Rechnerarchitektur 1 und 2

Prof. Dr. Fey

Semester: WS 2004/05, SS 2005

Vorwort

Dieses Skript ist eine Zusammenstellung von Informationen rund um die Vorlesung Rechnerarchitektur von Prof. Dr. Fey. Es ist als Vorbereitung auf die Diplomprüfung entstanden und keine Vorlesungsmitschrift.

Dieses Skript ist im Rahmen des Projekts „Vorlesungsskripte der Fakultät für Mathematik und Informatik“ entstanden und wird von dem Projekts weiter betreut. Das Skript wurde nach bestem Wissen und Gewissen erstellt. Dennoch garantiert weder der auf der Titelseite genannte Dozent, noch die Mitglieder des Projekts für dessen Fehlerfreiheit. Für etwaige Fehler und dessen Folgen wird von keiner der genannten Personen eine Haftung übernommen. Es steht jeder Person frei, dieses Skript zu lesen, zu verändern oder auf anderen Medien verfügbar zu machen, solange ein Verweis die Internetadresse `http://uni-skripte.lug-jena.de/` *des Projekts enthalten ist.*

Diese Ausgabe trägt die Versionsnummer 2571 und ist vom 4. Dezember 2009. Eine neue Ausgabe könnte auf der Webseite des Projekts verfügbar sein.

Jeder ist dazu aufgerufen, Verbesserungen, Erweiterungen und Fehlerkorrekturen für das Skript einzureichen bzw. zu melden oder selbst einzupflegen – einfach eine eMail an die Mailingliste `<uni-skripte@lug-jena.de>` *senden. Weitere Informationen sind unter der oben genannten Internetadresse des Projekts verfügbar.*

Hiermit möchten wir allen Personen, die an diesem Skript mitgewirkt haben, vielmals danken:

- *Fred Thiele* `ferdy@inf.uni-jena.de` *(2005)*
- *Christian Raschka* `chrisra@inf.uni-jena.de` *(2005)*
- *Jörg Sommer* `joerg@alea.gnuu.de` *(2007)*

Inhaltsverzeichnis

Teil I.

Zusammenfassung RA1

Definition: (LATENZ)
* Zeit zwischen Beginn und Ende einer Aktion, auch: (execution time).

Übersicht über Rechnerarchitektur 1

1. Kapitel: Was ist Rechnerarchitektur?

- 3 Definitionen für RA
- Historische Entwicklung

2. Kapitel: Formale Entwurfsmethoden

- Nutzen von FEM
- Hierarchie

2.1. Automaten

- Nutzen, Definition, Moore/Mealy Automat
- Was ist ein nichtdeterministischer Automat?
- Vergleich Moore/Mealy
- Überführung Moore <> Mealy
- Minimaler endlicher Automat
- Schwachpunkte der Modellierung mittels endlicher Automaten
- Erweiterungen (Statecharts, SDF, CSP)

- Statecharts
  - Eigenschaften (Hierarchiebildung, Nebenläufigkeit)

- Petrinetze
- Was sind Petrinetze, wozu sind sie gut?
- Definition + Beispiel
- Modellierung eines Mealy-Automaten durch ein Petrinetz

- Synchrone Datenflussgraphen (SDF)
- Beschreibung der Kommunikationsregel auf Basis der von Petrinetzen
- Anwendungsgebiet
- Definition
- Inkonsistenz (nicht-periodische Abläufe!)
- minimaler Vektor (-> Repetitionsvektor)
- Verklemmungen, Wie kann man die Berechnen, Algorithmus von LEE
- Schwachpunkte von SDF und Erweiterungen

- Communicating Sequential Process (CSP)
- Wozu dient es?
- Grundidee, Grundbegriffe
- Einzelne Prozesse, Parallele Prozesse
- Darstellung mittels Prozessbaum

- Rechner- und Systementwurfssprachen (RuS)
- Sechs Ebenen der Rechnerbeschreibung
- Unterschied Rechner-/Systementwurfssprache
- Aufgaben von RuS
- VHDL
- Vorteile von VHDL
- Grobe und feinere Strukturierung einer VHDL-Beschreibung
- Syntax u. Semantik:
- Datenfluss-, Struktur- und Verhaltensbeschreibung
- Konfigurationen (Was leisten diese?)
- Unterschied Signal/Variablen
- Aufbau eines VHDL-Simulationssystems:
- Analyse und Synthese
- Interner Ablauf einer VHDL-Simulation
- Zusammenfassung VHDL: mächtige Sprache -> schwierige Synthese
  - Synthetisierbarkeit
- Abstraktionsebenen
- -> Architektursynthese, Register-Transfersynthese, Logiksynthese, Schaltungssynthese
- Automatisierung ist teilweise problematisch
- Unterscheidung in kombinatorische und sequentielle Schaltung
  -> Anwendung von Inferenzregeln
- Technologieabbildung
- Abbildung einer technologieunabhängigen Netzliste (boolsche Netzliste) auf eine Zielstruktur (Abhängig von der Technologie des Schaltkreisherstellers)
- Abbildung unter Verwendung einer Kostenfunktion (Zeit+Fläche)
- Strategie: (1) Rückführung der Netzliste auf standardisierte Form (2) Versuch der Überdeckung durch Bibliothekselemente
- Optimale Technologieabbildung mit minimalen Kosten (TODO: Formel einfügen!)
- ^^ in der Gleichung bleibt ein Problem: Ein Gatterausgang kann nicht als Eingang weiterverwendet werden, Lösung: weitere Bedingung: Verfügbarkeitsbedingung

3. Halbleitertechnologie
- FET(spannungsgesteuert) vs. Bipolartransistor(stromgesteuert) (Leistungsaufnahme!)

- Aufbau von MOS
- Funktionsweise des Anreicherungstyps
- Schaltverhalten n-Kanal, p-Kanal-Mosfet
- Gatterlogik und Komplexgatter
- prinzipieller Aufbau von CMOS-Gattern
- Komplexgatter (TODO: Was ist das?) erlauben einstufige Realisierung
- Abbildung in dichte Layouts
- Gesetz von NOYCE/MOORE:
- Anteil der Geschwindigkeit der Entwicklung der Technologie an der Entwicklung des Mikroprozessors
- Schrumpfung der Bauelemente durch Skalierung
- alle 18-24 Monate erfolgt eine Verdopplung der Bauelemente auf einem Chip (empirisch gewonnene Aussage von Moore in den 60igern)
- natürliche Grenze: Leitungen nicht dünner als die sich darin bewegenden Elektronen
- Technologie: 1.-4. Generation, 5. Generation hat sich nicht durchgesetzt (-> Real World Computing, Earth-Simulator!)
- Mikroelektronik dominiert durch planare Halbleitertechnologie
  SKALIERUNG (1):
- Welche Parameter gelten für die Bauelemente: + Skalierung mit einem Faktor /alpha > 1 Realistisch: nicht alle Parameter werden mit dem gleichen Faktor skaliert
- Versorgungsspannung konstant (Kompatibilität!)
- Welche Vorteile hat die Skalierung? (Verzögerungszeit, Fläche, Verlustleistung, Kosten)
- Welche Probleme gibt es bei der Skalierung? (Bsp. Gatelänge, Weite der Verarmungsschichten von Source/Drain), Folgen und Maßnahmen!
  SKALIERUNG (2): SCHWIERIGKEITEN BEI VERBINDUNGEN:
- "interconnect crisis" - langsame globale Verbindungen, wenig externe Verbindungen; Herkunft dieser Probleme
- Laufzeitverzögerungen (lange Leitung ;-) -- Abnahme der Kanallängen um den Faktor /alpha
  - Gatterschaltzeiten nehmen ab, Laufzeitverzögerung nimmt zu bei gleich langer Leitung
- Quadratische Zunahme der # der Bauelemente (durch Skalierung UND/ODER Chipvergrößerung)
- Limitationen durch # der Pins -> Regel von RENT: Anzahl der Pins vs. Anzahl der verwendeten Gatter, P=B*N^s (B,s schaltkreisspezifisch) -> s ca. 0,7: Bedarf an externen Anschlüssen steigt stärker als die # der Pins
  - Lösung: explizite Laufzeitverzögerung durch spezielle Werkstoffe, um die RC-Konstante zu erniedrigen

- elementare kleine Befehlssätze
- festverdrahtete Leitwerke (Mealy-Automat)
- konsequentes Pipelining
  - Pipelining:
- überlappte Bearbeitung von Arbeitsteilschritten
- mindestens ein Dutzend Stufen (aktuelle Implementierungen)
  - Superskalares Pipelining:
- Gruppieren von mehreren Befehlen (mehrere Rechenwerke)
- Gleichzeitige Anwendung von Operationen auf einzelne Komponenten eines Vektors, aber: auch skalare Operationen müssen durchgeführt werden
- benötigt Befehlsgruppierer, dynamische Parallelisierung (Umordnung sequentiell einlaufender Befehle)
- Allgemeines Prinzip: Keine direkte Parallelität! Herausziehen von Parallelität aus dem sequentiellen Befehlsstrom!
  - Pipelining: Welche Leistungssteigerung ist (theoretisch) möglich? (TODO: Werte herausfinden!)
- Gesamtzeit Tk, Speed-Up Sk (in Abhängigkeit der # stufen, bzw # Instruktionen)
  - Hazards: (Struktur, Steuerung, Daten) -- Lösung per Rechnerarchitektur oder per Compiler (Codegenerierung)
- Strukturhazard: Bsp. Nur ein Speicherport, gemeinsamer Zugriff, Lösung: idlen der Instruktion
- Steuerungshazard: Bei Sprüngen, Verzweigungen, Aufrufen und Rücksprüngen; explizites Laden des Folgebefehls -> könnte erst spät bekannt sein, Folge: Aussetzen des Teilwerkes bis Ziel bestimmt
- Lösung für Steuerungshazards: spekulative Befehlsausführung:

(i) mehrere Befehlsströme

- Ein Ergebnis wird behalten, das andere verworfen

(ii) Vorabspeichern des Sprungziels + (iii) Puffer für Verzweigungsziele

  - branch target cache (Idee: Einmal angesprungenes Ziel wird nochmals angesprunge)

- Fifo-Befehlsspeicher zwischen Befehlscache und Instruktionsregister

TODO: Unterschied zum "normalen" Cache? FIFO!

