung der Eingabeoperanden zur Verfügung stehen
- Verdoppelt man auch die Zahl der ALUs, spielt auch dies keine Rolle mehr
- Während des Operandensammelns und Befehlbearbeitens wird die Sprungbedingung von einer der ALUs ausgewertet
- Dies klappt allerdings nur, wenn ausreichend viele Reservierungsregister und Schattenreservierungsregister zur Verfügung stehen, in denen lange genug gesammelt und zurückgeschrieben werden kann, bis die Sprungbedingung ausgewertet ist
- Stellt sich heraus, dass der Zweig, dessen Eingabeoperanden in den Schattenreservierungsregister gespeichert sind, der richtige ist, wird der Inhalt dieses Registersatzes an den Reorder Buffer weitergegeben
- Ansonsten werden die normalen Reservierungsregister an den Reorder Buffer weitergegeben

1.16.1 Nachteil von Eager Execution:

- In beiden Programmzweigen können erneut bedingte Sprungbefehle enthalten sein, für die ebenfalls doppeltes Befehlholen und Dekodieren nötig wäre

⇒ eine exponentielle Zunahme der benötigten Hardware-Ressourcen ist die Folge, was in der Praxis ab einer Vervierfachung der Ressourcen nicht mehr tragbar ist

1.17 Register Renaming

Def.: Echte Register sind nach aussen über den Debugger sichtbar und können über Assemblerbefehle adressiert werden, (Schatten)Reservierungsregister hingegen nicht.

- (Schatten)Reservierungsregister erhalten ihre Inhalte entweder von einem echten Register oder von einem ALU-Ausgang über den Common Data Bus
- Das Kopieren der Inhalte in echte Register kostet Zeit
- Deshalb existiert bei einigen RISC CPUs das Konzept des Registerumbenennens (Register Renaming), um die Programmausführung zu beschleunigen
- Dabei werden echte CPU-Register, die zu kopierende Eingabeoperanden enthalten, in den Operandenteil von (Schatten)Reservierungsregistern umbenannt, so dass das Kopieren entfällt

⇒ Register Renaming ist die Umbenennung einiger nach aussen sichtbarer Register in den Operandenteil von Reservierungsregistern, die nach aussen nicht sichtbar sind

- Die Umbenennung erfolgt zur Laufzeit, und zwar bevor die ALU, die zu den Reservierungsregistern gehört, die Eingabeoperanden benötigt

- **Die Gegenrichtung, d.h. das Umbennen des Operandenteils von (Schatten)Reservierungsregister in echte Register ist komplizierter, da die echten Register ihre Inhalte nur in-order ändern dürfen und deshalb der Reorder Buffer existiert**
- **Für die Umbenennung in Gegenrichtung sind weitere Zusatzeinrichtungen erforderlich, die hier nicht beschrieben werden**
- **Im Endergebnis ist ein echtes Register einmal Teil eines (Schatten)Reservierungsregisters und ein anderes Mal ein nach aussen sichtbares CPU-Register**

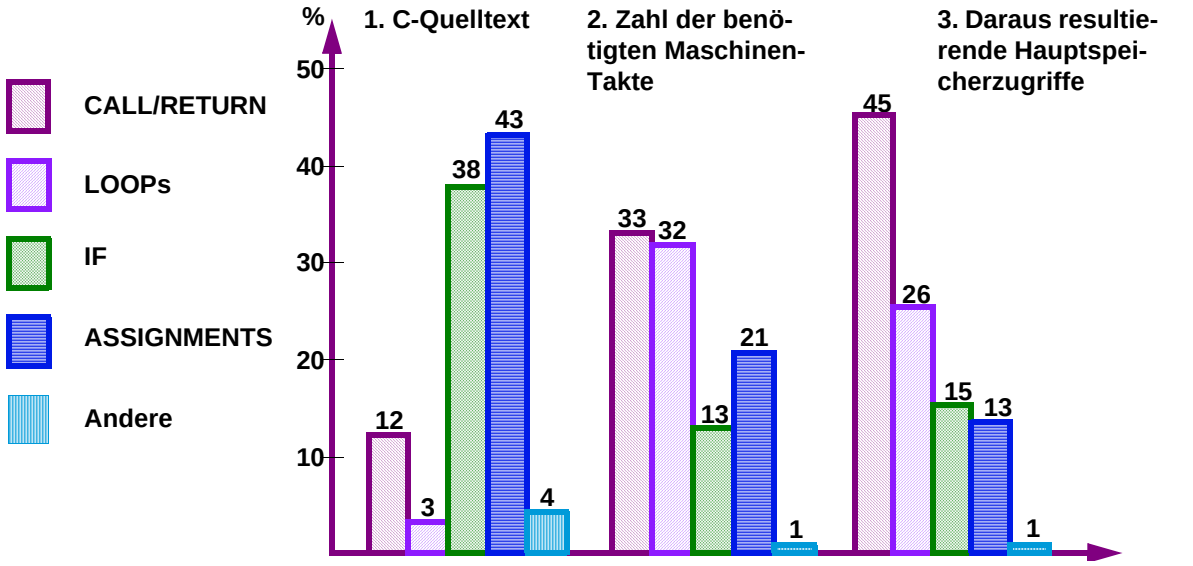
Beispiel: Die Pentium-Architektur hat keine Eager Execution aber Register Renaming

1.18 Registerfenster

- **Unterprogrammaufrufe übergeben Aufrufparameter vom rufenden an das gerufene Programm**
- **In Gegenrichtung werden Ergebniswerte vom gerufenen an das rufende Programm übergeben**
- **Beide Übergaben erfolgen automatisch durch die Assemblerbefehle CALL und RETURN**
- **Diese Befehle benützen dazu den Benutzer-Heap oder den Betriebsystem-Stack und greifen über Push/Pop-Operationen darauf zu**
- **Stacks liegen jedoch überweigend im Hauptspeicher, da sie aufgrund ihrer Größe nicht komplett ge-cached werden können, so dass ein Zugriff darauf viel Zeit kostet**

- **Registerfenster (Register Windows) dienen zur Beschleunigung von Unterprogrammaufrufen**
- **Sie wurden kommerziell bei Rechnern der Fa. Sun, jetzt Fa. Oracle, eingesetzt**
- **Um den Nutzen von Registerfenstern zu ermitteln wurden statistische Analysen der Häufigkeitsverteilung von Befehlen in C-Programmen vorgenommen**

1.18.1 Häufigkeitsverteilung von Befehlen in C-Programmen



Hinweis: [aus P. Horster u. a: RISC - Konzepte und Realisierungen; Hüthig-Verlag Heidelberg 1987]

- **Das Resultat der Analysen ist:**

1.) ca. 45% des Verkehrs mit dem Hauptspeicher entfallen auf Prozeduraufrufe, d.h. auf Stack Push/Pops zur Parameterübergabe aufgrund von CALL und auf Stack Push/ Pops für die Rückkehradresse und die Ergebniswerte durch RETURN

⇒ **Der Zeitbedarf für Unterprogrammaufrufe ist hoch, da der Hauptspeicher langsam ist.**

2.) Schleifen (Loops) erzeugen darüberhinaus zuzsätzliche ca. 26% des Speicherverkehrs aufgrund von Cache Misses

⇒ **Schleifenkörper sollten nicht zu lange sein, damit sie in den Befehls- und/oder Daten-Cache passen = Hinweis an Software-Entwickler!**

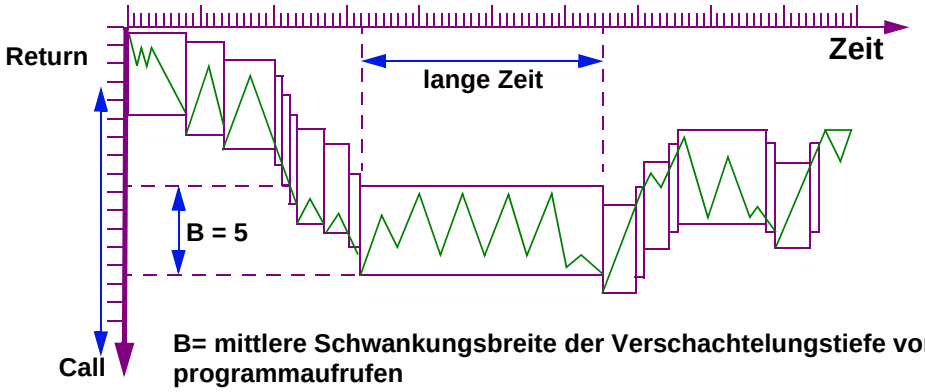
1.18.1.1 Verschachtelungstiefe

Def.: Die Verschachtelungstiefe von Unterprogrammaufrufen gibt an, in wie vielen gleichzeitigen Unterprogrammaufrufen sich ein Hauptprogramm befinden kann.

Beispiel: A ruft B, B ruft daraufhin C => Verschachtelungstiefe = 2

- **Betrachtet man alle Stellen des Hauptprogramms, an denen eine Unterprogrammaufruf erfolgt, dann ergeben sich je nach Stelle unterschiedliche Verschachtelungstiefen**
- **Eine zeitliche Analyse der Verschachtelungstiefen von Programmen ergibt das nachfolgende Beispielbild**

1.18.2 Zeitliche Analyse der Verschachtelungstiefen



B= mittlere Schwankungsbreite der Verschachtelungstiefe von Unterprogrammaufrufen

- Die zeitliche Analyse der Verschachtelungstiefen ergibt ein besonderes Lang- und Kurzzeitverhalten von Unterprogrammaufrufen: die mittlere Schwankungsbreite B der Verschachtelungstiefe beträgt nur ca. 5
- Es gibt rel. lange Codeabschnitte, in denen B=5 nicht überschritten wird (im obigen Diagramm durch rote Bänder gekennzeichnet)

⇒ Die Zahl der Unterprogrammaufrufe und damit der Zahl der CALL/RETURN-Übergaben von Parametern und Rücksprungadressen ist für bestimmte Zeitabschnitte klein

- Idee: die Variablenübergabe bei Prozeduraufruf/-rückkehr erfolgt nicht mehr durch den Stack im Hauptspeicher, sondern durch spezielle Register in der CPU, weil Register viel schneller sind
- Diese extra Register werden Übergaberegister genannt
- Übergaberegister für ca. 5 Prozeduren sind aufgrund des gemessenen B in der Regel ausreichend, um für lange Zeitabschnitte im Programm alle notwendigen CALL/RETURN-Übergabeparameter zu speichern
- Der Stack dient bei Registerfenstern nur noch zur Speicherung der Rückkehradresse ins rufende Programm und zur Speicherung des Prozessorstatusworts

⇒ Viele Transfers vom/zum Stack entfallen, d.h. Hauptspeicherzugriffe werden reduziert

- Insgesamt werden pro Prozedur folgende Registersätze bereitgestellt:
  - ein erster Registersatz für alle lokal deklarierten Variablen der Prozedur
  - ein zweiter Registersatz zur Parameterübergabe zwischen zwei Prozeduren
  - ein dritter Registersatz für global deklarierte (= gemeinsame) Variable aller Prozeduren
- Damit kann das sog. „Register-Window“-Konzept realisiert werden:

Hinweis: Verwechseln Sie nicht das Register-Window mit dem Instruction Window

  - Die Parameterübergabe geschieht dadurch, dass die Übergaberegister von rufender und aufgerufener Prozedur identisch sind (= Registerüberlappung)
  - Das Register-Window ist die Schnittmenge gemeinsamer Register zwischen rufender und aufgerufener Prozedur

- Es wird bei der Parameterübergabe null Mal kopiert und null Mal auf den Hauptspeicher zugegriffen

- Da alles, was das Unterprogramm an Input braucht, bereits in den Prozedur-eigenen Registern vorliegt, kann das Unterprogramm sofort starten, d.h. ohne vorhergehenden Parameter-Pop vom Stack
- Zusätzlich gilt: Register werden CPU-intern über sog. Unit-Adressen angesprochen

Hinweis: Unit-Adressen sind bereits von den Reservierungsregistern, d.h. von RO I bekannt

- D.h. jedes Register hat eine CPU-interne, eindeutige Unit-Adresse

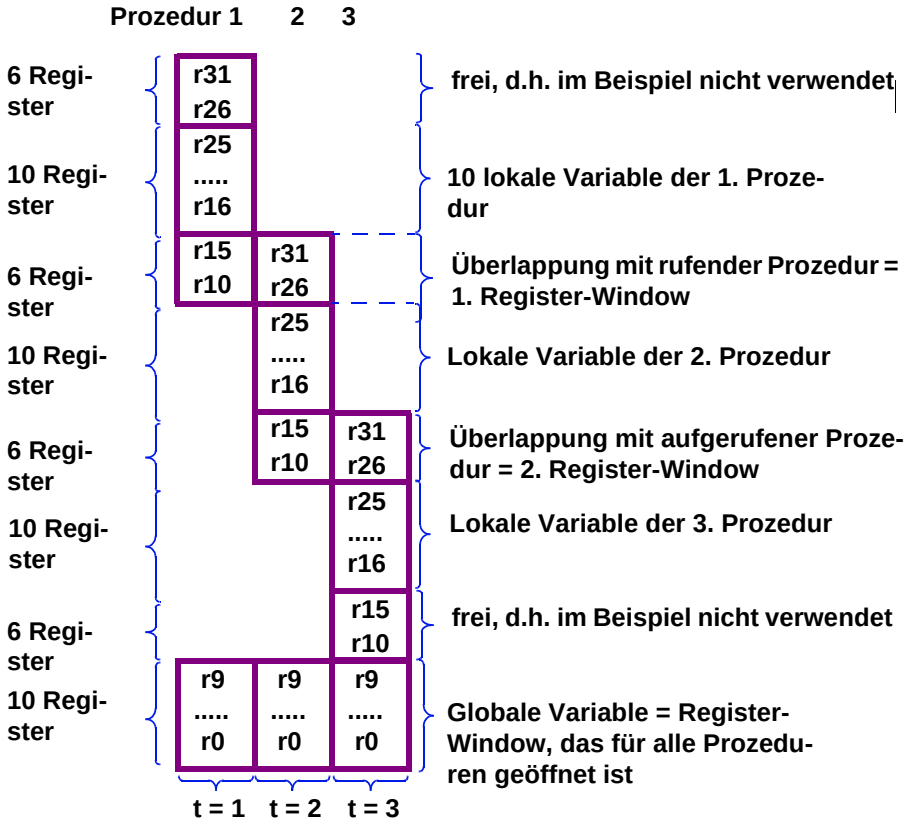
⇒ Die Implementierung von Register-Windows erfolgt einfach durch eine adressmäßige Überlappung von Unit-Adressen. Verschiedene Unit-Adressen liegen auf denselben Registern.

- Die Anzahl der benötigten Register für die Überlappung wird vom Compiler anhand der Anzahl der Übergabeparameter im Programmcode ermittelt
- Darüberhinaus implementiert der Compiler die überlappenden Register zwischen zwei Prozeduren mit Hilfe von zwei Zeigern, die auf verschiedene Unit-Adressen zeigen
- Diese Pointer heißen Upper and Lower Window Pointer der Prozedur
- Ist die Zahl von Parametern, die von Prozedur 1 nach Prozedur 2 übergeben werden, groß, bewirkt dies eine große Differenz der beiden Pointerwerte
- Im folgenden zeigt ein Beispiel, wie durch Registerfenster Parameter übergeben werden

- Zur Vereinfachung wird im Beispiel angenommen, dass die Unit-Adressen streng monoton aufsteigend sind
- Dies stellt eine Vereinfachung dar, die später aufgehoben wird
- Die Konsequenz dieser Vereinfachung ist, dass die Register linear angeordnet sind („ein Register nach dem anderen“)

Beispiel: Gegeben sei ein Prozessor mit 32 Registern pro Prozedur und folgender Programmkonstellation: Prozedur 1 ruft Prozedur 2, Prozedur 2 ruft Prozedur 3. Die Prozeduren übergeben ihre Parameter durch Register-Windows. Jede Prozedur habe die Unit-Adressen r10-r31 für ihre Register. Zusätzlich gebe es gemeinsam genutzte Variable für alle 3 Prozeduren in den Registern mit Unit-Adressen r0-r9.

1.18.3 Linear angeordnete Register



- Das obige Prinzipschaubild funktioniert zwar, aber es gibt verschiedene Probleme bei linear angeordneten Registern:
  - Zwar haben die Register jedes Registerfensters verschiedene Unit-Adressen, die sich überlappenden Register sind aber physikalisch identisch. Z.B. sind im Beispiel r15 von Prozedur 1 und r31 von Prozedur 2 adressmäßig verschieden aber physikalisch identisch
  - Andererseits existieren dieselben Unit-Adressen gleich mehrfach für unterschiedliche Register. Z.B. gibt es im Prinzipschaubild die Adresse r25 drei Mal.

⇒ **Zusatzaufwand zur korrekten Zuordnung von logischen Unit-Adressen zu physikalischen Registern nötig**

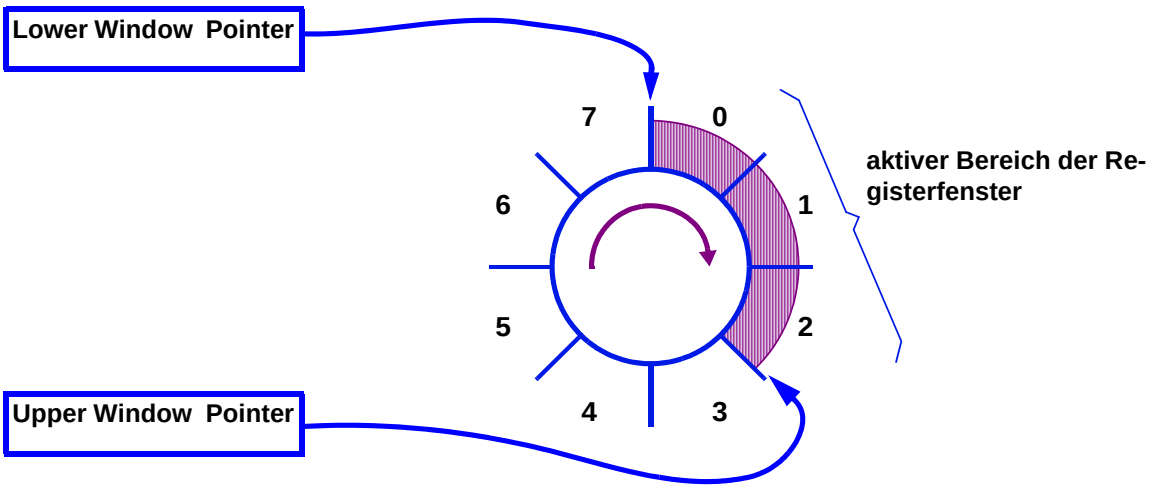
1.18.4 Zirkular angeordnete Register

- Zur Lösung der Probleme werden die Register nicht linear sondern modulo-mäßig adressiert, das entspricht zirkular angeordneten Registern
- Die zirkulare Anordnung bewirkt ein Aufheben des streng monotonen Ansteigens von Unit-Adressen, die eine Prozedur zur Verfügung hat
- Die zirkulare Anordnung hat außerdem einen sog. Ringpuffer (= „Modulo“-Puffer) zur Folge

Hinweis: Der Ringpuffer ist genauso aufgebaut, wie es beim Schiebefensterprotokoll der Vorlesung Rechnernetze I beschrieben wurde

- Es gibt einen aktiven Bereich innerhalb des Ringpuffers, der durch die Differenz zwischen Upper and Lower Window Pointer definiert ist und in dem die Prozedur arbeitet

- Der aktive Bereich im Ringpuffer dreht sich z.B. im Uhrzeigersinn durch den Ringpuffer, indem die Werte beider Pointer zunehmen, bis die LSBs der Pointer überlaufen
- Danach fangen die Pointer wieder von vorne an zu zählen (= Modulo-Puffer)



- Zusätzlich vergrößert oder verkleinert sich dabei der aktive Bereich je nach Pointer-Differenz

Hinweis: Der einzige Unterschied zum Schiebefensterprotokoll von RN I ist, dass jetzt der Ringpuffer nicht im Hauptspeicher liegt, sondern durch Register implementiert wird

- Oberhalb der Größe des Puffers werden höhere Unit-Adressen wieder auf niedere Register abgebildet

⇒ es gibt mehr logische Unit-Registerdressen als physikalische Register vorhanden sind

- Auch bei der modulo-mäßigen Adressierung der Register mit logischen Adressen gilt:
  - wenn Registerfenster mit mindestens 5 überlappenden Übergaberegistern vorhanden sind, wird meistens der Hauptspeicher-Stack zum Übergeben von Aufrufparametern nicht benötigt
- Es gibt leider auch bei zirkular angeordneten Registersätzen noch ein Problem:
  - Es existiert keine Trennung zwischen Übergabeparametern, die von Betriebssystemprozeduren übergeben werden, und zwischen denen, die von Benutzerprozeduren übergeben werden

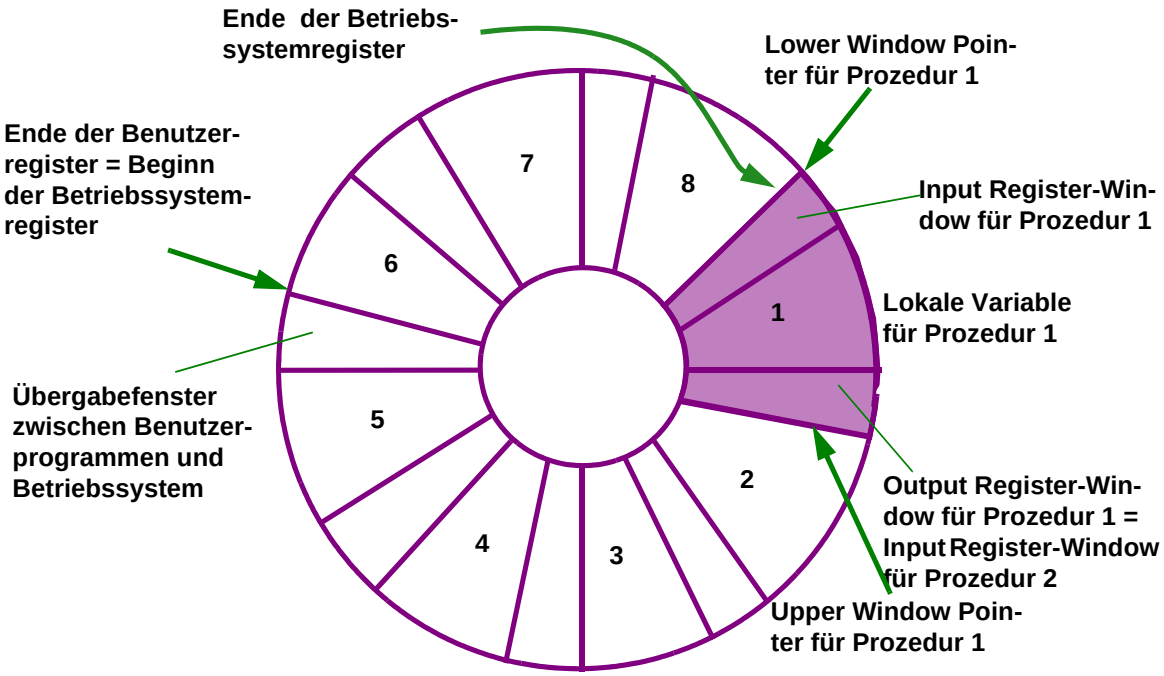
⇒ Die Isolation von Benutzerdaten und Betriebssystemdaten ist ungenügend

- Zur Lösung dieses Problems werden bei der Adressierung für Betriebssystemdaten sog. reservierte Unit-Adressen übersprungen, sobald eine Benutzerprozedur ausgeführt wird
- Dies wird erreicht durch ein Unit-Adressengenerator, der Sprünge in die Unit-Adressen einbaut

⇒ Die Benutzerprozedur kann auf die übersprungenen Unit-Adressen nicht zugreifen. Sie stehen exklusiv dem Betriebssystem zur Parameterübergabe zur Verfügung

1.18.5 Zirkulare Register mit Unit-Adressengenerator

Beispiel: Es gebe insgesamt 8 Registersätze zu je 16 Registern. Registersatz 1-5 sei für 5 Unterprogramme des Benutzers reserviert. Registersatz 6-8 sei für Unterprogramme des Betriebssystems reserviert



61

- Bei modul-mäßigen Unit-Adressen kann das Registerfenster folgendermaßen sehr einfach implementiert werden, was einer der Vorteile des Ringpuffers darstellt:
  - Wenn der Upper Window Pointer der rufenden Prozedur einen größeren Wert als der Lower Window Pointer der gerufenen Prozedur hat, kommt es zu der gewünschten Überlappung der Unit-Adressen, d.h. zum Registerfenster

1.18.6 Bewertung von Registerfenstern

- Vorteil: durch die Registerfenster-Technik sinkt der Anteil der Hauptspeicherzugriffe von ca. 45% auf ca. 20%

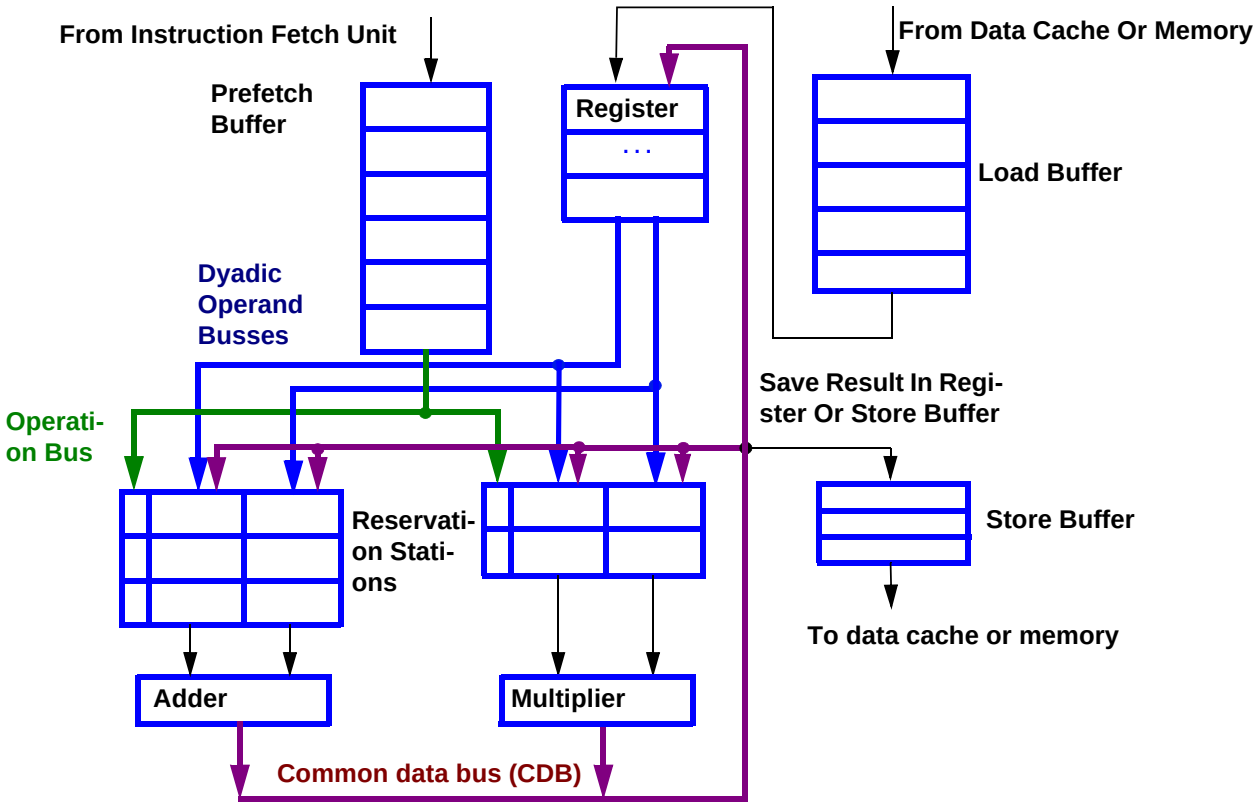
⇒ **Die Registerfenstertechnik erhöht die Systemleistung signifikant**

- Nachteile: die Registerfenstertechnik erfordert viele Register in der CPU (bis ca. 512)
- Da die Unit-Adressen von Betriebssystem-Registerfenster automatisch übersprungen werden müssen, um eine Isolation zum Betriebssystem zu erzielen, erhöht sich die Komplexität des Unit-Adressengenerators
- Zirkulare Registerfenster werden deshalb nur bei wenigen RISC-Prozessoren eingesetzt

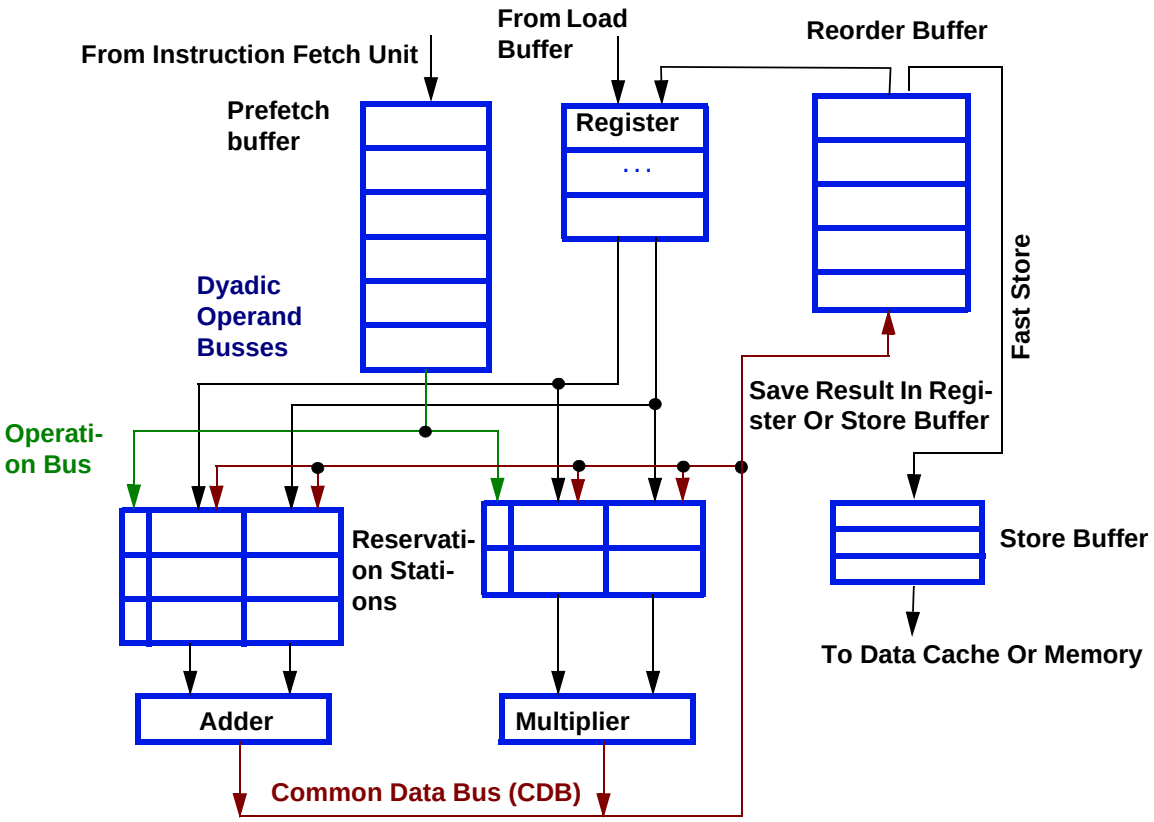
1.19 Beispiel für RISC-Prozessorarchitekturen

- Nachfolgend werden Beispiele kompletter RISC-Prozessorarchitekturen gezeigt

1.19.1 Einfache, superskalare, Tomasulo-basierte CPU

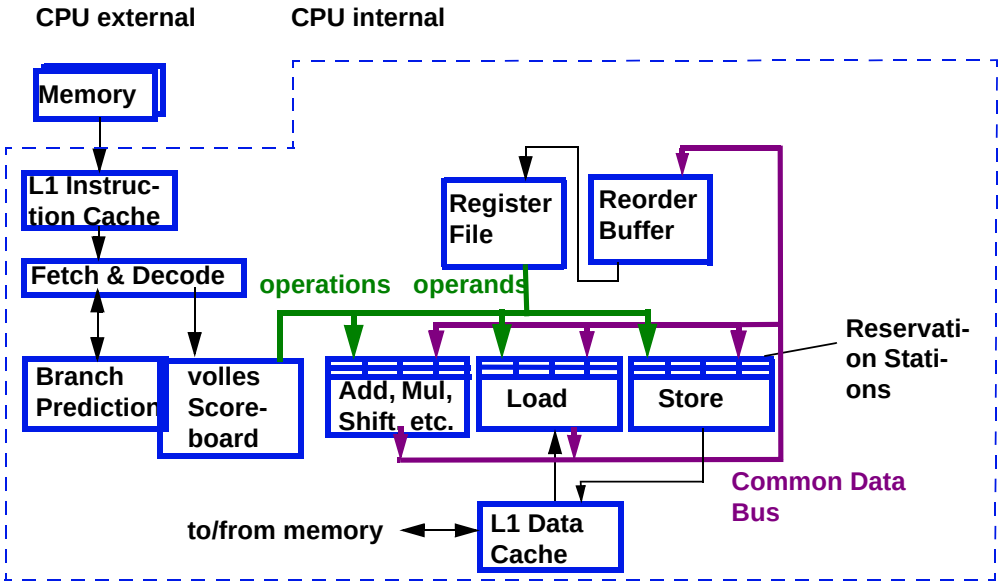


1.19.2 RISC-CPU mit spekulativer Befehlsausführung



1.19.3 Gesamtschaltbild RISC-CPU

- Nachfolgend ist das grobe Gesamtschaltbild einer einfachen, superskalaren RISC-CPU zu sehen, wie sie kommerziell vertrieben wurde



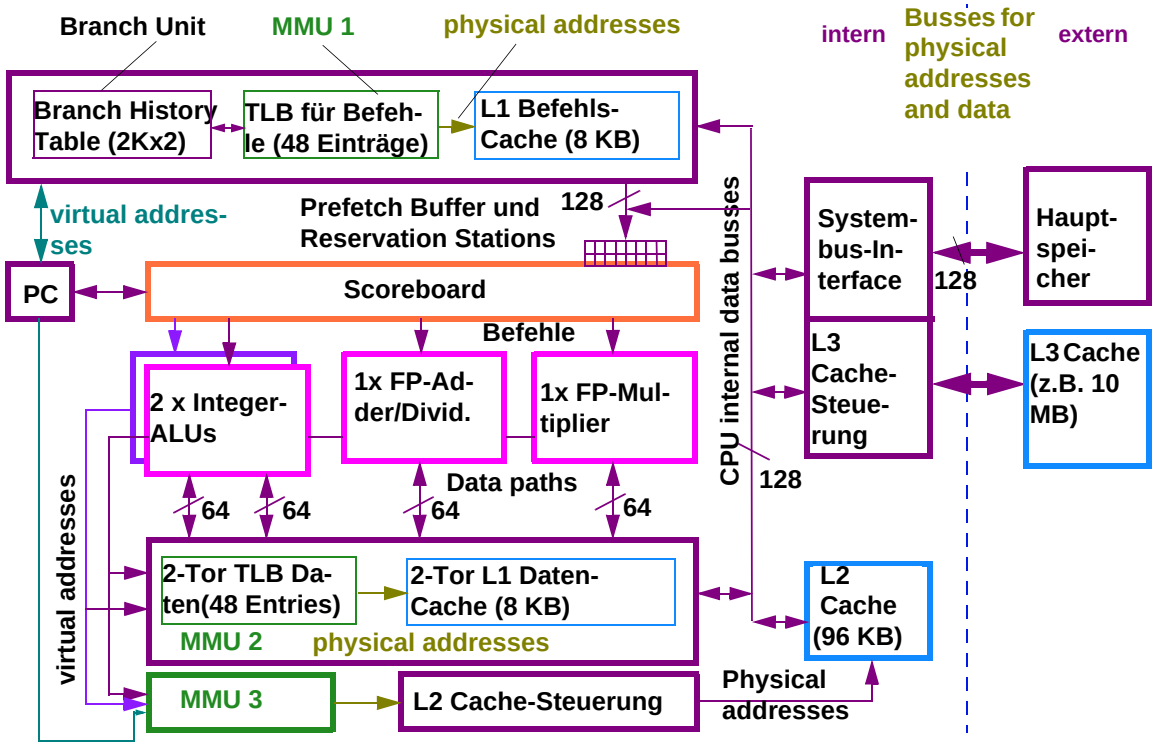
Hinweis: der Prefetch Buffer ist hier nicht eingezeichnet. Die Load- und Store Buffer sind Teil der Load- bzw. Store Unit. Die Reservation Stations sind nicht beschriftet.