(iv) stat./dyn. Verzweigungsvorhersage (branch prediction)

STATISCH:

- "branch not taken" ; "branch taken" ; "predict by opcode"
- Zusätzliches Berechnungshardware

DYNAMISCH:

- Aufzeichnen der Vorgeschichte
- "Taken/not taken"-Schalter ; "Branch History Table"

(v) verzögerte Verzweigung

- verzögerte Ausführung von Befehlen, die der Verzweigung folgen
- "delayed slots" - Ausfüllung mit von der Verzweigung unabhängigen Befehlen
- Probleme bei mehrfacher Ausführung (TODO: Rausfinden)
  - Datenhazards: 3 Typen: RAW, WAR, WAW
- RAW: Instr2 liest Operanden, den Instr1 noch nicht geschrieben hat
- WAR: Instr2 überschreibt Operanden, bevor Instr1 ihn gelesen hat
- WAW: Instr2 schreibt Operanden, der durch Instr1 überschrieben wird
- Gegenmaßnahme: Forwarding
  - Rückführung der Ergebnisse an die Eingänge der ALU (bypass)
  - komplexe, aber verkürzte Datenpfade!
  - Umordnung des Codes (Code Scheduling) (EPIC-Architektur)
- Gegenmaßnahmen in HW (Dynamic Scheduling)
  - Vorteile:
  - einfacher Compileraufbau
  - Platformunabhängigkeit des Codes
  - Berücksichtigung von zur Compilezeit unbekannten Abhängigkeiten
  - Grundprinzip:
  - out-of-order execution/completion
  - Verhindern des Anhaltens der Pipeline durch Ändern des ursprünglichen Ablaufplans
  - Nachteil:
  - Gefahr von WAR und WAW
- SCOREBOARDS: (part of dynamic scheduling)
- HW-Tabelle zur Verwaltug von Instruktionsfolgen
- Instruktionsausführung ohne Rücksicht auf ihre Reihenfolge im Code
- Vorraussetzung: Struktur- und Datenhazardfrei!
- Zweiteilung der Dekodierphase:
- (1) (DE1) (issue stage): Instruktionen dekodieren, Strukturhazards ausschliessen
- (2) (DE2) (read ops): Datenhazards ausschliesen, Operanden lesen
- Gefahr von WAR und WAW durch Änderung des Planes; Gegenmaßnahmen:
  - für WAR:
  - Kopie der Operanden bei der Operation mitführen
  - Register *nur* während der OP auslesen
  - für WAW:
  - Hazard erkennen und warten bis dieser aufgelöst ist

```
- Scoreboard überwacht
- Datenabhängigkeiten
- auszuführende Ops
- Instruktionszustände
- Scoreboard unterteilt ID in
- DE1 und DE2
- Vierstufige Ausführung:
- 1. DE1:
- Instruktion lesen und auf Strukturelle Hazards prüfen
- freie Funktionseinheit (FE) && keine Instruktion
   mit gleichem Zielregister?
- ja: Instruktion an FE senden, update Scoreboard
- nein: stalls für aktuelle Instruktion, keine
   weiteren Instruktionen aktivieren
- 2. DE2:
- Operanden lesen && warten bis Datenhazards
   ausgeschlossen -> Lesen der Register
    - Quelloperand nutzbar? (d.h. früher aktivierte
      Instruktion, die ihn benutzte, hat geendet ||
      unbenutztes Op-Register! -> Ausschluss von RAW)
    - ja: SB veranlasst Lesen des Registers
- 3. AS: (Operation ausführen)
- FE startet Operation
- bei längertaktigen Operation: Scoreboard informieren
   falls fertig
- 4. RS: (Rückschreibephase)
- auf WAR-Hazards testen
- ja: stalls, bis aufgelöst

- Scoreboard Datenstruktur:
enum TYPE_STATE =  issue, read_op, fe_ready, write_result ;

typedef struct InstructionT
TYPE_STATE instr_state;  // which state are we in
typedef struct FunctionEntityT
boolean busy;  // FE active/inactive
TYPE_OP op;  // type of operation
TYPE_REG dr;  // destination register
TYPE_REG sr1, sr2; // source registers 1 and 2
FunctionEntityT *fe1,*fe2;// FE, which create sr1 and
// sr2's data
boolean fsr1, fsr2; // indicate if sr1 and sr2 are
// usable
```

```
FunctionEntityT reg_state[NumberOfRegisters]; // shows which FE
 // accesses a reg
- Scoreboard Algorithmus:

InstructionT instr;
issue_stage(instr);

FunctionEntityT fe;

switch (instruction)
case issue:
wait_until(!fe.busy && !reg_state(dr));
fe.busy=TRUE; fe.op=instr.op;
...
...

- Fazit Scoreboard:
- einfache HW-Strukur, erhöhte Tendenz zu WAR- und WAW-Hazards
   durch out-of-order commit

- TOMASULO-Algorithmus: (Basis für alle heutigen dynamischen
   Ablaufpläne, ab P2)
- Verhindern der Steigerung von WAR- und WAR-Hazards
- Registerumbennung + Scoreboard
- Unterschied Scoreboard:
- Hazarderkennung und Ausführungssteuerung getrennt
- Ergebnisse gehen direkt an die Funktionseinheiten
   (Umgehung der Register)
- Struktur:
- Reservierungsstationen
- Reservierungstabellen an jeder Funktionseinheit triggern
   den Beginn einer Instruktion (Signal an Steuerung)
- Registerumbenennung
- gemeinsamer Datenbus
- dreistufige Ausführung:
- 1. Befehl installieren (BH+DE) (hole nächste Instruktion)
- Reservierungsstation frei?
- ja: d.h. kein Strukturhazard, Instruktion starten und Operanden
   übertragen (Register umbennen)
- 2. Ausführung (BA) (führe Operation aus)
- Beide Operanden nutzbar?
- ja: starte Ausführung
```

- nein: beobachte gemeinsamen Datenbus
- 3. Rückschreiben (RS) (schliesse Operation ab)
- Ergebnis über gemeinsamen Datenbus an alle wartenden Funktionseinheiten senden
- markiere nutzbare Reservierungsstation
- normaler Datenbus: Daten + Zieladresse
- gemeinsamer Datenbus: Daten + Quelladresse
- Vorteil Tomasulu:
- Vermeidung von WAR, WAW durch Registerumbenennung
- Abkürzende Datenpfade durch Forwarding (gemeinsamer Datenbus ermöglicht Anliegen des Ergebnisses einen Taktzyklus früher als bei Scoreboard)
- Nachteil:
- komplexe Hardware: komplexere Kontrolllogik, assoziative Speicherung (TODO: Assoziative Speicherung? Schlecht?)
- RISC/CISC- Vergleich
- CISC:
- Kosten für Software übersteigen die der Hardware
- Komplexität bei Hochsprachen
- Semantische Lücke
- In many layered systems, some conflicts when concepts at a high level of abstraction need to be translated into lower, more concrete artifacts. This mismatch is often called semantic gap.

- For example, semantic gap denotes the difference between the complex operations performed by high-level language constructs and the simple ones provided by computer instruction sets. It was in an attempt to try to close this gap that computer architects designed increasingly complex instruction set computers.
- Divergenz zwischen komplexen Softwareinstruktionen bei Programmen und Betriebssystemen und den einfachen Hardwareinstruktionen
- damit:
- große # Befehle, mehrere Adressierungsarten, Hardwareunterstützung für Hochsprachenkonstrukte
- Intention:
- einfachere Implementierung von Compilern
- Verbesserung der Ausführungseffizienz (Komplexoperationen als Mikrocode)
- Unterstützung von Hochsprachenkonstrukten
- RISC:
- Pattersenstudie 1982

- Untersuchung von bestimmten Eigenschaften bei der Befehlsausführung von Hochsprachenprogrammen
- Operationen
- 1. Zuweisungen, 2. Bedingungen
- Transport und Verzweigung auf Maschinenebene optimieren
- Operanden
- 1. lokale, skalare Variablen
- Optimieren des Zugriffs auf lokale Variablen
- Wichtige Lösung: lokale Variablen in Registern halten (=> Reduzierung der Speicherzugriffe)
- Notwendigkeit für RISC: großer Registersatz
- Hardwarelösung
- viele Register
- hohe # lokaler Variablen kann in Registern gehalten werden
- Softwarelösung
- Compiler allokiert Register optimal
- Allokierung basiert auf der Feststellung der höchsten Verwendung von bestimmten Variablen in einer Zeit
- komplexe Programmanalyse
- Bsp. Graphfärbungsalgorithmus
- Optimierung der # der verwendeten Register
- TODO: Wie funktioniert der?
- Bsp. Registerfenster
- bezüglich Prozeduraufrufen
- wenige Parameter, begrenzte Aufruftiefe
- kleinere Registermenge
- Aufruf: Umschalten auf eine andere Registermenge
- Rücksprung: Rückschalten auf eine vorherige Registermenge
- Überlappende Registerfenster sind Lösung
- TODO: Wie funktionieren die?
- RISC-Zusammenfassung:
- Ansatz: 1 Instruktion pro Takt
- Operationen nur auf Registern (Load/Store-Architektur)
- wenige, einfache Adressierungsmodi (TODO: Genauer, was sind Adressierungsmodi?)
- wenige, einfache Befehlsformate
- festes Befehlsformat
- kein Mikrocode (festverdrahteter Befehlsentwurf)
- mehr Compilezeit erforderlich (schärfere Analyse)
- CISC, warum nur, CISC?!?!
- Vereinfachter Compiler? Umstritten!

- komplexe Maschinenbefehle schwierig auszunützen
- schwierigere Optimierung
- Kleinere Programme?
- weniger Speicherbedarf (Speicher jedoch billig!)
- viele Befehle -> lange Opcodes
- Schnellere Programme?
- Trend zur Verwendung einfacher Anweisungen
- komplexere Leitwerke nötig
- Mikroprogrammspeicher ist größer -> Ausführung einfacher Programme benötigt mehr Zeit!
- Fazit: Unklar, ob CISC eine vernünftige Lösung ist!
- RISC-Prozessoren:
- sehr leistungsstark, Verzicht auf Kompatibilität
- CISC-Prozessoren:
- Kompatibilität im Vordergrund
- trotz geringerer Leistungsfähigkeit durch Mikroprogrammierung Marktführer!
- Kombination RISC/CISC:
- Ziel: Hohe Leistung && Hohe Kompatibilität
- RISC Techniken in CISC eingeführt
- Sprungzielspeicher
- Befehlssatz auf RISC-Ebene herunterbrechen und Teilmenge nach dem RISC-Prinzip abarbeiten, andere nach Mikroprogrammierung
- interne Dekodierung von CISC-Befehlen in elementare RISC-Befehle
- Betriebssystemerweiterung (erweiterte E/A mit CISC-Instruktionen)
- Ausblick:
- Derzeitige RISC-Prozessoren nähern sich CISC an
- Kompatibilität und On-Chip-Funktionalität erhöhen
- feinkörnige Parallelität bei sowohl RISC, als auch CISC
- verstärkte Nutzung von Pipelining (Superpipelining)
- höhere # Stufen, kürzere Befehlszyklen, höherer Takt
- mehrere Spezialisierte ALU’s (Superskalarität)
- FP, Int, Adressberechnungseinheiten mehrfach!
- vier Entwicklungen:
- (1) VLIW (Very long instruction word)-Architekturen
- statische oder explizite Parallelisierung
- (2) Multithreading
- Hyperthreading
- (3) Netzwerkprozessor
- basieren auf JVM

- (4) Chipmultiprozessoren
- Ziel von VLIW und Multithreading:
- Beschränkungen des Parallelismus bei superskalaren Architekturen beseitigen
- dynamische Paral. (immer nur ein Ausschnitt des Programmkodes)
- statische Paral. (im Prinzip das gesamte Programm verfügbar)
- (1) VLIW-Architekturen:
- Parallelisierung nicht zur Laufzeit sondern während Compilephase (Codeoptimierung)
- Compiler erzeugt lange Instruktionen (jeder ALU einen Maschinenbefehl zuordnen)
- Analyse mittels DAG (Datenabhängigkeitsgraph)
- PRO:
- Zeitaufwand nur in der Übersetzungsphase (Optimierung)
- schlankere Leitwerke (Steuerlogik vereinfacht sich)
- CON:
- Parallelisierung nicht so flexibel wie im Dynamischen
- Redundanz im Code
- Programme superskalarer Rechner sind inkompatibel zu VLIW-Rechnern
- EPIC-Architektur: (IA-64 HP+Intel), Ist eine VLIW-Implementierung
- Verzweigungen:
- spekulativ (superskalar)
- beide Pfade werden ausgeführt, der falsche wird später verworfen
- spekulatives Laden
- noch nicht benötigte Daten werden willkürlich aus dem Speicher geladen
- Gleitkommaverarbeitung
- MAC(?)-Operationen, "Multiply-and-Accumulate"
- Kompatibilität
- Einmal übersetzter Code wird im Speicher gehalten
- (2) Multithread-Architekturen
- Idee: Programm durch Programmierer oder Compiler in von einander unabhängige Teile (Fäden) zerlegen
- von einander unabhängige Befehle verschiedener Prozesse mischen
- Pro Prozessor ein Thread (Faden)
- Globaler Speicher zur Synchronisation
- Compiler ist für Synchronisierung zuständig
- jeder Thread hält lokalen Speicher