- In einer weiteren Stufe der Komplexität müssen physikalische von virtuellen Adressen unterschieden werden

1.19.4 Speicherverwaltungseinheiten (MMUs) für RISC-CPUs

- Der Befehlszähler (Program Counter) eines Cores gibt nur virtuelle Adressen aus, genau wie die Load/Store-Befehle
- Da RISC Cores eine Harvard Architecture (=split Caches) für Befehle und Daten sowie Load/Store-Einheiten haben, die alle parallel arbeiten, werden mehrere MMUs benötigt, um die virtuellen Adressen in reale Hauptspeicheradressen umzusetzen
- Im Idealfall hätte jeder Cache eines Cores eine eigene MMU, ebenso wie die Load- und die Store Unit
- Die ALUs greifen auf die Datencaches zu, solange wie die Operanden noch nicht in den Registern/Reservierungsregistern und auch nicht im Load Buffer zur Verfügung stehen
- Da ALUs parallel arbeiten bedeutet dies, dass am besten auch jedes Rechenwerk eine eigene MMU hätte, um auf die Datencaches zugreifen zu können
- Da zu viele MMUs technisch zu aufwendig sind, werden real nur ca. 3 MMUs eingebaut, aber einige davon haben mehrere Ports (sog. Multiport Caches oder Vieltor-Caches)
- Jeder Port erlaubt eine eigene Adressumsetzung, gleichzeitig zu anderen Ports

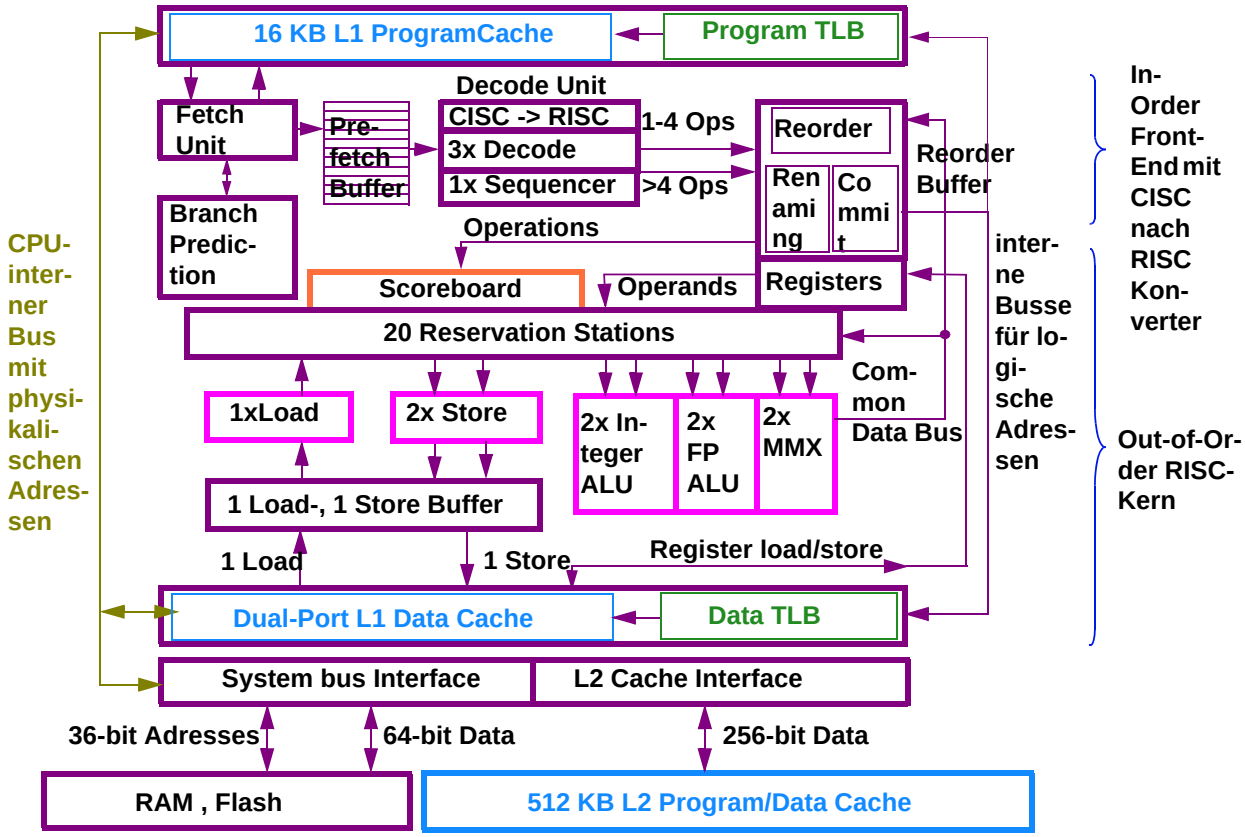
1.19.4.1 Beispiel des RISC-Prozessors HP Alpha 21264



Hinweis: TLB = translation look-aside buffer; PC = Program Counter; FP = floating point; 2-Tor Daten TLB = Dual Port TLB für Daten. Fetch, Decode, Register etc. sind aus Platzgründen weggelassen!

- **Der HP Alpha 21264 wurde in Arbeitsplatzrechnern (workstations) eingesetzt und weist folgende Merkmale auf:**
  - **Getrennte Befehls- und Daten-Caches (Harvard-Architektur)**
  - **Getrennte Speicherverwaltungseinheiten (MMUs) für Befehls- und Daten-Caches**
  - **Sprung- und Sprungzielvorhersage (Branch prediction)**
  - **Superskalarität**
  - **Out-of-order Execution**
  - **Spekulative Befehlsausführung**
  - **Pipelines für Befehle, Speicherzugriffe und Rechnen**
  - **Superpipelining durch Kopplung der Befehl-Pipelines mit den ALU-Pipelines und der Speicherpipeline**

1.19.4.2 Kombinierte RISC/CISC-Architektur der Intel x-86 CPUs



2 Die Organisation von Parallelrechnern

- Im folgenden werden Multi/Many Core-Prozessoren, Multiprozessoren, Multicomputer und Cluster-Computer definiert, die Spielarten der Parallelrechner sind

2.1 Multi Core-CPU

Def.: Multi Core-CPU: ein Multi/Many Core-Prozessor enthält auf einen einzigen Chip bis ca. 16 voneinander unabhängige, vollständige Prozessoren (Cores), bestehend aus Rechenwerken + Steuerwerk (=Multiple Instruction Multiple Data).

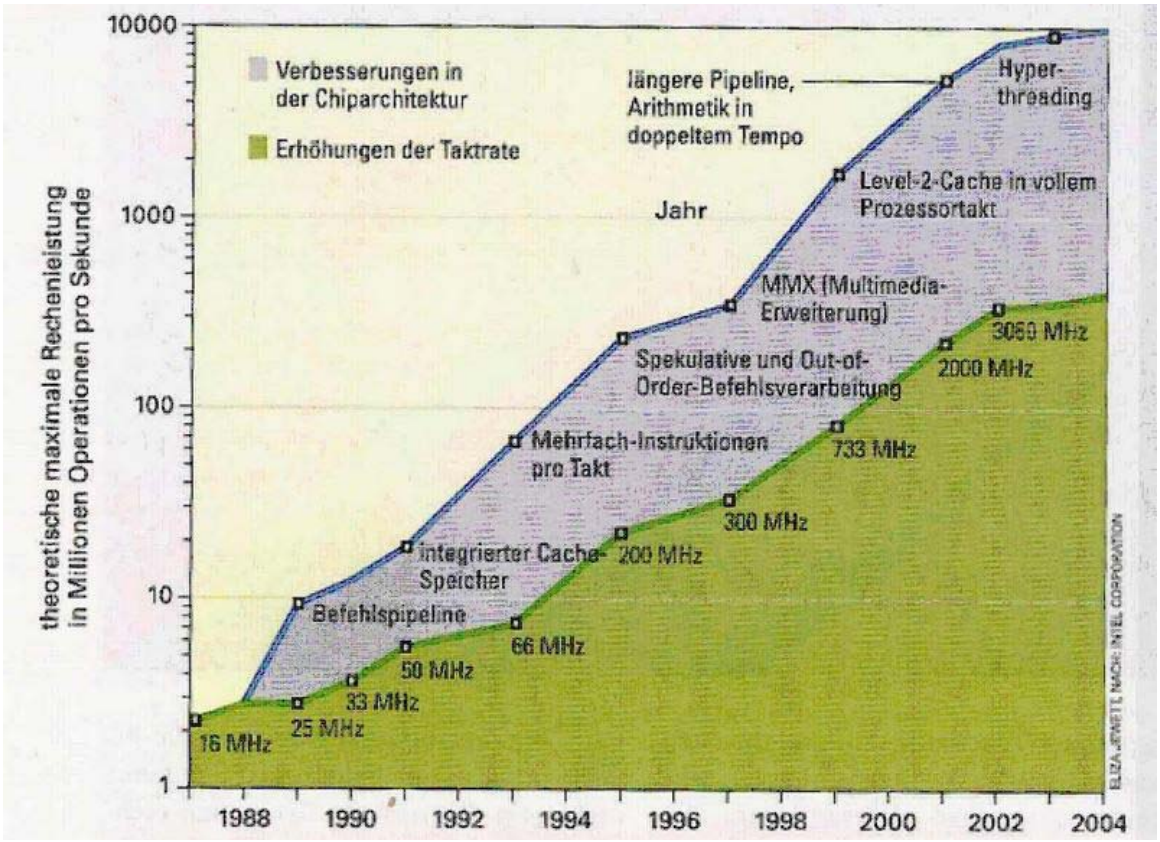
- Multi Core-CPUs werden über einen gemeinsamen 2nd oder 3rd Level Cache auf demselben Chip gekoppelt, da deren Zahl üblicherweise ≤ 16 ist, d.h. klein

Beispiel: für Multicore-Prozessoren sind IBM Power9, Intel Xeon, AMD Bulldozer, AMD APU für Sony Playstation 4, Fujitsu UltraSPARC 64 X

- Intel Multicore-Prozessor unterstützen hardware-mäßig sog. „Hyper Threading“

Hinweis: Beim Intel Hyperthreading wird pro physischem Core die Zahl der nach außen sichtbaren Register verdoppelt, und es können alle ALUs schnell von einem Registersatz zum anderen umgeschaltet werden. Dadurch entsteht der Eindruck, als ob 2 Cores statt einem vorhanden sind. => scheinbare Verdopplung der Core-Zahl im Chip.

2.2 Warum gibt es Multicore-CPUs?



⇒ **seit ca. 2002 stagniert die Entwicklung bei Taktrate und Rechnerorganisation**

- **Beides ist an ihre Grenzen gestoßen, es geht nur noch schneller durch Multi/Many Cores und einer entsprechenden parallelen Programmierung**
- **Die Marketing-Abteilungen der Prozessorhersteller verschleiern diese Tatsache erfolgreich seit Jahren, da parallele Programmierung schwierig ist**

2.3 Programmierung von Multicore-CPUs

- **Die Programmierung von Multicore-CPUs erfolgt typischerweise**
  - **entweder über gemeinsame Variable mit Hilfe der OpenMP-Programmierbibliothek, d.h. über gemeinsamen Speicher**
  - **oder über Botschaftenaustausch mit Hilfe der MPI-Programmierbibliothek, die den gemeinsamen Speicher benützt**

2.4 Many Core-CPU

Def.: Many Core-CPU: ein Many Core-Prozessor enthält auf einen einzigen Chip viele (>> 16), voneinander unabhängige, vollständige CPUs.

Beispiel: Intel Xeon Phi mit 50 Cores, die allerdings nur Co-Prozessoren sind.

- **Eine reine Cache-Kopplung ist hier aus Gründen der effizienten Interprozessorkommunikation nicht mehr möglich, vielmehr kommt ein sog. Verbindungsnetzwerk hinzu**
- **Die Programmierung ist schwieriger als bei Multi Core-CPUs, da viele Cores vorhanden**

2.5 Multiprozessoren

Def.: Multiprozessor: ein Multiprozessor ist ein System auf der Ebene einer Platine oder eines Gehäuses (engl. Crate), bestehend aus mehreren oder vielen voneinander unabhängigen Multi Core CPUs

- **Die Kopplung bei einem Multiprozessor erfolgt physikalisch über einen Third-Level Cache oder über einen gemeinsamen Hauptspeicher, beides auf der Platine**
- **Die Programmierung erfolgt wie bei den Multi Core CPUs ebenfalls über gemeinsame Variable (OpenMP) oder über Botschaftenaustausch (MPI)**
- **Die Entfernung zwischen den CPUs darf aus Gründen der notwendigen hohen Bandbreite und geringen Latenz nur einige Dutzend cm betragen**
- **Es gibt gelegentlich einen sog. Front End Computer, der für Bedienung und Programmentwicklung zuständig ist, während alle anderen Prozessoren nur rechnen, d.h. eine explit parallel Anwendung ausführen**

Beispiel: für Multiprozessoren sind High-End Server mit mehreren CPU-Sockeln

2.6 Multicomputer

Def.: Multicomputer: ein Multicomputer ist ein System auf der Ebene eines Schranks (engl. Rack) oder einer Menge von Schränken, bestehend aus mehreren oder vielen voneinander unabhängigen, kompletten Rechnern. Zwischen den Rechnern ist in Einzelfällen ein gemeinsamer Speicher oder sogar Cache-Kohärenz möglich.

- **Jeder Rechner verfügt über ein eigenes Platten-Betriebssystem**
- **Meist wird eine Kopplung über schnelle Kommunikationsnetzwerke (>> 1 Gb/s) eingesetzt (mindestens 10 Gbit/s-40 Gbit/s Ethernet/Infiniband mit Switch)**
- **Aufgrund der räumlichen Ausdehnung eines Multicomputers ist die Latenz der Kommunikation auch für den Anwender spürbar**
- **Die Programmierung erfolgt meist über Botschaftenaustausch (MPI)**

Beispiel: für Multicomputer sind einige Rechner von IBM, Cray und SGI

2.7 Cluster-Computer

Def.: Cluster-Computer: ein Cluster Computer ist eine kostengünstige Spezialisierung eines Multicomputers, der nur aus handelsüblichen PCs oder Servern oder aus Platinen von Spielekonsolen besteht. Meist ist ein High Performance-Verbindungsnetzwerk vorhanden (10-40 Gbit/s-Ethernet/Infiniband mit Switch).

- **Die Kommunikation zwischen Computern wird ausschließlich per Botschaftenaustausch (MPI) realisiert**
- **Einen gemeinsamen Speicher gibt es nicht**
- **Die größten Rechner der Welt sind Cluster Computer**

Beispiel: Sunway Taihu Light, 1.45GHz Taktrate, 10 Mio. Cores, ca. 100 PFLOP, ca. 15 MW Leistungsverbrauch (entspricht ca. 10-15 Windrädern bei Volllast).

2.8 Parallelrechner

Def.: Parallelrechner: Multicore/Manycore-CPUs, Multiprozessoren, Multicomputer und Cluster-Computer bilden zusammen die Klasse der Parallelrechner

2.9 Warum gibt es Parallelrechner?

1.) Weil es Echtzeitanwendungen mit hohem Rechenbedarf gibt

Beispiel:

- **Wettervorhersage für morgen (muss vor dem nächsten Tag fertig sein)**
- **Spracheingabe (muss fertig sein, wenn der Satz zu Ende ist)**
- **Mustererkennung in der Robotik (muss fertig sein, bevor ein Roboter gegen ein Hindernis gelaufen ist)**
- **Radarbildauswertung in Kampfjets (muss fertig sein, bevor der andere schießt)**

2.) Weil es Anwendungen mit so hohem Rechenbedarf gibt, dass die Rechnung andernfalls Jahre dauern würde

Beispiel:

- **Genaue Vorhersage der Klimaveränderung durch Simulation der globalen Erderwärmung**
- **Risikoabschätzungen in der Wirtschaft durch simulationsbasierte Prognosen von Börsekursen und Konjunkturverläufen**

- **Simulation von Atombombenexplosionen, um echte Atombombentests zu vermeiden (z.B. ASCI-Projekt des DoD der USA = traditionell größter Geldgeber und Kunde für Parallelrechnerentwicklungen)**

Hinweis: Hinweis: DoD = Department of Defence

- **Aufwendige Auswertung großer Datenmengen (=Data Mining, Big Data), Bsp. Google**

❑ **Diese Probleme können selbst mit dem schnellsten Einzelrechnersystem nicht gelöst werden**

⇒ **die gebündelte Rechenleistung sehr vieler Prozessoren bzw. Rechner ist nötig**

3.) Weil es praktisch ist, bei gestiegenen Rechenanforderungen in einer Firma, mehr Prozessoren/Rechner hinzufügen zu können, ohne gleich eine neue EDV-Anlage kaufen zu müssen (= „Horizontale Skalierung“)

4.) Weil mehrere langsame PCs zusammen billiger und schneller sind als eine einzige teure Workstation. Kosten werden teilweise auf die Programmerstellung verlagert.

5.) Weil es Anwendungen gibt, die einen sehr hohen Ein-/Ausgabebedarf haben (= I/O-intensiv)

Beispiel:

- **Web Server für große Firmen mit Millionen von Kunden, Bsp. Amazon**
- **Internet Server für Video on Demand, Bsp. Deutsche Welle (http://www.dw-world.de/)**

- **Datenerfassung in der Hochenergiephysik mit TByte an Daten/Sekunde (Bsp. CERN LHC-Experiment)**
- **Datenerfassung in der Plasmaphysik (Fusionsreaktorexperimente)**

6.) Weil es hochzuverlässige Anwendungen gibt, die redundante Prozessoren/Rechner benötigen

Beispiel: Flugbuchungssysteme, Fluglotsenrechner, Leitrechner für Stellwerke der Bahn

2.10 Ziele beim Entwurf eines Parallelrechners

- **Hohe bis sehr hohe Rechenleistung bei gleichzeitig moderatem Stromverbrauch**
- **Skalierbarkeit von kleinen zu großen Systemen**
- **Kosteneffizienz (bis 100 Mio € Investition pro Parallelrechner sind Einzelfälle)**
- **Bedienbarkeit, d.h. Vermeidung zu großer Komplexität in Handhabung und Programmierung**

2.11 Abgrenzung gegenüber verteilten Systemen auf LAN-Basis

- **Ein über ein LAN gekoppelte Sammlung von Rechnern ist nur dann ein Multicomputer, wenn:**
  - **Rechner-Hard- und Software homogen sind (keine unterschiedlichen Betriebssysteme etc.)**

  - **Bandbreite und Latenz der LAN-Kopplung besser als das übliche Maß sind**
  - **Dies schließt TCP/IP als Protokolle aus**
  - **Software (sog. Middleware) die Kopplung so unterstützt, dass effiziente parallele Anwendungen möglich sind**
- **In allen anderen Fällen spricht man von einem verteilten System**
- **WAN-gekoppelte Systeme sind stets verteilte Systeme**

2.12 Was sind die wesentlichen Punkte bei Parallelrechnern?

1.) Die Architektur und Organisation des Parallelrechners

2.) Die Art der Kopplung der Prozessoren/Rechner untereinander, d.h. wie ist der Datenaustausch organisiert („Verbindungsnetzwerk“)

3.) Die Art der Programmierung und ihr Schwierigkeitsgrad („Programmiermodell“)

4.) Die Software zur Unterstützung von Programmierung und Test (Tools und Middleware)

5.) Die Netto-Rechenleistung, wenn alle Prozessoren/Rechner dieselbe Aufgabe bearbeiten
  - **= Absolutwert in Milliarden Gleitkommaoperationen pro Sekunde [GFLOPs], oder in Billionen Gleitkommaoperationen TFLOPS, oder in Billiarden Gleitkommaoperationen [PFLOPS]**

- **GFLOP = GigaFLOP = 10^9, TFLOP = TeraFLOP = 10^{12}, PFLOP = PetaFLOP =10^{15}**
- **Stand der Technik: ca. 100 PFlop wurde in 2016 erreicht**

6.) Der Wirkungsgrad des Rechners beim Bearbeiten derselben Aufgabe

- **= Relativwert in Prozent bzgl. der Auslastung aller Cores**
- **Idealerweise sollte der Wirkungsgrad nahe 100% sein**
- **Dies lässt sich in der Praxis aber kaum realisieren und wird umso schwieriger, je mehr Prozessoren beteiligt sind (bei konstanter Problemgröße)**

2.13 Kategorien bei der Parallelrechnerorganisation

- **Es gibt 3 Hauptkategorien für Parallelrechnerorganisation:**
  - **Multiple Instruction Multiple Data (MIMD) = Stand der Technik**
  - **Single Instruction Multiple Data (SIMD) = wird nur noch selten verwendet**
  - **Multiple Instruction Single Data (MISD) = nicht eingesetzt, Analogien zum Gehirn**
- **Single Instruction Single Data (SISD) ist der klassische sequentielle Rechner**
- **Hier werden nur MIMD-Architekturen bzw. deren Verbindungsnetzwerke behandelt**
- **Bei den MIMD-Architekturen gibt es 2 Unterkategorien:**
  - **Uniform Memory Access (UMA): Speicherzugriffe dauern von jedem Prozessor auf alle Speicher gleich lange**
  - **Non Uniform Memory Access (NUMA): Speicherzugriffe dauern unterschiedlich lang**

- **Die Zugriffszeit hängt von der „Distanz“ zwischen Speicher und Prozessor ab (kurze Distanz => schneller Zugriff)**
- **Des weiteren unterscheidet man bei MIMD-Rechnern zwischen eng gekoppelten und lose gekoppelten Systemen**

Def.: Eng gekoppelt: Prozessor kann Speicher der anderen Prozessoren direkt lesen und schreiben => gemeinsamer Speicher über READ & WRITE

Def.: Lose gekoppelt: Zugriff auf entfernte Speicher nur indirekt über SEND & RECEIVE, d.h. mittels Senden und Empfangen von Botschaften

- **Das verwendete Verbindungsnetzwerk und die Art der Programmierung sind entscheidende Kriterien von Parallelrechnern**

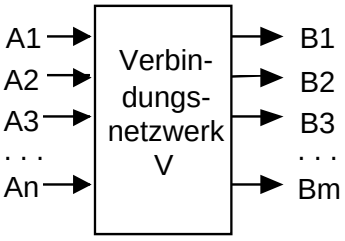
Hinweis: Ab hier sollten Sie zusätzlich das Buch „Verbindungsnetzwerke“ des Autors dieses Skripts nutzen.

2.14 Definition Verbindungsnetzwerk

Def.:

$$V: \{A_i\} \xrightarrow{f_t} \{B_j\},\ i = 1,2,\ldots,n,\ j = 1,2,\ldots,m,\ t = 1,2\ldots,p$$

- **Das Netzwerk V realisiert zeitlich veränderliche Kombination von Verbindungen**



- **Mit Hilfe eines Verbindungsnetzwerks kann man die beiden Arten von Parallelrechnern realisieren: UMA (Uniform Memory Access Computer) und NUMA (Non Uniform Memory Access Computer)**
- **Beide Formen der Rechnerorganisation werden im folgenden dargestellt**

2.15 Uniform Memory Access Computer (UMA)

Def.: Bei Uniform Memory Access Computer (UMA) ist der Zugriff der Prozessoren auf alle Daten in den gemeinsamen Speichermodulen gleich schnell

- **UMA haben einen globalen, aber keine lokale Speicher, denn lokale Speicher lassen sich schneller lesen und schreiben als der Globalspeicher, was der UMA-Eigenschaft widersprechen würde**
- **Aus Hardware-Sicht sind allerdings NUMA-Rechner (Non Uniform Memory Access Architecture) einfacher zu realisieren, da sie oft Bus/Speicher-Kopplung haben**

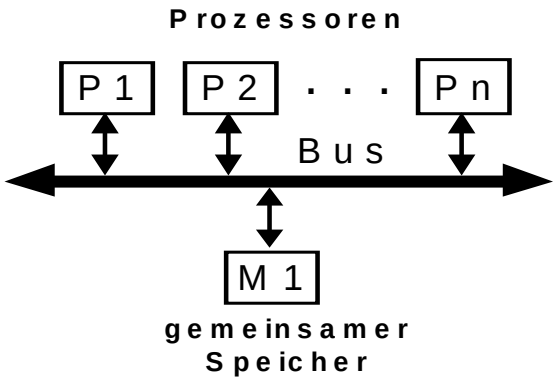
- **UMA-Rechner sind dagegen relativ einfach zu programmieren, da es egal ist, in welchem gemeinsamen Speichermodul eine gemeinsame Variable abgelegt wird**
- **Der Nachteil von UMA-Rechnern ist, dass sie die häufig vorhandene Datenlokalität paralleler Anwendungen nicht ausnützen, weil lokale Speicher nicht existieren => geringere Effizienz**

Def.: Datenlokalität paralleler Anwendungen heißt, dass parallel laufende Prozesse häufig mit denselben Prozessen kommunizieren und selten mit allen oder mit wechselnden Prozessen.

2.15.1 Bus/Speicherkopplung

- **Die Bus/Speicherkopplung mehrerer gleicher Prozessoren/Rechenknoten ist der einfachste Fall einer konventionellen Koppelung, aber auch der leistungsschwächste**
- **Hierbei kommunizieren Prozessoren/Rechenknoten über den Bus und einen gemeinsamen Speicher durch Schreiben und Lesen gemeinsamer Variablen**
- **Gemeinsamer Speicher ist bei Multicore CPUs ein on-chip 2nd oder 3rd Level Cache**
- **Nicht gemeinsame Variable und Programmkode sind ebenfalls im gemeinsamen Speicher enthalten**

⇒ **Zugriff auf Code und nicht gemeinsame Variable sowie die Interprozessor-Kommunikation muss wegen des einzigen Busses und einzigen Speichermoduls zeitlich nacheinander ablaufen = Sequentialisierung**



- Die im obigen Bild gezeigte Rechnerorganisation wird auch als Symmetric Multiprocessor (SMP) bezeichnet
- Der gemeinsamer Bus ist bei SMPs ein Engpaß, genau wie der gemeinsame Speicher

⇒ Effizienz eines großen SMPs mit >8 physikalischen Cores ist sehr begrenzt

- Wichtig ist auch der wechselseitige Ausschluss (mutual exclusion) von Prozessoren, die gleichzeitig auf dieselbe Speicherzelle schreiben wollen
- Erlaubt ist nur ein sequentieller Zugriff auf die Variable, i.e. den gemeinsamen Speicher
- Um dies sicherzustellen gibt es mehrere Möglichkeiten: Semaphore, Monitore und Bus Arbiter

- im obigen Blockdiagram wird der sequentieller Zugang zum einzigen Bus und Speichere wird durch Schiedsrichten („Arbitrierung“) der Zugangswünsche zum Bus in Hardware erreicht

Hinweis: to arbitrate: schiedsrichten

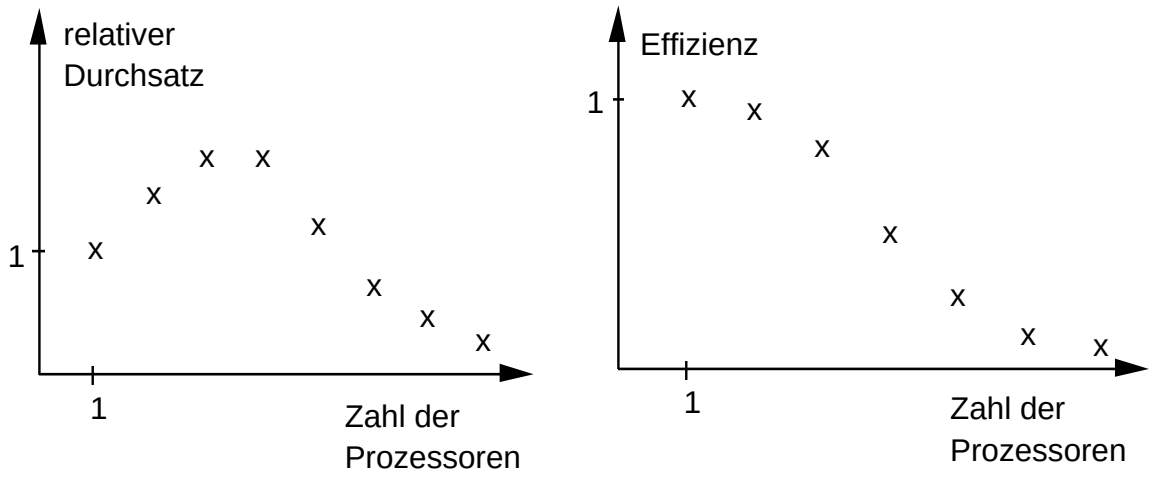
- Die Bus-Arbitrierung kann entweder zentral von einem Bus Arbiter erbracht werden, oder dezentral in den einzelnen Prozessoren/Cores erfolgen
- Bei dezentraler Arbitrierung liegt ein sog. Multimaster-Bus vor, da er mehrere Bus-Master als „Arbiter“ hat
- Diese sind die Prozessoren/Cores, die an den Bus angeschlossen sind
- Zu jeder Zeit ist aber nur ein Prozessor/Core als Arbiter aktiv

2.15.2 Sättigungseffekt bei der Bus-Speicher-Kopplung

- Durchsatz und Effizienz paralleler Programme, die über Bus und Speicher gekoppelt sind, sinken bereits ab einer kleinen Zahl (>4) von Prozessoren ab
- Richtlinie: Bus- und Speicherbandbreiten sollten so sein, dass sie ungefähr der additiven Kommunikationsbandbreite der Rechenknoten entsprechen
- Die Grenzen des technisch Machbaren werden damit schnell überschritten (=nicht erfüllbar)

⇒ Von einer kleinen Prozessorzahl (>4) ab sinkt Durchsatz und Effizienz nicht nur bei parallelen, sondern sogar bei sequentiellen Programmen unter die Werte des Einzel-

prozessorsystems ab, da sich die Prozessoren/Cores gegenseitig beim Speicherzugriff behindern

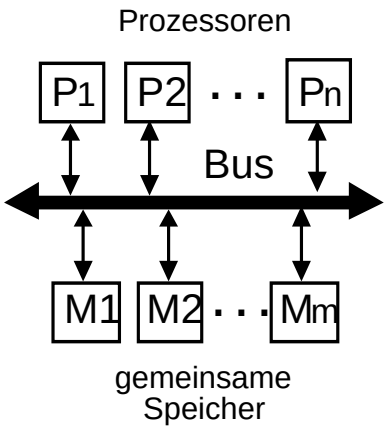


- **Ein Multiprozessorsystem dieser Organisationsform wird oberhalb der Sättigungsgrenze ineffizient**
- **Diese Art der Kopplung ist Stand der Technik bei einigen Dual/Quad/Oct Core CPUs, der Bus und der Speicher sind dabei Chip-intern**

2.15.3 Erhöhung der Bandbreite bei der Bus/Speicherkopplung

- **Die Speicherbandbreite kann durch folgende Maßnahmen erhöht werden:**

1.) Durch Einführen mehrerer Speichermodule, die parallel arbeiten



⇒ **Lese- oder Schreibwünsche von n>1 Prozessoren können von m Moduln parallel bearbeitet werden, sofern der gemeinsame Bus die erforderliche Bandbreite hat**

2.) Durch Wortbreiteerhöhung der Speichermodule: Bei jedem Speicherzugriff werden mehr Worte adressiert und transferiert als der Wortbreite eines Prozessors entspricht

⇒ **Wegfallen der Zugriffe auf nachfolgende Adressen, sofern der Prozessor nachfolgende Worte lesen oder schreiben will (ist nicht immer der Fall)**

3.) Durch Einführen mehrerer Speichermodule mit sog. Adressverschränkung

Def.: Adressverschränkung: Adressmäßig nachfolgende Worte werden auf n>1 unabhängige Speichermodule („Speicherbänke“) mit Zugriffszeit T verteilt.

Beispiel: Wort 1 auf Bank 1, Wort 2 auf Bank 2 u.s.w. für bis zu n Bänke.

- **Voraussetzung für Adressverschränkung: Ein oder mehrere Prozessoren legen möglichst lange nur aufeinanderfolgende Adressen an den Adressbus an**

Beispiel: Über den Bus werden im Abstand t nacheinander n nachfolgende Schreib- oder Leseadressen ausgegeben. Wenn die Summe aller Zeitversetzungen t bei n Bänken kleiner als die Zykluszeit T einer Bank ist, können ununterbrochen neue Adresse an den Bus gelegt werden

⇒ **Im Mittel steht alle T/n Zeiteinheiten ein neues Wort aus dem gemeinsamen Speicher zur Verfügung**

4.) Durch Einführen eines geteilten Buszyklus (Split Transaction)

- **Ein Buszyklus wird unterbrochen, sobald die Speicheradresse vom Bus an den Speicher ausgegeben wurde („suspended“). Der Speicher arbeitet danach abgekoppelt vom Bus und adressiert nur noch intern die betreffende Speicherzelle(n)**
- **In der Zwischenzeit, die der Speicher benötigt, um auf die gewünschte Adresse zu reagieren (Speicherzugriffszeit), liegen am Bus neue Adressen an, und eine andere Speicherbank beginnt, auf die dortigen Speicherzellen zuzugreifen**
- **Nach Ablauf der ersten Speicherzugriffszeit, die größer als die Buszykluszeit sein muss, wird der unterbrochene Buszyklus wiederaufgenommen (=“resumed“)**
- **Dann werden die jetzt vorhandenen Daten des ersten Speichers gelesen und über den Bus zum Empfänger transferiert**
- **Anschließend werden die Daten der zwischenzeitlich ebenfalls adressierten Bänke gelesen und zu deren Empfängern transportiert**

⇒ **Timesharing desselben Busses durch zeitliche Verschränkung (=Überlappung) mehrerer Speicherzyklen**

- **Vorteil ist eine geringere Blockierung der Prozessoren aufgrund zu langer Speicherzykluszeiten**
- **Ist zusammen mit einer mehrfädigen Programmierung („Multi Threading“) von großem Vorteil**

Hinweis: bei Multi Threading wird ohne Prozeßwechsel, d.h. sehr schnell von einem Prozeßfaden zum nächsten umgeschaltet. Dafür gibt es keinen Speicherschutz zwischen den Threads.

5.) Durch Einführen von Pipelining im Speichermodul

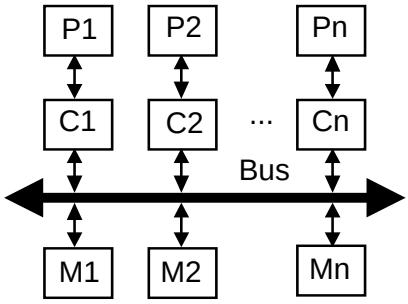
- Ein Speicherzugriff der Zeitdauer T wird in eine Reihe von n elementaren Operationen untergliedert. Bsp. für n=8:
  - Zeilenadresse anlegen
  - Chip Select-Leitung aktivieren + R/$\overline{W}$ aktivieren
  - RAS aktivieren
  - Spaltenadresse anlegen
  - CAS aktivieren
  - Speicherzugriffszeit abwarten
  - Datum lesen/schreiben
  - Chip Select-Leitung deaktivieren
- Jede dieser Operationen benötigt im Idealfall die Zeit T/n
- Alle n Operationen können fließbandmäßig verkettet, d.h. als Pipeline ausgeführt werden

⇒ **Im eingeschwungenen Zustand können die Speicheradressen n-fach schneller angelegt werden**

6.) **Durch Einführung von lokalen Caches gemäß nachfolgendem Blockdiagramm:**

- Lokale Caches entlasten den Systembus, denn sie erlauben es, Kopien von gemeinsamen Variablen anzulegen, die mit geringerer Latenz gelesen und geschrieben werden können, da sie direkt im jeweiligen Prozessor gespeichert sind und schneller als die M_i arbeiten

Hinweis: Das ist allerdings bereits eine NUMA-Architektur



- Problem: Wie erreicht man die Konsistenz von Kopien derselben Variablen in verschiedenen Caches, wenn das Original irgendwo in M_1-M_n steht?

2.15.3.1 Konsistenzproblem bei Cache-Kopien

- Die Aufrechterhaltung der Konsistenz zwischen einer oder mehrerer Cache-Kopien und einem Hauptspeicherblock ist ein Problem
- Immer dann, wenn eine Kopie geschrieben wird, müsste ein Update aller Kopien erfolgen => sehr zeitaufwendig
- Zusätzlich müsste auch das Original im Hauptspeicher aktualisiert werden => sehr zeitaufwendig
- Stets muss die Aktualisierung von Kopien und des Originals erfolgen, bevor ein veralteter Wert gelesen wird

- Der frühestmögliche Zeitpunkt für die Aktualisierung aller Kopien und des Originals ist beim ersten Schreiben einer Kopie
- Der spätestmögliche Zeitpunkt für eine Aktualisierung aller Kopien und des Originals ist kurz vor dem ersten Lesen einer veralteten Kopie
- Eine spezielle Cache-Steuerung sorgt dafür, dass das Konsistenzproblem gelöst wird (= zeitaufwendig und teuer)
- Diese automatische Konsistenzsicherung wird durch Beobachten der Schreib-/Leseaktivitäten auf dem Systembus erreicht, dem sog. Bus Snooping

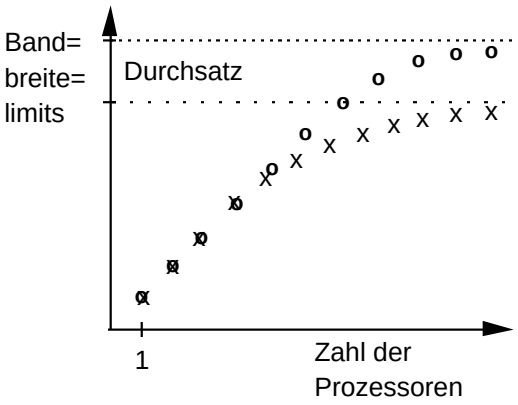
Hinweis: to snoop = schnüffeln

- Wesentlicher Teil des Bus Snoopings ist ein endlicher Automat mit Ein-/Ausgabe, der das "Update"-Protokoll „MESI“ abwickelt
- Bei MESI hat jede Cache Line in jedem Cache einen von vier Zuständen: „Modified, Exclusive, Shared od. Invalid“
- Je nach Zustand der Cache Line variiert der Update-Vorgang der Cache-Line durch einen endlichen Automaten mit Ein-/Ausgabe
  - Bei „Invalid“ ist die Cache-Kopie nicht gültig
    - Sie kann nicht mehr gelesen werden
  - Bei „Exclusive“ ist die Cache-Kopie gültig und stimmt mit dem Original überein, und es gibt nur eine Kopie des Originals
    - Wird das Original geschrieben, wird die Kopie „Invalid“
    - Wird die Kopie geschrieben, geht sie in den Zustand „Modified“ über

  - Bei „Modified“ ist die Cache-Kopie gültig, stimmt aber nicht mit dem Original überein, und es gibt nur eine Kopie des Originals
    - Soll das Original gelesen werden, muss es zuvor durch die Kopie aktualisiert werden
  - Bei „Shared“ ist die Cache-Kopie gültig und stimmt mit dem Original überein, und es gibt mehrere Kopien des Originals
    - Wird in eine Kopie geschrieben, gehen alle anderen Kopien in den Zustand „Invalid“ und die geschriebene Kopie in den Zustand „Modified“
    - Wird das Original geschrieben, werden die Kopien „Invalid“

2.15.4 Bewertung der Beschleunigungsmaßnahmen bei Bus/Speicher-Kopplung

- Um die zuvor erläuterten Beschleunigungsmaßnahmen bei der Bus/Speicher-Kopplung bewerten zu können, müssen zuvor drei Fragen beantwortet werden:
  - Frage 1: Um wieviel steigt der Durchsatz des Systems nach Einführung aller Beschleunigungsmaßnahmen an? (=gut)
  - Frage 2: Um wieviel steigt die Verzögerung beim Speicherzugriff an? (=schlecht)
  - Frage 3: Gibt es eine Grenze für Durchsatz und Verzögerung, die sog. Sättigungsgrenze?
- Antworten liefern folgende Diagramme: (x ist ohne Maßnahmen, o mit Maßnahmen)
- Es zeigt sich:
  - durch die beschleunigenden Maßnahmen wird zwar die Prozesserzahl, von der ab die Bandbreite in Sättigung gerät und die Latenz der Interprozessorkommunikation nichtlinear ansteigt, erhöht, aber das Verhalten ändert sich nicht grundsätzlich



- **insgesamt können die beschriebenen Maßnahmen die Sättigungsgrenze der Bus/Speicher-Kopplung nur hinausschieben aber nicht umgehen**

⇒ **Die Bus/Speicherkopplung ist nur begrenzt einsetzbar. Daraus ergeben sich u.a. Konsequenzen für Multicore-Prozessoren: mehr als 8-16 Cores lassen sich so nicht koppeln, um an demselben Programm gleichzeitig zu arbeiten.**

Hinweis: sog. ein- und mehrstufige Verbindungsnetzwerke sind beliebig skalierbar

- **Die gezeigten Diagramme, die durch Messung entstanden sind, lassen sich auch durch Simulation, sowie durch die sog. Warteschlangentheorie (Queueing Theory) vorhersagen**

- **Bei der Warteschlangentheorie werden die beiden Kenngrößen Bandbreite (Bandwidth) und Latenz (Latency) durch die Parameter Durchsatz (Throughput) und Verzögerung (Delay) ersetzt, die dazu analog sind**
- **Im folgenden wird für einige Beschleunigungsmaßnahmen diskutiert, wie sie sich auf die Bandbreite auswirken und was sie im Vergleich zum erzielten Effekt kosten**

2.15.5 Kosten/Nutzen-Analyse der Beschleunigungsmaßnahmen

1.) Wortbreiteerhöhung und/oder Adressverschränkung:

- **erfordert sehr breite Busse und sehr breite Speicher**
- **ist teuer (ca. 70% der Gesamtkosten eines sog. Vektor-Supercomputers entfallen auf das Speichersubsystem)**
- **Wird seit Jahrzehnten bei Vektor-Supercomputern erfolgreich eingesetzt**
- **Aus Kostengründen nur bedingt für Standard-Parallelrechner oder Multi Core CPUs einsetzbar**

2.) Gemeinsame Speichermodule mit geteiltem Buszyklus (Split Transaction):

- **Realisierung ist relativ preisgünstig**
- **Multi Threading-Programmierung oder Hyper Threading-Hardware in der CPU verbirgt außerdem einen langsamen Speicherzugriff („Latency Hiding“), da kurz nach dem Speicherzugriff auf anderen Thread umgeschaltet wird**
- **Wird bei Vektor-Supercomputern und bei Parallelrechnern angewandt**

3.) Speicher-Pipelining

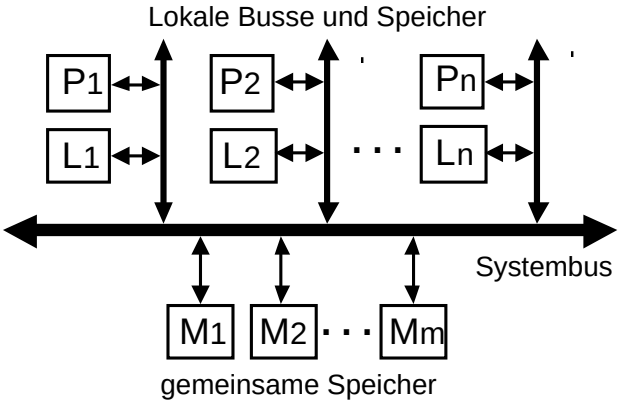
- **Wird angewandt**

2.16 Non-Uniform Memory Access Computer (NUMA)

- **Eine Verbesserung von Durchsatz und Verzögerung (throughput and delay) kann durch Lokalspeicher und/oder durch lokale Caches erreicht werden**
- **Dazu geht man vom Konzept der gleichen Zugriffszeit auf alle Hauptspeicher-Module (UMA) ab und zu NUMA über**

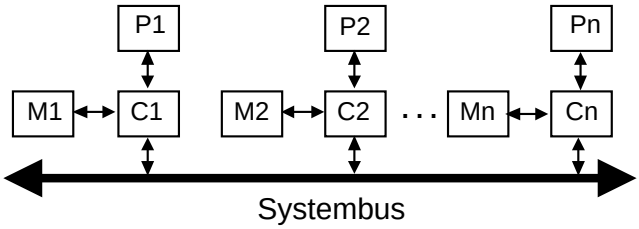
2.16.1 Beschleunigung durch Lokalspeicher

- **Direkt an der jeweiligen CPU werden zusätzliche Lokalspeicher angebracht**
- **Lokale Daten sind nicht über den Systembus zugänglich, sondern nur über den lokalen Speicherbus der jeweiligen CPU**
- **Alle nicht gemeinsamen Daten werden in den Lokalspeichern L_1 - L_n gehalten, da sie dort schneller zugreifbar sind**
- **Der Zugriff auf gemeinsame Module M_1 - M_m erfolgt nur beim Lesen und Schreiben von tatsächlich gemeinsam genutzten Variablen**
- **Vorteil: kurze lokale Punkt-zu-Punkt-Verbindungen aus nur zwei Teilnehmern (P_i + L_i) können aus elektrischen Gründen breitbandiger als der Systembus ausgelegt werden, der lang sein muss und >2 Teilnehmer hat**



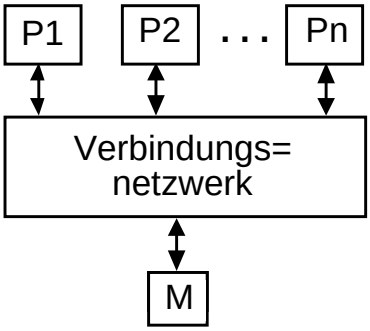
- **Nachteile:**
  - **Gemeinsamer Speicher wird nicht effizient genutzt, sobald es wenige gemeinsame Variable gibt**
  - **Lokalspeicher sind nicht effizient genutzt, sobald es wenige lokale Variable gibt**
  - **Gemeinsamer Bus wird überlastet, wenn es zu viele gemeinsame Variable gibt**
  - **Erhöhte Anforderungen an den Compiler und Programmierer bzgl. der Allokation von Variablen: „Wo soll welche Variable abgelegt werden?“**

2.16.2 Beschleunigung durch lokale Caches



- **Genau wie ein CPU-externer Lokalspeicher kann ein CPU-interner Cache die Programmausführung beschleunigen**
- **Wenn die Caches mit einem gemeinsamen Systembus verbunden sind, dann kann die Konsistenzsicherung zwischen den Caches und den Speichermoduln über das MESI-Protokoll erfolgen**
- **Diese Art der Kopplung hat sich als ein Standardverfahren bei Multi Core CPUs etabliert**
- **Dabei hat jeder Core getrennte 1st Level Befehls- und Datencaches und einen eigenen 2nd Level Cache, der für Befehle und Daten gemeinsam ist**
- **Die Cores werden über einen 3rd Level Cache gekoppelt, der für alle Cores gemeinsam ist**
- **Dieses Verfahren ist ähnlich zur Kopplung über ein sog. Multiport Memory, wie sie bei Supercomputern angewandt wurde**

2.16.3 Prozessor/Core-Kopplung über Multiport Memory

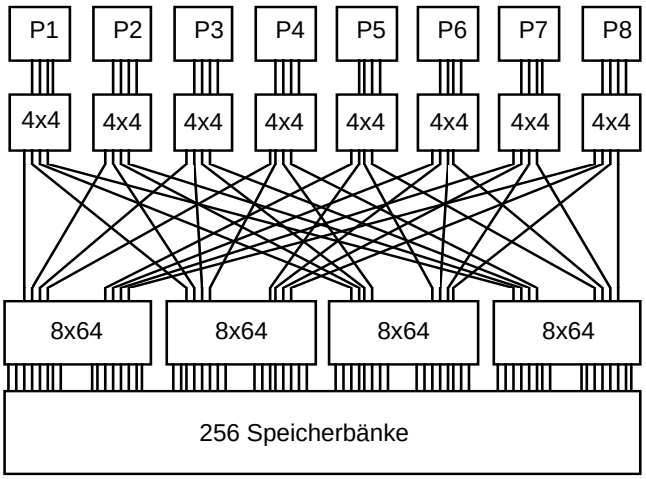


Def.: Multiport Memories sind Speicher, die von mehr als einem physikalischen Speicheranschluss („Port“) gleichzeitig gelesen und geschrieben werden können

- **Multiport Memories können auf zwei Arten realisiert werden:**
  - **Durch Speicher-Chips mit echten Mehrfach-Anschlüssen erfolgen (= sehr aufwendig und teuer)**
  - **Durch Standardspeicher-Chips, deren Anschlüsse durch Multiplexbetrieb und ein Verbindungsnetzwerk auf mehrere Ports aufgeweitet werden (= langsamer aber preisgünstiger)**

2.16.3.1 Beispiel für Multiport Memory-Kopplung

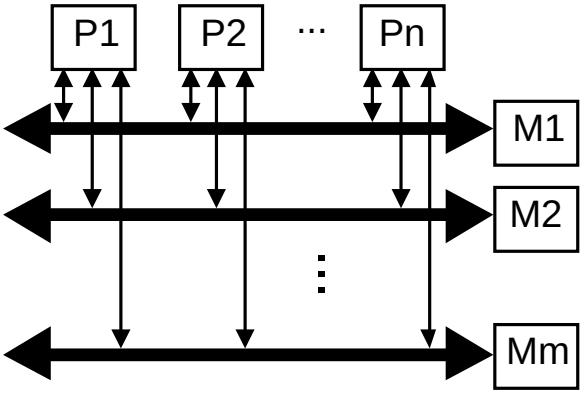
- **Die Fa. Cray ist ein wichtiger Hersteller von Supercomputern und hat in der Vergangenheit ein großes Multiport Memory über ein Verbindungsnetzwerk und 256 Speichermodule realisiert**



2.16.4 Parallelbussysteme

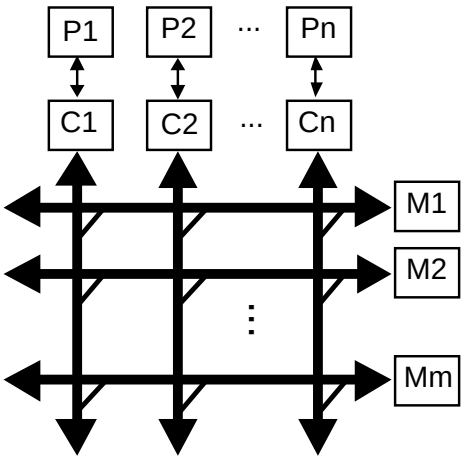
- **Ein ganz andere Möglichkeit der Kopplung stellt die Einführung paralleler oder hierarchischer Bussysteme dar**

- **Im folgenden ist ein Bussystem gezeigt, in dem entlang der horizontalen Richtung mehrere parallele Busse betrieben werden, die ebensoviele Speichermodule ansteuern**



2.16.5 2D-Busmatrizen (Kreuzschienenverteiler)

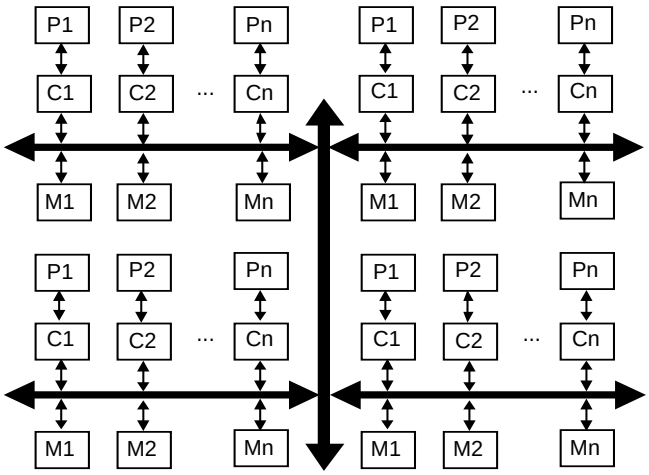
- Die Idee der parallelen Bussysteme kann auch auf zwei Dimensionen erweitert werden



- Der Übergang zwischen horizontalen und vertikalen Bussen erfolgt über sog. Busbrücken

2.16.6 Hierarchische Bussysteme

- Bei hierarchischen Bussysteme verbindet ein Hauptbus mit besonders hoher Bandbreite z.B. vier gleichartige Rechensysteme mit Unterbussen miteinander
- Das nachfolgende Blockschaltbild zeigt eine zweistufige Bushierarchie



- Jedes dieser Rechensysteme besteht aus einer „teilweisen“ UMA-Architektur
- UMA ergibt sich dadurch, dass Zugriffe auf entfernte Hauptspeicherbänke über den zentralen Hauptbus abgewickelt werden müssen und deshalb alle gleich lang dauern

- Die Architektur ist aber nur teilweise UMA, weil jeder Prozessor einen lokalen Cache und lokale gemeinsame Speichermodule hat, was NUMA ist

2.17 Skalierbarkeit von Verbindungsnetzwerken

- Die Diskussion über die Sättigungsgrenze bei der Bus/Speicherkopplung läßt sich auf die Skalierbarkeit allgemeiner Verbindungsnetzwerke ausdehnen

Def.: Skalierbar heißt, dass kleine, mittlere und große Netze dieselbe Struktur (= Topologie) haben. Deren Bandbreite wächst mit der Netzgröße N linear an, während der Hardware-Aufwand zu ihrer Realisierung nicht wesentlich schneller als linear ansteigen darf, um bezahlbar zu bleiben. Bei der Latenz ist ein O(logN)-Anstieg realisierbar.

- Skalierbarkeit erlaubt, Parallelrechner unterschiedlicher Größe aber mit denselben Eigenschaften bzgl. der Interprozessorkommunikation zu bauen
- Prinzipiell skalierbar sind allerdings nur dynamische Verbindungsnetzwerke, genauer gesagt, alle sog. logN-Netze, alle sog. Banyans, sowie das Benes-Netz, das doppelte Baseline-Netz nach Wu und Feng und ein vom Verfasser des Skripts entwickeltes Netz
- Es zeigt sich, dass der Aufwand für die Realisierung skalierbarer Verbindungsnetzwerke stets nichtlinear mit deren Größe ansteigt
- Der kleinstmögliche Anstieg im Aufwand ist O(NlogN). Weniger geht nicht.

Beispiel: O(NlogN) ist noch tolerabel, aber $O(N^2)$ wäre zuviel.

- In der Praxis zeigt sich bei skalierbaren Netzen bzgl. der Latenz ein zusätzliches Problem in Form sog. Hot Spots

Def.: Hot Spots sind Stellen in einem Verbindungsnetzwerk, wo besonders viel Datenverkehr herrscht.

- Insbesonders bei großen Netzen können verstärkt Hot Spots entstehen, da das gesamte Datenaufkommen größer ist als bei kleinen Netzen
- Hot Spots können die Latenz so sehr erhöhen, dass sie u.U. wesentlich schneller als linear mit der Netzgröße ansteigt
- Ähnlich wie bei einem „MegaStau“ auf der Autobahn geht dann gar nichts mehr
- Es müssen deshalb Zusatzmaßnahmen ergriffen im Netz ergriffen werden, um den Anstieg der Latenz zu bremsen
- Diese sind:
  - Adaptive Wegewahl, um Hot Spots zu umgehen: überlastete Netzknoten werden vom Switching/Routing ausgespart.
  - Durch das sog. „Data Packet Combining“ wird das gleichzeitige Lesen einer Speicherzelle durch mehrere Prozessoren/Cores effizienter gemacht, da durch das Netz nur einmal auf die Zelle zugegriffen wird. Der Wert wird dann vom Netz auf alle lesenden Prozesse verteilt.

2.18 Übersicht zur Programmierung von Parallelrechnern

- Die Programmierung von Parallelrechnern ist wesentlich schwieriger als bei Einzelprozessoren, weil:
  - die zeitliche Interaktion zwischen Cores bzw. Prozessen berücksichtigt werden muss

⇒ Das Debugging eines einzelnen Prozesses ist nicht ausreichend

  - die hohe Zahl von Cores (bis derzeit 10^7 in einem Supercomputer) es unmöglich, das Programm jedes Cores einzeln zu testen

⇒ die Kommunikation zwischen Cores bzw. Prozessen muss durch Abstraktion vereinfacht werden, so dass ein menschlicher Programmierer wieder in der Lage ist, die Situation zu überblicken

- Es findet deshalb eine sog. Modellbildung der Kommunikation statt, d.h. eine Vereinfachung auf wesentliche Aspekte
- Der Programmierer muss sich außerdem entscheiden, ob er eine sog. Funktionsparallelisierung oder eine Datenparallelisierung anstrebt, oder eine Kombination aus beidem

2.18.1 Funktionsparallelisierung versus Datenparallelisierung

Def.: Funktionsparallelisierung bedeutet, dass eine beliebige Aufgabe in mehrere Teilaufgaben zerlegt wird, die voneinander verschieden sind und die parallel bearbeitet werden können.

- Jede Teilaufgabe heißt „Funktion“
- I.d.R. kommunizieren die Funktionen miteinander und werden von verschiedenen Cores ausgeführt
- Die Zahl der Teilaufgaben liegt in der Regel in niedrigen zweistelligen Bereich

⇒ mit Funktionsparallelisierung kann nur eine kleinere Zahl von Cores programmiert werden, da i.d.R. nur eine kleinere Zahl paralleler Funktionen möglich ist

- Funktionsparallelisierung ist analog zum Programmiermodell des Multiple Instruction, Multiple Data (MIMD)

Def.: Datenparallelisierung bedeutet, dass in einer Rechenaufgabe Vektoren oder Matrizen bearbeitet werden. Die Parallelisierung erfolgt dadurch, dass derselbe Code gleichzeitig auf vielen Cores ausgeführt wird, allerdings jedesmal mit unterschiedlichen Teilbereichen innerhalb der Vektoren oder Matrizen, d.h. mit verschiedenen Elementen.

- Die Größe der Vektoren oder Matrizen kann sehr hoch werden (bis Milliarden von Elementen)
- D.h. damit können Millionen von Cores programmiert werden

- Datenparallelisierung ist analog zum Programmiermodell des Single Instruction, Multiple Data (SIMD), allerdings ist der einzige Instruktionsstrom zeitlich nicht straff synchronisiert, sondern liegt in jedem einzelnen Core als Kopie vor

2.18.2 Modelle der Kommunikation

- Für den Programmierer gibt zwei grundsätzlich verschiedene Modelle der Kommunikation:
  - Gemeinsame Variable (Shared Variables) mit Semaphoren für gemeinsamen Speicher
  - Botschaftenaustausch (Message Passing) mit Kanälen
- Semaphore sind zur Synchronisation des Zugriffs von schreibenden und lesenden Prozessen auf gemeinsame Variable unabdingbar
- Bei gemeinsamen Variablen erfolgt die Kommunikation mittels read/write
- Kanäle erlauben, zwischen Prozessen Botschaften zu senden
- Bei Botschaften erfolgt die Kommunikation mittels send/receive
- Meist wird innerhalb eines Rechners das Kommunikationsmodell der gemeinsamen Variablen verwendet und zwischen den Rechnern das Kommunikationsmodell des Botschaftenaustauschs

2.18.3 Testen paralleler Programme

- Das Testen paralleler Programme beschränkt sich oft nur noch darauf, den Austausch von Botschaften bzw. die Veränderung gemeinsamer Variablen zu beobachten

- Für beide Modelle der Kommunikation gibt es sowohl Spezialsprachen als auch Kommunikationsbibliotheken als Zusatz für die üblichen Sprachen wie C, C++. Java und C#
- Spezialsprachen haben sich nicht durchgesetzt
- Vielmehr werden Standardsprachen + extra Bibliotheken für gemeinsame Variable oder für Botschaftenaustausch verwendet
- Das sog. Message Passing Interface (MPI) ist die Bibliothek für Botschaftenaustausch
- Es existieren mehrere Varianten von MPI wie z.B. OpenMPI
- MPI ist eine Bibliothek mit ca. 200 Funktionsaufrufen
- Für manche Anwender ist MPI zu aufwendig
- Deshalb gibt es auch sog. Low Level-Kommunikationsbibliotheken mit wenigen und einfachen Funktionen, die aber auch weniger mächtig sind
- Low Level-Bibliotheken für Botschaftenaustausch sind Active Messages und Fast Messages
- Für den Programmierstil der gemeinsamen Variablen ist OpenMP die Bibliothek
- Sie ist allerdings ähnlich groß wie MPI
- Low level-Bibliotheken für gemeinsame Variable sind SHMEM („Shared Memory“) und POSIX threads
- Eine umfangreiche, wenn auch nicht aktuelle Gliederung von Sprachen, Bibliotheken und Konzepten zur parallelen Programmierung ist unter: http://www.vcpc.univie.ac.at/~ian/hotlist/computing/hpc/models.shtml

2.18.4 Implementierung der Kommunikationsmodelle

- Es gibt 2 grundsätzlich verschiedene Implementierungen für beide Modelle:
  - Über gemeinsamer Speicher = Shared Memory für Message Passing und für Shared Variables
  - Über Kommunikations-„Kanäle“ zwischen Prozessoren = Kanal auf ISO-Schicht 1-2 oder 1-3 für Message Passing und für Shared Variables
- Gemeinsamer Speicher kann unterschiedlich realisiert werden:
  - gemeinsamer 3rd Level Cache (= Stand der Technik bei Multi Core-Prozessoren)
  - gemeinsamer Hauptspeicher (wird bei einigen Supercomputern eingesetzt)
- Gemeinsamer Hauptspeicher gibt es in zwei verschiedenen Varianten:

1.) Mit gemeinsamen physikalischen Adressen = verteilter gemeinsamer Speicher = distributed shared memory

2.) Mit gemeinsamen logischen (virtuellen) Adressen = shared virtual memory

Def.: Distributed Shared Memory ist ein verteilter gemeinsamer Speicher, der über gemeinsame physikalische Adressen von mehreren Rechnern angesprochen werden kann.

Def.: Shared Virtual Memory ist eine Variante von Distributed Shared Memory bei der gemeinsame Variable nicht über reale Hauptspeicheradressen sondern über virtuelle Adressen, so wie sie im Befehlszähler der CPU stehen, angesprochen werden.

- Dazu wird das Paging (Seitentauschen) im Betriebssystem erweitert
- Entfernte Speicherbereiche können dann von Prozessoren analog wie Festplatten-Records von entfernten Speichermoduln geholt und durch Seitentauschen wieder in diese zurückgeschrieben werden
- Grundsätzlich gilt: beide Programmiermodelle (Message Passing und Shared Variables) können über beide Arten der physikalischen Kopplung realisiert werden, allerdings mit unterschiedlicher Effizienz
- Am besten passen gemeinsame Variable zu gemeinsamem Speicher und Botschaftenaustausch zu Kanalkopplung

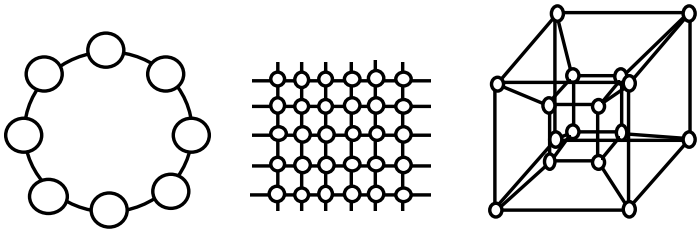
2.18.5 Zusammenfassung Parallelrechnerprogrammierung

- Der Programmierer eines Parallelrechners hat die Wahl zwischen gemeinsamen Variablen und/oder Botschaftenaustausch und zwischen Funktionsparallelisierung (MIMD) und/oder Datenparallelisierung (SIMD)

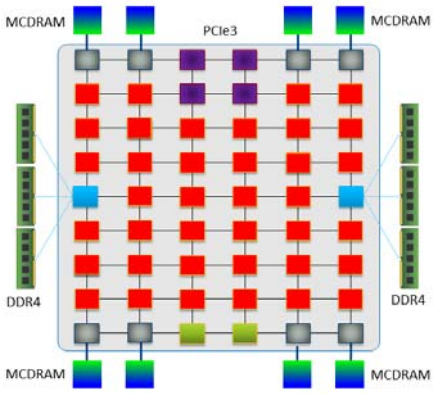
3 Grundlagen statischer und dynamischer Netze

Def.: Statische Netze definieren einen Verbindungsgraphen („Topologie“) zwischen Prozessoren oder Rechnern. Das Netz existiert in Form der Verbindungsleitungen zwischen Prozessoren/Rechnern und durch Netzwerkanschlüsse in diesen.

- **Im folgenden sind einige Beispiele statischer Netze gezeigt: ein Ring, der eigentlich ein geschlossener Polygonzug ist, ein zweidimensionales Gitter und ein vierdimensionaler „Überwürfel“ (Hypercube)**

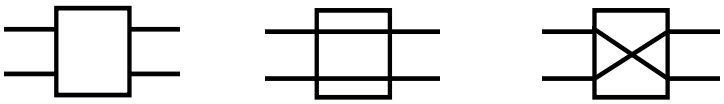


Beispiel: Die Intel Xeon Phi-Prozessorfamilie verwendet ein 2D-Gitter.



Def.: Ein dynamisches Netz ist eine eigene Hardware-Komponente in einem Parallelrechner. Es schaltet zeitlich veränderliche Verbindungen zwischen Prozessoren oder Rechnern.

- **Ein häufig verwendetes Schaltelement in dynamischen Netzen ist der 2x2-Kreuzschalter, der symbolisch als Box mit zwei Eingängen und zwei Ausgängen dargestellt wird**

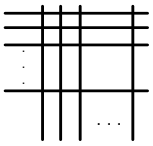


a) 2x2-Schalter **b) paralleler Durchgang** **c) gekreuzter Durchgang**

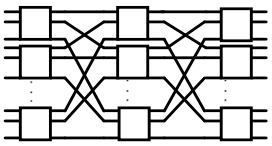
- **Der 2x2-Kreuzschalter hat die beiden Schaltmöglichkeiten „=“ und „x“**
- **Aus solchen Schaltern können zweidimensionale Anordnungen (Schaltfelder) erzeugt werden**
- **Für die Schaltfelder gilt:**
  - **sie sind in Zeilen und Spalten gegliedert**
  - **die Ausgänge vorangehender Spalten sind mit den Eingängen nachfolgender Spalten über Leitungen verbunden**
  - **die Leitungen stellen Permutationsfunktionen dar**

- Die Spalten des so entstehenden Netzes werden auch als „Stufen“ bezeichnet
- Je nach der Zahl der Stufen und der Art der Verschaltung ergeben sich dynamische Netze mit unterschiedlichen Eigenschaften
- Nachfolgend sind einige Beispiele dynamischer Netze gezeigt: ein sog. „Kreuzschienenverteiler“ bestehend aus einer Stufe, ein „Clos-Netz“ aus drei Stufen und ein „Benes-Netz“ aus 5 Stufen

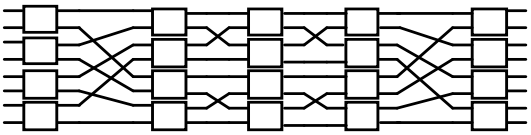
a) Kreuzschienen-verteiler



b) Clos-Netz



c) Benes-Netz



- Sowohl mit statischen als auch mit dynamischen Netzen kann man eng und lose gekoppelte Systeme realisieren
- UMAs kann man nur mittels eines zentralen dynamischen Verbindungsnetzwerks implementieren, über das alle Prozessoren auf gemeinsame Speichermodule zugreifen
- NUMAs werden oft über statische Verbindungsnetzwerke realisiert, bei denen jeweils ein Paar aus Prozessor und Lokalspeicher ein sog. „Prozessor/Speicher-Modul“ bilden
- Viele solche Prozessor/Speicher-Module werden dann über ein statisches Netz gekoppelt

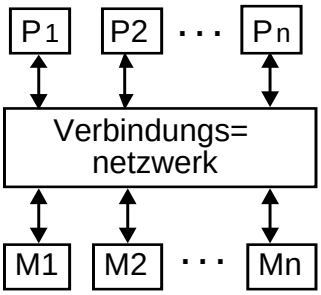
Beispiel: Ein Core ist ein Prozessor/Speicher-Modul, da es aus einem Prozessor mit first und second Level Cache besteht.

3.1 Eng gekoppelte Systeme

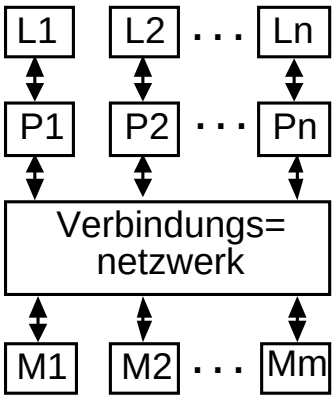
- Ein erstes Beispiel für ein eng gekoppeltes System beinhaltet verteilten gemeinsamen Speicher (Distributed Shared Memory, DSM) mit homogenen Speicherzugriffszeiten (Uniform Memory Access, UMA) zu allen Speichermoduln

3.1.1 DSM UMA-Architektur

- Das Verbindungsnetzwerk ist dynamisch



3.1.2 DSM NUMA-Architektur



- **Ein zweites Beispiel für ein eng gekoppeltes System besteht aus verteiltem gemeinsamen Speicher mit inhomogenen Speicherzugriffszeiten (DSM NUMA)**
- **Durch die Lokalspeicher L_1-L_n, die als CPU-externe Speicher, oder als CPU-interne Caches, oder als beides realisiert werden können, kann jeder Rechner/Prozessor/Core schneller auf seinen Speicher zugreifen**
- **Auf den gemeinsamen Speicher wird nur dann zugegriffen, wenn gemeinsam genutzten Variablen benötigt werden**
- **Das Verbindungsnetzwerk ist meist statisch**

- **Die nutzbare Speicherbandbreite ist bei geschickter Allozierung von Variablen zu Lokalspeichern hoch**

⇒ **Die Lokalspeicher L_i sind aufgrund des Lokalitätsprinzips von Befehlen und Daten effizient nutzbar**

- **DSM UMAs und DSM NUMAs werden als MIMD-Rechner betrieben**
- **DSM UMAs werden auch in Single Instruction Multiple Data (SIMD)-Rechnern eingesetzt, DSM NUMAs hingegen nicht, da sich ihre Ausführungsgeschwindigkeiten bei unterschiedlichen Speicherzugriffsverhalten unterscheiden**

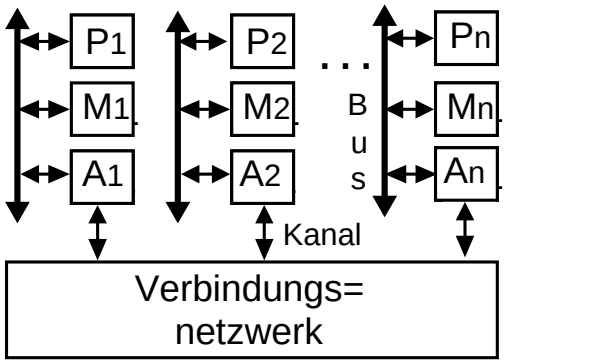
Beispiel: für DSM UMA: IBM/Sony/Toshiba Cell Prozessor einer früheren Sony Playstation.

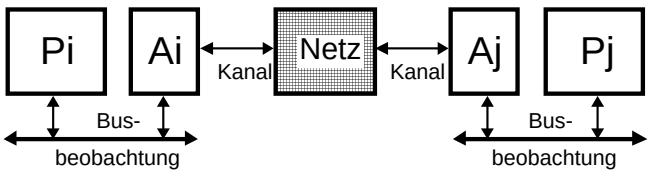
3.1.2.1 Transparenter Zugriff auf entfernten Hauptspeicher

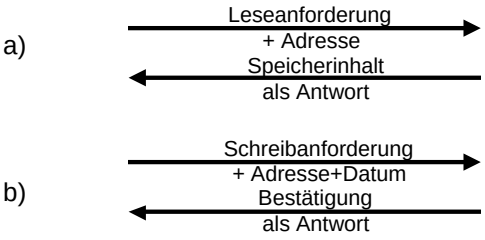
- **Verwendet man sog. „A-Controller“ auf dem Motherboard oder in einer PCI-Karte, dann kann ein automatischer Zugriff auf entfernte Hauptspeicher erfolgen, der für die Anwendungsprogramme der Prozessoren unbemerkt, d.h. transparent abläuft**

Hinweis: Die Bezeichnung „A-Controller“ ist ohne tiefere Bedeutung und dient nur zur Unterscheidung zu K-Controllern.

- **Ein Controller A_i kommuniziert mit einem Controller A_j über das Verbindungsnetzwerk**
- **Der A_j-Controller liest und schreibt den entfernten Speicher von P_j, der für A_j lokal ist, im Auftrag des A_i-Controllers, der wiederum im Auftrag seines Prozessors P_i handelt**





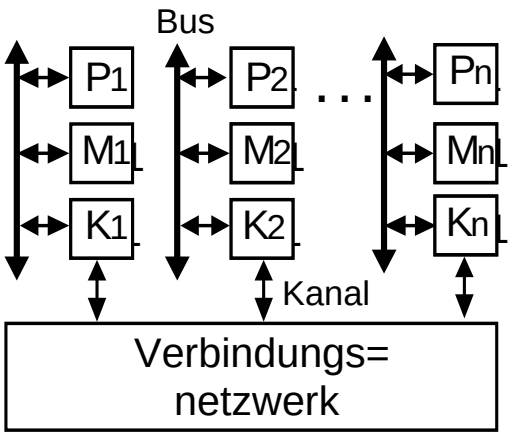


- Hierbei gibt ein Prozessor P_i eine Hauptspeicheradresse aus, die entfernt ist
- Mit Hilfe zweier A-Controller wird der Inhalt dieser physikalischen Adresse geholt
- Dazu wird von A_i eine Leseanforderung (Read Request) an A_j geschickt und von A_j mittels des Speicherinhalts beantwortet (Read Response) => Zweiphasen-Handshake
- Alternativ wird eine Schreibanforderung (Write Request) geschickt und auf die Bestätigung gewartet, dass das Schreiben stattgefunden hat (Write Response)
- Beidesmal findet ein Zweiphasen-Handshake statt, der die Zuverlässigkeit des Datentransfers erhöht

3.1.3 SVM NUMA-Architektur

- Das dritte Beispiel für ein eng gekoppeltes System besteht aus gemeinsamen virtuellen Speicher (Shared Virtual Memory, SVM) und inhomogenen Speicherzugriffszeiten (SVM NUMA)
- SVM entsteht, wenn man zwischen Rechnern einen virtuellen Hauptspeicher mit Hilfe von Speicherverwaltungseinheiten (MMUs) aufbaut, bei dem Speicherseiten (pages) nicht nur zwischen Festplatten und Hauptspeichern, sondern auch zwischen Hauptspeichern ausgetauscht werden
- Die dazu notwendigen „K-Controller“ haben umfangreichere Aufgaben als zuvor die A-Controller, da sie in das paging-Konzept des virtuellen Speichers eingebunden sind

⇒ **Ein Blocktransfer für ganze Records (Speicherseiten) ist erforderlich, anstelle des Transfer einzelner Variablen**



- Daraus resultieren folgende Probleme bei Shared Virtual Memory:

1.) Es muss stets ein ganzer Block gelesen oder geschrieben werden, selbst wenn nur auf ein Wort zugegriffen wird

⇒ **hohe Latenz beim Wortzugriff aufgrund der Seitengröße von 4-8 KB**

2.) Ständiges Hin- und Herwandern einer Seite zwischen zwei oder mehr Prozessoren kann vorkommen, wenn diese gleichzeitig und mehrfach auf dieselbe Variable zugreifen

⇒ es ensteht dann sog. „Seitenflattern“ (page thrashing). Seitenflattern führt zu erhöhtem Verkehr auf dem Netz und im Speicher.