- einfacher Prozesskontext pro Thread
- aktuelle Variante: Hyperthreading (synchrones Multithreading)
- mehrere logische Prozessoren auf einem physikalischen Prozessor
- überproportionaler Anstieg von elektrischer Leistung und Chipfläche gegenüber Zuwachs an Rechenleistung
- Parallelismus auf Threadebene:
- time slice multithreading
- Zeitscheibenmechanismus
- event driven multithreading
- Schalten bei Ereignissen, die lange Latenzzeiten nach sich ziehen
- simultaneous multithreading
- kein Schalten, sondern auswählen
- logische Prozessoren durch Architekturzustand gegeben
- eigener Registersatz, Kontrollregister, Maschinenstatusregister
- eigenes Steuerregister für Interrupts (APIC)

- Das Rechenwerk
- Prozessor besteht aus Leitwerk und Rechenwerk
- IEEE754: Spezifikation:
- vier Fliesskommazahlenformate
- single, double prec, single, double extended
- -> Genauigkeitsanforderungen für die Formate
- | Vz | HB | Ex | .. | Ex | Mt | .. | Mt |
- 31 30 29 23 22 0
- TODO: Wie genau funktioniert die Darstellung?
- Schnelle Addierer
- Grundlage 1-Bit FA:
- S = ~AB~C v A~B~C v ~A~BC v ABC
 = ~C(~AB v A~B) v C(~A~B v AB)
 = ~C(A x B) v C(~(A x B))
 = not(C) and (xor (a,b)) or C and (not(xor(a,b)))
- C = AB v AC v BC (alternativ: C=AB v C(AxB))
- Serieller Addierer mit FA:
- Schiebe-Register A, B geben pro Takt ihre Werte an den FA, der gibt sum aus und legt Carry für den nächsten Takt in FlipFlop ab (für Eingang am FA)
- Ripple-Carry-Addierer
- n-Bit-Wörter A(1..n) und B(1..n) legen ihre Werte jeweils an FA(1..n) an
- das von FA(c) berechnete carry wird dabei an den Eingang von FA(c+1) angelegt (Latenz!)
- Vorteil:

- einfache Struktur
- gut für VLSI: Transmission-Gate, Manchester-Addierer
- Nachteil:
- hohe Latenzzeit
- drei Mechanismen zum schnellen Addieren:
- (1) schneller Übertragsdurchlauf
- Transmissiongate Addierer (TODO: Wie arbeitet das Ding, was bedeuten die Eingänge Pi und ~Pi?)
- (2) parallele Übertragsberechnung
- (3) hierarchische Addiererstrukturen
- Conditional-Sum-Addierer
- Bsp. Carry-Look-Ahead
- Alternative: Carry-Select-Addierer
- bestehend aus Conditional-Sum-Addierer
- hierbei werden zwei Fälle berechnet:
- Fall 1: Kein Übertrag entsteht
- S(0) = A x B, C(0) = A ^ B
- Fall 2: Übertrag entsteht
- S(1) = ~(A x B), C(1) = A v B
- in Abhängigkeit der tatsächlichen Situation werden die Bits gewählt (TODO: WIE???)
- Mehr-Operanden-Addierer (Carry-Save-Adder)
- TODO: WIE???

- Multiplikation: Reduzierung der # der Teilprodukte
- Bsp. Radix-4-Multiplication
- Reduzierung der Teilprodukte auf m/2
- Gleichzeitige Auswertung von 2 Bit des Multiplikators
- Probleme:
- erhöhter Hardwareaufwand
- größere Latenz der Einzelschritte
- Bsp. Feldmultiplizierer
- Pipeliningprinzip (Höherer Durchsatz)
- Operandenbits liegen in Matrixform vor:

```
-  a3 a2 a1 a0     |
-                   ------------------------------+---
-a3b0 a2b0 a1b0 a0b0   |b0
-a3b1 a2b1 a1b1 a0b1    |b1
-a3b2 a2b2 a1b2 a0b2    |b2
-a3b3 a3b3 a2b3 a1b3    |b3
-----------------------------------------------------+--
-P7 P6 P5 P4 P3 P2 P1 P0    |
```

- Bsp, für Feldmultiplizierer: Carry-Save-Array-Multiplier
- Funktionalität eines FA um ein AND erweitert

- S = (ab)x s’ x c’ , C= abs’ v abc’ v s’c

- Speicherwerk
- entscheident für Leistungsfähigkeit und Kosten eines Rechners
- Ideal: (aus wirtschaftl. Gründen nicht realisierbar)
- ausreichende Kapazität
- Zugriffszeit in der Größenordnung der Prozessorgeschwindigkeit
- Lösung: mehrstufige Speicherhierarchie (Cache!)
- jeder Speicher einer niedrigeren Hierarchiestufe hält den Ausschnitt des Speicher des nächstgrößeren
- Charakteristika eines Speichers:
- Ort
- Prozessor, Intern, Extern
- Performanz
- Zugriffszeit, Zykluszeit, Transferrate
- Kapazität
- Wortbreite, # der Wörter
- Physischer Typ
- Halbleiter, Magnetisch, Optisch, Magneto-Optisch
- Transfereinheit
- Wort, Block
- Zugriffsmethode
- Sequentiell, Direkt, Zufällig, Assoziativ
- Physische Charakteristika
- flüchtig, nicht-flüchtig
- Prozessorzugriff auf schnelleren Speicher überhaupt möglich, da eine zeitliche und räumliche Lokalität von Daten oder Befehlen vorliegt!
- Cache:
- kleiner, schneller Pufferspeicher zwischen Register und Hauptspeicher
- Funktion: Überbrückung der Prozessor/Speicherlücke bzgl. Leistung
- Struktur: Blöcke ähnlich dem Hauptspeicher
- Typen: Primär- und Sekundärcache, (L3- Tertiärcache)
- Primär: Daten und Befehle getrennt, kurze Blöcke bis 64 Byte (Harvard-Cache)
- Sekundär: Daten und Befehle gemeinsam, längere Blöcke (unified cache)
- Zugriff:
- Steuerwerk überprüft anhand Adresse, ob das Datum im Speicher vorhanden ist
- Vorhanden? HIT: Datum auslesen

- Nicht vorhanden? MISS: Wort aus Hauptspeicher in den Cache kopieren und angefordertes Datum lesen
- Organisationsprobleme:
- (1) Platzierungsproblem (in welchem Cacheblock wird der Hauptspeicherblock abgebildet)
- (2) Identifikationsproblem (Wiederauffinden eines Datums im Cache)
- Möglichkeiten für Platzierung:
- (1) direct mapping (einfach assoziativ)
- jeder Adresse B eines Hauptspeicherblocks wird direkt ein Block m bestehend aus N Cache-Blöcken zugewiesen (zbsp. m = B % N)
- (2) full associative (vollassoziativ)
- jede Adresse eines Hauptspeicherblocks kann in einen beliebigen Cache-Speicherblock abgebildet werden
- (3) n-way associative (n-fach assoziativ)
- Cache Blöcke in s-Mengen mit jeweils n Blöcken unterteilen (s = N/n)
- Hauptspeicherblock nach "direct mapping" in eine CacheBlock-Menge abbilden, innerhalb der Menge beliebig
- n=N: voll-assoziativ, n=1: einfach assoziativ
- Abbildung ist surjektiv
- Kennung zur Identifikation der Adresse des Hauptspeicherblocks notwendig (TAG)
- | Tag-Index 24Bit | Index 6Bit | BlockOffset 2Bit|
- mit steigender Assoziativität verschiebt sich die Tag/Index-Grenze nach Rechts
- Zusätzlich noch ein (V)alidy-Bit -> Eintrag gültig?
- n-fach assoziativ -> n-Komparatoren benötigt
- direct mapping -> nur ein Komparator, ABER:
- mehrere Adressen bilden auf den selben Block ab: viele Kollisionen (häufiges Umladen)
- ergo: Assoziativität verringert die Wahrscheinlichkeit von Kollisionen, optimal: voll-assoziativ, aber langsam/teuer
- real: 2,4 oder 8fach-assoziativ
- Aktualisierungsstrategie:
- Inkonsistenz zwischen Hauptspeicher und Cache
- Wann/Wie Hauptspeicher aktualisieren?
- (1) write through
- jede Änderung sofort im Hauptspeicher aktualisieren

- Pro: Konsistenz gegeben
- Con: Hohe Last für CPU/Speicherbus
- Anwendung in Primärcaches
- (2) write back
- Aktualisierung bei Verdrängung,
  Ausgabeoperation, Zugriff eines anderen
  Prozessors (SMP)
- dirty bit -> Rückschreiben nicht-modifizierter
  Einträge vermeiden
- Anwendung in Sekundärcaches
- Ersetzungsstrategie:
- cache miss -> Fehlzugriff im Cache tritt auf
- Nachladen des Datums aus dem Hauptspeicher
- Ersetzen von vorhandenen Daten (möglicherweise) nötig
- direct mapping: easy
- assoziativ:
- LRU (least recently used)
- Alterungsinformationen mitführen
- LFU (least frequently used)
- Benutzungsstatus (Zähler o.ä.)
- FIFO (first in - first out)
- z.bsp. durch Ringpuffer realisierbar
- ersetzen der am längstem im Puffer
  befindlichen Zeile
- random
- geringer Hardwareaufwand durch Wegfall von
  Stapel- oder Pufferstrukturen
- pseudo random -> ermöglicht auch debugging
- Leistungsbetrachtungen
- mittlere (effektive) Speicherzugriffszeit:
-
- $T_a = T_h + m * T_m$
-
- m - miss rate, T_m - miss penalty (Nachladezeit),
  T_h - hit time (mittlere Zugriffszeit)
- Optimierung von T_a
- Reduzierung von T_h, m und T_m
- Aber: Reduzierung von einer Größe bedeutet häufig
  Verschlechterung der beiden anderen
- Kompromiss finden
- Reduzierung von m (miss rate)
- Erhöhung der Kapazität oder der Assoziativität
- Erhöht T_h (Zugriffszeit)
- Erhöhung der Blockgröße

- Erhöht T_m (Nachladezeit)
- Maßnahme für direkt abbildende Caches:
- victim cache (voll-assoziativer Nebencache)
  für zuletzt verdrängte Blöcke
- prefetching (vorrausgreifendes Laden)
- Reduzierung von T_h (Zugriffszeit)
- bei Primärcache: entscheident für die
  Prozessortaktrate
- bei Sekundärcache: entscheident für die
  Wartezyklen
- kleinerer Cache -> besser, aber: erhöht wiederum
  die miss rate m
- Reduzierung von T_m (Nachladezeit)
- maßgeblich durch Prozessor-/Systembus bestimmt
- Einführung eines Sekundärcaches
- T_m wird dann ebenfalls wie T_a berechnet
- Einführung eines non-blocking cache
- Nachladen und Zugriff gleichzeitig möglich bei
  Treffern
- mehrere ausstehende Speicherzugriffe
  gleichzeitig
- Verhältnis 2-Ebenen-Speicher S2/S1 (TODO: Formel
  nachtragen)
- Zugriffseffizienz T1/T2 (TODO: Formel nachtragen)
- Klassifikation von Fehlzugriffen:
- (1) compulsory
- erster Zugriff trifft nicht den Block
- Block muss nachgeladen werden
- first reference miss
- (2) capacity
- cache ist zu klein um alle Blöcke der aktuellen
  Befehlsfolge zu enthalten
- (3) conflict
- collision or interference misses
- in n-way associative oder direct mapping caches
  möglich
- Adresskonflikte führen zum Überschreiben von
  Blöcken (und müssen ggfs. später nachgeladen
  werden)
- # der Fehlzugriffsraten kann durch steigenden
  Kapazität vermieden werden
- Hauptspeicher
- Zykluszeit = Zeit, bis nächste Adresse angelegt werden kann
- SRAM (static RAM -> Cache und Hochleistungsspeicher)