3.) Falls zwei voneinander unabhängige gemeinsame Variablen, die von zwei oder mehr Prozessoren benötigt werden, ungeschickterweise in derselben Seite abgelegt werden (= False Sharing), kommt es ebenfalls zu Seitenflattern

3.1.4 COMA-Architektur

- Das vierte Beispiel für ein eng gekoppeltes System beruht auf dem NUMA-Konzept
- Es hat keinen Hauptspeicher, sondern nur Caches und verwendet aus Gründen besserer Skalierbarkeit ein Verbindungsnetzwerk anstelle eines Busses
- Man spricht deshalb von einer „Nur-Cache“-Architektur (Cache-Only Memory Architecture, COMA)
- Vorteil von COMA-Architekturen:

1.) Statt ganzer Records müssen bei COMA nur rel. kurze Cache Lines transferiert werden

2.) Durch den Transfer von 16 B statt 4 oder 8 KB sinkt die Latenz der Interprozessorkommunikation erheblich, und das Verbindungsnetzwerk wird wesentlich weniger belastet

3.) Zusätzlich erhält jeder Prozessor in seinem Cache früher oder später Kopien derjenigen Variablen, die er benötigt

4.) Durch lokale Kopien von gemeinsamen Variablen entfällt das Seitenflattern komplett

- COMA-Rechner sind effektiv, setzen aber große Caches mit einem dezentralem Cache Update voraus
- Probleme bei COMA-Architekturen:

1.) In einem Verbindungsnetzwerk ist Bus Snooping zur Aufrechterhaltung der Cache-Konsistenz nicht möglich, weil es keinen Bus mehr gibt

⇒ Aufrechterhaltung der Cache-Konsistenz ist bei COMA ein Problem, das mit der Zahl der Prozessoren, i.e. der Zahl der Caches zunimmt

2.) Cache-Speicher ist viel aufwendiger und damit teurer als normaler Hauptspeicher

- Cache Only Memory-Architekturen sind im Prinzip eine gute Idee, aber haben sich wegen der im Verbindungsnetz schwer zu realisierenden Cache-Konsistenz und der hohen Kosten für Caches als zu aufwendig herausgestellt

3.1.5 Programmierung eng gekoppelter Systeme

- Die Programmierung eng gekoppelter Systeme erfolgt i.d.R. über gemeinsam genutzte Variable, auf die mehrere Prozessoren lesend und schreibend zugreifen können
- Beim Zugriff auf gemeinsame Variablen treten allerdings drei neue Situationen auf, die es bei sequentiellen Programmen nicht gibt, und die alle drei zu einem Programmierfehler führen:

1.) **Ein Prozessor schreibt eine Variable während ein zweiter diese liest => Das Ergebnis des Lesens ist unbestimmt**

2.) **Zwei Prozessoren schreiben gleichzeitig dieselbe Variable => Das Ergebnis des Schreibens ist unbestimmt**

3.) **Prozessoren lesen eine Variable, die nie geschrieben wird. Das Ergebnis des Lesens ist dann unbestimmt, wenn es keine automatische Variableninitialisierung zu Programmbeginn gibt.**

- **Um diese Fehlersituationen zu vermeiden, ist es notwendig, den Zugriff auf gemeinsame Variablen abzustimmen: die sog. Zugriffssynchronisation**

3.1.5.1 Zugriffssynchronisation für gemeinsame Variable

- **Fakt 1: Nur ein Prozess darf eine Variable zu einer Zeit schreiben, keine zwei oder mehr**
- **Fakt 2: Gleichzeitiges Lesen einer Variablen durch mehrere Prozesse geht problemlos**
- **Fakt 3: Lesen während des Schreibens ist nicht zulässig**
- **Um alle drei Regeln zu berücksichtigen, muss der Schreibzugriff auf eine gemeinsame Variable exklusiv, d.h. wechselseitig ausschließend für die beteiligten Prozesse sein**
- **Das Schreiben einer gemeinsamen Variablen wird deshalb über Semaphore oder über Monitore geregelt**

3.1.5.2 Semaphore

- **Semaphore (“Zeichenträger“) erlauben einen zeitlich unteilbaren Zugriff auf Daten im Hauptspeicher**
- **Semaphore ermöglichen dadurch den kontrollierten Zugang zu gemeinsam genutzten Variablen, die von mindestens einem Prozess geschrieben und von mindestens einem anderen Prozess gelesen werden**
- **Zwischen Lesen und Schreiben der Semaphore darf kein anderer Prozess den laufenden Prozess unterbrechen und seinerseits die Semaphore lesen, andernfalls würde das Ergebnis des Lesens und damit auch des Schreibens verfälscht => Exklusivzugriff**
- **Semaphore setzen voraus, dass eine Variable in einem einzigen Maschinenbefehl wie z.B. „TestAndSet“ zuerst gelesen und abhängig vom gelesenen Werte u.U. auch geschrieben werden kann**
- **Semaphore sind ganz normale Variable im Hauptspeicher, auf die über Spezialbefehle zugegriffen wird, wie z.B. über „TestAndSet“**
- **Wird eine ungesetzte Semaphore gelesen, kann sie noch vom selben Maschinenbefehl gesetzt werden (= bedingtes Schreiben)**
- **Semaphore sind damit ähnlich wie Datenbanken mit ACID-Eigenschaft (ACID= Atomar, Consistent, Isolating, Durable), die ebenfalls einen unteilbaren Zugriff auf Daten erlauben**
- **Wird eine bereits gesetzte Semaphore gelesen, bleibt der lesende Prozess nach der Ausführung von Test and Set „stehen“**

- Der stehengebliebene Prozess verbraucht während des Wartens keine CPU-Zeit, sondern wird vom Betriebssystem aus der Liste der rechenbereiten Prozess entfernt (=descheduled)
- Setzt der Prozess, der die Semaphore gesetzt hat, diese wieder zurück, wird der nachfolgende Prozess, der stehengeblieben ist, vom Betriebssystem automatisch wieder in Gang gesetzt (=rescheduled) und kann weiterrechnen

Hinweis: Semaphore erlauben nicht nur den exklusiven Zugriff auf eine Variable im Speicher. Sie können auch den exklusiven Zugriff auf andere Betriebsmittel wie z.B. Drucker regeln.

3.1.5.3 Peer-to-Peer-Kommunikation über Semaphore

Beispiel: Ein rechnender Prozess schreibt auf eine Variable, die er gemeinsam mit einem anderen Prozess hat. Der zweite Prozess wartet, bis der erste fertig ist, da er den Wert dieser Variablen braucht.

- Eine gesetzte Semaphore blockiert den READ-Prozess solange, bis die Semaphore vom aktuellen Inhaber der gemeinsamen Variable (= WRITE-Prozess) zurückgesetzt worden ist
- Nach dem Zurücksetzen wird der blockierte Prozess wieder freigegeben und kann auf die gemeinsame Variable zugreifen

⇒ Wartender Prozess (= READ Prozess) hat eine Konsumentenrolle von Information. Schreibender Prozess (= WRITE Prozess) hat eine Produzentenrolle von Information.

- Zusammenfassung:

- Semaphore sind das Hilfsmittel, um den Zugriff auf gemeinsam genutzte Betriebsmittel wie z.B. eine gemeinsam genutzte Variable wechselseitig auszuschließen
- Dies erfolgt so, dass zu jedem Zeitpunkt die Variable nur von einem Prozess geschrieben werden kann, aber nicht von zwei oder mehr, und dass während des Schreibens kein Lesen durch einen anderen Prozess möglich ist
- Dadurch ist ein kontrollierter Zugriff möglich, und die Variable hat stets konsistente Werte
- Dazu muss es in der Rechnerarchitektur des verwendeten Prozessors einen Maschinenbefehl geben, der Lesen und ein unmittelbar nachfolgendes Schreiben abhängig vom gelesenen Wert unterstützt

3.1.5.4 Netzwerkweite Semaphore (=Entfernte Semaphore)

- Für die Synchronisation des Zugriffs auf gemeinsam genutzte Daten, die im Parallelrechner verteilt sind, muss das Netzwerk des Parallelrechners auch Semaphor-Operationen in entfernten Speichen aufweisen
- Das Lesen und bedingte Schreiben von entfernten Semaphoren, sowie das automatische De-/ und Rescheduling von Prozessen, die auf eine Semaphore im Hauptspeicher eines anderen Rechners zugreifen, ist deutlich schwieriger zu implementieren, als Semaphor-Operationen im selben Rechner
- Die obigen Funktionen werden deshalb als netzwerkweite Semaphor-Operationen bezeichnet

Def.: Netzwerkweite Semaphore bedeuten, dass es einen Maschinenbefehl gibt, der über das Verbindungsnetzwerk eine Semaphore im Hauptspeicher eines anderen Rechners lesen und diese unterbrechungsfrei setzen kann, wenn sie nicht gesetzt ist.

- **Netzwerkweite Semaphore arbeiten über Rechnergrenzen hinweg**
- **Netzwerkweite Semaphore sind in einem Parallelrechner die einzige schnelle Möglichkeit des wechselseitigen Ausschlusses gemeinsam genutzter Variablen**
- **Netzwerkweite Semaphore bedeuten auch, dass das De- und Reschulding bei gesetzter bzw. zurückgesetzter Semaphore netzwerkweit von allen lokalen Betriebssystemen unterstützt werden muss**
- **Netzwerkweite Semaphore sind kaum verbreitet, da technisch schwierig zu realisieren**

3.1.5.5 Monitore

Def.: Ein Monitor ist ein Programm, das von verschiedenen Prozessen gleichzeitig um das Lesen oder Schreiben einer gemeinsamen Variablen gebeten werden kann. Der Monitor sequentialisiert die Zugriffe so, dass eine vorab definierte Zugriffsreihenfolge entsteht.

- **Der Monitor ist der „Verwalter“ einer gemeinsam genutzen Variablen**
- **Er kann mit einer C++ Klasse verglichen werden, die parallele Methodenaufrufe erlaubt, die ihrerseits eine gemeinsame Variable im Auftrag lesen oder schreiben und dabei eine bestimmte Reihenfolge einhalten**

- **Die Zugriffsreihenfolge bewirkt eine Priorisierung der Zugriffe und definiert eine bestimmte Semantik bei gleichzeitigem Zugriff**
- **Ein Monitor ist langsamer als eine Semaphore, da Software**

3.1.5.6 Critical Sections

- **Das Konzept der gemeinsamen Variablen kann auf ganze Speicherbereiche ausgedehnt werden, d.h. auf einen zusammenhängenden Block von Speicherzellen = Critical Section**
- **Der Zugriff auf eine Critical Section wird analog wie der Zugriff auf eine einzelne gemeinsame Variable über Semphore oder Monitore durchgeführt, um Programmierfehler zu vermeiden**

3.2 Lose gekoppelte Systeme

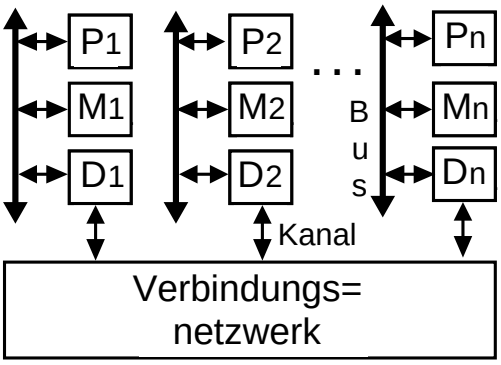
- **Lose gekoppelte Systeme erlauben keinen lesenden oder schreibenden Zugriff auf entfernte Speicher**
- **Statt dessen tauschen parallele Prozesse Informationen durch Botschaften aus**

⇒ **READ und WRITE bei eng gekoppelten Systemen wird durch SEND und RECEIVE bei lose gekoppelte Systemen ersetzt**

- **Die Botschaften werden zwischen Sende- und Empfangspuffer ausgetauscht**
- **Speicher sind beim Datentransfer nur indirekt in Form von Puffern beteiligt**

3.2.1 Botschaftenaustausch

- Ein typisches Beispiel für ein lose gekoppeltes System, das auf Botschaftenaustausch (Message Passing) zwischen entfernten Prozessoren basiert, ist im nächsten Bild gezeigt



- Der Botschaftenaustausch wird über sog. „D“-Controller durchgeführt
- Zwischen den D-Controllern für Message Passing und den A-Controllern für Shared Memory gibt es Unterschiede, die das Resultat unterschiedlicher Anforderungen sind
- Beim Botschaftenaustausch existieren folgende Anforderungen an den Transfer:

1.) Transfers erfolgen explizit über SEND und RECEIVE statt implizit über READ und WRITE

2.) Botschaften aus dem Sendepuffer werden per DMA als Block transferiert

3.) Den kommunizierenden Prozessen wird für die Dauer des DMA-Transfers die CPU entzogen (Descheduling)

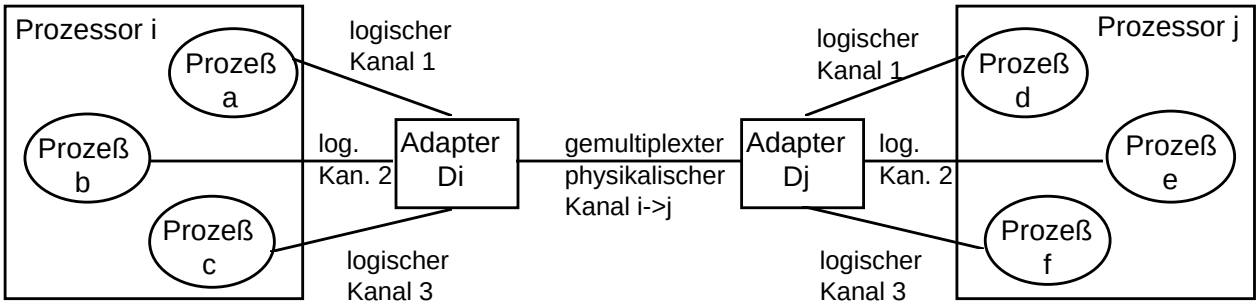
⇒ höherer Overhead und damit höhere Latenz bei der Interprozessorkommunikation über Botschaften als über gemeinsame Variable

4.) Ein Multiplexen mehrerer Datenströme auf denselben physikalischen Kanal ist möglich

⇒ virtuelle Kommunikationskanäle

3.2.1.1 Virtuelle Kommunikationskanäle mittels D-Controller

- D-Controller ermöglichen eine Kommunikation über virtuelle Kanäle zwischen entfernten Prozessen
- Anders als A- oder K-Controller implementieren die D-Controller Interprozesskommunikation statt Interprozessorkommunikation => höherer Overhead => höhere Latenz, aber bessere Handhabbarkeit für den Benutzer
- D-Controller/Adapter sind vergleichbar mit Switches/Routern in einem Rechnernetz plus Portadressierung in einem Rechner arbeiten aber nur auf ISO-Schicht 1-2 oder 1-3



- **D-Controller benützen Adressen zur Unterscheidung der virtuellen Kanäle, die man mit physikalischen Ports der ISO-Schicht 2 oder mit Software-Ports der ISO-Schicht 4 gleichsetzen könnte**
- **Jede Portadresse wird über das Betriebssystem oder über das Laufzeitsystem der verwendeten parallelen Programmiersprache mit einem dazu gehörenden Sende- bzw. Emfangsprozess gekoppelt**

Hinweis: Im Falle von Rechnernetzen ist die feste Zuordnung zwischen Schicht 4-Ports in einem Rechner und dessen Prozessen eine der Aufgaben des Betriebssystems.

- **Pro Prozessor können mehrere Prozesse existieren, die entweder im Zeitscheibenverfahren nebenläufig oder auf eigenen Cores tatsächlich parallel ausgeführt werden**

3.2.2 Programmierung lose gekoppelter Systeme

- **Die Programmierung lose gekoppelter Systeme erfolgt über synchronen oder asynchronen Datenaustausch mittels Botschaften**

3.2.2.1 Synchrone/asynchrone Kommunikation über Botschaften

- **Asynchroner Datenaustausch ist analog zu dem Verfahren, das Briefträger und Briefkasten praktizieren = Mailbox-Prinzip**
- **Dasselbe Verfahren wird auch bei E-mails angewandt.**
- **Synchroner Datenaustausch entspricht einem „Rendezvous“ zwischen Sende- und Empfangsprozess, da beide zur selben Zeit kommunikationsbereit sein müssen**
- **Sowohl asynchroner als auch synchroner Botschaftenaustausch kann dazu verwendet werden, um entfernte Prozesse miteinander zu synchronisieren**

3.2.2.2 Prozess-Synchronisation durch Botschaften

- **Prozesssynchronisation zwischen zwei Prozessen heißt, dass ein Prozess weiß, dass sich ein anderer Prozess gerade an einer bestimmten Stelle im Code befindet**
- **Diese Information ist z.B. dann wichtig, wenn der zweite Prozess kommunikationsbereit ist und auf den ersten wartet, weil er ein (Teil)ergebnis fertig berechnet hat**
- **Prozesssynchronisation zwischen zwei Prozessen ist für einen der beiden Prozesse i.d.R. mit Warten verbunden, da nur selten zwei Prozesse exakt zur selben Zeit ihre Rechenaufgabe abschließen oder kommunikationsbereit sind**

- **Das Ziel bei der parallelen Programmierung ist es deshalb, solche Wartezeiten durch Umprogrammieren zu minimieren, da sie die Effizienz des Parallelrechners reduzieren**
- **Idealerweise ist jeder Prozessor/jeder Core ständig entweder mit rechnen oder mit kommunizieren beschäftigt**
- **Im Laufe einer Interprozesskommunikation gibt es zwei grundsätzliche Situationen:**

1.) Prozess 1 ist mit einer Teilberechnung fertig und hat seine Ergebnisvariable geschrieben. Prozess 1 wartet danach auf Prozess 2, bis dieser sie gelesen hat.

2.) Prozess 1 ist mit einer Teilberechnung noch nicht fertig (rechnet noch). Prozess 2 hingegen wartet auf Prozess 1, um die Ergebnisvariable zu lesen.

- **Problem: woher weiß ein wartender Prozess, wann ein anderer Prozess fertig ist, ohne dass er ständig dessen Status in einer Schleife abfragt? (Polling kostet zuviel Zeit).**
- **Lösung:**

1.) Im Falle von gemeinsamen Variablen wird das Problem über eine Semaphore gelöst. Es erfolgt automatisches Aufheben des Wartens durch das Betriebssystem, sobald die Semaphore zurückgesetzt ist.

- **dies setzt voraus, dass es einen gemeinsamen Speicher zwischen den beiden Prozessen gibt**
- **Sind die beiden Prozesse auf verschiedenen Rechnern angesiedelt, ist ein gemeinsamer Speicher nur durch eine aufwendige Zusatz-Hardware möglich**

2.) Im Falle von synchronem Botschaftenaustausch wird über ein sog. Rendezvous synchronisiert, das auf der gleichzeitigen Existenz eines Produzenten und eines Konsumenten von Daten und deren Kommunikation beruht

Def.: Rendezvous heißt: zwei Prozesse kommunizieren zur selben Zeit über einen gemeinsamen Kanal

- **das Rendezvous setzt voraus, dass es einen gemeinsamen Kanal zwischen den beiden Prozessen gibt**
- **sind die beiden Prozesse auf verschiedenen Rechnern angesiedelt, ist ein gemeinsamer Kanal rel. leicht realisierbar**
  - **im einfachsten Fall wird der Kanal über zwei Ethernet-Karten und einen Ethernet Switch realisiert, der die Karten verbindet**

- **Das Rendezvous ist i.d.R. für einen der beiden Kommunikationspartner ebenfalls mit Warten verbunden, solange bis der andere Prozess zur Kommunikation bereit ist**
- **Der wartende Prozess wird bis zum Beginn des Datenaustauschs von demjenigen Betriebssystem automatisch descheduled, in dem er ausgeführt wird („Waiting for IO“), so dass der Prozess keine CPU-Zyklen vergeudet**
- **Treffen Daten auf dem gemeinsamen Kanal ein, re-scheduled das betreffende Betriebssystem den Prozess automatisch wieder**
- **Zum Zeitpunkt des Rendezvous wissen beide Prozesse, an welcher Stelle des Programms der jeweils andere sich befindet, nämlich an der Stelle des Rendezvous**

- Das Rendezvous entspricht einem synchronen Datenaustausch, da beide Prozesse zur selben Zeit miteinander kommunizieren
- Eine Alternative zum synchronen Datenaustausch besteht in einem asynchronen Datenaustausch
- Der asynchronen Datenaustauschs erlaubt keine Aussage über den aktuellen Zustand des anderen Prozesses, d.h. über dessen Programmzähler
- Ist es dennoch notwendig, etwas über den momentanen Befehlszähler (= Zustand) des anderen Prozesses zu erfahren, kann ersatzweise zwei Mal hintereinander ein asynchroner Botschaftenaustausch mit wechselnder Kommunikationsrichtung durchgeführt werden gemäß:
  - A sendet zuerst nach B. B wartet bis A gesendet hat. Dann weiß B, wo A steht. Danach sendet B an A, dass B die Nachricht von A empfangen hat. Dann weiß A, wo B steht. Zum Schluss wissen beide, wo der andere steht.
- Dies entspricht einem Zwei-Phasen Handshake-Protokoll wie bei einem Rechnernetz
- Darüberhinaus ist, im Gegensatz zu einem gleichzeitigem WRITE, bei gleichzeitigem SEND zweier Sender an denselben Empfänger ein wechselseitiger Ausschluss der Sender bei Botschaftenaustausch nicht nötig
- Dies ist deswegen so,
  - weil Sender ihre Daten an verschiedenen Stellen im gleichen Empfangspuffer ablegen können, oder alternativ
  - weil Sender Daten in verschiedene Empfangspuffer transferieren können

- Semaphore sind deshalb bei Botschaftenaustausch nicht nötig
- Sie wären bei getrennten Rechner sowie so nur durch eine aufwendige Zusatz-Hardware realisierbar

3.2.2.3 Produzenten und Konsumenten von Daten

- Prozesse können nicht mehr isoliert voneinander betrachtet werden, sobald Daten ausgetauscht werden müssen
- Beim Datenaustausch gibt es zwei grundsätzliche Rollen:

1.) Ein Prozess stellt Daten für einen anderen Prozess bereit = Produzentenrolle

2.) Ein Prozess verwendet Daten von anderem Prozess, um weiterrechnen zu können = Konsumentenrolle

- Im Laufe der Ausführung eines parallelen Programms können sich die Rollen umdrehen
- Aus dem Produzent wird dann ein Konsument und umgekehrt
- Diese Situation wird auch als Peer-to-peer-Kommunikation bezeichnet

Def.: Peer-to-peer-Kommunikation: bidirektionale Interprozesskommunikation mit gleicher Hierarchiestufe beider Partner

- Die Peer-to-peer-Kommunikation wird im Falle gemeinsamer Variable über Semaphore und im Falle von Botschaftenaustausch über das Rendezvous einer synchronen Kommunikation abgewickelt

3.2.2.4 Peer-to-Peer-Kommunikation über Rendezvous

- Wird READ und WRITE durch synchrones SEND und RECEIVE ersetzt, terminieren beide Operationen erst dann, wenn der Datenaustausch abgeschlossen ist
- SEND hat dabei die Produzentenrolle und RECEIVE die Konsumentenrolle
- Man spricht in diesem Falle auch von „blockierendem“ SEND/RECEIVE
- Blockierendes SEND/RECEIVE implementiert eine synchrone Kommunikation, d.h. ein Rendezvous
- Nach der synchronen Kommunikation terminieren SEND und RECEIVE gleichzeitig
- Nicht blockierendes SEND/RECEIVE hingegen implementiert eine asynchrone Kommunikation
- Asynchrone Kommunikation kann über eine Mailbox oder einen FIFO realisiert werden, worin der sendende Prozess Daten ablegt und woraus der empfangende Prozess sie irgendwann liest

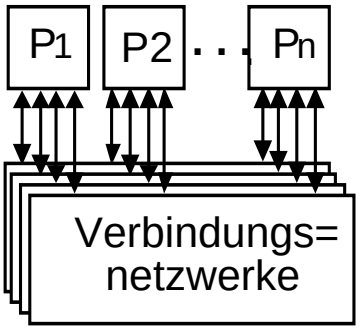
3.2.2.5 Bewertung von READ/WRITE und von (a)synchronem SEND/RECEIVE

- READ/WRITE mit Semaphore und synchrones SEND/RECEIVE haben bei der Parallelrechnerprogrammierung eine hohe praktische Bedeutung
- Asynchrones SEND/RECEIVE hingegen wird selten eingesetzt, READ/WRITE ohne Semaphore wird nie verwendet, da nicht deterministisch

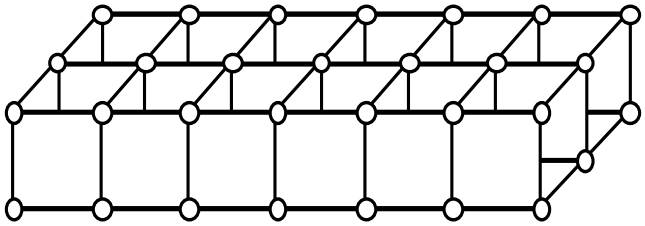
3.3 Erweiterte Konstruktionsprinzipien von Netzen

- Hat man in einem Parallelrechner ein bestimmtes Verbindungsnetz, dann kann man daraus mit Hilfe von vier grundlegenden Prinzipien neue Netze generieren
- Diese vier Prinzipien sind Parallelschaltung, Hierarchie, Rekursion und Modularität bekannter Netze
- Alle vier werden nachfolgend erläutert

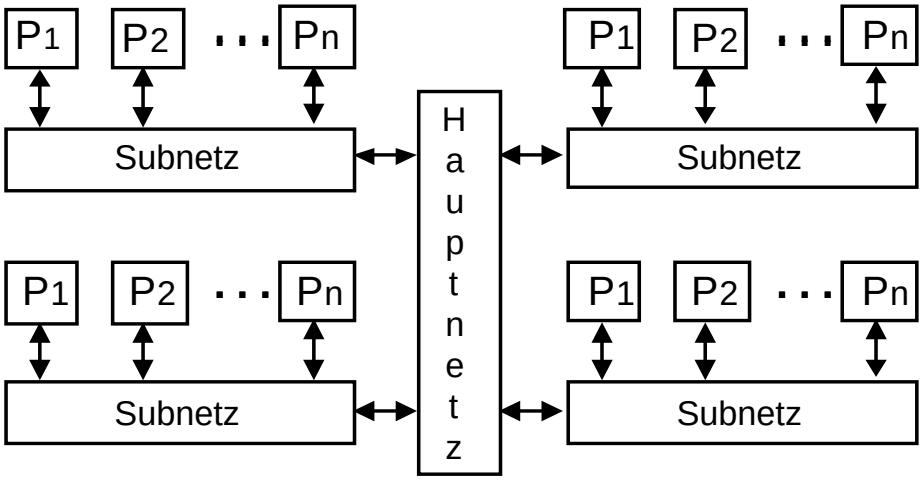
3.3.1 Parallelschaltung



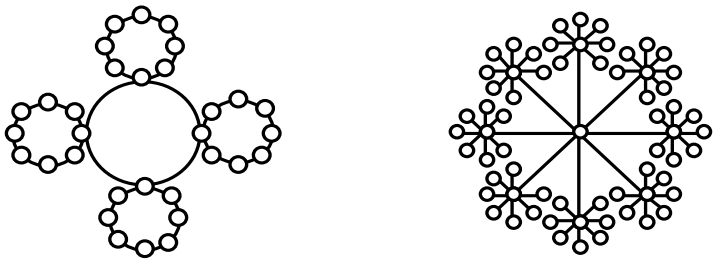
Beispiel: Parallelschaltung mehrerer 2-D-Gitter zu einem 3D-Gitter.



3.3.2 Hierarchie



3.3.3 Rekursion



a) b)

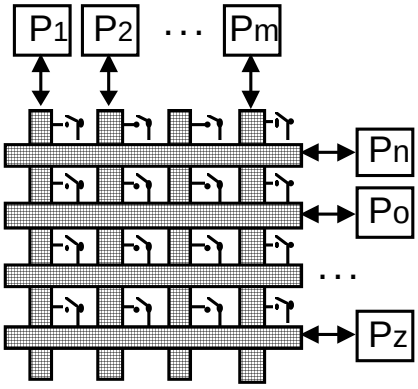
- **Im Beispiel a) hat man einen Ring aus Ringen, im Beispiel b) einen Stern aus Sternen**

Hinweis: unendliche Fortsetzung der Rekursion führt zu einer fraktalen Geometrie

3.3.4 Modularisierung

- **Das Verbindungsnetzwerk ist aus baugleichen Einheiten (sog. Moduln) aufgebaut**

Beispiel: das Verbindungsnetzwerk sei ein Kreuzschienenverteiler aus baugleichen horizontalen und vertikalen Bussen



3.3.5 Modularisierung und Rekursion gemeinsam

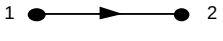
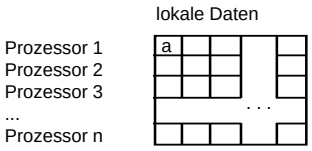
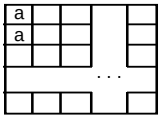
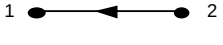
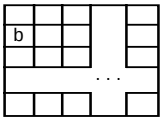
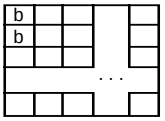
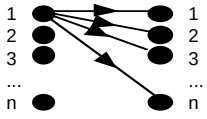
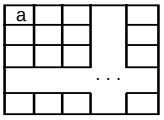
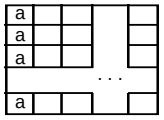
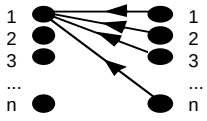
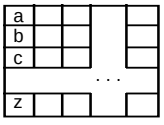
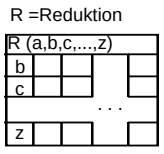
- **Das gleichzeitige Auftreten von Modularisierung und Rekursion ist möglich**

Beispiel: Ein Kreuzschienenverteiler bestehe aus baugleichen horizontalen und vertikalen Moduln. Jedes Modul sei kein Bus sondern wiederum ein Kreuzschienenverteiler (= zweistufige Rekursion)

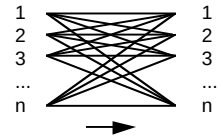
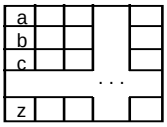
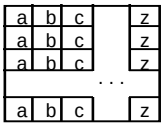
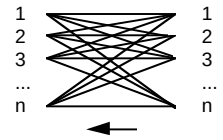
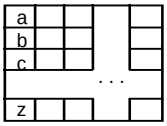
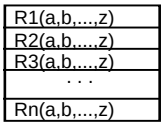
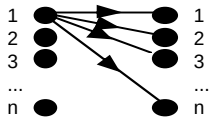
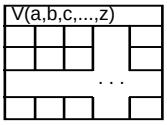
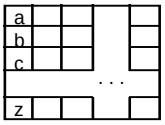
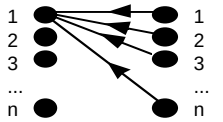
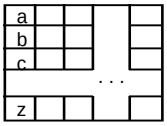
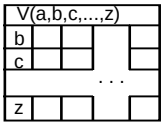
Beispiel: Ein Netz bestehe aus baugleichen Subnetzen, die wiederum aus baugleichen Subnetzen bestehen

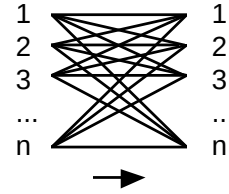
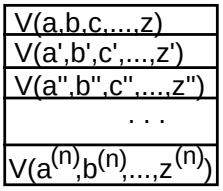
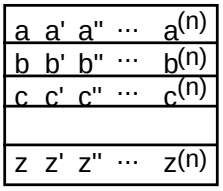
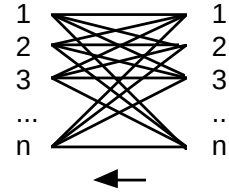
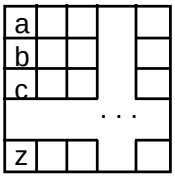
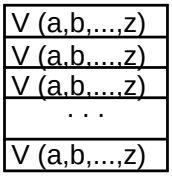
3.4 Verbindungstypen bei eng und lose gekoppelten Systemen

- **Bei eng und lose gekoppelten Systemen können verschiedene Arten von Verkehr existieren, die Resultat der verwendeten parallelen Programmierung sind**
- **Letztlich geht es dabei immer um die Frage, wer, wann mit wem Botschaften austauscht und ob dabei ein reguläres Kommunikations-Muster vorliegt**
- **Ein reguläres Kommunikations-Muster ist ein Datenaustausch, der über einfache Regeln spezifiziert werden kann**
- **In nachfolgender Tabelle sind alle Fälle von regulären Mustern beim Datenaustausch, d.h. bei Interprozessorkommunikation aufgelistet**

Bezeichnung	Typ	Vorher	Nachher
Punkt-zu-Punkt Verbindung			
Inverse Punkt-zu-Punkt Verbindung			
Multi/Broadcast Verbindung			
Inverse Multi/Broadcast Verbindung			

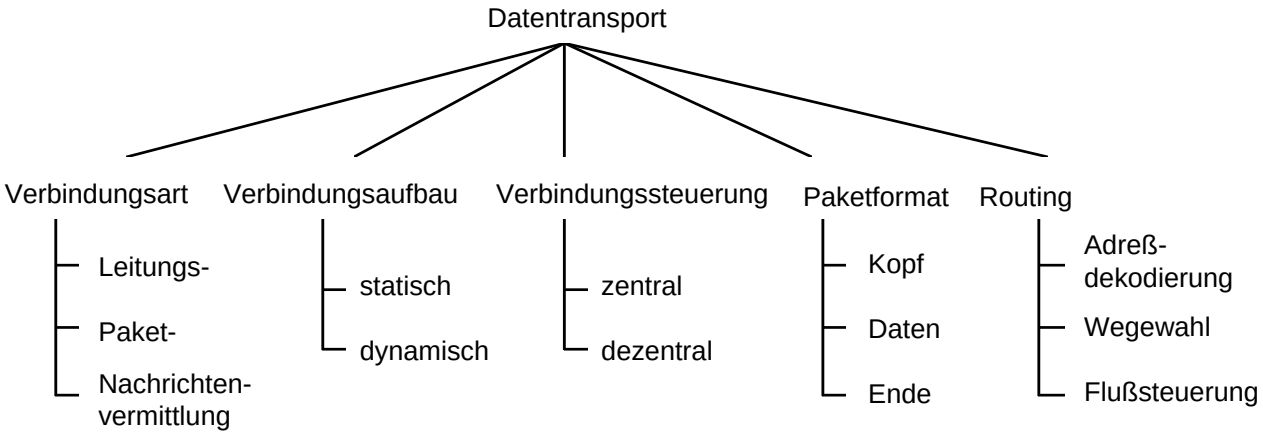
- **Komplexere Muster sind der verallgemeinerte oder personalisierte Broad-/Multicast sowie die Inversionen dazu**

Bezeichnung	Typ	Vorher	Nachher
allgemeine Multi/Broadcast Verbindung			
Inverse, allgemeine Multi/Broadcast Verbindung			 R=Reduktion
Personalisierte Multi/Broadcast Verbindung			
Inverse, personalisierte Multi/Broadcast Verbindung			V=Vektor 

Bezeichnung	Typ	Vorher	Nachher
Allgemeine, personalisierte Multi/Broadcast Verbindung			
Inverse, allgemeine, personalisierte Multi/Broadcast Verbindung			 V=Vektor

3.5 Datentransport bei eng und lose gekoppelten Systemen

- **Die Gliederung des Datentransfers bei eng und lose gekoppelten Systemen ist äquivalent zu der in statischen und dynamischen Netzen**
- **Es gibt dabei stets auch Analogien zum Datentransfer in Rechnernetzen**
- **Die Gliederung erfolgt bei eng und lose gekoppelten Systemen anhand der Kriterien Verbindungsart, Verbindungsaufbau, Verbindungssteuerung, Paket- bzw. Rahmenformat und Wegewahl (Routing bzw. Switching):**



3.5.1 Rahmen/Pakete- und Nachrichtenformate

- **Anstelle von Datenpaketen können in eng und lose gekoppelten Systemen, s.h. in statischen und dynamischen Netzen auch Datenrahmen übertragen werden**
- **Ein Paket besteht dabei aus einem oder mehreren Datenrahmen**
- **Die Nomenklatur erfolgt in Analogie zu Rechnernetzen**
- **Pakete implizieren eine Wegewahl im Verbindungsnetzwerk durch einen Router, Datenrahmen beinhalten eine Wegewahl durch einen Switch**
- **Jeder Datenrahmen wiederum besteht aus sog. flits bzw. phits**

Def.: Ein phit ist die kleinste Einheit, die das Verbindungsnetzwerk übertragen kann.