- Speicherzelle: FlipFlop
- zerstörungsfreies Lesen
- 6-8 Transistoren
- schneller (8fach)
- geringere Kapazität (8fach)
- Zugriffszeit = Zykluszeit = 0.3 - 0.5ns
- DRAM: 5-6ns Zykluszeit
- Größere Wortbreiten: parallele Anordnung
- DRAM (breiter Einsatz)
- Speichermatrix mit einer oder einigen 1Bit Speicherzellen an Knotenpunkten
- Vorteil:
- Kompakt
- Nachteil:
- zerstörendes Lesen, benötigt refresh cycle (Leckströme bedingen 8ms Zyklus)
- Adressierung via Zeile/Spalte im Multiplexbetrieb
- Einsparung von Pins
- RAS (Row Adress Strobe), CAS (Column Adress Strobe)
- Aufbau von Speicher blockorientiert
- byte-adressierbarer 16MByte-Speicher mit 32Bit-Wortbreite aus 4x1-Bit-DRAM
- | 31-24 | 23 - 2 | 1 0 |
- ^^ADR^^^ ^Byte^
- memory controler zur
- Adressinterpretation
- Wortadressierung
- Auswahl einer oder mehrerer byte-Blöcke
- memory bank
- parallel angeordnete Speicherbausteine + memory controller
- Speicherverschränkung (memory interleave)
- Zykluszeit bremst Prozessor
- Lösung: Benachbarte Worte liegen in unterschiedlichen Bänken -> Speicherzugriffe auf unterschiedliche Bänke können überlappen (TODO: genauer angucken)
- Fehlerkorrektur in Halbleiterspeichern
- Verwendung von K zusätzlichen Prüfbits
- (TODO: Fehlerkorrektur in HLS! Wie funktioniert das?)
- memory gap
- Latenzzeit verringert sich jährlich nur um 10%, langsamer als die der CPU
- Lösung: (weitere Architekturmaßnahmen)
- Nibble-, Page- oder Static-Column-Modus (beim Speicherzugriff gleich mehrere Folgebits in der

  aktiven Zeile auslesen)
    - EDO-Ram (Extended Data Out) DRAM
    - PageMode mit zusätzlichem Zyklus zum einlesen, Überlappung bringt Vorteile im Pipelining Stil
    - PageMode:
    - row im DRAM wird offengelassen um nocheinmal gelesen zu werden (Column, Nibble sind Varianten hiervon)
- E(nhanced)DRAM, C(ached)DRAM
- Cache auf dem Speicherchip
- S(ynchron)DRAM
- Betrieb synchron zum CPU-/Speicherbus (nicht asynchron wie normaler DRAM)
- zusätzliche Speichermatrizen (memory interleaving erhöhen)
- burst mode (schnelle Blockübertragung)
- mehrere Adressen "gleichzeitig" auslesen, mit adressinkrementeller logik on-chip
- bei 100MHz -> 10ns für Folgezugriffe (Latenz)
- Taktsignal treibt endlichen Automaten on chip an, der Pipelining ermöglicht
- D(ouble)D(ata)Ram
- Datenübertragung bei fallender und steigender Taktflanke
- DDR, DDR-2, DDR-3 Standards
- fallende Versorgungsspannung 2.5V -> 1.6V
- unterschiedliche Geschwindigkeit
- D(irect)R(ambus)DRAM (a.k.a. RIMM)
- ähnlich DDR, jedoch unterschiedliche Signalgebung, die höhere Taktraten ermöglicht
- besteht aus mehreren Kompononenten (RAMBUS-Controller, RDRAM-Speichermodule, RAMBUS-Kanal)
- das Design arbeitet mit bis zu 400MHz -> zwei Taktflanken ermöglichen 800MHz
- kritisch: Mainboarddesign (hohe Taktraten müssen erzeugt werden),
- Technik:
- kurze Signalwege, nur 16 Bit Wortbreite
- Entflechtung der Leiterbahnen, weniger Pins
- non-volatile memory (nicht-flüchtig)
- Flash-Eprom (TODO: Wie funktioniert?)
- MRAM: (neue Technologie)

- Speicherung wird realisiert durch die Ausrichtung magnetischer Momente in benachbarten magnetisierten Schichten
- Speicherzelle als 3-lagiges "Sandwitch"
- Oben/Unten Fe, Mitte Isolator, Dicke ca 3-6nm
- GMR: Mitte = Nicht-Magnetischer Leiter
- TMR: Mitte = Isolator
- Speicherung beruht auf magneto-resistance
- elektrischer Widerstand ändert sich durch Magnetfeld (Lorentzkraft)
- anti-parallel: hoher Widerstand: logisch 0
- parallel: niedriger Widerstand: logisch 1
- GMR-Prinzip: spin-abhängige Streuung
- TMR-Prinzip: spin-abhängige Tunnelung
- E/A-Werk
- externe Speicher (magnetisch, optisch, magneto-optisch)
- Bussysteme
- alle an einer Leitung
- Alternative: P2P
- funktionelle Unterteilung in Steuerbus, Adressbus, Datenbus
- strukturelle Unterteilung in
- Cachebus, Systembus, Peripheriebus
- Notwendig: Chipsets (Buskontroller)
- steuern den Busverkehr
- Anliegen: Entlasten der CPU von Zugriff auf Peripherie bzw. Hauptspeicher
- benötigt wird ein Busprotokoll
- Busprotokoll
- asynchrones Protokoll:
- Handshake (req,ack)
- skalierbar (laufzeiten unerheblich)
- langsamer als synchrone Protokolle
- meist für langsamere Peripherie genutzt (Drucker)
- synchrones Protokoll:
- taktgesteuert, keine Handshakes nötig
- nicht skalierbar (Geräte müssen im Takt mitarbeiten)
- Einsatz über kürzere Distanzen, Cachebus, Systembus und schnelle Peripheriebusse
- Cachebus
- paralleler Zugriff auf Cache- und Systembus
- Cachebusse breiter und schneller getaktet als Systembusse

- führt zu Problemen bei Multiprozessorsystemen (Cache-Kohärenz)
- Systembus (a.k.a. CPU-Bus)
- Verbindet CPU mit Hauptspeicher/Peripherie bzw. Chipset
- entweder als
- 1 Systembus oder
- Front-Side und Back-Side-Bus
- BSB: Verbindung zu Cache
- FSB: Verbindung zu Hauptspeicher/Periph.
- als 1 Systembus:
- doppelt so breite Wortbreite als CPU
- keine ständige Anpassung an steigende CPU-Taktfrequenzen
  - Lösung mit Datenpufferung und Zugriff auf größere Datenblöcke
- Verwendung von burst access
- 1x Adresse übergeben
- Sender/Empfänger erhöhen die Adresse pro Taktzyklus
- Übertragung (meist) 4 Datenblöcke in der Busbreite
- Angabe nach wievielen Takten ein Datenblock eintrifft: (2-1-1-1, oder 3-2-2-2)
- CPU-nahe/schnelle Peripheriebusse sind AGP,PCI,PCI-X,PCIe
- AGP:
- grundsätzlich auf 66MHz-Basis (Durchsatz: 266MB/s) und kennt zwei Betriebsmodi:
  - AGP-1x: nur steigende Taktflanken
  - AGP-2x: + fallende TF
  - AGP-4x: niedrigere Spannung(TODO: ?), höherer Takt
- PCI-Bus:
- Bustakt synchron zum CPU-Takt: max. 33/66MHz
- Busbreite 32 oder 64Bit
- Adress- und Datenmultiplexing
- (TODO: Wie gehtn das hier?)
- bei 32Bit Busbreite max. 66MB/s (schreiben) und 44MB/s (lesen)
- PCI-bursts ermöglichen Steigerung des Durchsatzes auf 133MB/s (32Bit), oder 266MB/s (64Bit)
- Einsatz von Bridges (PCI-to-ISA-Bridge) zur Verbindung zu anderen Bussen
- Bus-Master, Slave-Prinzip
- PCI-BusMaster kann R/W-Zugriffe auf den Speicher

tätigen, ohne CPU-Einsatz
- Slave -> nur Empfänger (Bsp. Grafikkarte, Soundkarte)
- Plug & Pray - Prinzip
- Konfiguration der Karte via ROM-Bios
- Konflikte von Interrupts oder Adressen beantwortet das ROM-Bios durch Änderung der Werte oder Abschalten bei Fehlfunktionen
- Hierarchischer Aufbau Systembus-PCIBus
- PCI-Express:
- AGP und PCI haben ausgedient
- neue, schnelle serielle Verbindungen
- höherer Takt
- schnellerer Speicher und langsamerer I/O-Controller ersetzt durch einen einzigen Datenpuffer (genannt Bridge)
- wenige, schnell-multigeplexte Leitungen (lanes in links)
- USB: (universal serial bus)
- Stern-Topologie mit bis zu 127 Endgeräten
- Steuerung durch USB-Controller (Host), keine direkte Verbindung zwischen Endgeräten, Host ist einziger Interrupt-Träger
- Geschwindigkeit: USB 1.0 (1.5Mbit), USB 1.1 (12Mbit), USB 2.0 (240MBit)
- hot-plugging
- FireWire:
- Alternative zu USB2.0
- IEEE1394
- 800MBit

5. Leistungsbewertung
- Was ist Leistung?
- Instruktionen pro Zeit
- Antwortzeit
- Kommunikation
- Durchsatz
- Genauigkeit,
- ...
- Was ist Bewertung?
- Zuteilung quantitativer Werte
- Was ist Leistungsbewertung?
- Versuch, objektive Aussagen über Rechensysteme anhand messbarer Größen zu erwerben

- Schwierigkeit der Leistungsfeststellung:
- keine eindeutigen objektiven Vergleichskriterien
- Leistung für bestimmte Rechnerkonfiguration unter
  Verwendung bestimmter Nebenbedingungen + Software
- subjektives Empfinden
- Einflüsse durch:
- Algorithmen
- Software
- Hardwarekonfiguration
- Betriebssysteme
- Netzwerk/Kommunikation
- Was will man also mit Leistungsbewertung erreichen?
- Entwurf, Optimierung, Weiterentwicklung von Rechnersystemen
- Auffinden von Flaschenhälsen
- ungezielte Aufrüstung von Rechnerteilen sinnlos
- Analyse, Auswahl und Konfiguration von Gesamtsystemen aus
  HW/SW
    - standardisierte Messprogramme benötigt (-> Benchmarking)
- 4 Aspekte der Leistungsbewertung:
- (1) zu untersuchendes System
- (2) Leistungskenngrößen
- (3) Belastung oder Last
- (4) Methode zur Ermittlung der Leistungskenngrößen
- Siehe getexte Ausarbeitung

6. Fehlertoleranz

%warum die zweite Klammer?

Teil II.

Fehlertoleranz und Leistungsbewertung

1. Leistungsbewertung

1.1. Kenngrößen der Zeit

Mittlere Operationszeit:

$$T = \sum_{i=1}^{n} t_i \cdot p_i$$

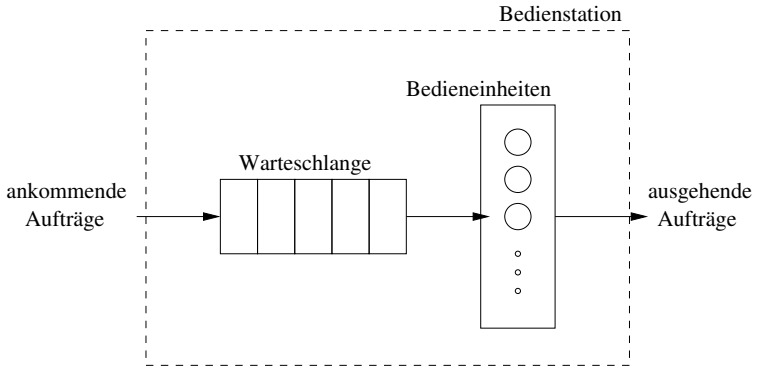
t_i ... Operationszeit des i-ten Befehls
p_i ... Relative Häufigkeit des i-ten Befehls im Befehlssatz

Mittlere Operationszeit: *alternativer Ausdruck*

$$T = CPI \cdot T_C + S \cdot T_S[s]$$

CPI ... cycles per instruction
T_C ... Taktzykluszeit
S ... Speicherbedarf bei Durchschnittsbefehl $[bit]$
T_S ... Speicherzugriffszeit [s/bit]

Verweilzeit, Wartezeit, Bedienzeit *Warteschlangenmodell*



Bedienzeit ... Dauer der Ausführung
Wartezeit ... Dauer des Aufenthalts in der Warteschlange
Verweilzeit... Verzögerungszeit, Latenz, Antwortzeit

1.2. Kenngrößen zum Durchsatz

MIPS, MOPS, MFLOPS:

$$\text{MIPS/MOPS:} \quad \frac{1}{\text{mittlere Operationszeit}} = \frac{1}{T}$$

Da weder die Leistungsfähigkeit der einzelnen Operationen, noch die relativen Geschwindigkeiten der beteiligten CPUs einberechnet werden eignet sich dies nicht zum Vergleich von verschiedenen CPUs. MFLOPS sind ebenfalls ungünstig, da sie sich auf Programme beschränken, die sehr Fliesskomma-zentriert sind. Auch die relative Komplexität dieser Operationen wird nicht mit einbezogen.

Grenzdurchsatz, Verlustrate: Für Warteschlangenmodell relevant:

In Rechnernetzen spricht man beim *Grenzdurchsatz* von der *Bandbreite*, wobei dies somit den maximalen Durchsatz darstellt. Da die *Latenz* in Netzen, die nahe der Grenzfrequenz operieren, oft sehr hoch ist, ist als interessantes Maß die Anzahl der Aufträge, die eine bestimmte obere Grenze der Antwortzeit (Latenz) nicht überschreiten von Interesse. Die *Verlustrate* hingegen beschreibt die Anzahl der Einheiten, die pro Zeiteinheit verloren gehen.

1.3. Auslastung

Hierbei von Bedeutung ist das Verhältnis von tatsächlich erreichtem Durchsatz und Grenzdurchsatz. Dazu wird für folgendes Beispiel der *Speed-Up*, also die Beschleunigung definiert:

$$S_p(n) = \frac{T_1(n)}{T_p(n)}$$

Es sei $T_p(n)$ die Laufzeit eines Algorithmus, der zur Ausführung p Prozessoren benötigt, bei einer Eingabe der Länge n. Die Berechnung der *Effizienz* $E_p(n)$ berechnet sich nun aus dem Verhältnis des berechneten Speed-Ups $S_p(n)$ zu dem maximal erreichbaren Speed-Up:

$$E_p(n) = \frac{S_p(n)}{p}$$

Die maximal erreichbare Beschleunigung ist p-fach, da wir davon ausgehen, dass das Problem sich durch die Verteilung auf p-Prozessoren idealerweise um diesem Faktor beschleunigt. Damit gilt für die Effizienz folgende Ungleichung:

$$\frac{1}{p} \leq E_p(n) \leq 1$$

Daher bezeichnet man einen parallelen Algorithmus mit $E_p(n) = 1$ als *optimal*.

1.4. Amdahls Gesetz

Amdahls Gesetz besagt, dass die *Leistungsverbesserung* durch den Einsatz neuer Hardware abhängig ist von

- dem Anteil an Ausführungszeit r, die ein Programm erbringt und
- der Beschleunigungsrate b, die durch die bessere Hardware erreicht wird, wenn das *gesamte* Programm getestet wird.

Mit anderen Worten besagt Amdahls Gesetz, dass sich der Aufwand für die Verbesserung nur dann lohne, wenn dadurch der häufig eintretende Fall verbessere, also die neue Hardware häufig eingesetzt würde.

„making the common case fast“

Es ergibt sich also der Anteil an Ausführungszeit r, der sich parallelisieren läßt und $(1-r)$, der Anteil für den das nicht klappt. Damit gilt für die *Gesamtausführungszeit* T:

$$T = T \cdot (1-r) + T \cdot r$$
$$T_{\text{verbessert}} = T \cdot (1-r) + \frac{T \cdot r}{b}$$
$$\rightarrow \text{Speed-Up} = \frac{T}{T_{\text{verbessert}}} = \frac{1}{(1-r) + \frac{r}{b}}$$

1.5. Methoden

Viele Methoden zur Leistungsbewertung lassen sich in die drei Klassen *analytische Methoden, Messungen und Simulation* einordnen. Analytische Methoden eignen sich gut zur schnellen Orientierung, Simulationen sind dagegen sehr flexibel und Messungen liefern genaue Aussagen.

Analytische Methoden. Hierbei wird das Bewertungsproblem auf der Grundlage von *Gleichungen* approximativ oder numerisch gelöst. Diese Gleichungen sind mathematisch *abgeschlossen* oder bestehen aus Modellen. Sie werden abgeleitet aus z.Bsp. Warteschlangenmodellen, Wartenetzen oder stochastischen Petrinetzen. Berechnet werden quantitative Leistungsmaße wie *Mittelwert, Durchsatz* oder *Latenz*. Das betroffene System wird dabei immer im *stationären Zustand* betrachtet. Mit der Zunahme der Komplexität eines Systems wird auch das Auffinden eines geeigneten analytischen Modells schwieriger und aufwendiger.

Simulation. Ohne das der zu bewertetende Rechner existieren muss, wird eine Annahme über dessen Eigenschaften, also *Struktur, Betrieb, Betriebsprozesse*, getroffen und darauf ein *Ablaufmodell* gewonnen. Mit diesem Modell wird versucht genaue Leistungsaussagen über den Rechner zu gewinnen. Simulationen sind *Programme*, die geschaffen werden, um das Verhalten eines Rechners auf einen anderen abzubilden und werden meist in höheren Programmiersprachen, oder Simulationssprachen geschrieben. Sehr häufig wird dabei eine *ereignisorientierte* Simulation

(event driven simulation) angewendet. Dabei wird die Zeit *diskretisiert* und die Steuerung wird durch bestimmte Ereignisse, wie z.Bsp. dem Ende einer Auftragsbearbeitung ausgelöst. Grundsätzlich werden mit *deterministischen* und *stochastischen* Simulationen zwei Arten definiert, von denen es aber auch Mischformen gibt. Bei den stochastischen Simulationen werden *Zufallswerte* als Betriebsdaten angenommen. Es ensteht hierbei das Problem der Generierung vernünftiger Zufallszahlen. Variablen sind während des gesamten Simulationsprozesses abrufbar und erlauben somit wesentlich aussagekräftigere Feststellungen als dies bei analytischen Methoden möglich ist. Es lassen sich somit beispielsweise *einzelne Komponenten* einer Bewertung zuführen. Auf der anderen Seite stehen die deterministischen Simulationen. Mit ihnen wird das *funktionale* und *Zeitverhalten* realer Systeme genauer abgebildet als bei den stochastischen Simulationen. Der Haupteinsatz dieser Simulation findet sich bei dem Entwurf neuer oder der Optimierung bereits bestehender *Funktionseinheiten* eines bereits existierenden Systems. Detaillierte und deterministisch arbeitende Modelle zu entwerfen ist hier möglich. Es wird hier weiterhin unterschieden in die *Trace*-gesteuerten und *ausführungsgesteuerten* Simulationen. Bei ersterem wird beispielsweise das Verhalten von Speicherhierarchien simuliert. Mit letzterem werden auf dem Simulationsmodell komplette Programme, sogenannte *Benchmark*-Programme, ausgeführt und erlauben somit eine noch realitätsnähere Betrachtung.

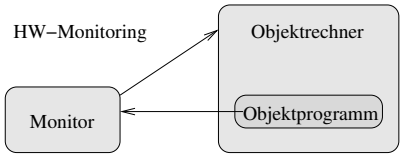
Allgemein läß sich für Simulationen festhalten:

1. Simulationen bieten eine gute Möglichkeit zum Entwurf von Rechnern.
2. Sie erlauben eine Auswahl an Konfigurationsänderungen vorzunehmen.
3. Mit der Zunahme der Komplexität steigen jedoch auch ihre Kosten.

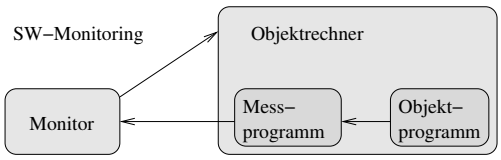
Messungen. An existierenden System können zur Leistungsfeststellung, also zum Erhalt von *Leistungsmaßzahlen*, Messungen durchgeführt werden. Man nennt dies *beobachtende Leistungsbewertung*, deren Ziel es sein kann, entweder einzelne Komponenten oder aber auch komplette Rechenanlagen zu vermessen. In dieser Zielstellung steckt der Wunsch verschiedenene Rechenanlagen *vergleichen* zu können, theoretische Leistungsaussagen zu *validieren*, sowie den Betrieb *überwachen* zu können. Daher auch der Begriff „beobachtende Leistungsbewertung“, im Englischen *monitoring*. Das monitoring kann *ereignisbezogen* und *ereignisunabhängig* sein. Es werden drei Typen von Messungen unterschieden:

1. Monitoring
2. Benchmarking
3. Hybrides Monitoring

Das Monitoring wird weiterhin unterschieden in *Hardwaremonitoring* und *Softwaremonitoring*. Mit ersterem werden *Messfühler* an das Messobjekt angelegt und die Ergebnisse vom Hardwaremonitor aufgesammelt. Vorteilhaft ist hier, dass die Messung den Ablauf des Objektrechners nicht beinflußt, hingegen muss mit einer großen Datenflut umgegangen werden.



Hingegen wird beim Softwaremonitoring die Messung durch das *Objektprogramm* angestoßen und durch ein *Messprogramm* dem Monitor zugeführt. Vorteilhaft hier ist, dass nur wenig Information über den Objektrechner notwendig ist und das die Messprogramme auf vielen Rechnern laufen. Jedoch ist negativ zu bemerken, dass durch den Eingriff in den Programmablauf das dynamische Verhalten des Objektrechners verfälscht wird.



Eine spezielle Problemstellung stellt das Monitoring in verteilten Systemen dar, in denen es keinen Masterrechner gibt, der die Daten über die beteiligten Knoten zur Verfügung stellen kann. Daher werden *Knotenmonitore* zur Überwachung von einzelnen Knoten oder *Verbindungsmonitore* zur Überwachung von Clustern von Rechnern eingesetzt. Dabei stellt die *unterschiedliche* Uhrzeit in den beteiligten Rechnern das Hauptproblem dar, dass durch software- oder hardwaremässige Synchronisation, oder durch den Einsatz lokaler Zeitstempel gelöst werden kann.

Das Monitoring erfährt seine Grenzen in der aus der zunehmenden Integration resultierenden Schwierigkeit auf interne Signale zugreifen zu können. Weiterhin wird durch *virtuellen Speicher* eine eindeutige Zuordnung zum Prozessor erschwert. Der Zugriff auf Daten- oder Befehlscache wird verhindert. RISC-Prozessoren ändern die Reihenfolge der Befehlsausführung und sorgen damit für Probleme beim Monitoring. Schliesslich kann die Steigerung der Taktraten zu einer übermäßigen Datenflut führen.
Unter dem Begriff **Benchmarking** versteht man die *vergleichende Ausführung* ein und desselben (Mess-)programms auf unterschiedlichen Rechnern. Der Rechner, der das Programm am schnellsten ausführt ist demnach der schnellere Rechner. Hier liegt das Problem begraben. In der Praxis ist es häufig vorgekommen, dass Compiler oder Befehlssätze manipuliert wurden, damit sie auf speziellen Benchmarks bessere Leistung erzeugen. Benchmarks ermöglichen es, zusätzliche Informationen über Compiler, Betriebssystem und auch die Genauigkeit der Ergebnisse zu gewinnen. Es gibt vier Typen von Benchmarks:

1. Reale Programme (häufig verwendete Programme)
2. Kernels (kurze kritische Passagen aus realen Programmen)
3. Toy Benchmarks (kleine, einfache, schnell übertragbare Programme)
4. Synthetische Benchmarks (testen spezielle Instruktionen)

Zu den synthetischen Benchmarks gehören die mittlerweile veralteten Dhrystone und Wheatstone Benchmarks, der LINPACK-Benchmark und die SPECint, sowie SPECfp-Benchmarks.