Hinweis: phit = physical transfer unit.

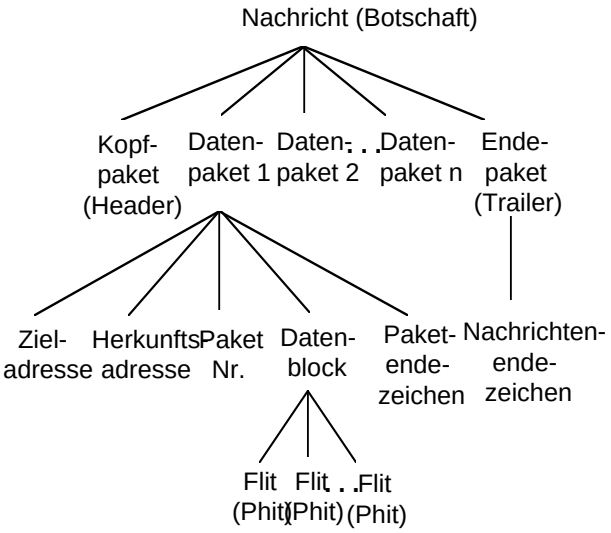
- **Das kann ein Bit, ein Byte oder ein Wort sein**

Def.: Ein flit ist die kleinste Einheit, für die es im Verbindungsnetzwerk eine Flusssteuerung gibt, sofern diese existiert.

Hinweis: flit = flow control digit.

- **Meistens sind phit und flit identisch**
- **Umgekehrt können mehrere Pakete zu einer einzigen Nachricht gehören, die per Send/Receive übertragen wird**
- **Das nachfolgende Schaubild zeigt den allgemeinen Aufbau einer Nachricht**

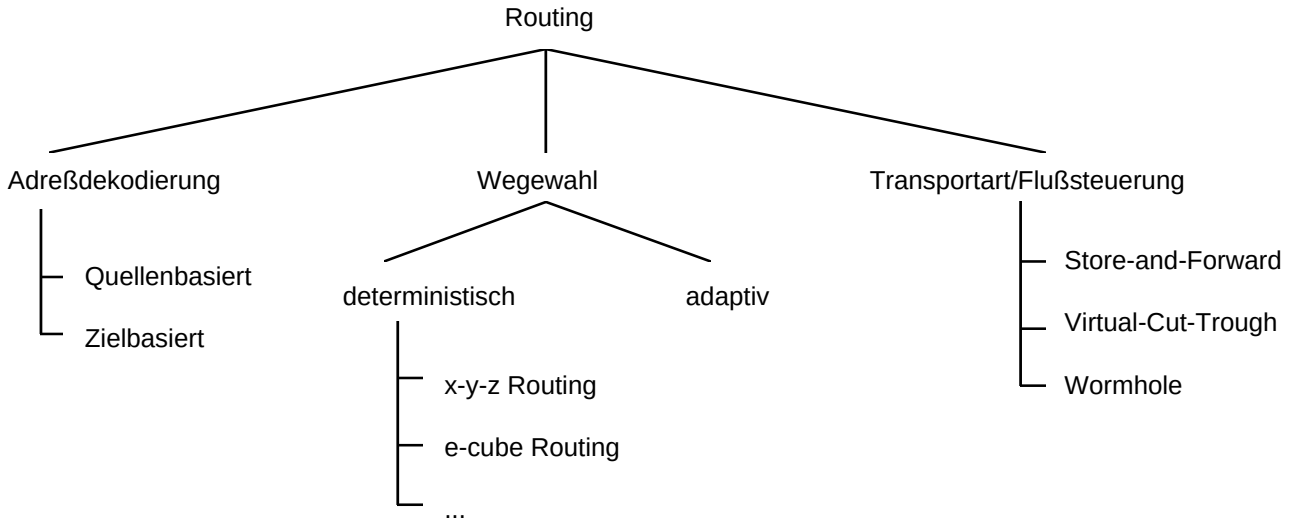
- **Hierbei wird weggelassen, dass Pakete als Datenrahmen übertragen werden (die Rahmenebene ist aus Gründen der Übersichtlichkeit nicht dargestellt)**



3.5.2 Paket-Routing bzw. Rahmen-Switching

- **Bei Verbindungsnetzwerken für Parallelrechner werden Routing und Switching begrifflich leider nicht sauber voneinander getrennt**
- **Dies gilt es beim Studium der betreffenden Literatur zu berücksichtigen**

- **Die Untergliederung des Routings/Switchings erfolgt anhand der Parameter Adressdekodierung, Algorithmus der Wegewahl (Routing/Switching) und Transportart bzw. Flusssteuerung**
- **In jeder Kategorie gibt es weitere Unterkategorien gemäß des nachfolgenden Schaubilds**



3.5.3 Transportart/Flusssteuerung

- Bei der Transportart wird zwischen **Store-and-Forward, Virtual-Cut-Through und Wormhole** unterscheiden
- Alle drei Varianten betreffen die Flusssteuerung zwischen Sende- und Empfangsknoten und die Frage, wo ein Paket/Rahmen auf dem Weg zum Ziel zwischengespeichert wird, falls Flusssteuerung erfolgt
- Flusssteuerung heißt, dass der Empfänger den Sender bremsen können muss, um nicht mit Daten überflutet zu werden, falls er langsamer als der Sender ist
- Die Flusssteuerung zwischen Sender und Empfänger erfolgt entweder über ein zurückgesendetes Acknowledge in einem Datenrahmen oder Paket oder über ein spezielles Rückleitungssignal per Hardware

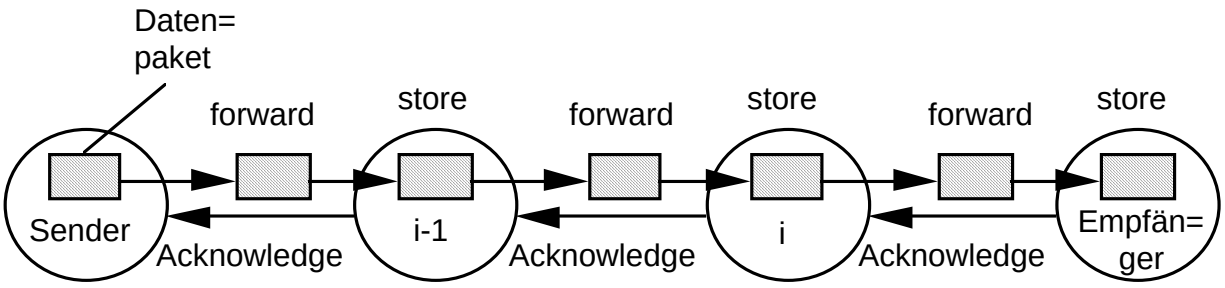
3.5.3.1 Store-and-Forward Routing

- Store-and-Forward Routing ist das Verfahren, das auch im Internet zwischen Routern angewendet wird

Def.: Store-and-Forward bedeutet, dass ein Datenpaket zwischen einem Sender und einem Empfänger komplett übertragen werden muss, bevor der Empfänger das Datenpaket inspizieren kann.

- Die Konsequenz ist, dass der Empfänger das Paket vollständig in einen Empfangspuffer einstellen muss

- In jeder Zwischenstation zwischen Sender und Empfänger erfolgt ein Zwischenspeichern des kompletten Pakets, bevor es weitertransportiert werden kann
- Dies ist in nachfolgendem Schaubild gezeigt:



3.5.3.2 Virtual-Cut-Through Routing

- Virtual-Cut-Through heißt, dass es nicht notwendig ist, das ganze Paket bei einem Zwischenknoten einzuspeichern, bevor es inspiziert werden kann

- **Um eine Wegewahlentscheidung zu treffen reicht es aus, wenn nur der Header des Pakets bei einem Zwischenknoten eingetroffen ist, so dass dieser den Header inspizieren und danach seine Wegewahlentscheidung treffen kann**

⇒ **Bei Virtual-Cut-Through wird das Paket unmittelbar nach Eintreffen des Headers zum nächsten Knoten weitergeschickt, sofern dieser nicht die Flusssteuerung einsetzt und auf die Bremse tritt**

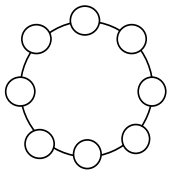
- **Der nächste Koten ist entweder ein sog. Zwischenknoten oder der Empfänger**
- **Bremst ein Zwischenknoten oder der Empfänger, dann speichert derjenige Zwischenknoten das ganze Paket, bei dem sich momentan der Header befindet**

3.5.3.3 Wormhole Routing

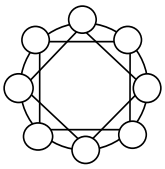
- **Im nicht-blockierenden Fall sind Virtual-Cut-Through und Wormhole genau gleich**
- **Im blockierenden Fall werden bei Wormhole Routing alle Zwischenknoten auf dem Weg zum Ziel als temporärer Speicher für einzelne Pakete/Rahmen/flits benutzt**
- **Das hat den Vorteil, dass der benötigte Zwischenspeicher viel kleiner als bei Virtual-Cut-Through oder Store-and-Forward sein kann**

4 Statische Verbindungsnetzwerke

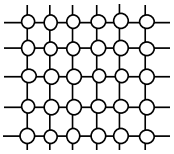
- **Die nächsten Diagramme zeigen Beispiele statischer Verbindungsnetzwerke**



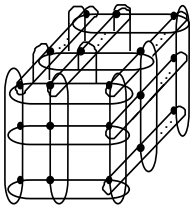
Ring



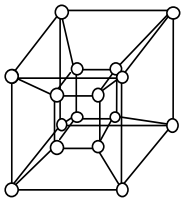
Sehnenring



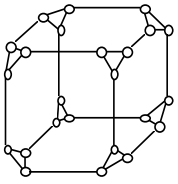
2D-Gitter



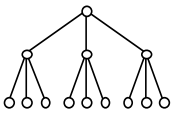
3D-Torus



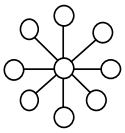
4D-Hypercube



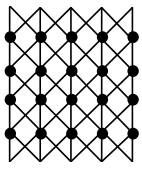
3D-Cube Connected Cycles



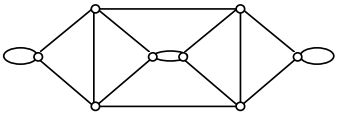
vollständiger Baum



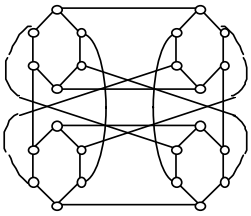
Broadcast-Stern



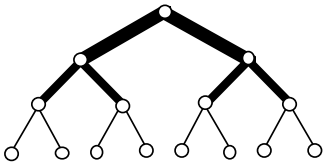
Systolisches Feld



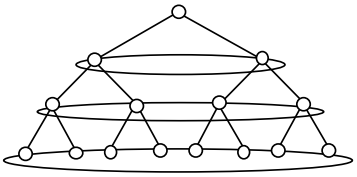
De Brujin-Graph



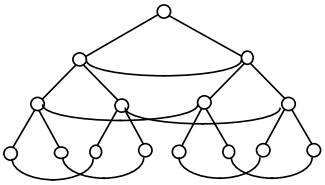
Star-Graph



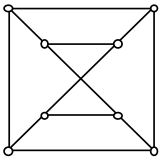
Fat Tree



X-Tree

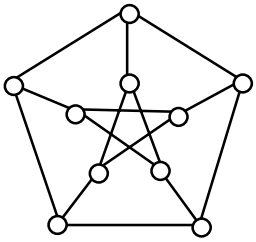


Hypertree

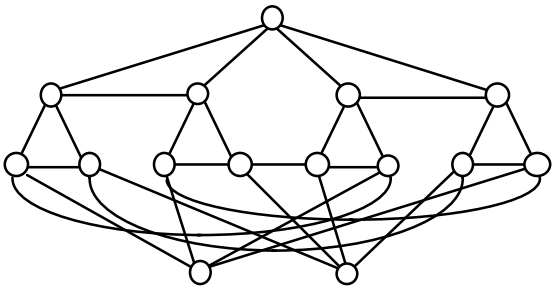


Twisted Cube

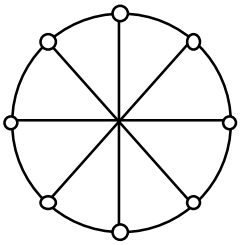
155



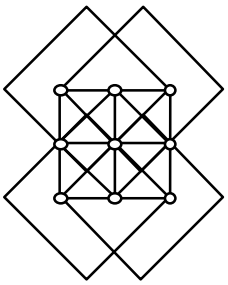
Moore-Graph für N=10
(= Petersen-Graph)



Reduzierter Kneser-Graph (N=15)

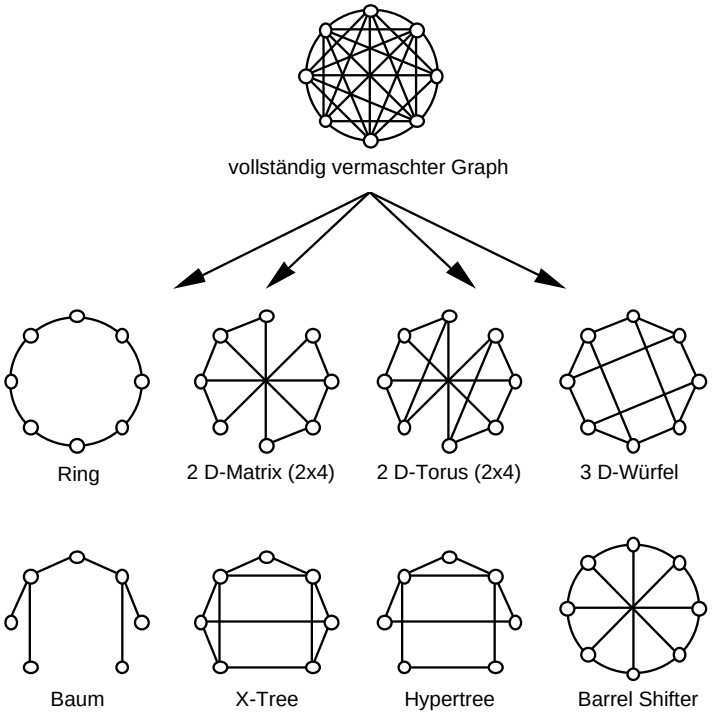


Cayley-Graph (N=8)



Balanced Incomplete Block Design (N=9)

156

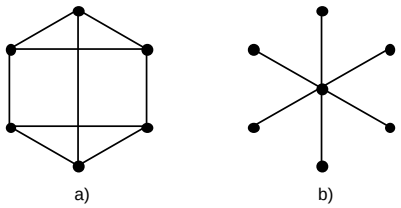


4.1 Symmetrie bei statischen Netzen

Def.: Regelmäßiger (regulärer) Graph: an jedem Knoten sind gleich viele Kanten.

- **Reguläre Graphen setzen die Existenz von Knotensymmetrie oder von Kantensymmetrie voraus**
- **Aus Knotensymmetrie oder Kantensymmetrie alleine folgt aber noch nicht die Regularität eines Graphen => notwendige aber nicht hinreichende Bedingung**

Beispiel: a) Knotensymmetrie aber keine Kantensymmetrie, b) Kantensymmetrie aber keine Knotensymmetrie



- **Vorteile von Graphen, die knoten- und kantensymmetrisch sind:**
  - **Der Graph sieht aus der Sicht jedes Knoten gleich aus**
  - **Dies erleichtert die Programmierung des darauf basierenden Parallelrechners, denn auf jedem Prozessor kann derselbe parallele Algorithmus ohne Software-Änderung ausgeführt werden**

- **Spezialfälle, wie z.B. Prozessoren am Rand eines Gitters, etc., gibt es nicht**
- **Broadcast, Multicast etc. sind leichter zu implementieren**
- **Programme sind einfacher und übersichtlicher als im unsymmetrischen Fall**
- **Der Verkehr auf den Kanten kann gleichmäßig verteilt werden, da es keine ausgezeichneten Kanten gibt**
- **Das Routing ist für jeden Prozessor gleich**
- **Routing-Tabellen entfallen. Sie werden durch einen einheitlichen Routing-Algorithmus ersetzt.**
- **Die Leistungsanalyse des Verbindungsnetzwerkes in Bezug auf Durchsatz und Latenz, sowie seine Herstellbarkeit, Testbarkeit und Skalierbarkeit sind vereinfacht**

4.2 Metriken bei statischen Netzen

- **Wichtige Metriken zur Beurteilung eines Netzes: Grad, Abstand zwischen 2 Knoten, Durchmesser, mittlerer Knotenabstand, Halbierungsbandbreite**

Def.: Grad = Zahl der Kanten pro Knoten bei einem regulären Graphen

Def.: Abstand zwischen 2 Knoten = Minimum aller Pfadlängen zwischen den beiden Knoten

Def.: Durchmesser = Maximum aller Abstände

Def.: mittlerer Knotenabstand = Mittelwert aller Abstände

Def.: Halbierungsbandbreite = Minimum der Zahl von Kanten, die zwei beliebige aber gleich große Hälften miteinander verbinden, multipliziert mit der Bandbreite pro Kante

- **Weitere Metriken: mittlere Nachrichtendichte, kleinste Erweiterung**

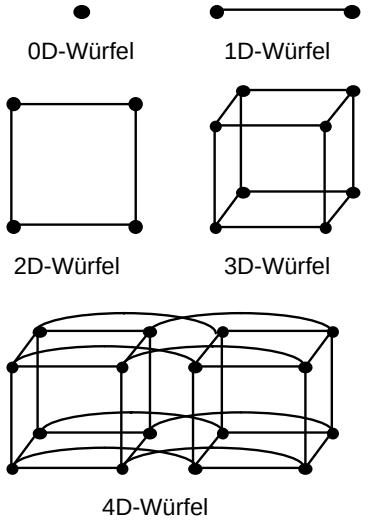
Def.: mittlere Nachrichtendichte = Zahl der Datenpakete/-rahmen pro Zeiteinheit auf einem Kanal, gemittelt über alle Kanäle

Def.: kleinste Erweiterung = kleinste Zahl von Knoten, die unter Beibehaltung der Grundstruktur (Topologie) hinzugefügt werden kann

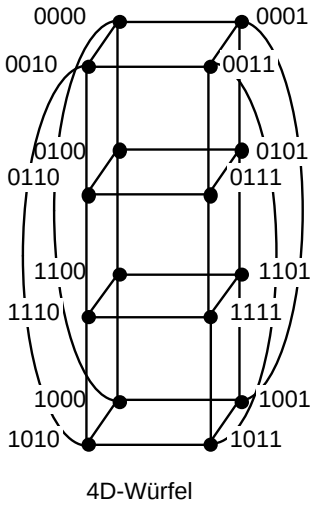
- **Das Ziel bei allen statischen Topologien für Parallelrechner ist ein möglichst kleiner Grad und ein möglichst kleiner Durchmesser**
- **Dies sind einander widersprechende Forderungen**
- **Der Grad des statischen Verbindungsnetzwerkes bestimmt seine Kosten, der Durchmesser bestimmt seine Latenz**
- **Neben Grad und Durchmesser spielt auch die Halbierungsbandbreite eine Rolle**
- **Die Latenz sagt, wie lange es dauert, bis Daten zum Ziel kommen, die Halbierungsbandbreite ist ein Maß dafür, wie viele Daten pro Sekunde unterwegs zum Ziel sein können**
- **Ein guter Kompromiss zwischen Grad und Durchmesser ist der Hypercube**
- **Im Hypercube können außerdem viele andere Topologien „eingebettet“ (=untergebracht) werden**
- **Für den Hypercube wurden deshalb viele parallele Algorithmen entwickelt, und es wurden Parallelrechner dieser Topologie in größeren Stückzahlen gebaut**

4.3 Konstruktion eines n-dimensionalen Überwürfels (Hypercube)

- **Es gibt zwei Arten der Erzeugung: 1) rekursiv von klein nach groß und 2) konstruktiv**



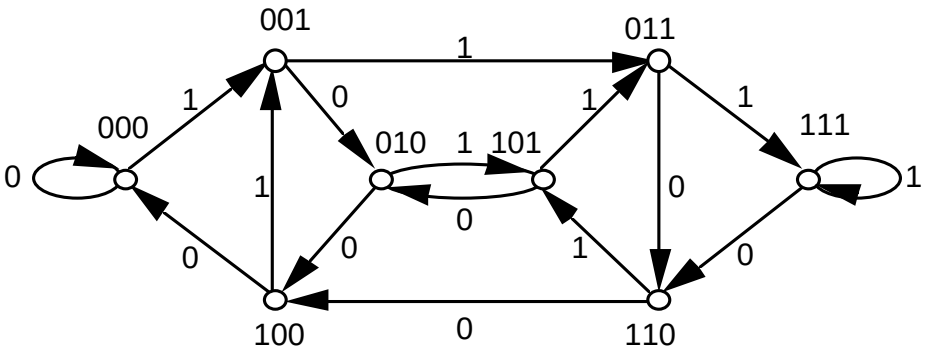
a) rekursive Erzeugung



b) konstruktive Erzeugung

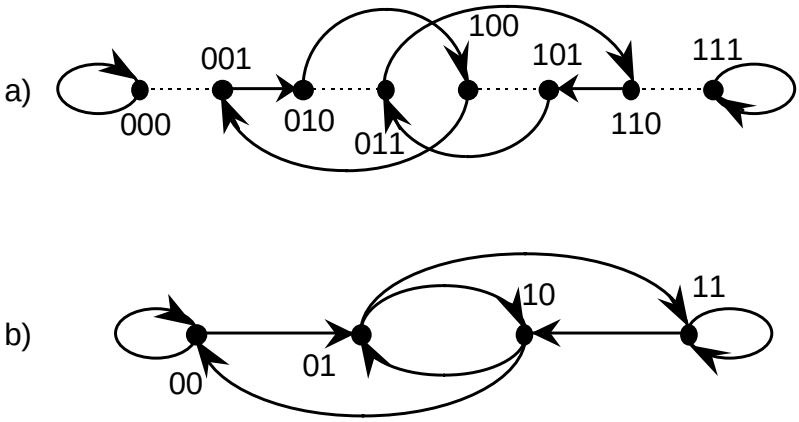
4.4 Konstruktion eines de Bruijn-Graphen

- **Der de Bruijn-Graph ist bezgl. Grad, Durchmesser und Halbierungsbandbreite ebenfalls rel. gut, allerdings noch weniger anschaulich als der Hypercube**



- **Der Graph eignet sich speziell für die Berechnung der parallelen schnellen Fouriertransformation und für andere Algorithmen aus der digitalen Signalverarbeitung**
- **Der Grund dafür ist seine topologische Verwandschaft mit der sog. Shuffle-Exchange-Permutation, die bei der schnellen Fouriertransformation eine wichtige Rolle spielt**

Beispiel: für die topologische Verwandschaft zwischen de Brujin-Graph (b) und Shuffle-Exchange (a)



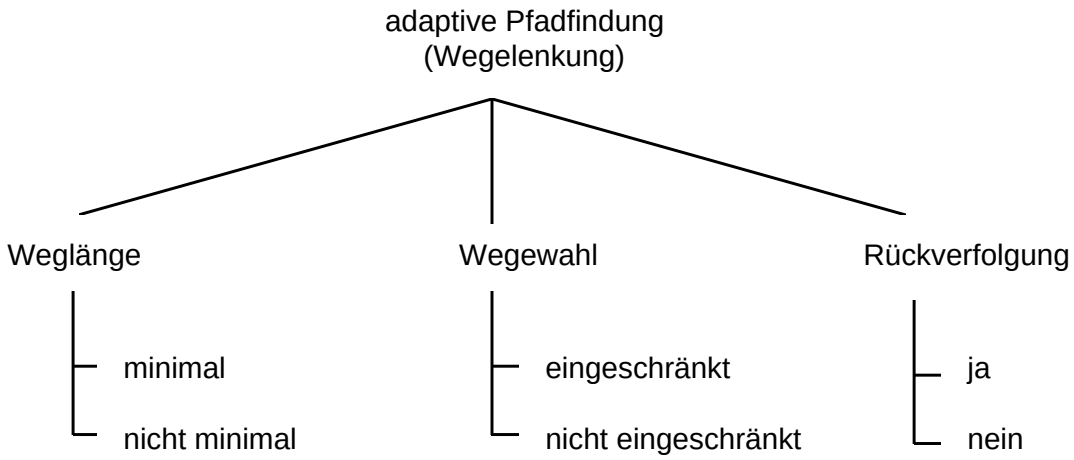
- **Betrachtet man im Shuffle-Exchange-Graphen für 8 Cores (a) nur die 2 MSBs der Knotenadressen, dann verschmelzen alle Knoten miteinander, die sich im LSB unterscheiden, und man erhält den de Brujin-Graph (b) für 4 Cores**

4.5 Routing in statischen Netzen

- **Es gibt in statischen Netzen deterministisches und adaptives Routing**
- **Leistungsfähiger aber auch schwieriger in der Implementierung ist das dyn. Routing**

Beispiel: deterministisches Routing x-y-z-Routing beim 3D-Gitter und beim 3D-Torus, sowie e-Cube-Routing beim Hypercube.

- **Beim adaptiven Routing gibt es eine größere Zahl von Parametern:**



- **Weglänge: minimal heißt, dass nur kürzeste Wege erlaubt sind**
- **Wegewahl: eingeschränkt heißt, dass von allen möglichen Alternativen einige ausgespart werden, weil sie z.B. zu einem Deadlock führen könnten**
- **Rückverfolgung (Backtracking): ja heißt, dass Datenrahmen/-pakete auf ihrem Weg auch ein Stück zurücksetzen dürfen, um eine andere Wegealternative zu wählen**

4.6 Das Deadlock-Problem bei Interprozessor-Kommunikation

- **Das Deadlock-Problem ist bei der Parallelrechnerprogrammierung gravierend und tritt u.a. in folgender Situation auf:**

1.) Ein Datenrahmen belegt eine Ressource im Parallelrechner wie z.B. einen Kanal oder einen Puffer, da sie die Ressource zum Senden eines Datenrahmens benötigt

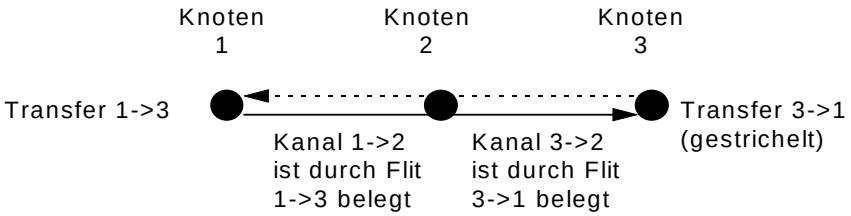
2.) Ein zweiter Datenrahmen benötigt zur Durchführung seines Datenaustauschs die unter 1. belegte Ressource

3.) Der erste Datenrahmen kann nur dann seine Ressource freigeben, wenn der zweite Datenrahmen zugestellt worden ist

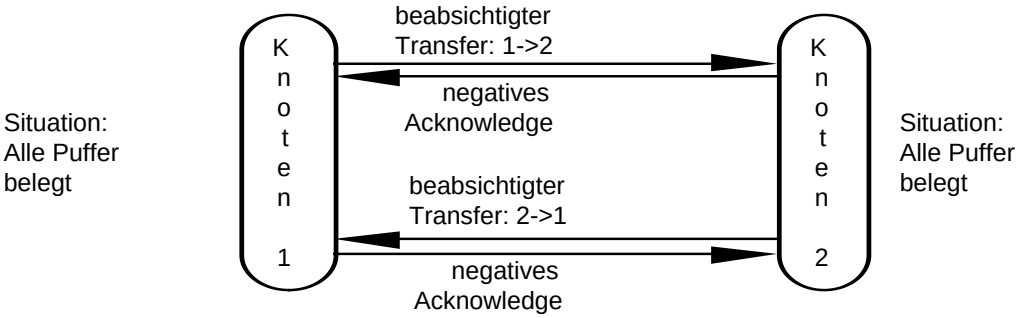
⇒ **gegenseitiges Warten im Kreis (=Verklemmung)**

- **Das Beispiel kann auf n>2 Rahmen verallgemeinert werden**

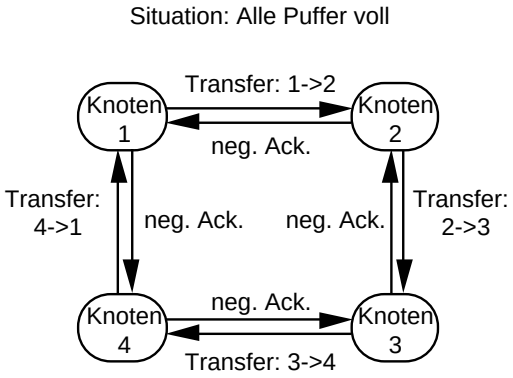
4.6.1 Verklemmung aufgrund eines belegten Kanals



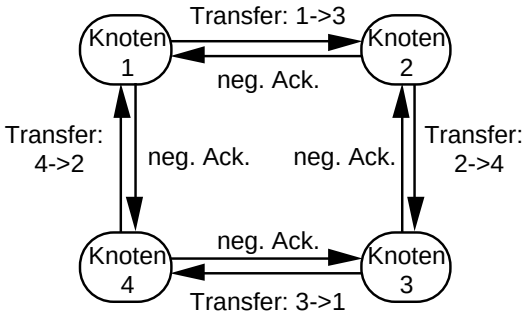
4.6.2 Verklemmung aufgrund zweier belegter Puffer



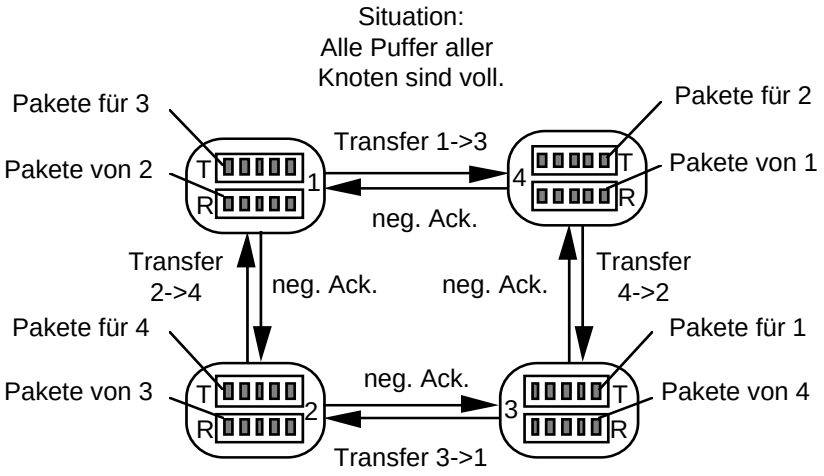
4.6.3 Verklemmung aufgrund vier belegter Puffer



4.6.4 Verklemmung aufgrund vier belegter Kanäle



4.6.5 Verklemmung trotz getrennter Sende- und Empfangspuffer



Hinweis: Die Situation ist dieselbe wie im vorherigen Bild, allerdings sind die Nummern der Knoten 2 und 4 vertauscht, weshalb sich 2->4 in 4->2 ändern muss, sowie 4->2 in 2->4, um dieselbe Situation darzustellen.