2. Fehlertoleranz

Rechnersysteme werden auch in sensiblen Bereichen eingesetzt, in denen ein Ausfall oder Fehlfunktion eklatante Folgen haben können. Es wird daher nach hoher Verlässlichkeit gefragt. Die *Fehlerintoleranz* beschreibt Methoden zur *Vermeidung* von Fehlverhalten. Die *Fehlertoleranz* versucht dagegen erwünschtes Verhalten durch ein System auch dann zu erhalten, wenn mit einer bestimmen Anzahl von Fehlern gerechnet werden muss. Fehler lassen sich aufgrund verschiedener Eigenschaften klassifizieren. So beschreibt die *Dauer eines Fehlers*, ob ein Fehler permanent oder nur vorrübergehend auftritt. Die *Auswirkung eines Fehlers* erklärt, ob ein Fehler nur auf eine Komponente wirkt ($\rightarrow$ Einfachfehler) oder ob eine Fehlerquelle mehrere Fehler erzeugt ($\rightarrow$ Mehrfachfehler). Finde sich ein *Fehler im Lebenszyklus eines Rechners*, so besteht die Frage ob es sich um einen Konstruktionsfehler (Entwurfs-, Implementierungs- oder Herstellungsfehler) handelt oder ob es ein Betriebsfehler ist, also ob er während des Betriebs(Bedienungs- oder Wartungsfehler) auftritt. Ein weiteres Klassifizierungsmerkmal ist der *Ort des Fehlers*, also ob er in der Hardware oder Software angesiedelt ist Ein weiteres Klassifizierungsmerkmal ist der *Ort des Fehlers*, also ob er in der Hardware oder Software angesiedelt ist.

2.1. Zuverlässigkeit und Verfügbarkeit

Zur Erfassung der Fehlertoleranz werden quantitative Größen relevant. Eine wichtige Größe ist die *Zuverlässigkeit* $R(t)$, die bedingte Wahrscheinlichkeit, dass ein System das Zeitintervall $[0,t)$ überlebt. Es gilt:

$$R(0) = 1 \longrightarrow \text{ Das Gerät ist zum Zeitpunkt 0 funktionstüchtig.}$$
$$R(\infty) = 0 \longrightarrow \text{ Nach genügend langer Zeit fällt das Gerät aus.}$$

Eine weitere Größe ist die *Ausfallrate* $\lambda(t)$, für die im nicht konstanten Fall gilt:

$$\lambda(t) = -\frac{\delta R(t)}{dtR(t)}$$

Ist die Aufallrate λ konstant, so gilt beispielsweise:

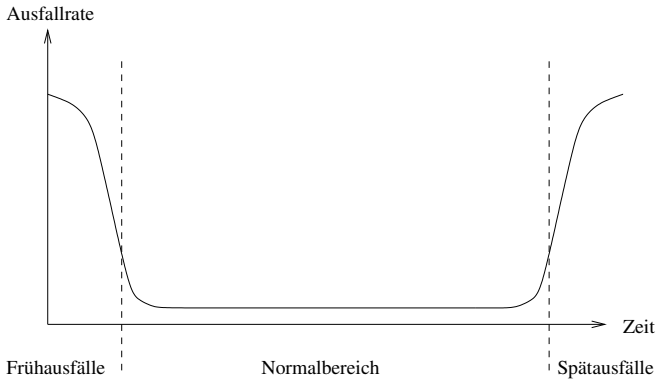
$$R(t) = e^{-\lambda t}$$

Dieses Beispiel nutzen wir für die Beschreibung einer weiteren Größe, nämlich der *mittleren Funktionszeit* (mean team to failure):

$$MTF = \int_{i=0}^{\infty} R(t)dt = \frac{1}{\lambda}$$

Mit der sogenannten Badewannenfunktion lassen sich Frühausfälle und Spätausfälle beschreiben. Eine andere wichtige Größe stellt die *Verfügbarkeit* dar. Für eine nicht-redundante Einheit mit Ausfallrate λ und Reparaturrate μ (Kehrwert der Reparaturzeit *MTR* gilt

$$A = \frac{MTF}{MTR + MTF} = \frac{\lambda}{\lambda + \mu}$$



So lassen sich beispielsweise für die Halbleiterproduktion die Frühausfälle durch die Technik des *burn in* lokalisieren und die entsprechend fehlerhaften Produkte entfernen, was die Lebensdauer von bereitgestellten Halbleitern natürlich erhöht.

2.2. Redundanztechniken

Der Begriff *Redundanz* beschreibt hier das Hinzufügen von Hardware oder Software in ein System um Fehlererkennung und Reaktion darauf zu ermöglichen. Die zentralen Begriffe sind hierfür die *Fehlerdiagnose* und *Fehlerbehandlung*. Die Fehlerdiagnose findet als *Selbstest* im System im laufenden Betrieb statt. Die Fehlerbehandlung setzt sich aus der *Rekonfiguration*, also der Isolierung der fehlerhaften Teile in HW und SW und aus dem *Wiederanlauf* zusammen. Beim Wiederanlauf wird das System nach dem Ausschluss fehlerhafter Teile in der Grundzustand versetzt. Ausserdem sollen *Sicherungspunkte* angelegt werden, die als spätere Rücksetzpunkte dienen können.

Die Redundanz wird unterschieden in *dynamische* und *statische* Redundanz. In der statischen Redundanz sind alle Elemente, die für die Fehlertoleranz notwendig sind, ständig in Betrieb, während bei der dynamischen Redundanz notwendige Elemente on-demand zugeschalten werden.

2.3. Beispiele fehlertoleranter Systeme

Hierzu gelten *Universalrechnersysteme*, wie transaktionsorientierte Datenbankanwendungen, die hohe Zuverlässigkeit erfordern. Es sind dabei alle funktionalen Komponenten, also CPU/ Speicher/Busse/Peripherie statisch und/oder dynamisch redundant. Ebenso sind Lüfter- und Spannungsversorgung redundant. Als anderes Beispiel seien *eingebettete Rechnersysteme* genannt, bei denen geringe Kosten und hohe Kompatibilität im Vordergrund stehen. Hierbei wurden

Mechanismen entwickelt, die den Austausch von Komponenten zur Laufzeit ermöglichen (hot-swap, hot-plugin). Dazu werden zumeist Bussysteme verwendet, die dies unterstützenm wie USB. Ein letztes Beispiel sei mit RAID genannt. RAID steht für *Redundant array of inexpensive disks* und stellt ein Peripheriesystem dar. Der Grundgedanke ist, dass mechanische Komponenten wie hier zum Beispiel Festplatten eine sehr viel geringe *MTF* haben. Die *MTF* sinkt sogar noch weiter, wenn man die Anzahl der Festplatten in einem System erhöht. RAID bietet hierfür eine Lösung. Drei Eigenschaften sind kennzeichnend:

1. Eine Menge *physischer* Platten wird durch das Betriebssystem als *eine logische* Platte aufgefasst.
2. Die Datenspeicherung erfolgt verteilt.
3. Ausnutzen der Redundanz der verfügbaren Kapazitäten um im Fehlerfall die Wahrscheinlichkeit zur Wiederherstellung zu erhöhen.

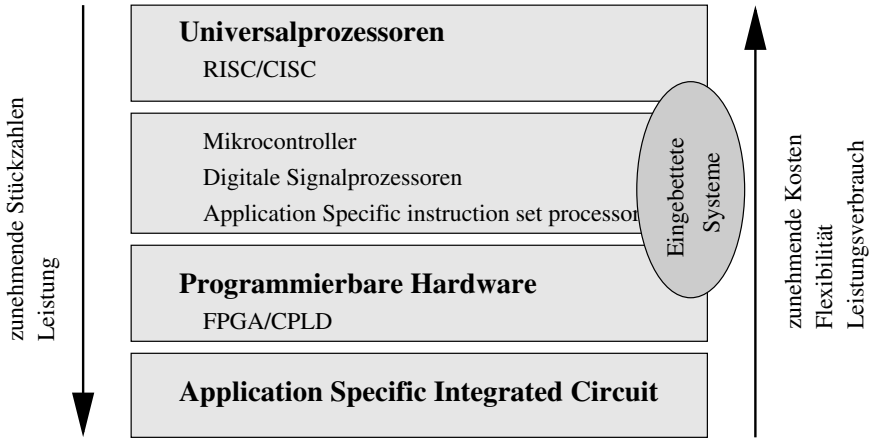
Die Vorteile liegen auf der Hand. Es wird eine wesentlich höhere Zuverlässigkeit ermöglicht und der parallele Zugriff auf die Platten erhöht die Leistungsfähigkeit der E/A.

Teil III.

Zusammenfassung RA2

3. Spezialprozessoren

Im Unterschied zu Universalprozessoren sind Spezialprozessoren weniger flexibel einsetzbar, können dafür jedoch mit einer größeren Leistungsfähigkeit bei speziellen Problemen aufwarten.



Unter den Spezialprozessoren werden die *Eingebetteten Systeme*, die *digitalen Signalprozessoren* (als Spezialfall eingebetteter Systeme), sowie *rekonfigurierbare Hardware* behandelt.

Die Universalprozessoren stellen Hochleistungsprozessoren für allgemeine Computeranwendungen mit PCs oder Workstations. Sie sind nützlich für allgemeine Software und existieren mit komplexen Betriebssystemen. Die Frage ist, ob der PC/Workstation die Lösung für alle Probleme ist?

3.1. Eingebettete Systeme

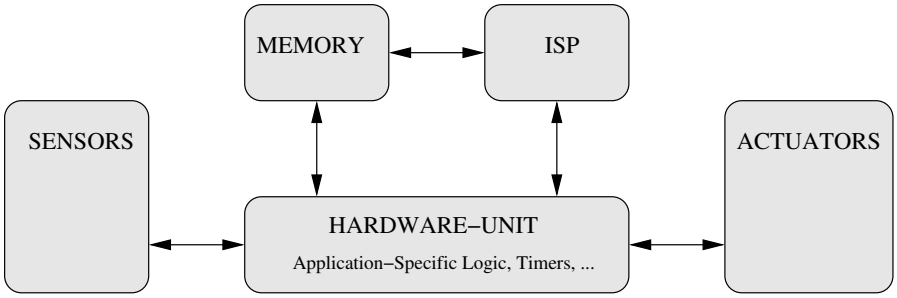
Einen Großteil der Anwendungen digitaler Systeme findet man in elektronischen Alltagsgegenständen verborgen wieder. Sie treten der Umwelt mit *analogen Schnittstellen* entgegen und ihre Rechenfunktionalität wird von außen nicht wahrgenommen. Trotzdem sind in ihrem Inneren leistungsfähige Rechner verborgen: **Eingebettete Systeme**.

3.1.1. Mikrocontroller

Mikrocontroller stellen einen Vertreter eingebetteter Systeme dar. Sie sind vor allem für *steuerungsdominante* Anwendungen interessant. Sie sind darauf ausgerichtet besonders *Kontrollfluss-orientierten* Code auszuführen, weniger jedoch arithmetische Operationen. Weitere Merkmale sind ein *geringer Datendurchsatz*, *Multitasking*, *geringe Kosten* und dementsprechend *hohe Stückzahlen*. Mikrocontroller arbeiten auf *niedrigen Bitbreiten*, üblich sind 8Bit. Es werden viele

Bit und Logikoperationen ausgeführt und Register werden oft als RAM realisiert. *Kontextwechsel* werden durch Zeigeroperationen durchgeführt, woraus eine minimale (durch Unterbrechungen hervorgerufene) *Latenz* resultiert. Mikrocontroller weisen die höchsten Stückzahlen auf. Meist sind periphere Einheiten direkt auf dem Mikrocontroller integriert, etwa A/D, D/A-Wandler und dergleichen.

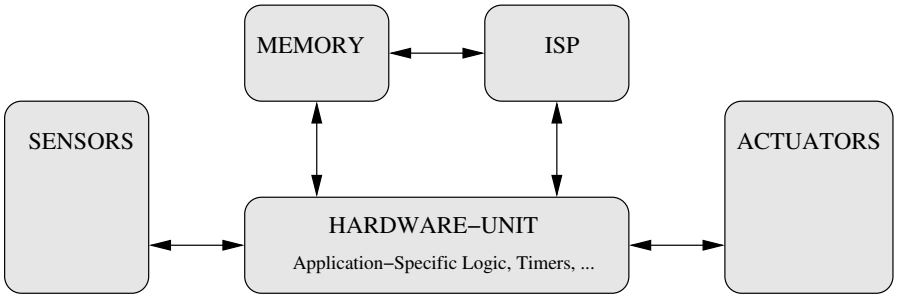
Einfache, kostengünstige Mikrocontroller (µC) arbeiten mit 4/8Bit-Prozessoren und die Codegröße nimmt den meisten Platz auf der Chipfläche ein und dominiert damit die Kosten des µC. Die Anforderungen an seine Rechenleistung sind eher gering. Die Klasse der Hochleistungs-µC hingegen arbeitet mit 16/32/64 Bit Prozessoren und wird in steuerungsrelevanten Anwendungen, die zusätzlich hohe Datenraten erfordern, eingesetzt, z.Bsp. in der Automobiltechnik oder in der Telekommunikation. An diese µC werden hohe Berechnungsanforderungen gestellt, etwa in der Regelungstechnik oder der Signalverarbeitung. Diese µC bilden oft den Kern eines *System-On-A-Chip*.



Wie auf der Abbildung ersichtlich stellen eingebettete Systeme die Möglichkeit dar, *anwendungsspezifische* Architekturen auch für hohe Leistung bereitzustellen.

3.1.2. Bestandteile eingebetteter Systeme

Allgemein gilt für eingebettete Systeme, dass sie in größere Umgebungen integriert sind. Mit der Definition von Ernst: „... ein Computersystem, was in ein technisches System eingebettet ist, jedoch selbst nicht als Computer erscheint.“ Das wichtigste Merkmal ist, dass diese Systeme in *analoge Umgebungen* eingebettet sind und mit der Umwelt über Sensoren Kontakt aufnehmen. Diese Systeme werden *anwendungsspezifisch* entworfen und führen innerhalb des Gesamtsystems genau definierte Funktionen aus. Als Prozessoren kommen in eingebetteten Systemen Mikrocontroller, Mikroprozessoren oder eigens für einen bestimmten Zweck entworfene Schaltkreise zum Einsatz.



Eingebettete Systeme spielen eine *wichtige* Rolle in der industriellen Automation, bei Werkzeugmaschinen, in der Robotik und der Unterhaltungsindustrie. In Europa, speziell in Deutschland überwiegt der Einsatz von solchen Systemen.

3.1.3. Entwurf eingebetteter Systeme

Bei dem Entwurf dieser Systeme spielen eine Reihe von Faktoren eine wichtige Rolle.

Komplexe Funktionalität. Mikroprozessoren erreichen mittlerweile sehr hohe Funktionalität und stellen somit sehr komplexe Gebilde dar. Heutige Systeme enthalten mehrere 10 Millionen Transistoren und bearbeiten mehrere 10 000 Zeilen Hochsprachencode. Es werden *Methoden aus der Softwaretechnik* auch für den Entwurf von eingebetteten Systemen angewendet.

Benutzerschnittstellen. Mehrfach-Menus, kleine Anzeigeflächen bestimmen hier. Scrollende Karten bei GPS-Geräten wären ein Beispiel für die wachsende Komplexität der Benutzerinterfaces.

Realzeitoperation. Das Nicht-Einhalten der zeitlichen Randbedingungen kann hier ein *Versagen* darstellen (*harte Anforderungen*), oder aber lediglich als *Leistungseinbruch* betrachtet werden (*weiche Anforderungen*). Eine besondere Herausforderung stellt das *deadline-driven programming* dar, also die Programmierung von Anwendungen mit festem Ausführungszeitrahmen. Dies muß trotz unvorhersagbaren Verhaltens (Caches etc.) eingehalten werden. Ein weiterer Punkt ist die *Multiratenverarbeitung*, also die gleichzeitige Verarbeitung von Daten verschiedener Anwendungen. Hierfür müssen dann Schedulingmechanismen o.ä. von Betriebssystemen herangezogen werden.

Niedriger Energieverbrauch. Die Kapazität der Spannungsversorgung ist durch die Faktoren *Gewicht, Kosten* oder *Rauschen* beschränkt. Die Lebenszeit von Batterie/Akku soll möglichst lang sein, ebenso die Einsatzzeit. Diese Faktoren wirken sich natürlich auf die Kosten aus.

Kosten. Es handelt sich bei eingebetteten Systemen häufig um Massenware, die per se niedrige Kosten erforderlich macht. Die Kosten werden bestimmt durch die Art des eingebetteten Prozessors, Speichers und E/A-Einheiten.

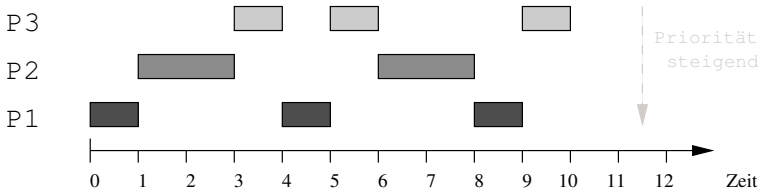
Entwicklungszeit. Diese sogenannte *time to market* wird mit 6 Monaten angegeben. Es wird HW/SW-Codedesign angewendet.

Beispiel: Das *rate-monotonic scheduling* stellt eine Strategie für die Ablaufplanung eines Echtzeit-Systems dar. Sie ist *statisch*, da den Prozessen feste Prioritäten zugewiesen werden. Es wird von einer einfachen Struktur ausgegangen, die folgende Grundlagen hat:

1. Alle Prozesse laufen *periodisch* auf *einer* CPU.

2. Die Kontextwechselzeit wird ignoriert.
3. Zwischen den Prozessen herrscht keinerlei Datenabhängigkeit.
4. Die Ausführungszeit der einzelnen Prozesse bleibt konstant.
5. Alle *deadlines* befinden sich am Ende einer Periode.
6. Der Prozess mit der höchsten Priorität wird zur Ausführung ausgewählt.

	Ausführungszeit	Periode
P1	1	4
P2	2	6
P3	3	12

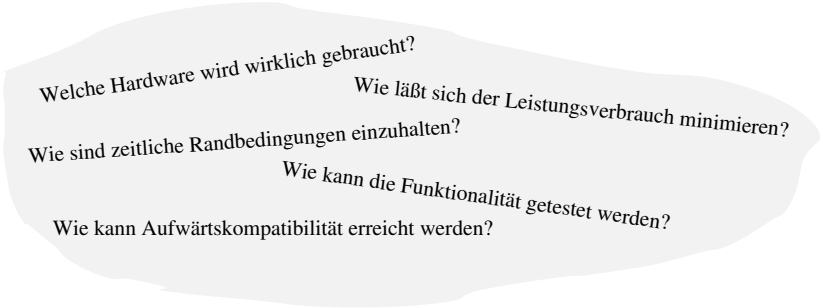


Bewertung: Eine relativ einfache Strategie ist optimal. Die Prioritäten werden nach Größe der Perioden vergeben. Optimal bedeutet, dass bei einer möglichst großen CPU-Auslastung die *deadlines* der einzelnen Prozesse dennoch eingehalten werden können. Jedoch gibt es keine 100-prozentige CPU-Auslastung, daher sind *dynamische* Strategien interessant.

Auswahl von Prozessoren für eingebettete Systeme. Viele *Universalprozessoren* werden nicht für universelle Aufgaben, sondern in eingebetteten Systemen eingesetzt. Diese Lösung ist möglich, kann jedoch sehr kostenintensiv werden. Für den Einsatz in *steuerungsdominanten* Anwendungen sind *Mikrocontroller* prädestiniert. Für spezielle Aufgaben aus der *digitalen Signalverarbeitung* werden *digitale Signalprozessoren eingesetzt*. Hingegen werden *anwendungsspezifische Prozessoren*, sogenannte ASIPs (*application specific instruction processors*), für klar umrissene Anwendungen entworfen, für die auf dem Markt keine Prozessorlösung vorhanden ist und für die ein eigens entwickelter Befehlssatz notwendig ist.

Was spricht also für eine Anwendung von Mikroprozessoren und was für kundenspezifische Lösungen? Für die *Mikroprozessoren* spricht die höhere Effizienz, die Möglichkeit zur Schaffung von Produktfamilien, wie sie in der *Generizität* der Mikroprozessoren begründet liegt und weiterhin die Geschwindigkeitsvorteile, die mit dem *Mikroprozessorparadoxon* erklärt werden können. Es besagt, dass trotz des höheren Aufwandes an Logik, der zur Erfüllung einer Funktion aufgebracht werden muß, die Geschwindigkeit besser ist als in der kundenspezifischen Lösung. Dies resultiert aus der optimierten Architektur (RISC, Parallelverarbeitung), den großen Entwicklungsteams um diese Prozessoren sowie des Einsatzes neuester VLSI−Technologien. Für *kundenspezifische Lösungen* spricht der niedrigere Leistungsverbrauch, die geringere Größe, und die Möglichkeit spezielle Anforderungen direkt zu realisieren, etwa eine integrierte Signalerfassung (OPTO−ASIC).

Beim Entwurf stellen sich unter anderem diese Fragestellungen:



Den in diesen Fragen steckenden Problemstellungen begegnet man mit einer geeigneten *Entwurfsmethodik*. Warum ist dies so? Entwurfsmethodik bietet die Möglichkeit einer *Bewertung*, etwa ob die Anforderungen, also *Funktionalität, Userinterfaces, Herstellungskosten, Leistungsverbrauch, ...* erfüllt werden. Sie ermöglicht die Verwendung von `CAE/CAD`-Werkzeugen, also die Aufteilung des gesamten Entwicklungsprozesses in überschaubare Teilschritte. Weiterhin ist mit der Entwurfsmethode auch die *Kommunikation* im Entwicklerteam definiert.

Der Entwurfsprozess wird in *Abstraktionsebenen* unterteilt, die eine schrittweise Verfeinerung während eines *top-down*-Prozesses und die Überprüfung der Randbedingungen durch einen folgenden *bottom-up*-Prozess ermöglichen. Wir gehen nun kurz auf die einzelnen Ebenen ein und nennen wesentliche Bestandteile und Mechanismen, die in jeder Ebene betrachtet werden sollten.

Die **Anforderungsebene** stellt zunächst eine informelle Beschreibung der Kundenwünsche dar. Dann werden *funktionale* von *nicht-funktionalen* Anforderungen getrennt. Ersteres beschreibt die Ausgabe als Funktion der Eingabe, letzteres hingegen

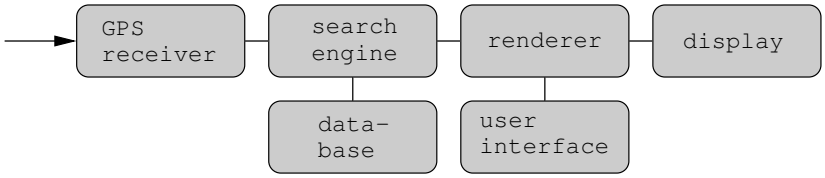
- wie lang es dauert, bis die Ausgabe verfügbar ist,
- Größe, Gewicht und Kosten,
- Leistungsverbrauch,
- Zuverlässigkeit.