- **Bei einem Deadlock über vier Knoten aufgrund von belegten Puffern hilft es nicht, getrennte Sende- und Empfangspuffer einzuführen**
- **Eine Lösung dieses Deadlock-Problems besteht z.B. in der Einführung von sog. Pufferklassen und zwar genau so vielen, wie Knoten am Deadlock beteiligt sind**

- Sind die Datenrahmen in ausreichend vielen verschiedenen Pufferklassen gespeichert, nehmen sie sich nicht gegenseitig den benötigten Puffer weg
- Im obigen Beispiel würden vier statt zwei Puffer pro Knoten die Verklemmung auflösen

5 Dynamische Verbindungsnetzwerke

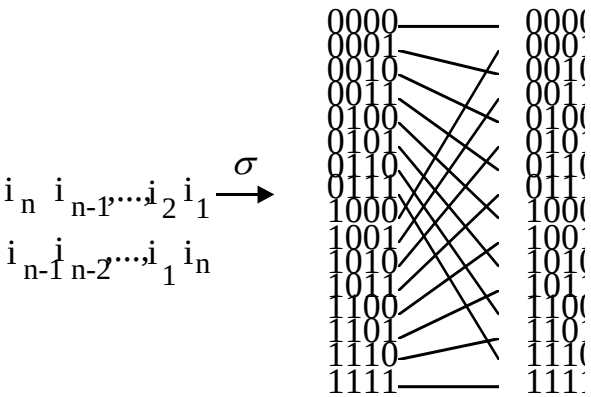
- Dynamische Verbindungsnetzwerke sind Serienschaltungen von Permutationsfunktionen, die durch eine Verdrahtung zwischen Kreuzschaltern realisiert werden

5.1 Permutationsfunktionen

- Im folgenden werden wichtige Permutationsfunktionen erläutert
- Diese sind die Perfect Shuffle-Permutation, die Butterfly-Permutation und die Reversal-Permutation sowie ihre inverse Abbildungen
- Hinzu kommen noch Varianten bei der Perfect Shuffle-Permutation und der Butterfly-Permutation, die als Sub-/Super-Shuffle bzw. als Sub-/Super-Butterfly bezeichnet werden
- Verallgemeinerungen dieser Permutationsfunktionen sind ebenfalls möglich, z.B. indem man von einem binären zu einem ternären oder quaternären Zahlensystem übergeht

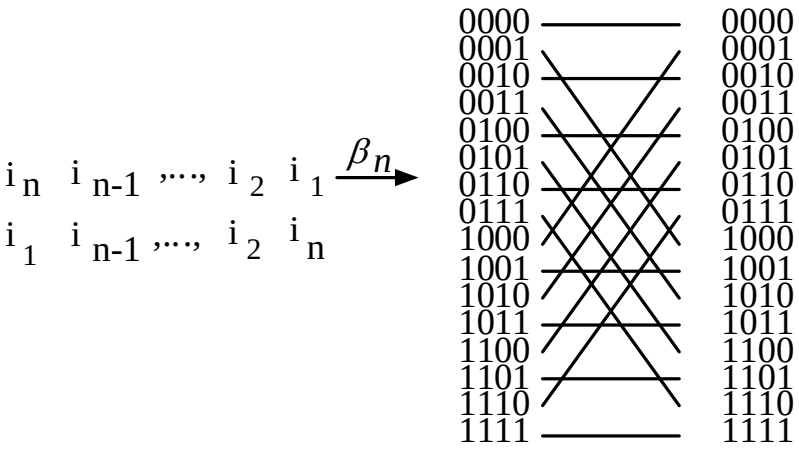
5.1.1 Die Perfect Shuffle-Permutation

- Bei der Perfect Shuffle-Permutation σ_n hat ihren Namen vom Mischen beim Kartenspielen
- Bei σ_n werden die n Adressbits zyklisch um eine Position nach links rotiert



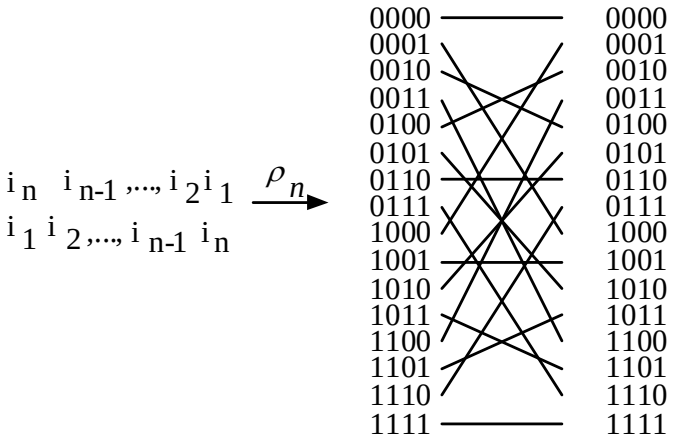
5.1.2 Die Butterfly-Permutation

- **Bei der Butterfly-Permutation β_n wird das Bit 1 (LSB) mit dem Bit n (MSB) getauscht**



5.1.3 Die Reversal-Permutation

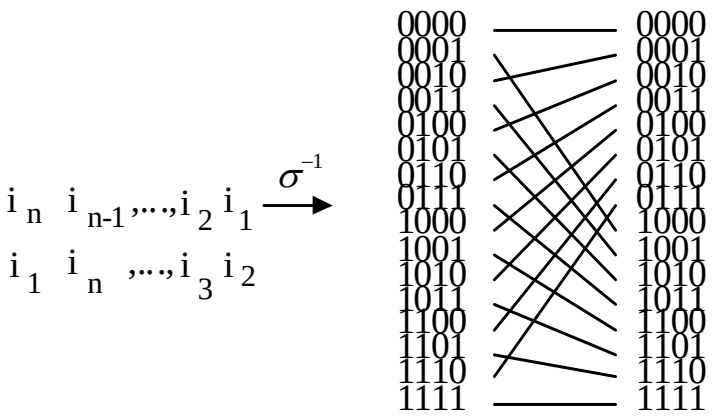
- **Bei der Reversal-Permutation ρ_n wird die Reihenfolge der Adressbits umgedreht (gespiegelt)**



- **Nachfolgend werden einige wichtige Varianten von Permutationsfunktionen dargestellt**

5.1.4 Die inverse Perfect Shuffle-Permutation

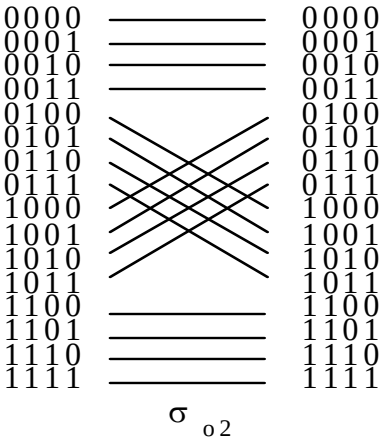
- **Hierbei werden die Adressbits zyklisch nach rechts rotiert**



5.1.5 Die Supershuffle-Funktion

- **Bei der Supershuffle-Funktion $\sigma_{ok}^{(n)}$ erfolgt eine zyklische Linksrotation der oberen k Bits (MSBs) bei insgesamt n Bit Wortbreite**

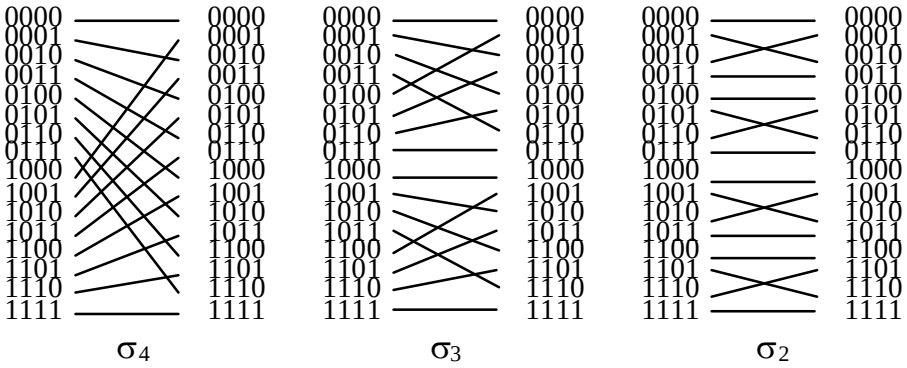
Beispiel: ***$\sigma_{ok}^{(n)}$ für n=4 und k=2***



5.1.6 Die Subshuffle-Funktion

- **Bei der Subshuffle-Funktion $\sigma_{uk}^{(n)}$ erfolgt eine zyklische Linksrotation der unteren k Bits (LSBs) bei insgesamt n Bit Wortbreite**

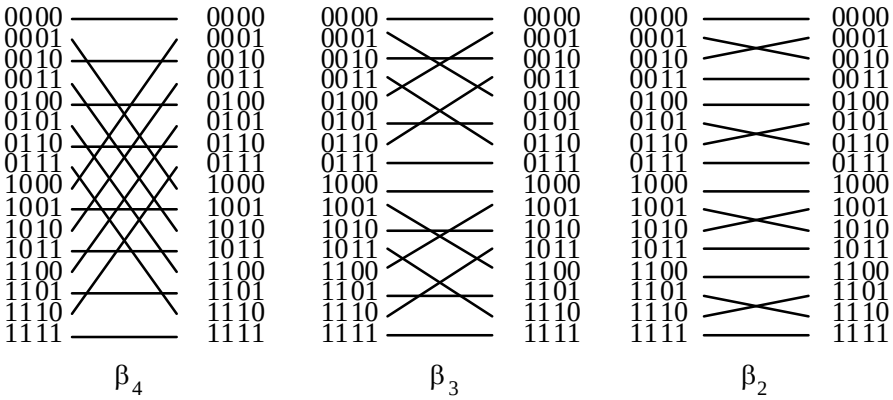
Beispiel: *$\sigma_{uk}^{(n)}$ für n=4 und k=4, 3 und 2.*



5.1.7 Die Super-/Subbutterfly-Funktion

- **Bei der Superbutterfly-Funktion $\beta_{ok}^{(n)}$ wird das MSB wird mit dem k-ten Bit von links getauscht (1<k<n)**
- **Bei der Subbutterfly-Funktion $\beta_{uk}^{(n)}$ wird das LSB mit dem k-ten Bit von rechts getauscht (1<k<n)**
- **Die inversen Abbildungen zu Superbutterfly und Subbutterfly sind identisch mit den normalen Superbutterfly- bzw. Subbutterfly-Funktionen, d.h. zwischen normal und invers braucht nicht unterschieden zu werden**
- **Bei Parallelrechnern sind nur $\sigma_{uk}^{(n)}$ und $\beta_{uk}^{(n)}$ relevant, da die Super-Varianten keine echte Durchmischung der Daten liefern**
- **Beliebige Abbildungen von Eingängen eines dynamischen Verbindungsnetzwerkes auf seine Ausgänge sind nur bei echten Durchmischungen möglich**

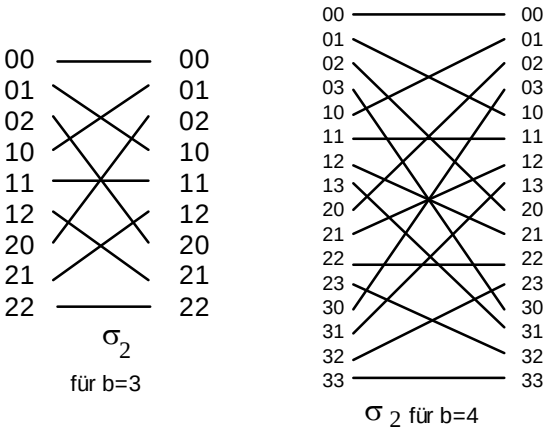
Beispiel: $\beta_{uk}^{(n)}$ ***für n=4 und k=4, 3 und 2.***



5.1.8 Ternäre und quaternäre Shuffle-Permutation

- **Hierbei wird von der Zahlenbasis zwei auf die Zahlenbasis drei bzw. vier übergegangen, um die Zahl der Ein- und Ausgänge feiner abstufen zu können**
- **Es erfolgt dazu eine zyklische Linksrotation von Adressziffern 0,1,2 (=ternär) bzw. 0,1,2,3 (=quaternär) anstelle von Adressbits**

Beispiel: $\sigma_{ternär}$ ***für N=9, b=3 und k=2 (= ternär) bzw.*** $\sigma_{quaternär}$ ***für N=16, b=4 und k=2 (=quaternär)***

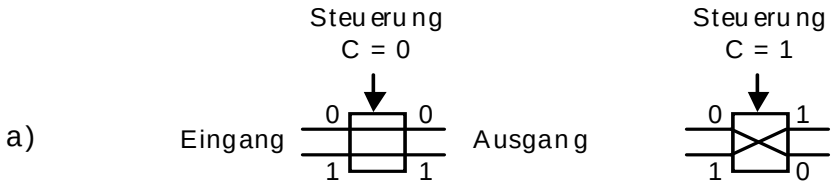


5.2 Kreuzschalter

- **Kreuzschalter sind bei dynamischen Netzen wichtig. Es gibt sie in insgesamt drei Variationen: 1) normal, 2) mit Broadcast, 3) mit allen Abbildungsmöglichkeiten**

5.2.1 Kreuzschalter mit Broadcast

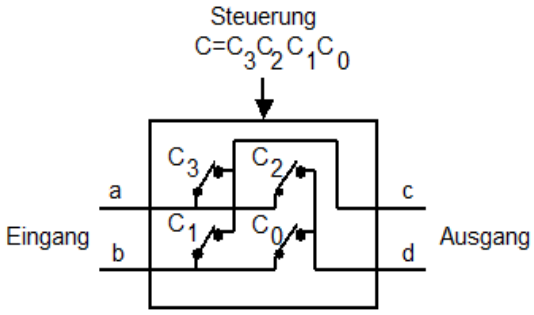
- **Kreuzschalter können auf eine Broadcast-Funktion erweitert werden, um dyn. Verbindungsnetzwerke mit Broadcast zu bauen (einer sendet an alle)**



- **Dazu muss das Steuerbit des Kreuzschalters um ein Bit erweitert werden**

5.2.2 Kreuzschalter als Kreuzschienenverteiler

- **Eine nochmalige Erweiterung der Kreuzschalter resultiert in einem Kreuzschienenverteiler der Größe 2x2, der alle Abbildungen von 2 Eingängen auf 2 Ausgängen realisiert**



5.2.3 Die Verbindungsmöglichkeiten des 2x2 Kreuzschienenverteilers

		$C_1 C_0$			
		00	01	10	11
C_3C_2	00	k.V.	b->d	b->c	b->c,d
	01	a->d	n.e.	a->d; b->c	n.e.
	10	a->c	a->c; b->d	n.e.	n.e.
	11	a->c,d	n.e.	n.e.	n.e.

0 = offen

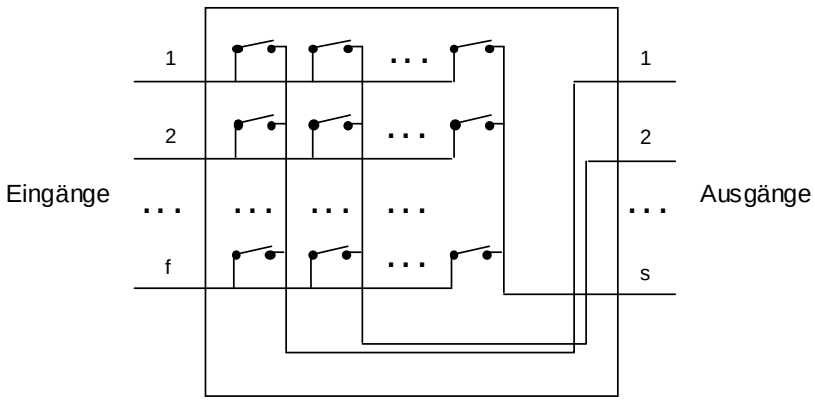
1 = geschlossen

k.V. = keine Verbindung

n.e. = nicht erlaubt

- **In logN-Verbindungsnetzwerkes sind nur der normale Kreuzschalter und seine Broadcast-Erweiterung relevant**
- **2x2 Kreuzschienenverteiler werden nur in Clos-Netzen in einer nochmaligen Verallgemeinerung verwendet, dem sog. fxs-Schalter mit f,s >2**

5.2.4 Ein fxs-Schalter und ein fxs-Kreuzschienenverteiler sind identisch



5.2.5 Funktion eines 2x2-Kreuzschalters beim Durchgang eines Datenrahmens

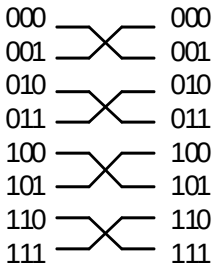
$i_1 o_k$	O	C
00	0 (gerade)	0 ('=')
01	1 (ungerade)	1 ('x')
10	0 (gerade)	1 ('x')
11	1 (ungerade)	0 ('=')

- **Es sei i_1 das LSB der Adresse eines Datenrahmens am Eingang eines 2x2-Kreuzschalter, und o_k die Adresse desselben Datenrahmens am Ausgang des 2x2-Kreuzschalters**
- **Dann kann man sagen, dass der 2x2-Kreuzschalter beim Durchgang des Datenrahmens i_1 durch o_k ersetzt, weil der Datenrahmen entweder am oberen oder am unteren Ausgang erscheinen kann**
- **Zusätzlich kann o_k je nach Steuerbit C den Wert 0 oder 1 haben**
- **Mit Hilfe von C kann man somit i_1 gezielt verändern, d.h. den Datenrahmen verschieben**
- **Wählt man o_k so, das es den Wert der gewünschten Zieladresse an einer wählbaren Bitstelle annimmt, dann nähert sich der Datenrahmen mit jeden Durchgang durch einen 2x2-Kreuzschalter schrittweise dem Ziel, sofern eine geeignete Verdrahrung zwischen den Kreuzschaltern vorhanden ist**

- **Z.B. könnte die k-te Stufe eines dyn. Verbindungsnetzwerkes des LSB der jeweiligen Zieladresse so verändern, dass es dem k-ten Bit der gewünschten Zieladresse entspricht**
- **Allgemein gilt: haben die Herkunfts- und die Zieladressen n Bits, dann kann der an das Netzeingang angelegte Datenrahmen in n Schritten zu einer beliebigen Ausgangsadresse ge-routet werden**
- **Mathematisch ausgedrückt realisiert der 2x2-Kreuzschalter eine sog. Exchange-Permutation, weil er $i_1$durch o_k ersetzt ($1 \le k \le n$)**

5.2.6 Die Exchange-Permutation ε

Beispiel: *ε für n=3*



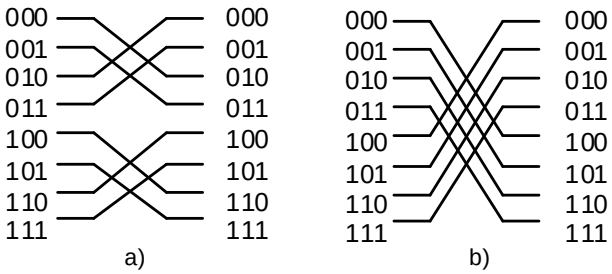
- **Allerdings ist die Exchange-Permutation für jeden 2x2-Kreuzschalter einzeln einstellbar, d.h. manche Schalter stehen auf „Exchange“ andere nicht**

- **Dies unterscheidet eine Schalterstufe in einem dyn. Verbindungsnetzwerk von einer echten Exchange-Permutation, bei der alle Eingänge gekreuzt werden**

5.2.7 Die Subexchange-Permutation

- **Bei der Subexchange-Permutation $\varepsilon_{k,n}$ (=allgemeine Exchange-Permutation) wird nicht das LSB sondern das k-te Bit von rechts (= k-tes LSB) durch sein Komplement ersetzt, bei insgesamt n Bits Wortlänge**

Beispiel: *$\varepsilon_{k,n}$ für n=3 und k=2 (Fall a) bzw. k=3 (Fall b)*



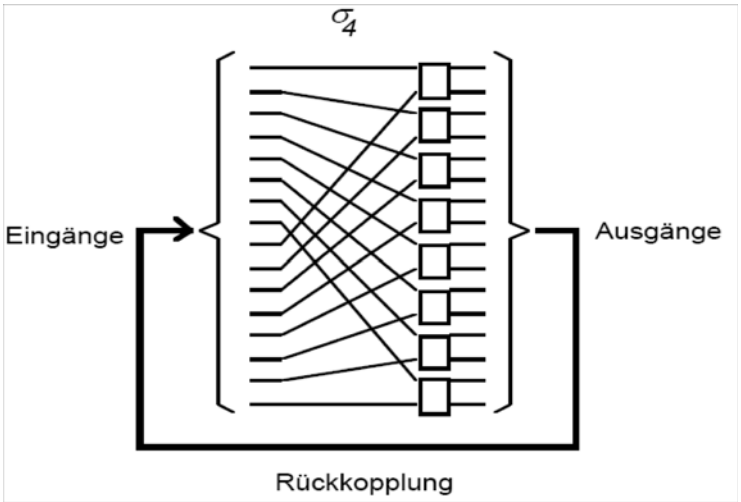
5.3 Die klassischen LogN-Netze

- **Im folgenden werden die klassischen logN-Netze dargestellt, die aus den zuvor erläuterten Permutationsfunktionen und 2x2-Kreuzschaltern aufgebaut sind**

5.3.1 Das Shuffle-Exchange-Netzwerk

- **Das Shuffle-Exchange-Netzwerk wurde von H. Stone erfunden**

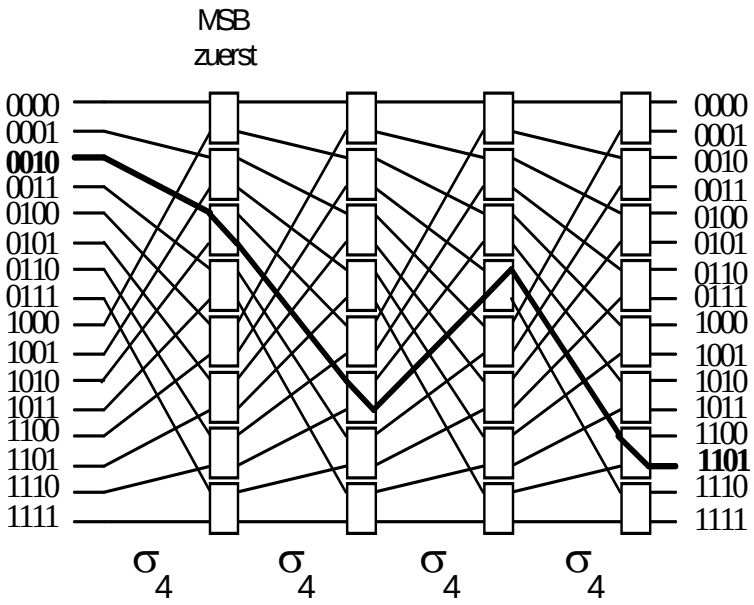
Beispiel: *für N = 16 Ein-/Ausgänge*



5.3.2 Das Omega-Netz

- Das Omega-Netz wurde von D. Lawrie erfunden

Beispiel: für N = 16 Ein-/Ausgänge

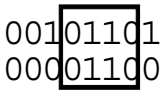


5.3.2.1 Self Routing beim Omega-Netz

- Die Wegewahl im Omega-Netz hat eine besondere Eigenschaft: das Paket kann alleine anhand der Zieladresse selbstständig den Weg durch das Netz finden
- Eine Wegewahl mit Hilfe von Routing-Komponenten des Netzes ist nicht notwendig
- Diese Eigenschaft wird als **Self Routing** bezeichnet
- Alle logN-Netze sowie die SW- und CC-Banyans haben **Self Routing**

5.3.2.2 Kollision zweier Pfade im Omega-Netz

- Alle logN-Netze sowie die SW- und CC-Banyans können jeden Eingang mit jedem Ausgang verbinden, allerdings nicht für alle Eingänge gleichzeitig
- Bereits ab zwei Paketen, die gleichzeitig im Netz sind, kann es zu Kollisionen kommen
- Zur Feststellung, ob zwei Pfade in einem Omega-Netz kollidieren, kann man die Zieladressen mit Hilfe einer rechteckigen Schablone miteinander vergleichen

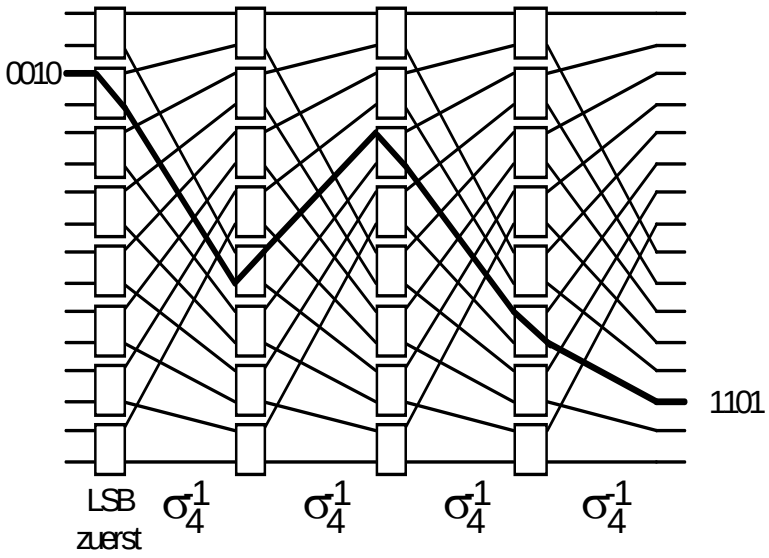


- Die Schablone wird bitweise von links nach rechts durch die konkatenierten Strings von Quell- und Zieladressen beider Pfade geschoben
- Wenn zwei gleiche Teilstrings in der Schablone sichtbar werden, tritt eine Kollision auf
- Die Stelle der Kollision entspricht dem Wert, der durch die Schablone ablesbar ist

5.3.3 Das Flip-Netz

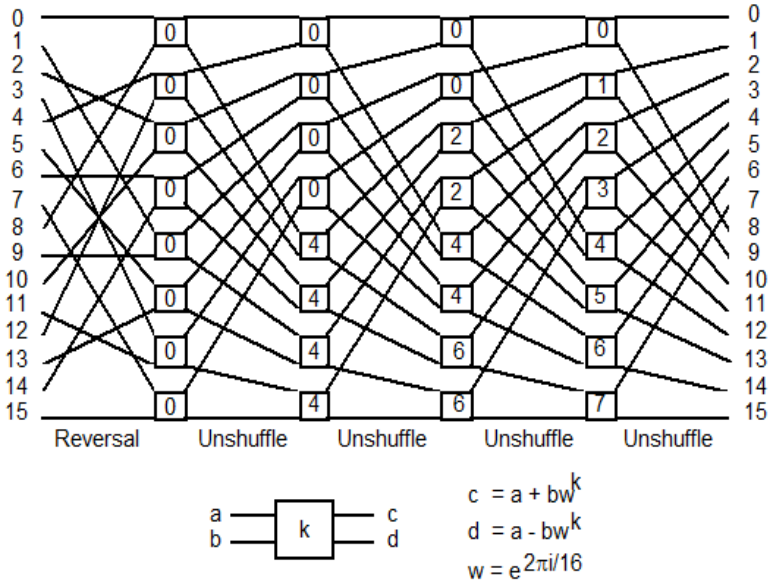
- **Das Flip-Netz wurde von K. Batcher erfunden**

Beispiel: für N = 16



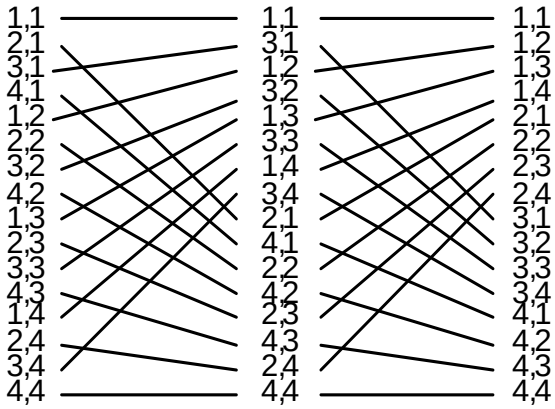
5.3.3.1 Analogie zwischen dem Flip-Netz und dem Signalflussgraph der Pease FFT

Beispiel: für 16 Punkte



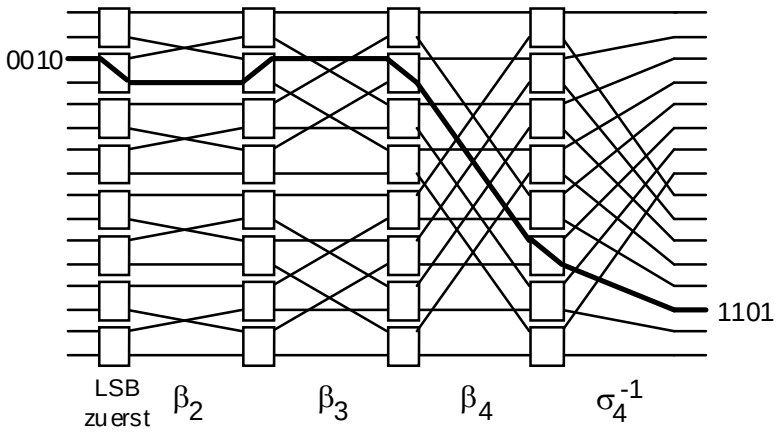
5.3.3.2 Analogie zwischen dem Flip-Netz und der Transposition einer Matrix

Beispiel: [4x4]-Matrix



5.3.4 Das Indirect Binary n-Cube-Netz

Beispiel: N=16



5.3.4.1 Funktion eines 16x16 Indirect Binary n-Cube-Netzes

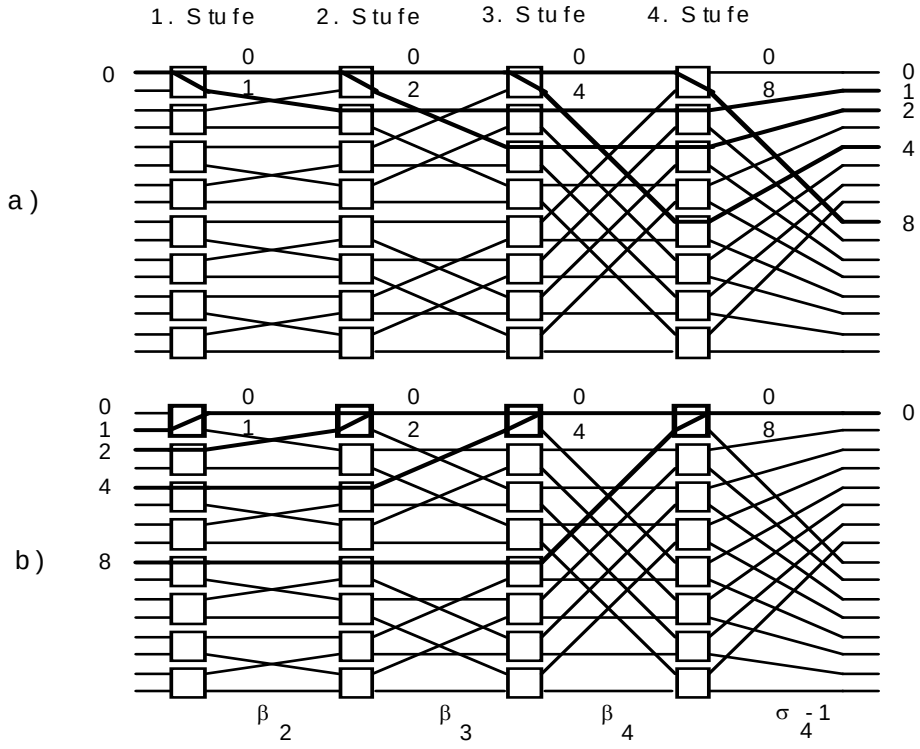
$$i_4 i_3 i_2 i_1 \xrightarrow{E(i_1,o_1)} i_4 i_3 i_2 o_1 \xrightarrow{\beta_2}$$
$$i_4 i_3 o_1 i_2 \xrightarrow{E(i_2,o_2)} i_4 i_3 o_1 o_2 \xrightarrow{\beta_3}$$
$$i_4 o_2 o_1 i_3 \xrightarrow{E(i_3,o_3)} i_4 o_2 o_1 o_3 \xrightarrow{\beta_4}$$
$$o_3 o_2 o_1 i_4 \xrightarrow{E(i_4,o_4)} o_3 o_2 o_1 o_4 \xrightarrow{\sigma_4^{-1}}$$
$$o_4 o_3 o_2 o_1 = O$$

- **Funktion eines Indirect 2^nx2^n Binary n-Cube-Netz**

$$I = i_n i_{n-1},\ldots,i_2 i_1 \xrightarrow{E(i_1,o_1)} i_n i_{n-1},\ldots,i_2 o_1 \xrightarrow{\beta_2}$$
$$i_n i_{n-1},\ldots,o_1 i_2 \xrightarrow{E(i_2,o_2)} \ldots$$
$$i_n o_{n-2},\ldots,o_2 o_1 i_{n-1} \xrightarrow{E(i_{n-1},o_{n-1})} i_n o_{n-2},\ldots,o_2 o_1 o_{n-1} \xrightarrow{\beta_n}$$
$$o_{n-1} o_{n-2},\ldots,o_2 o_1 i_n \xrightarrow{E(i_n,o_n)} o_{n-1} o_{n-2},\ldots,o_2 o_1 o_n \xrightarrow{\sigma_n^{-1}}$$
$$o_n o_{n-1} o_{n-2},\ldots,o_2 o_1 = O$$

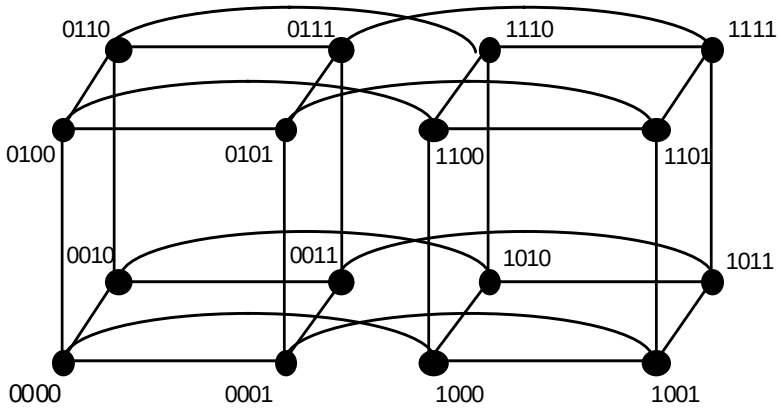
5.3.4.2 Analogie zwischen dem Indirect Binary n-Cube-Netz und dem Hypercube

- **Verbinden von Eingang 0 mit Ausgängen 1,2,4 oder 8 (a) und umgekehrt (b)**

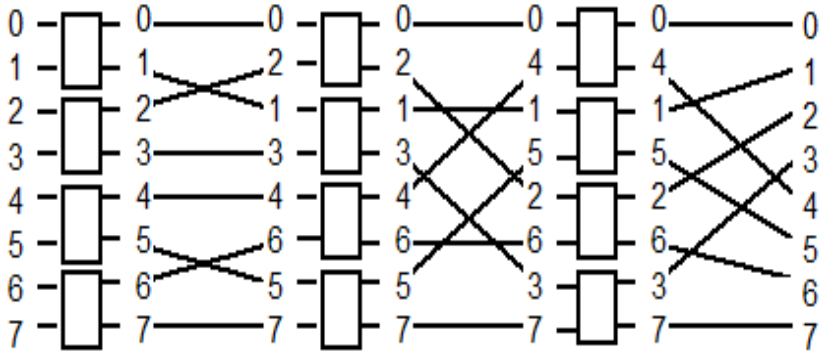


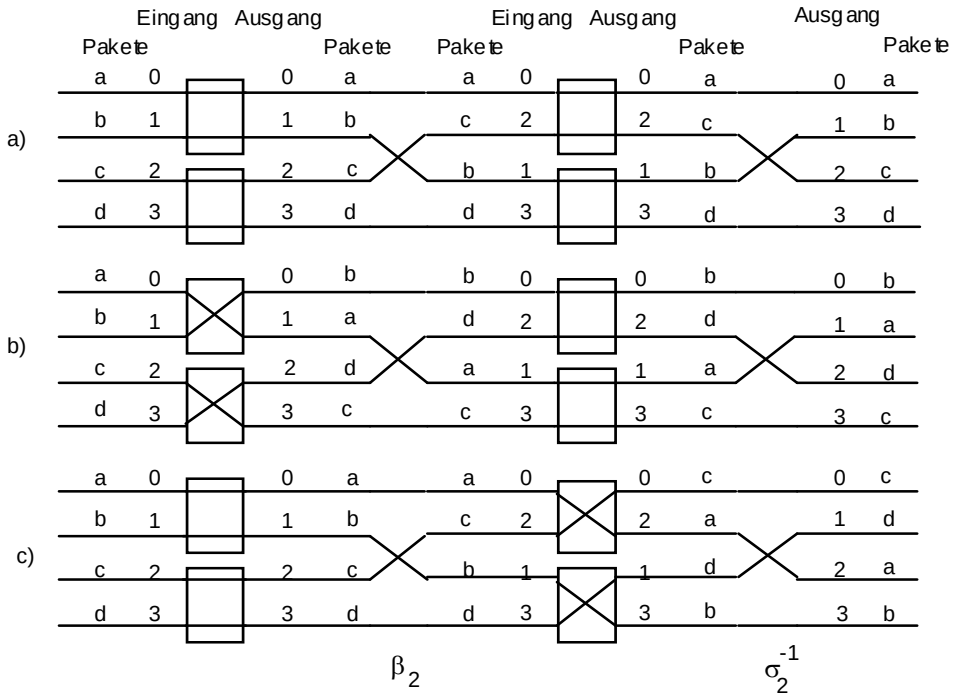
195

- **Zur Erinnerung: der Hypercube**



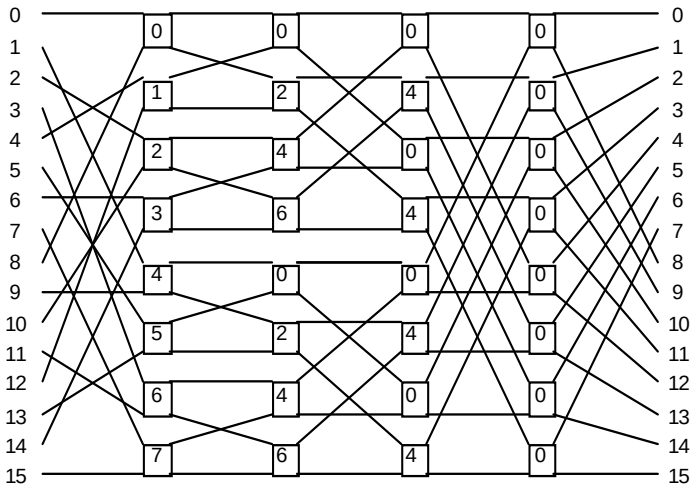
- **Alternatives Nummerierungsschema im Indirect Binary n-Cube-Netz**
- **Beispiele eines Indirect Binary n-Cube-Netz bei alternativer Nummerierung**


0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 2 1 3 4 6 5 7
0 2 1 3 4 6 5 7
0 4 1 5 2 6 3 7
0 4 1 5 2 6 3 7
0 1 2 3 4 5 6 7


Eingang Ausgang
Pakete
Eingang Ausgang
Pakete
Ausgang
Pakete
a)
b)
c)
a 0 b 1 c 2 d 3
0 a 1 b 2 c 3 d
a 0 c 2 b 1 d 3
0 a 2 c 1 b 3 d
0 a 1 b 2 c 3 d
a 0 b 1 c 2 d 3
0 b 1 a 2 d 3 c
b 0 d 2 a 1 c 3
0 b 2 d 1 a 3 c
0 b 1 a 2 d 3 c
a 0 b 1 c 2 d 3
0 a 1 b 2 c 3 d
a 0 c 2 b 1 d 3
0 c 2 a 1 d 3 b
0 c 1 d 2 a 3 b
β2
σ2-1

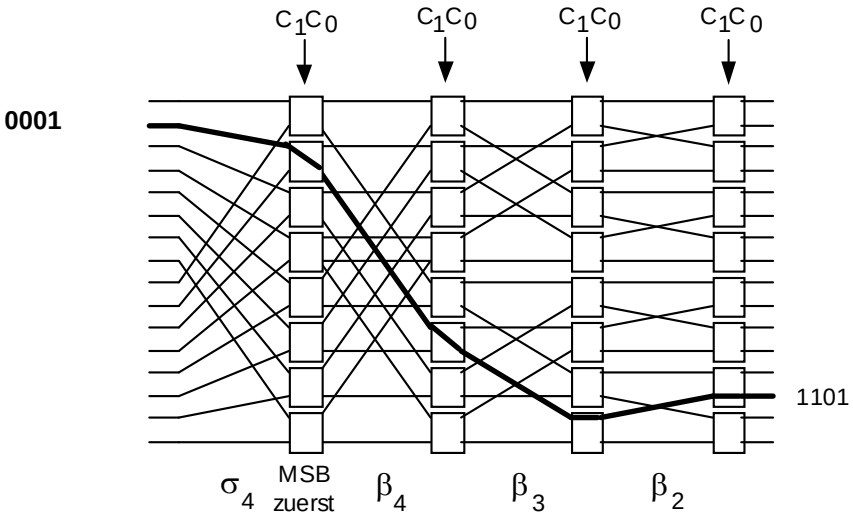
5.3.4.3 Analogie zwischen dem Indirect Binary n-Cube-Netz und der Cooley-Tukey FFT

Beispiel: (N = 16)



5.3.5 Das Generalized Cube-Netz

Beispiel: für 16 Ein-/Ausgänge



5.3.5.1 Routing im Generalized Cube-Netz

Beispiel: 16x16

$$i_4 i_3 i_2 i_1 \xrightarrow{\sigma_4} i_3 i_2 i_1 i_4 \xrightarrow{E(i_4, o_4)}$$

$$i_3 i_2 i_1 o_4 \xrightarrow{\beta_4} o_4 i_2 i_1 i_3 \xrightarrow{E(i_3, o_3)}$$

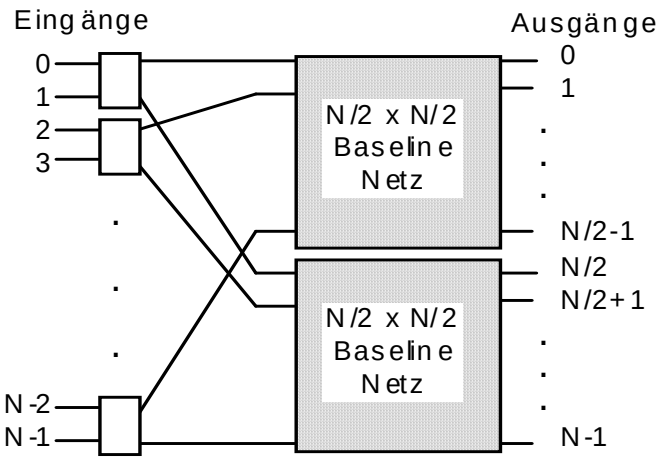
$$o_4 i_2 i_1 o_3 \xrightarrow{\beta_3} o_4 o_3 i_1 i_2 \xrightarrow{E(i_2, o_2)}$$

$$o_4 o_3 i_1 o_2 \xrightarrow{\beta_2} o_4 o_3 o_2 i_1 \xrightarrow{E(i_1, o_1)}$$

$$o_4 o_3 o_2 o_1 = O$$

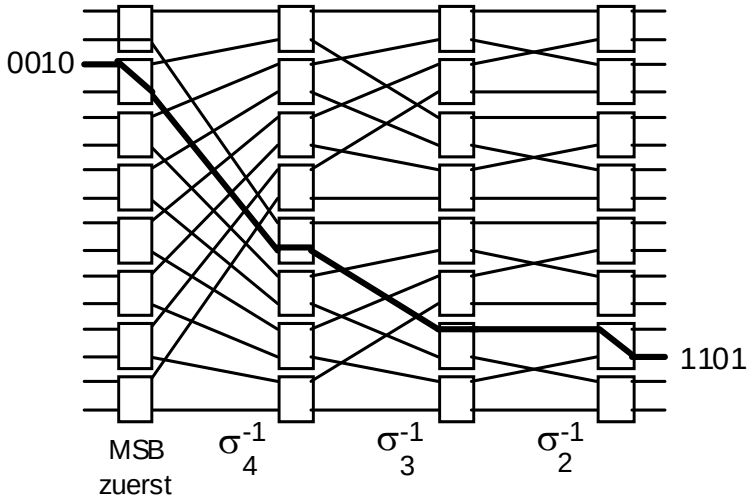
5.3.6 Das Baseline-Netz

- **Rekursive Konstruktion eines Baseline-Netzes**



- **Das Baseline-Netz wurde von Wu und Feng erfunden**

5.3.6.1 Routing im Baseline-Netz



5.3.6.2 Funktion des Baseline-Netzes (N=16, n=4)

$$i_4 i_3 i_2 i_1 \xrightarrow{E(i_1,o_4)} i_4 i_3 i_2 o_4 \xrightarrow{\sigma_4^{-1}}$$

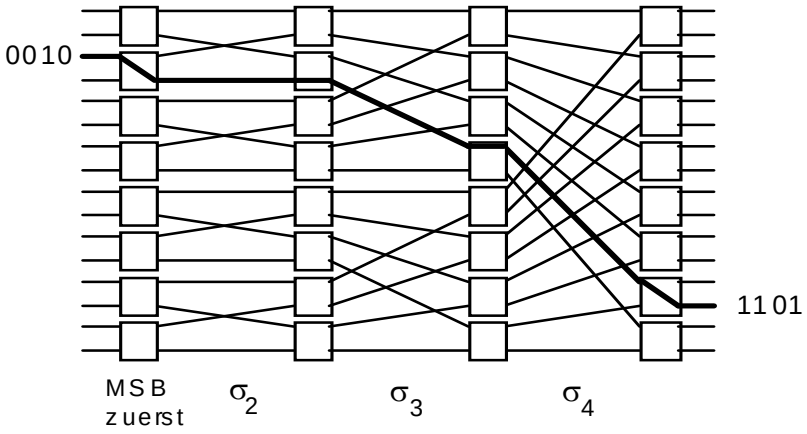
$$o_4 i_4 i_3 i_2 \xrightarrow{E(i_2,o_3)} o_4 i_4 i_3 o_3 \xrightarrow{\sigma_3^{-1}}$$

$$o_4 o_3 i_4 i_3 \xrightarrow{E(i_3,o_2)} o_4 o_3 i_4 o_2 \xrightarrow{\sigma_2^{-1}}$$

$$o_4 o_3 o_2 i_4 \xrightarrow{E(i_4,o_1)} o_4 o_3 o_2 o_1 = O$$

5.3.7 Das inverse Baseline-Netz

Beispiel: N=16, n=4



5.3.7.1 Funktion des inversen Baseline-Netzes

Beispiel: (N=16, n=4)

$$i_4 i_3 i_2 i_1 \xrightarrow{E(i_1, o_4)} i_4 i_3 i_2 o_4 \xrightarrow{\sigma_2}$$

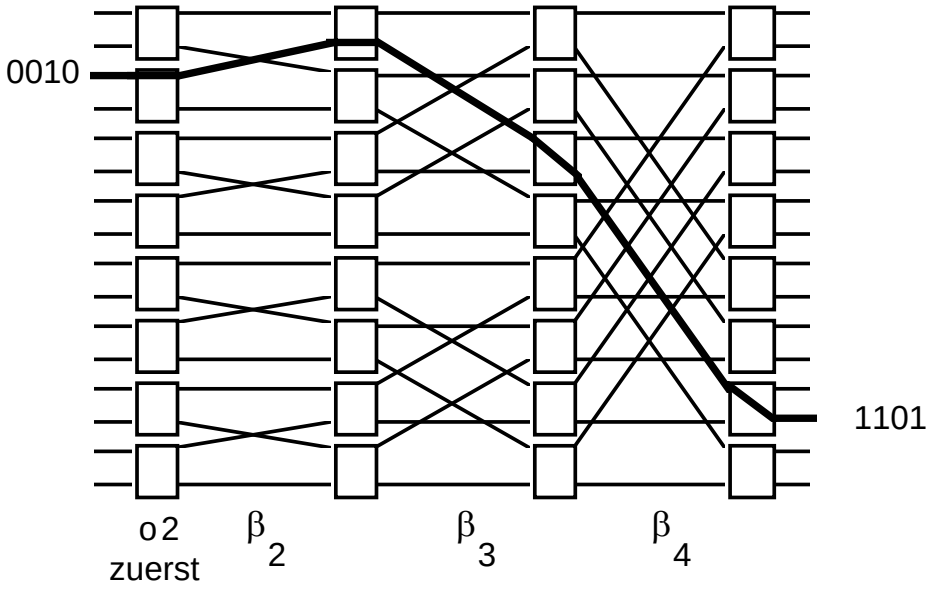
$$i_4 i_3 o_4 i_2 \xrightarrow{E(i_2, o_3)} i_4 i_3 o_4 o_3 \xrightarrow{\sigma_3}$$

$$i_4 o_4 o_3 i_3 \xrightarrow{E(i_3, o_2)} i_4 o_4 o_3 o_2 \xrightarrow{\sigma_4}$$

$$o_4 o_3 o_2 i_4 \xrightarrow{E(i_4, o_1)} o_4 o_3 o_2 o_1 = O$$

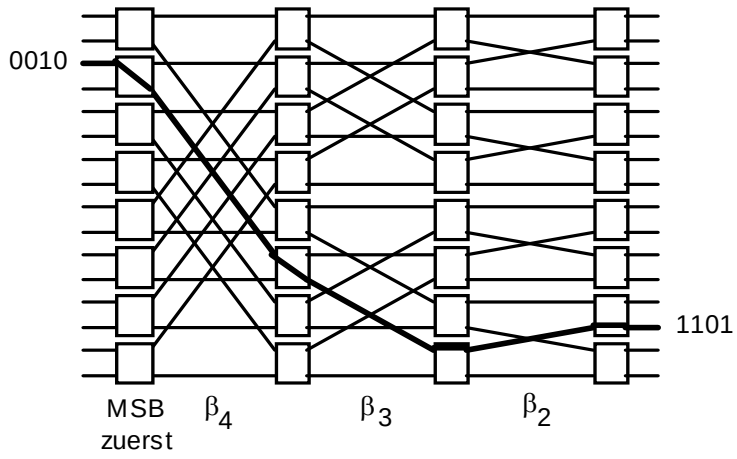
5.3.8 Der Butterfly-Banyan

Beispiel: N=16, n=4



5.3.9 Der inverse Butterfly-Banyan

Beispiel: (N=16, n=4)



5.3.10 Definitionsgleichungen der logN-Netze

Netzname	Definition
Omega -Netz	$\Omega_n = (\sigma_n \circ E)^n$
Flip - Netz	$F_n = \Omega^{-1} = (E \circ \sigma_n^{-1})^n$
Indirect Binary n - Cube	$IC_n = E\beta_2 E\beta_3 \cdot \ldots \cdot E\beta_n E\sigma_n^{-1}$
Generalized Cube	$IC_n^{-1} = \sigma_n E\beta_n E\beta_{n-1} \cdot \ldots \cdot E\beta_2 E$
Baseline - Netz	$BL_n = E\sigma_n^{-1} E\sigma_{n-1}^{-1} \cdot \ldots \cdot E\sigma_2^{-1} E$
inverses Baseline - Netz	$BL_n^{-1} = E\sigma_2 E\sigma_3 \cdot \ldots \cdot E\sigma_n E$
Butterfly - Netz	$BF_n = E\beta_2 E\beta_3 \cdot \ldots \cdot E\beta_n E$
inverses Butterfly - Netz	$BF_n^{-1} = E\beta_n E\beta_{n-1} \cdot \ldots \cdot E\beta_2 E$

5.3.11 Routing in logN-Netzen

- **Relevante Routing-Bits der 1. Stufe**

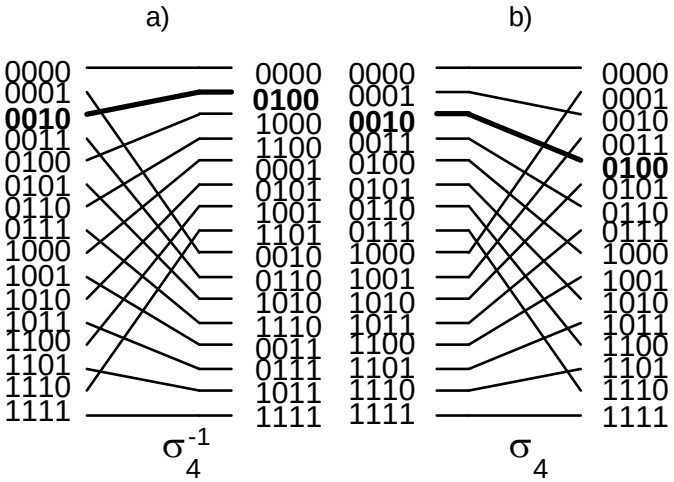
Netztopologie	gespiegelte Topologie
Butterfly: o_2	**inverser Butterfly: MSB**
Baseline: MSB	**inverser Baseline: MSB**
Indirect Binary n-Cube-Netz: LSB	**Generalized n-Cube: MSB**
Omega: MSB	**Flip: LSB**

Hinweis: MSB = Most Significant Bit der Zieladresse O. LSB = Least Significant Bit der Zieladresse O. o_2 = Bit der Zieladresse O mit Wertigkeit 2^1.

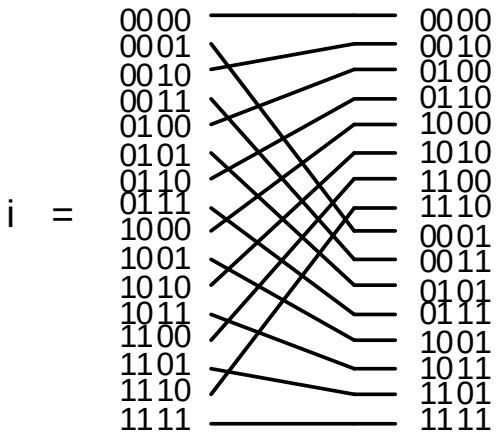
5.3.12 Äquivalenz der logN-Netze

- **Alle logN-Netze sind topologisch äquivalent, d.h. sie lassen sich zerstörungsfrei ineinander umwandeln**
- **Für die Praxis bedeutet das, dass man von allen logN-Netzen am besten das Baseline-Netz verwendet, da dessen Verdrahtung zwischen den Stufen die kleinste Vermaschung aufweist**
- **Dies ist positiv für die Herstellung des Netzes auf einer gedruckten Platine oder auf einem Chip**

Beispiel: Angebliche Umwandlung einer inversen Perfect Shuffle-Permutation in eine normale Perfect Shuffle-Permutation durch Umbenennen der Ausgangsadressen

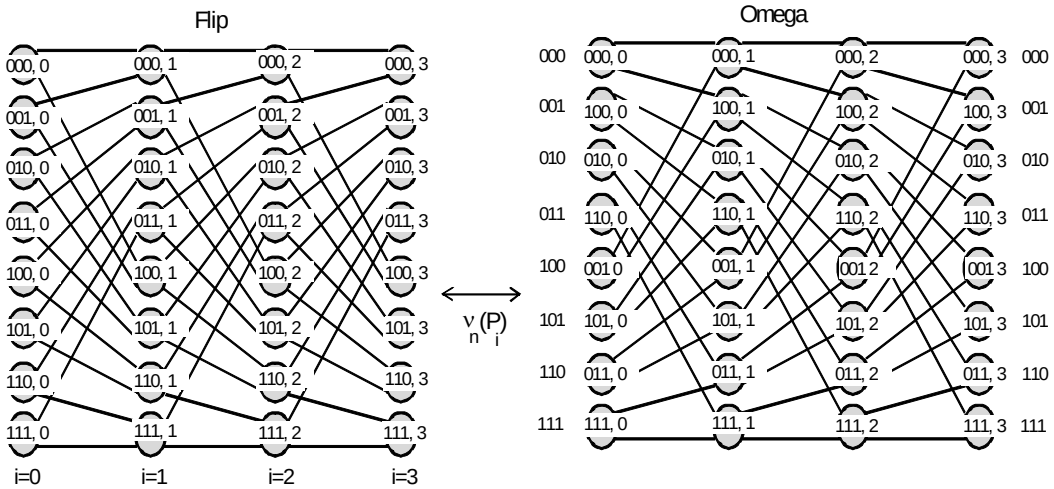


Beispiel: Neutrale Abbildung trotz angeblicher Unshuffle-Topologie

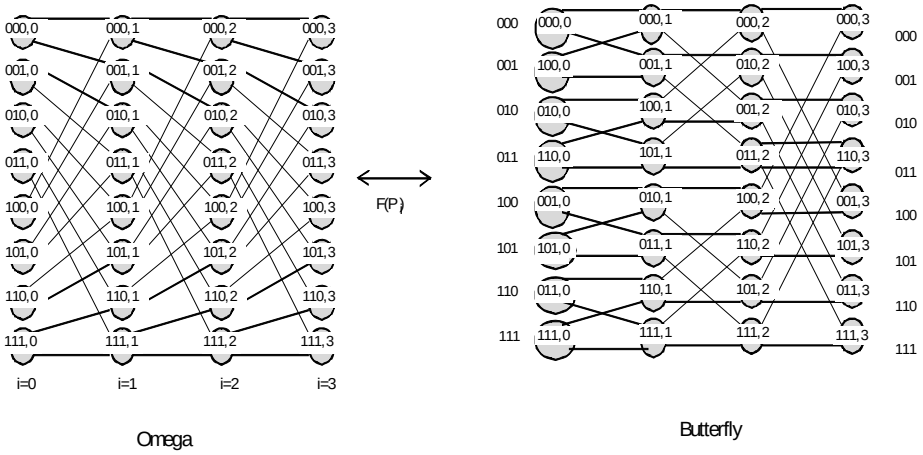


5.3.12.1 Umwandlung eines Flip-Netzes in ein Omega-Netz

- **Die Umwandlung eines Flip-Netzes in ein Omega-Netz erfolgt durch Schaltertauschen**

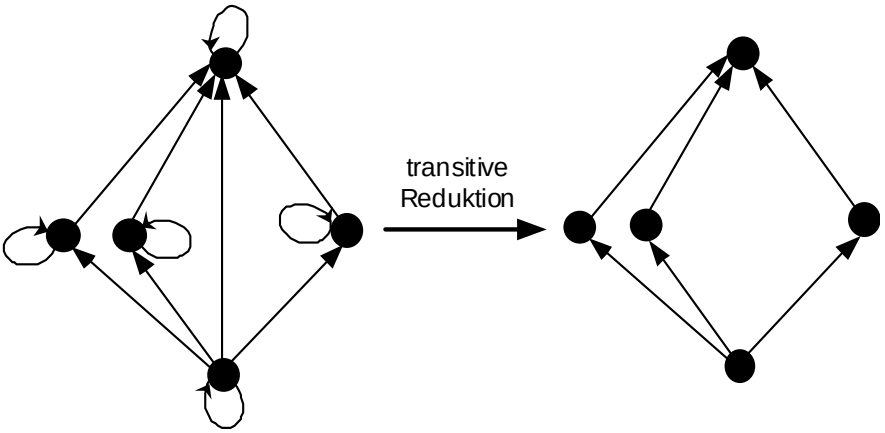


5.3.12.2 Umwandlung eines Omega-Netzes in ein Butterfly-Banyan

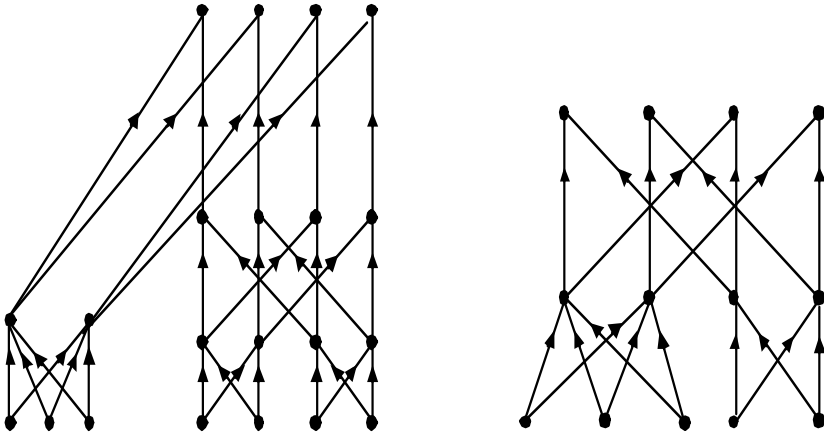


5.4 Allgemeine Banyans

Beispiel: Umwandlung einer partiellen Ordnung (links) in ihr Hasse-Diagramm (rechts)

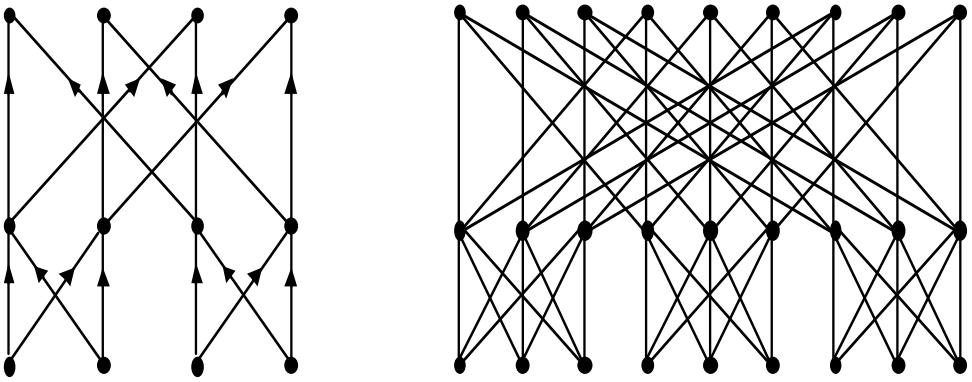


Beispiel: Ein Nicht-n-Ebenen- und ein n-Ebenen-Banyan



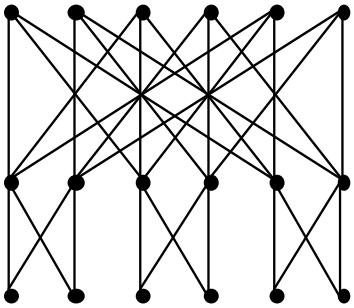
- **Folgende 3 Parameter charakterisieren alle n-Ebenen-Banyans: (f,s,n)**
  - **f = Zahl der auf einen Knoten zulaufenden Kanten (fan in)**
  - **s = Zahl der von einem Knoten abgehenden Kanten (Ausgangsverzweigung, spread)**
  - **n = Zahl der Pfeilebenen; n ist um 1 kleiner als die Zahl der Knotenebenen!**
- **In jeder Knotenebene i gibt es eine Zahl N_i an Knoten**

Beispiel: Zwei Banyans, die regelmäßig und rechteckig sind

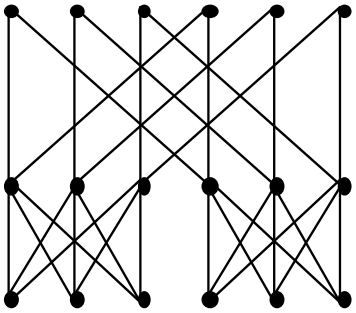


N=4, n=2, s=f=2 N=9, n=2, s=f=3

Beispiel: Zwei Banyans, die unregelmäßig aber rechteckig sind

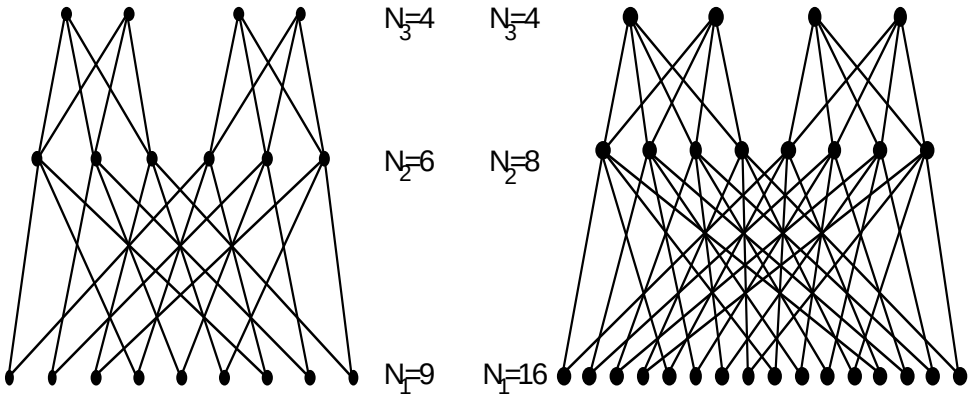


N=6, n=2
1. Ebene: $s_1 = f_1 = 2$
2. Ebene: $s_2 = f_2 = 3$



N=6, n=2
1. Ebene: $s_1 = f_1 = 3$
2. Ebene: $s_2 = f_2 = 2$

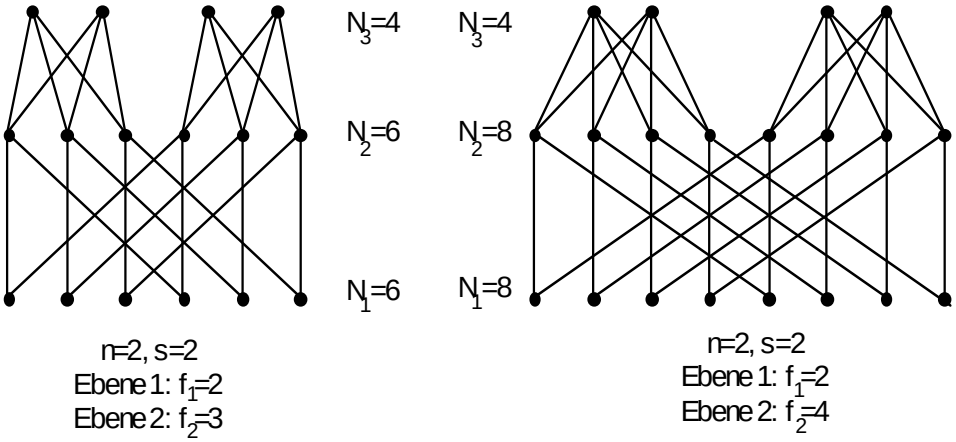
Beispiel: Zwei Banyans, die regelmäßig aber nicht-rechteckig sind



n=2, s=2, f=3

n=2, s=2, f=4

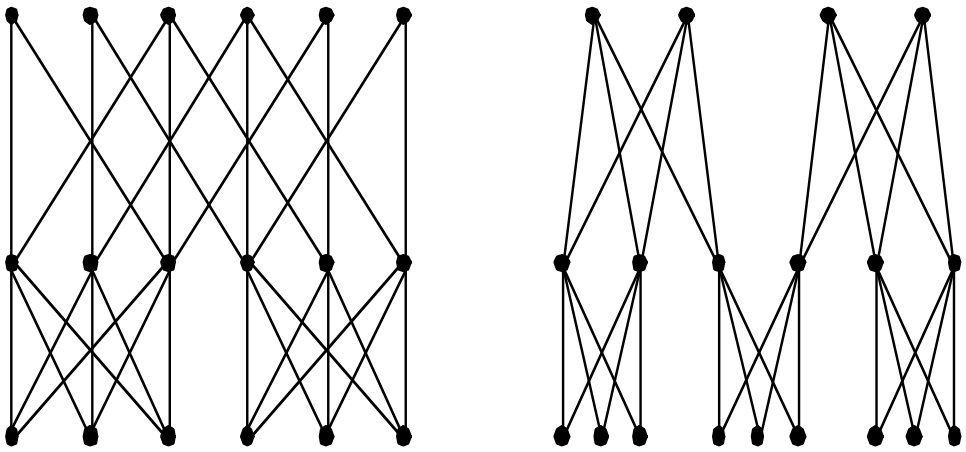
Beispiel: Zwei Banyans, die unregelmäßig und nicht-rechteckig sind



n=2, s=2
Ebene 1: f_1=2
Ebene 2: f_2=3

n=2, s=2
Ebene 1: f_1=2
Ebene 2: f_2=4

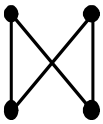
Beispiel: Zwei Graphen, die keine Banyans sind



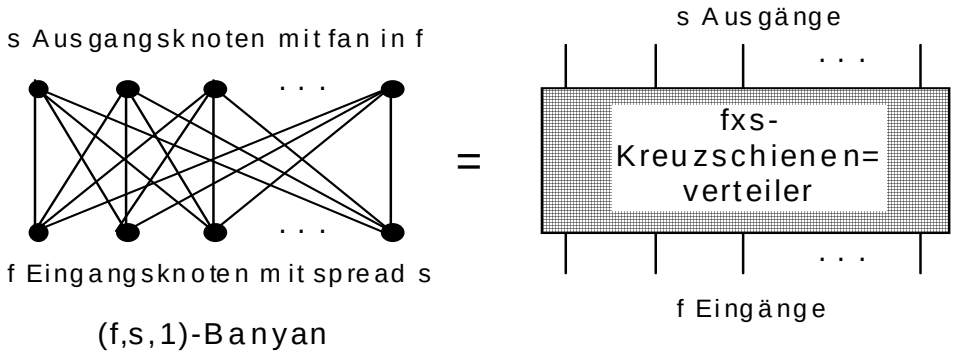
- **Nicht jeder Ausgang ist von jedem Eingang aus erreichbar!**

5.5 Regelmäßige Banyans

5.5.1 Einziger Banyan der Größe n=1 und f=s=2

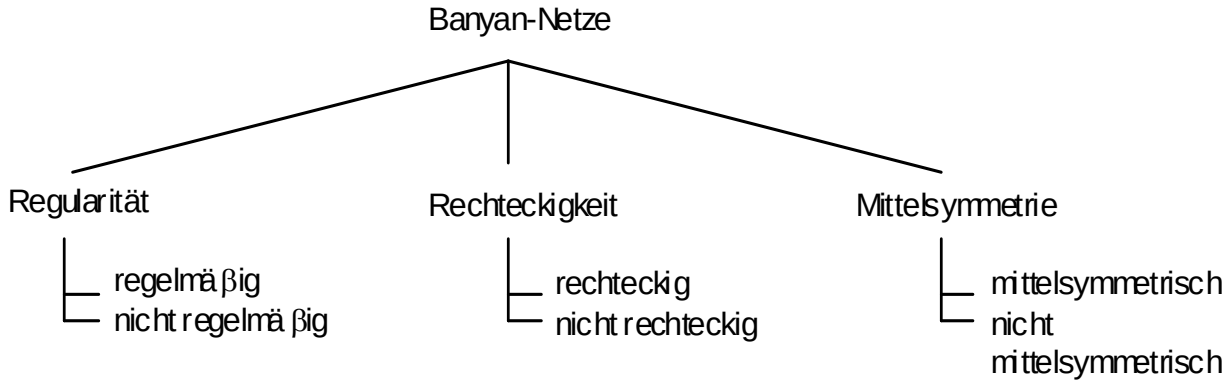


5.5.2 Verallgemeinerung zum (f,s,1)-Banyan ist ein fxs-Kreuzschienenverteiler



- **Der (f,s,1)-Banyan ist der bekannte fxs-Kreuzschienenverteiler**

5.6 Klassifikation der Banyans



- **Für die Praxis relevant sind die regelmäßigen Banyans, die rechteckig sind**

5.7 Regelmäßige und rechteckige Banyans

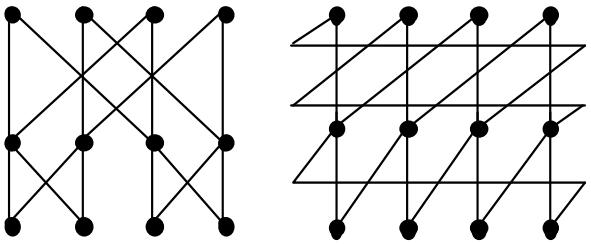
- **Innerhalb der Kategorie der regelmäßigen und rechteckigen Banyans sind die „SW- und CC-Banyans“ die beiden wichtigsten Untergruppen**

Hinweis: SW-Banyan. SW steht für „Switch“

Hinweis: CC-Banyan. CC steht sowohl für „Cylindrical Cross Hatched“ als auch für „Conical Cross Hatched“, ist also nicht eindeutig.

Hinweis: Cylindrical Cross Hatched = Zylindrisch quer gestreift; Conical Cross Hatched = Konisch quer gestreift

Beispiel: (2,2,2)-SW- und (2,2,2)-Cylindrical Cross Hatched-Banyan



(2,2,2)-SW Banyan

(2,2,2)-CC Banyan

Def.: SW-Banyan heißt: $N_1=N_2=...=N_{n+1}=N=const$, $s=f$, $n=\log_s N$ und Verdrahtung gemäß SW-Topologie.

Def.: Cylindrical Cross Hatched-Banyan heißt: $N_1=N_2=...=N_{n+1}=N=const$, $s=f$, $n=\log_s N$ und Verdrahtung gemäß Cylindrical Cross Hatched-Topologie.

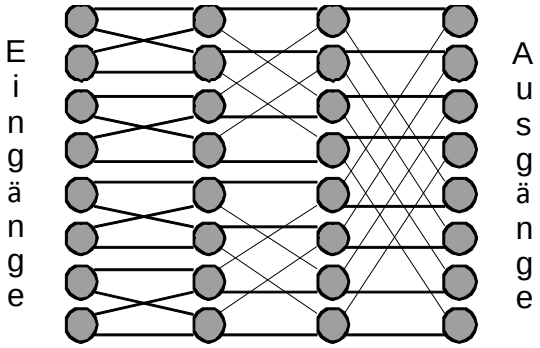
Def.: Conical Cross Hatched-Banyan heißt: $N_1>N_2>...>N_{n+1}$, $s>f$, und Verdrahtung gemäß Conical Cross Hatched-Topologie.

- **Im folgenden werden zuerst die SW- und dann die CC-Banyans erläutert**

5.7.1 Die SW-Banyans

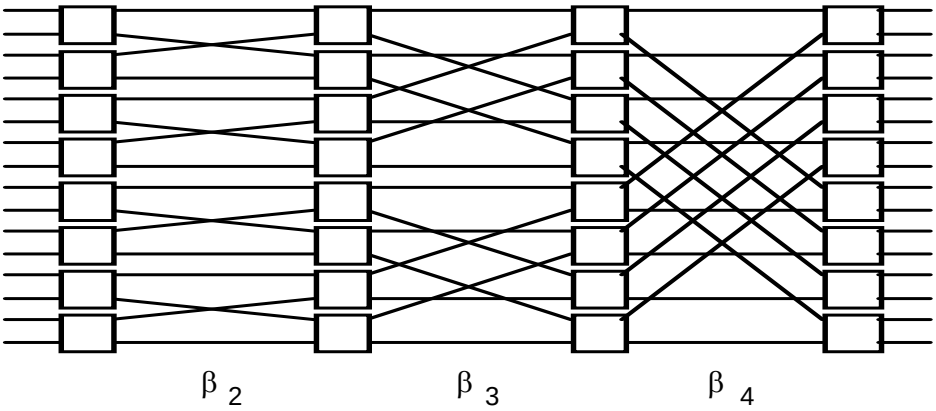
- **Die SW-Banyans werden anhand der Beispiele des (2,2,1)-, (2,2,2) und des (2,2,3)-SW-Banyans genauer dargestellt**

5.7.1.1 Der (2,2,3)-SW-Banyan



5.7.1.2 Der (2,2,3)-SW-Banyan in der Aktivknoteninterpretation

- **Aktivknoteninterpretation heißt: jeder Knoten ist ein 2x2-Kreuzschalter**



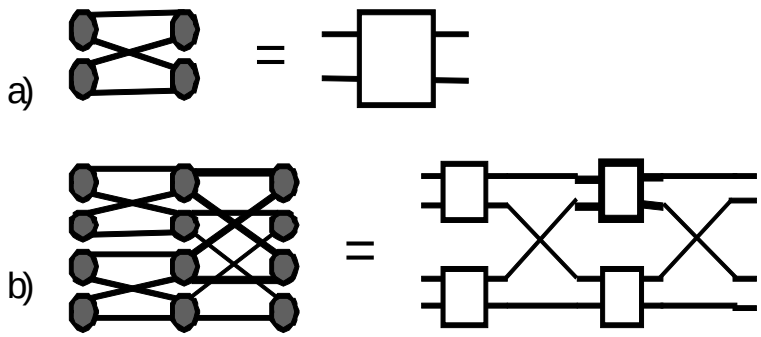
- **Der (2,2,3)-SW-Banyan entpuppt sich als der bekannte Butterfly-Banyan**

5.7.1.3 Funktionsweise des (2,2,3)-SW-Banyans in der Aktivknoteninterpretation

$$i_4 i_3 i_2 i_1 \xrightarrow{E(i_1, o_2)} i_4 i_3 i_2 o_2 \xrightarrow{\beta_2}$$
$$i_4 i_3 o_2 i_2 \xrightarrow{E(i_2, o_3)} i_4 i_3 o_2 o_3 \xrightarrow{\beta_3}$$
$$i_4 o_3 o_2 i_3 \xrightarrow{E(i_3, o_4)} i_4 o_3 o_2 o_4 \xrightarrow{\beta_4}$$
$$o_4 o_3 o_2 i_4 \xrightarrow{E(i_4, o_1)} o_4 o_3 o_2 o_1$$

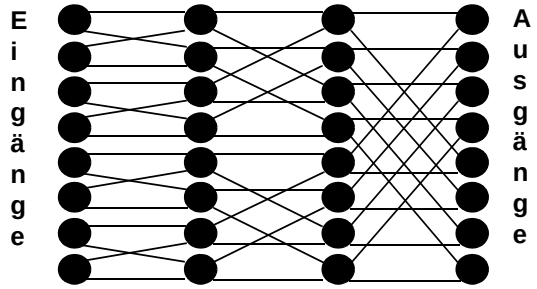
5.7.1.4 Der (2,2,3) Banyan in der Passivknoteninterpretation

- **Passivknoteninterpretation heißt:**
  - **jede Kante zwischen zwei Knoten ist ein Schalter**
  - **Schalter, die in Schmetterlings-Topologie verbunden sind, werden zu einem Kreuzschalter zusammengefasst**

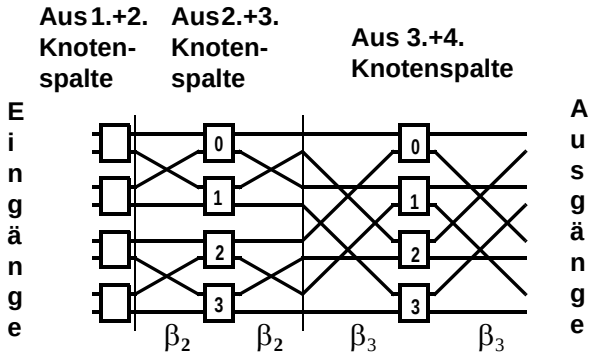


5.7.1.5 Konstruktion des (2,2,3) Banyans in der Passivknoteninterpretation

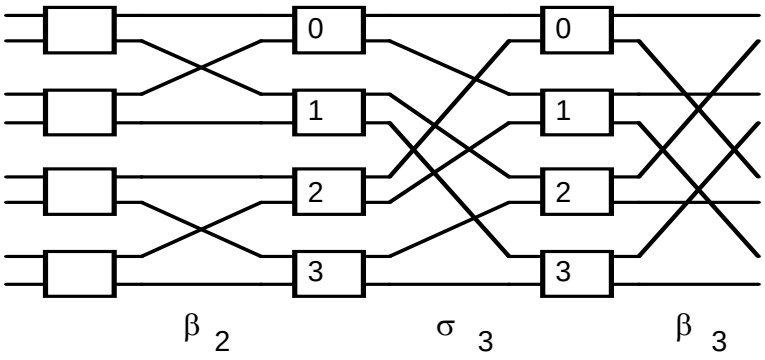
a)



b)



- **Der (2, 2, 3)-SW-Banyan in der Passivknoteninterpretation sieht nach Verketten der beiden mittleren Verdrahtungen auf den ersten Blick aus wie ein neues Netz N_{neu}, das zur Klasse der logN-Netze gehört**



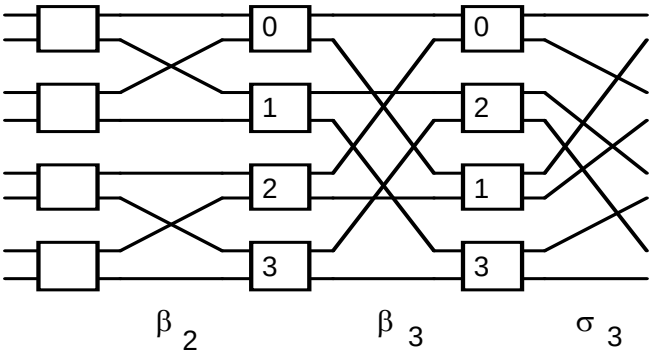
5.7.1.6 Funktionsweise des Netzes N_{neu}

- **Auch das Netz N_{neu} funktioniert, wie die Abbildungsfunktionen zeigen**

$$i_3 i_2 i_1 \xrightarrow{E(i_1, o_1)} i_3 i_2 o_1 \xrightarrow{\beta_2}$$
$$i_3 o_1 i_2 \xrightarrow{E(i_2, o_2)} i_3 o_1 o_2 \xrightarrow{\sigma_3}$$
$$o_1 o_2 i_3 \xrightarrow{E(i_3, o_3)} o_1 o_2 o_3 \xrightarrow{\beta_3}$$
$$o_3 o_2 o_1$$

5.7.1.7 Topologierhaltende Umwandlung des Netzes N_{neu}

- **Die topologierhaltende Umwandlung des Netzes Nneu erfolgt durch Verschieben der beiden mittleren Schalter in der letzten Stufe**
- **Dies wird am Beispiel 8 Ein-/Ausgänge gezeigt**



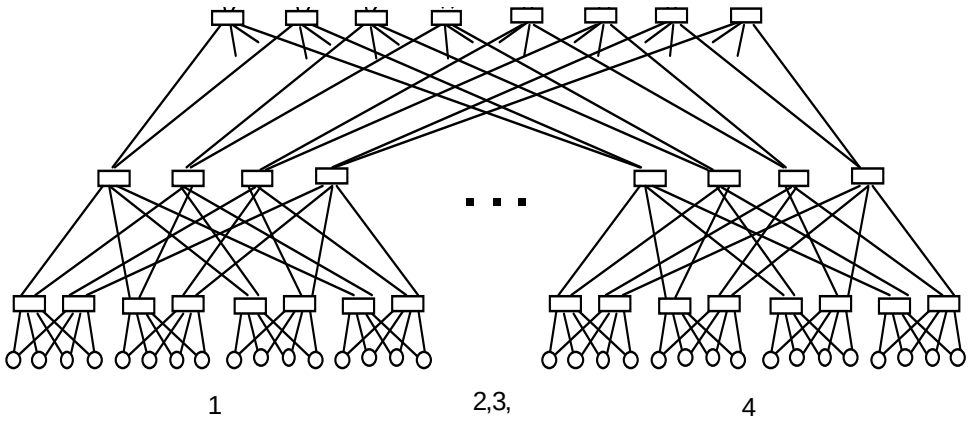
- Das Netz N_{neu} entpuppt sich als vollständiger Indirect Binary n-Cube, ist also nicht neu

5.8 Anwendungen von SW-Banyans

5.8.0.1 Parallelrechner mit 64-Prozessoren in „Fat Tree-Topologie“

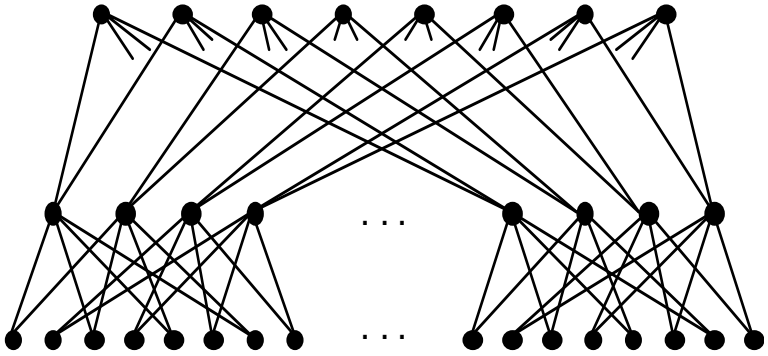
- Parallelrechner in Fat Tree-Topologie wurden in Stückzahlen gebaut

Beispiel: „SuperMUC“ = Cluster Computer in Fat Tree-Topologie



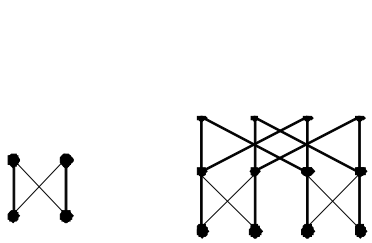
- Die beiden abgehenden Leitungen der mittleren Schalter sind doppelt so schnell, wie die vier zulaufenden Leitungen => „Fat Tree“, weil sich die Geschwindigkeit der Leitungen bei jedem 4x2-Schalter auf dem Weg zur Baumwurzel verdoppelt
- Alle Schalter zusammen bilden einen regelmäßigen, nicht rechteckigen (4,2,2)-Banyan mit N_1=32, N_2=16 und N_3=8
- Außerdem bildet jede Gruppe aus 16 CPUs/Cores unten, die mit den beiden untersten Schalterebenen verbunden sind, einen (4,2,2)-Banyan mit N_1=16, N_2=8 und N_3=4

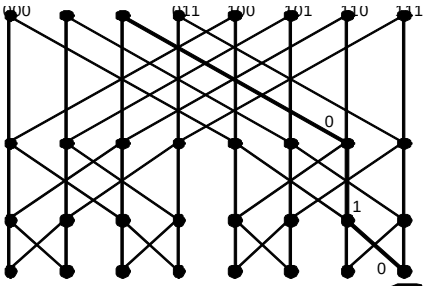
5.8.1 Topologie des (4,2,2)-Banyans mit N_1=32 und N_2=16 und N_3=8



5.8.2 Systematische Konstruktion von Banyans

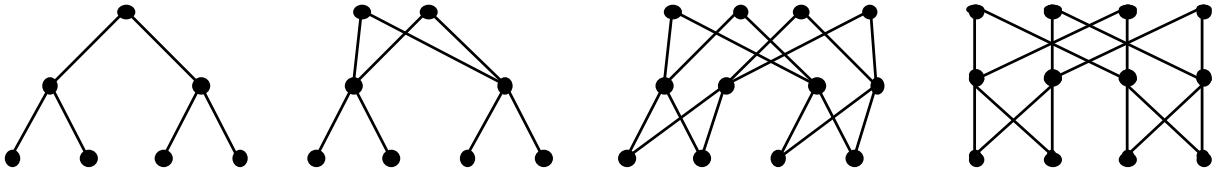
5.8.2.1 Sequenz der ersten drei SW-Banyans (2,2,1), (2,2,2) und (2,2,3)





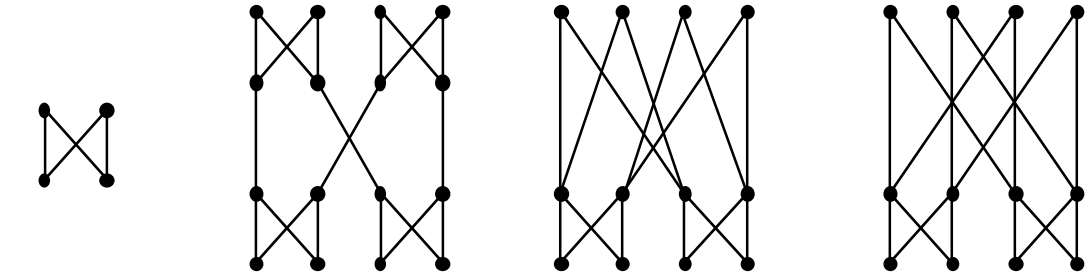
- **Die Sequenz der ersten drei (2,2,x)-SW-Banyans liefert Hinweise zu deren Konstruktion**

5.8.2.2 Additive Konstruktion eines (2,2,2)- SW-Banyans aus 4 Binärbäumen



a) b) c) d)

5.8.2.3 Rekursive Konstruktion eines (2,2,2)-SW-Banyans aus (2,2,1)-SW-Banyans



a) (2,2,1)-Banyan

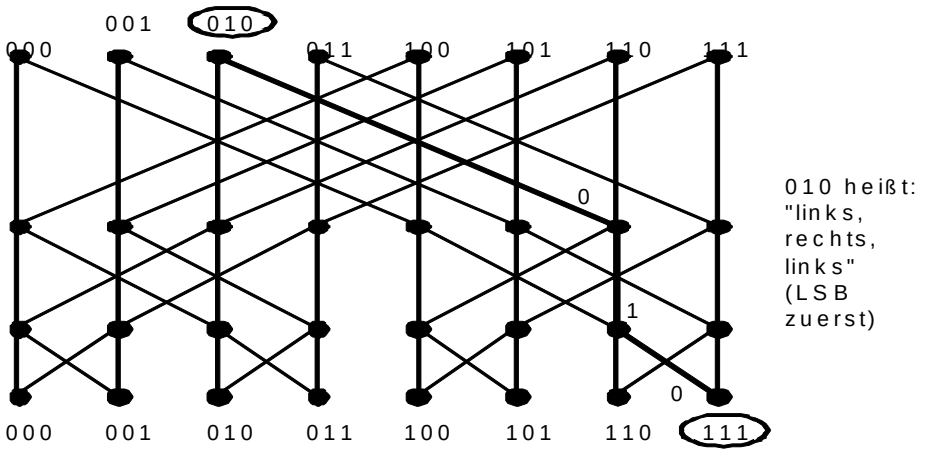
b) Rekursives Ersetzen

c) Streichen der urspr. Kanten und Verschmelzen der mittleren Knoten

d) Umsortieren in parallele Kanten

5.8.3 Routing in einem regelmäßigen und rechteckigen Banyan

Beispiel: Routing in einem (2,2,3)-SW-Banyan



5.8.3.1 Routing im (2,2,n)-SW-Banyan

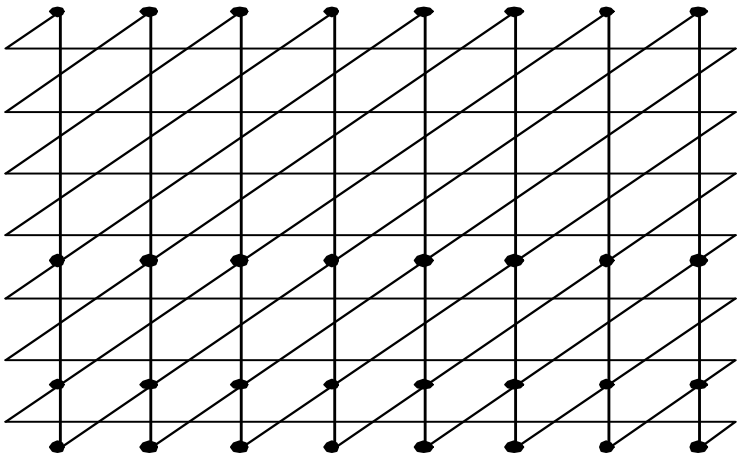
- **Die Ausgänge werden binär von links nach rechts durchnummeriert (ohne Lücke)**
- **Das LSB der Zieladresse wird von der untersten Knotenebene (Eingänge) ausgewertet**
- **Das MSB der Zieladresse wird von der obersten Knotenebene (Ausgänge) ausgewertet**

- Alle Bits der Zieladresse zwischen LSB und MSB werden von den Knotenebenen zwischen den Ein- und Ausgängen ausgewertet
- „0“ im relevanten Bit der Zieladresse heißt: erste Ausgangskante eines Knotens, gezählt von links, „1“ heißt: zweiter Ausgangskante, etc.
- Genau wie bei den logN-Netzen ist das Routing unabhängig von der Herkunftsadresse und hängt nur von der Zieladresse ab

5.8.4 Die CC-Banyans

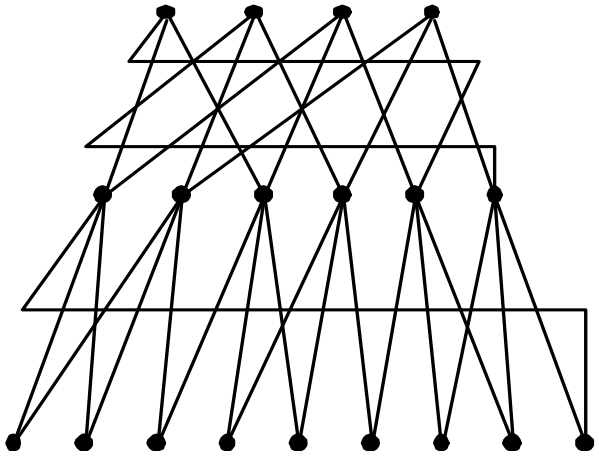
- CC-Banyans sind im Gegensatz zu den SW-Banyans nicht mittelsymmetrisch
- Bei den CC-Banyans gibt es die beiden Untergruppen **Cylindrical- und Conical Cross Hatched-Banyans**
- Sie unterscheiden sich dadurch, dass nur Cylindrical Cross Hatched-Banyans rechteckig sind
- Im weiteren werden beide Untergruppen anhand der Beispiele des (2,2,3)-Cylindrical- und des (3,2,2)-Conical Cross Hatched-Banyans erläutert

Beispiel: Der (2,2,3)-Cylindrical Cross Hatched-Banyan



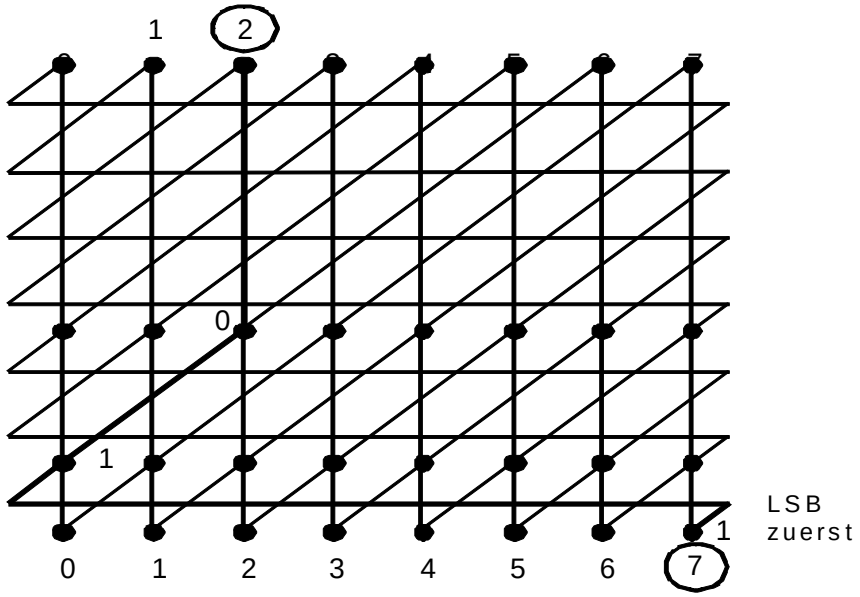
- Kennzeichen: Wrap around-Verbindungen von rechts aussen nach links aussen

Beispiel: Der (3,2,2)-Conical Hatched-Banyan



- **Kennzeichen: Wrap around-Verbindungen und nach oben kleinere Knotenzahl**
- **Verwendung für skalierbare Datenkonzentratoren für die Analyse großer Datenmengen (Big Data)**

Beispiel: Routing im (2,2,3)-Cylindrical Cross Hatched-Banyan



- **Sei $N= 2^n = 8$, Z = 2, S = 7, dann ist RA = 2-7 mod 8 = -5 mod 8 = +3**

Hinweis: es wird der mathematische modulo-Operator verwendet, der auch für negative Zahlen definiert ist

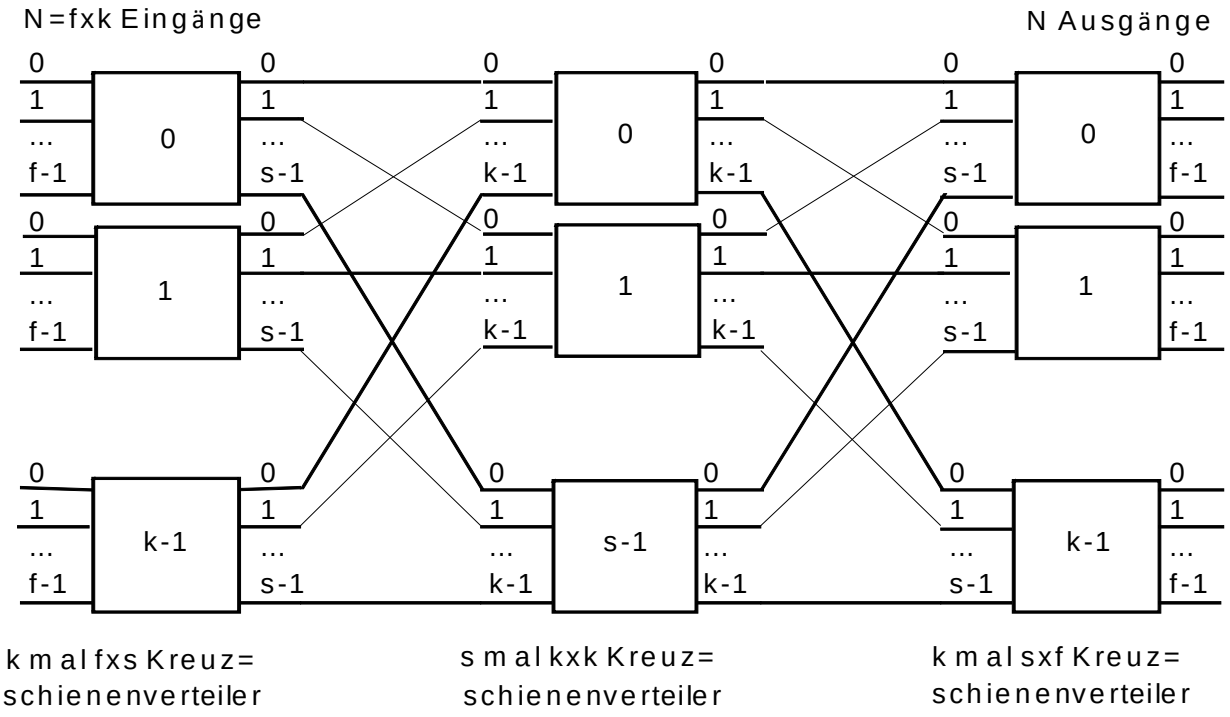
5.8.4.1 Routing im (2,2,n)-Cylindrical Cross Hatched-Banyan

- **Ein- und Ausgänge werden binär von links nach rechts durchnummeriert (ohne Lücke)**
- **Im Gegensatz zu den logN-Netzen werden Adressdifferenzen zum Routing verwendet**
- **Das Routing ist von der Differenz zwischen Ziel- und Herkunftsadresse abhängig**
- **Die Adressdifferenz wird als relative Zieladresse bezeichnet**
- **Sei S = Startadresse zur Basis 2, Z=Zieladresse zur Basis 2, dann ist die relative Adresse RA= Z-S mod 2^n**

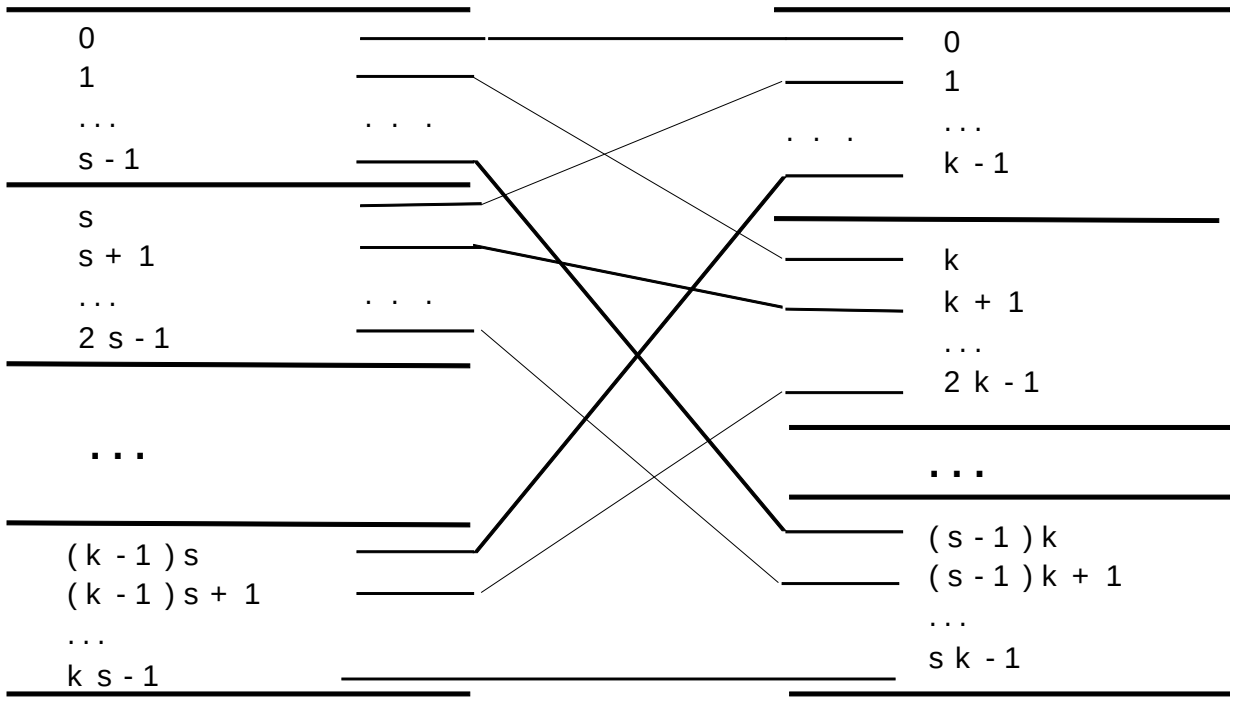
Hinweis: es wird der mathematische modulo-Operator verwendet, der auch für negative Zahlen definiert ist

- **Danach erfolgt Self Routing wie bei den logN-Netzen, aber auf Basis relativer Zieladressen gemäß:**
  - **das MSB der relativen Zieladresse wird von der obersten Knotenebene (Ausgänge) ausgewertet**
  - **das LSB der relativen Zieladresse wird von der untersten Knotenebene (Eingänge) ausgewertet**
  - **Alle Bits der relativen Zieladresse zwischen LSB und MSB werden von den Knotenebenen zwischen den Ein- und Ausgängen ausgewertet**
  - **„0“ im relevanten Bit der relativen Adresse heißt: gewählt wird die erste Ausgangskante eines Knotens, gezählt von links**
  - **„1“ heißt: gewählt wird die zweite Ausgangskante, etc.**

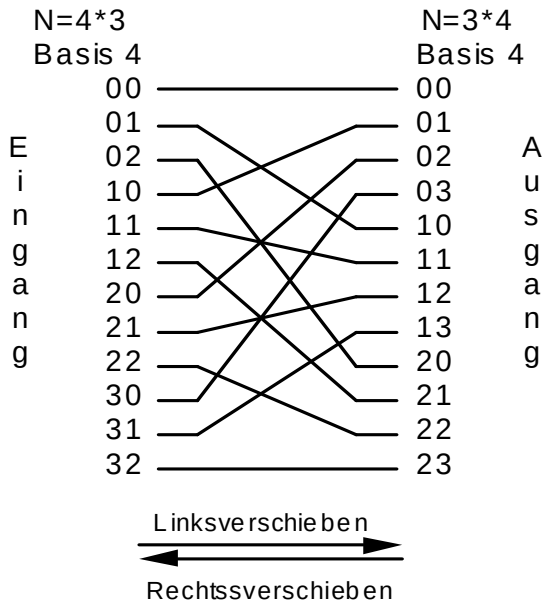
5.9 Das Clos-Netz



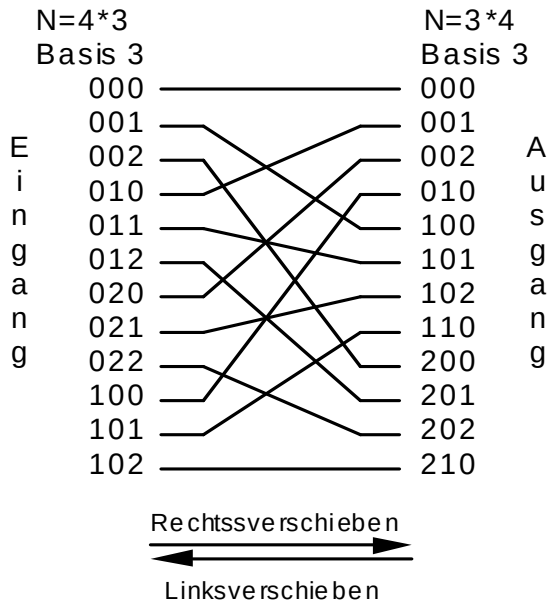
- **Die allgemeine Perfect Shuffle-Permutation**



Beispiel: Die allgemeine Perfect Shuffle-Permutation für N=12 zur Basis 4 und 3 (Bild a bzw. b)

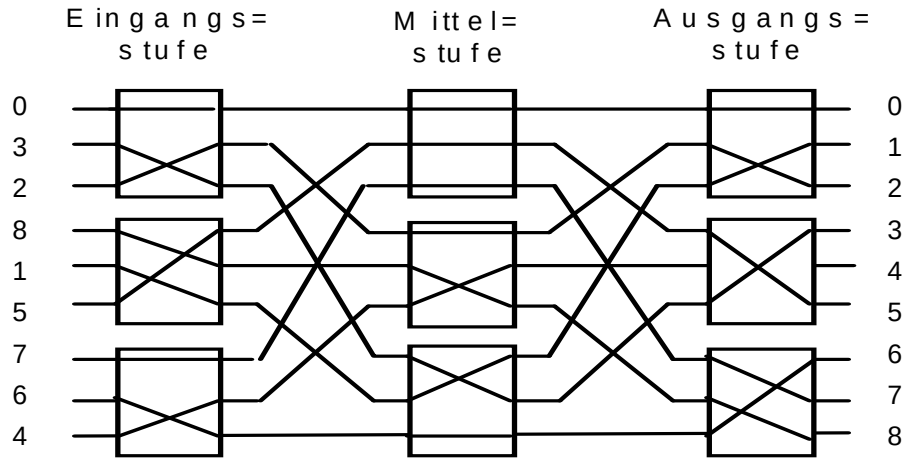


a)

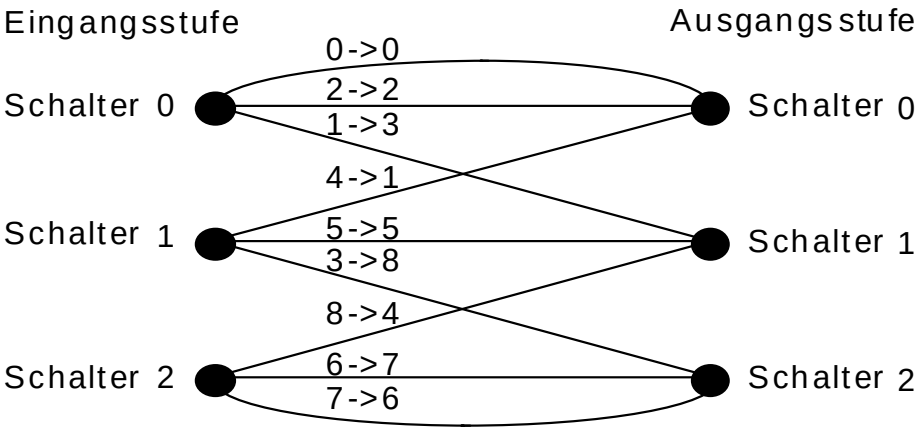


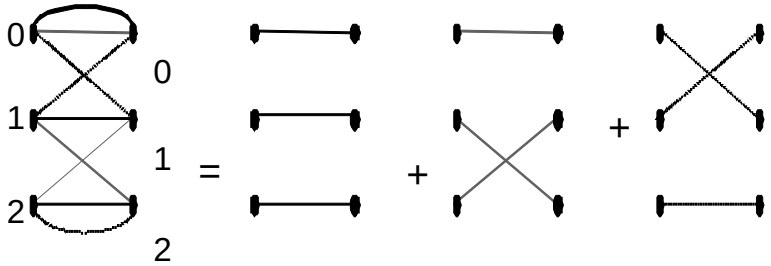
b)

- **Routing Beispiel für das 9x9 Clos-Netz**



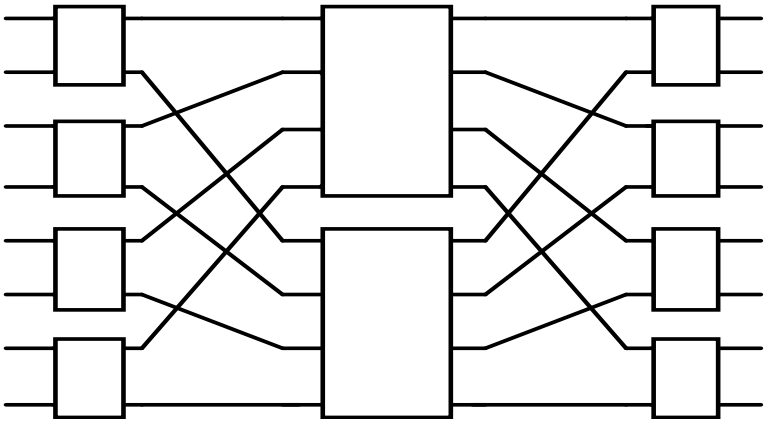
- **Multigraph der Permutation (0 3 2 8 1 5 7 6 4)**
- **Graphenzerlegung durch Kantenfärben**



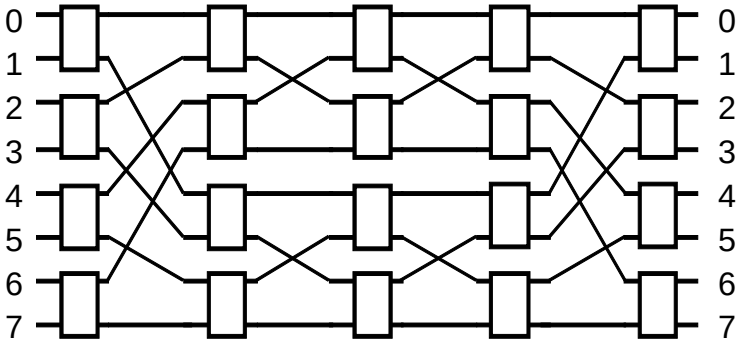


5.10 Das Benes-Netz

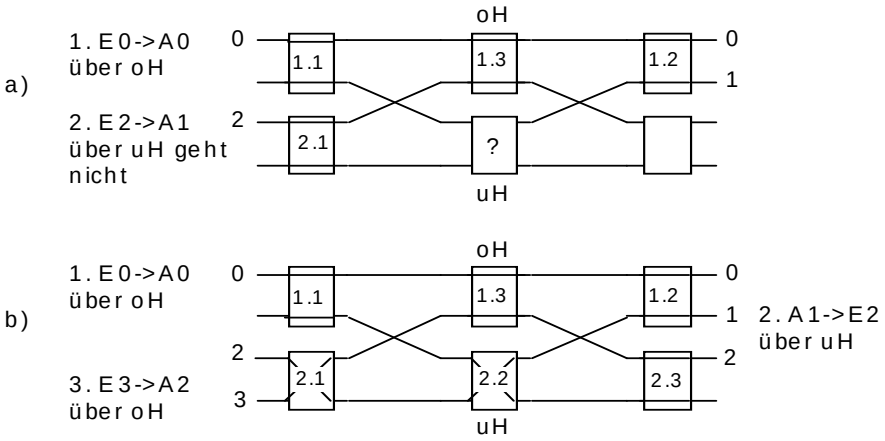
- **1. Schritt in der Konstruktion des Benes-Netzes**



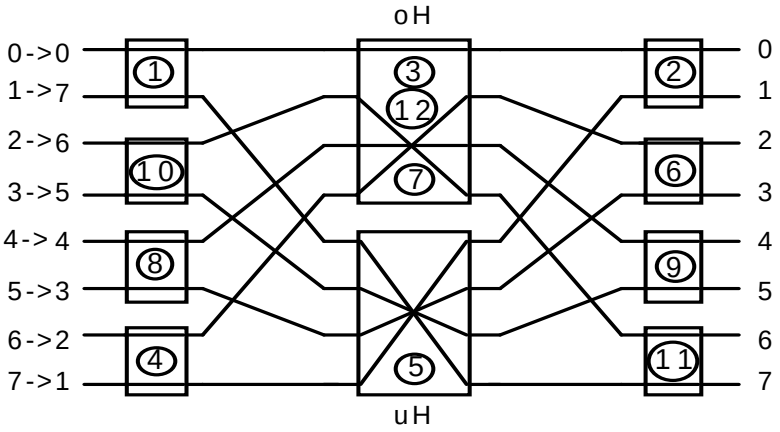
Beispiel: Benes-Netz für N=8 Ein-/Ausgänge



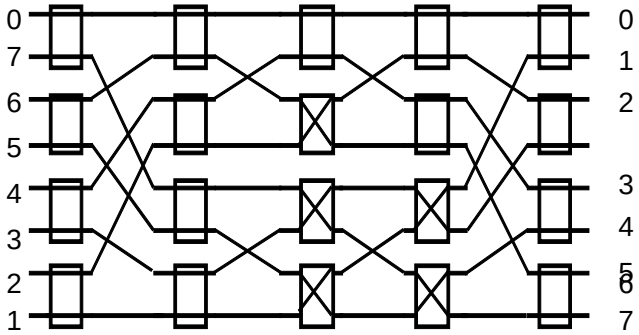
- **Routing-Beispiel im Benes-Netz a) nicht erfolgreich b) erfolgreich**



Routing in der Ein- und Ausgangsstufe

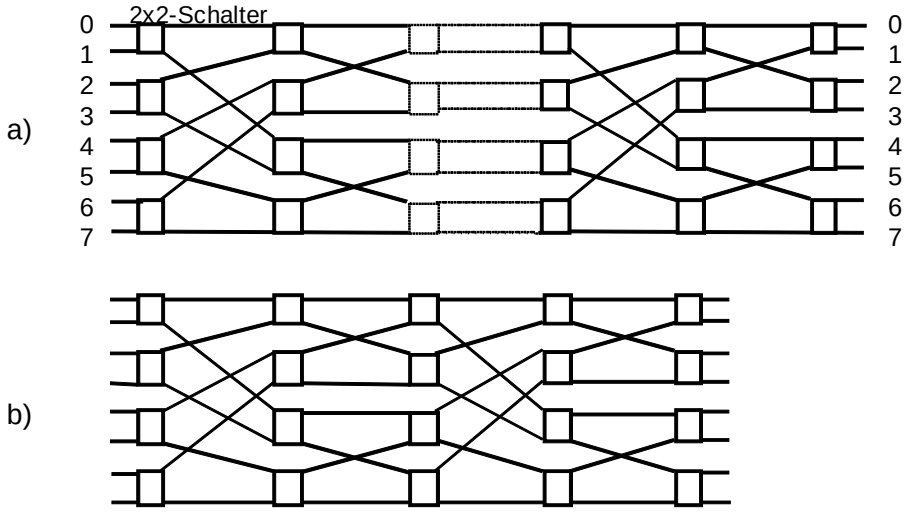


Beispiel: Routing im Benes-Netz



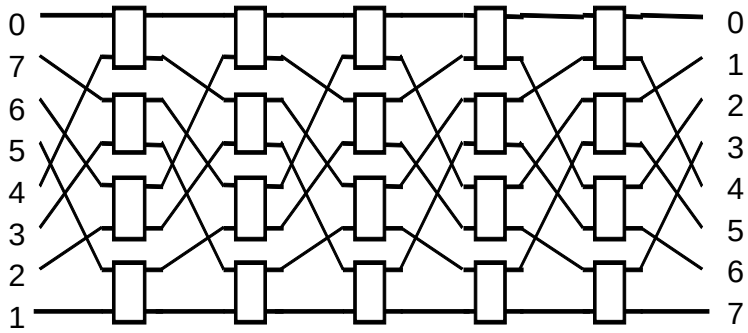
5.11 Das doppelte Baseline-Netz nach Wu und Feng

Beispiel: für N=8 Ein-/Ausgänge



5.12 Das Lee-Netz

Beispiel: für N=8



Vielen Dank für die Aufmerksamkeit!