In der Anforderungsphase wird weiterhin ein Modell, oder eine Attrappe des Endprodukts erstellt, um dem Kunden einen Eindruck von Gewicht und Dimensionen des Gerätes zu vermitteln. Schliesslich wird ein *Pflichtenheft* angelegt, dass die Anforderungen grob definiert und als Ausgangspunkt für den weiteren *top-down*-Prozess verwendet wird.

Auf der folgenden **Spezifikationsebene** wird eine *detaillierte* Beschreibung des Systems vorgenommen, die als ein Vertrag zwischen dem Kunden und Systemarchitekt verstanden wird.

Sie sollte ausformuliert sein, oder als mathematische Beschreibung, etwa durch UML−Diagramme, Kontroll- und Datenflussgraphen (CDFG), Statecharts oder Automaten vorliegen.

Nun wird auf der **Architekturebene** der Plan für die gesamte Systemarchitektur erstellt. Während die Spezifikation die Frage nach dem „*was*“ beantwortet hat, wird hier auf das „*wie*“ eingegangen. Dabei werden die *wichtigsten Operationen* in einem Blockdiagramm erfasst. Beispielsweise könnte für ein GPS−System das Blockdiagramm wie folgt aussehen:

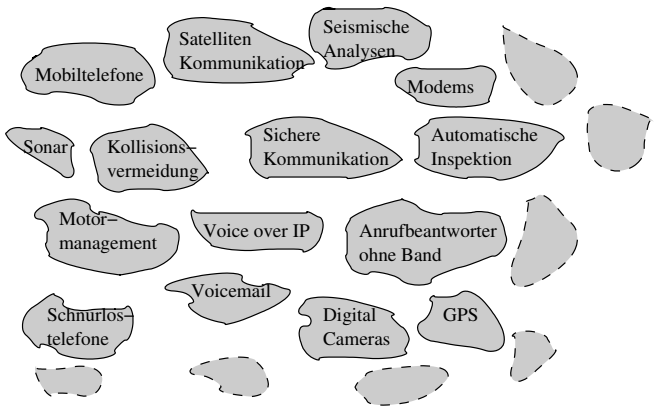


Im nächsten Schritt auf der Architekturebene wird das Blockdiagramm jeweil für die *Hardware* und für die *Software* spezialisiert. Neben diesen *funktionalen* Beschreibungen müssen auch *nicht-funktionale* Randbedingungen auf der Architekturebene beachtet werden. Diese basieren meist auf *Erfahrung* und aus dem *bottom-up*-Prozess, sowie aus Simulationsergebnissen.

Die **Komponentenentwurfsebene** dient zur Abschätzung der Möglichkeiten, ob man bereits *verfügbare Komponenten* einbinden kann, oder ob diese aus vorhergehenden Entwürfen entnehmbar sind. Hier wird dann auch die Schaffung einer neuen Komponente realisiert, falls dies notwendig ist.

Schliesslich wird auf der **Systemintegrationsebene** eine Zusammenführung aller Komponenten vorgenommen. Hier werden dann viele Fehler erstmals sichtbar.

3.2. Signalprozessoren



Die wichtigste Klasse eingebetteter Systeme stellen die sogenannten digitalen Signalprozessoren (DSPs) dar. Diese könnte man als „*elektronische Systeme zur digitalen Signalverarbeitung*“ definieren.

Wesentliches Charakteristikum von digitaler Signalverarbeitung ist die Anwendung *mathematischer Operationen* auf *digital-repräsentierte*, ursprünglich analoge, Signale. Diese Signale

werden als Folgen von Abtastwerten, sogenannten *Samples*, aufgenommen. Man erhält sie aus *physischen* Signalen durch die Anwendung von *Transducern*[1] oder AD−Wandlern.

Das Aufzeichnen von analogen Signalen ist mit einem *Verlust von Information* behaftet, da das Aufzeichnen wie ein Filter wirkt. Demhingegen hat die *digitale* Aufzeichnung, also die Repräsentation der Signale durch Zahlen, den Vorteil einer *stabilen* Speicherung. Man kann ohne Verzerrungen speichern, übertragen, wiedergeben und eine *(De-)Kodierung* wird simplifiziert. Auch bei niedrigen Frequenzen sind analoge Filter zur Speicherung schwierig zu realisieren, digitale dagegen recht einfach.

Eingesetzte Algorithmen drehen sich um *Filterung im Frequenzbereich*, *Frequenz-Zeit-Transformation* und *Korrelation*. Wesentliche Aufgaben umfassen *iterative, numerische* Berechnungen mit Beachtung numerischer Präzision, eine durch Arrayzugriffe hohe Speicherbandbreite und natürlich Realzeit-Verarbeitung. DSPs müssen ihre Aufgaben effizient ausführen bei gleichzeitiger Minimierung von Kosten, Leistungsverbrauch, Speicherbedarf und Entwicklungszeit.

Da DSP−intensive Aufgaben heutzutage den Flaschenhals in vielen Rechneranwendungen darstellen und unter Berücksichtigung des Kostenfaktors sind *general-purpose processors* (GPPs) nicht zum Einsatz in diesem Bereich geeignet. DSPs werden gewöhnlich für *ein* Programm entwickelt, daher fällt die Komplexität für das zugehörige Betriebssystem relativ niedrig aus. Sie müssen häufig *harten Realzeitanforderungen* genügen und sie verarbeiten einen potentiell unendlichen Datenstrom. Trotz der Kostenbeschränkungen finden in einigen Bereichen auch GPPszunehmend Betätigungsfelder, etwa Sprach- und Audiodatenkompression, Filterung, Modulation, Spracherkennung oder Signalsynthese und v. m. Mit dem LINGO wurde ein graphisches Format zur Darstellung von DSP−bezogenen Formeln geschaffen.

3.2.1. Algorithmen für digitale Signalprozessoren

In digitalen Signalprozessoren sind Algorithmen fest verdrahtete, oder auch Firmwareimplementierungen zur Berechnung von *Standardfunktionen*, ohne die Multiplikation. Wichtige Algorithmen sind *CORDIC*- und *Bit*-Algorithmen. Ein wesentliches Ziel war die *Vermeidung von Multiplikationen* um kostenträchtige Chipfläche einzusparen. Heute werden jedoch immer mehr schnelle Multiplizierer, etwa der Radix−4− Multiplizierer[2] eingesetzt. Für die Zukunft ist jedoch wieder mit einem Rückgang der Multiplikationen zugunsten *massiv-paralleler* Signalprozessoren auf *einem* Chip zu rechnen.

Zu den angesprochenen *Standardfunktionen* gehören trigonometrische Funktionen, Logarithmus- und Exponentialfunktionen, Quadratwurzelbildung und Division/Multiplikation. Die Berechnung ist grundsätzlich mit drei Verfahren realisierbar:

1. Reihenentwicklung
2. Interpolationsverfahren
3. **Konvergenzverfahren**

[1] Gerät, das Töne, Temperaturen, Druck, Licht oder andere Signale von/nach elektronische Signale konvertiert.
[2] Vorlesung, Rechnerarchitektur I.

Der CORDIC-Algorithmus stellt ein solches Konvergenzverfahren dar, dass auf dem Prinzip der *Koordinatentransformation* aufbaut. Die Idee ist, aufwendige Multiplikationen, wie sie in den Reihenentwicklungen vorkommen, durch *einfache* Operationen, wie Addition, Schiebeoperationen und Vergleiche zu ersetzen. CORDIC-Prozessoren sollen eine Möglichkeit schaffen mit ein und derselben Hardwarestruktur eine Menge von Standardfunktionen berechnen zu können.

Der Ansatz des CORDIC-Verfahrens ist die Tatsache, dass man bei der Drehung des Vektors $\binom{1}{0}$ um den Winkel θ den Vektor $\binom{\cos\theta}{\sin\theta}$ erhält – also todo: Bildchen malen den Sinus und den Cosinus des Winkels θ. Diese Drehung kann man durch Teildrehungen nachvollziehen, deren Berechnung so gewählt ist, dass sie ohne Multiplikation bzw. nur mit Tabellenabfragen und Schiebeoperationen auskommt.

Die allgemeine Beziehung für die Drehung eines Vektor $\binom{x}{y}$ um einen Winkel θ wird durch die Gleichung 3.1 beschrieben. Zur Vereinfachung beschreibt man diese mit nur einer Winkelfunktion; Gleichung 3.2.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.1}$$

$$= \frac{1}{\sqrt{1+\tan^2\theta}} \cdot \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.2}$$

Die Drehung um den Winkel θ zerlegt man in Drehungen um kleinere, fest vorgegebene Winkel α_i ($i = 0, \ldots, n-1$), so dass θ eine Linearkombination der α_i ist; Gleichung 3.3. Man beschreibt also die Drehung schrittweise um fest vorgegebene Winkel, wobei man in mathematisch negativer Richtung dreht, wenn man den Winkel θ überschritten hat. Dies wird durch den Vorfaktor σ_i gesteuert. Die α_i wählt man so, dass sich $\tan\alpha_i = 2^{-i}$ ergibt.

$$\theta = \sum_{i=0}^{n-1} \sigma_i \cdot \alpha_i \qquad \text{wobei} \qquad \sigma_i \in \{-1, 1\} \tag{3.3}$$

Für zwei Winkel β und γ gilt:

$$\begin{aligned} \begin{pmatrix} \cos(\beta+\gamma) & -\sin(\beta+\gamma) \\ \sin(\beta+\gamma) & \cos(\beta+\gamma) \end{pmatrix} &= \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} \cos\gamma & -\sin\gamma \\ \sin\gamma & \cos\gamma \end{pmatrix} \\ &= \frac{1}{\sqrt{1+\tan^2\beta}} \cdot \begin{pmatrix} 1 & -\tan\beta \\ \tan\beta & 1 \end{pmatrix} \\ &\quad \cdot \frac{1}{\sqrt{1+\tan^2\gamma}} \cdot \begin{pmatrix} 1 & -\tan\gamma \\ \tan\gamma & 1 \end{pmatrix} \end{aligned}$$

Verallgemeinert gilt also für die Drehung um einen zusammengesetzten Winkel $\theta = \sum_{i=0}^{n-1} \sigma_i\alpha_i$ die Gleichung 3.4. Da alle $\sigma_i \in \{-1, 1\}$ sind und der Tangens symmetrisch zum Koordinatenursprung ist, gilt $\tan(\sigma_i\alpha_i) = \sigma_i \tan\alpha_i$, und da die α_i so gewählt sind, dass $\tan\alpha = 2^{-i}$ gilt, kann man die Gleichung 3.4 zu Gleichung 3.5 umformen.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+\tan^2(\sigma_i \alpha_i)}} \cdot \begin{pmatrix} 1 & -\tan(\sigma_i \alpha_i) \\ \tan(\sigma_i \alpha_i) & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.4}$$

$$= \underbrace{\prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}}}_{=k} \cdot \prod_{i=0}^{n-1} \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \tag{3.5}$$

Den Vorfaktor k kann man im Voraus berechnen und als Konstante abspeichern, Da die Teilwinkel α_i fest gewählt sind. Die Berechnung kann man jetzt mit folgender Iterationsvorschrift beschreiben:

$$x_{i+1} = x_i - \sigma_i 2^{-i} y_i$$
$$y_{i+1} = y_i + \sigma_i 2^{-i} x_i$$

Um die Drehrichtung – also den Wert von σ_i – zu bestimmen, führt man eine Hilfsvariable z_i (Gleichung 3.6) ein, die den noch zu drehenden Winkel misst. Ist dieser kleiner null – wurde also zu weit gedreht –, muss die kommende Drehung in die mathematisch negativer Richtung erfolgen – $\sigma_i = -1$ –, andernfalls ist $\sigma_i = 1$.

$$z_0 = \theta \tag{3.6}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i = z_i - \sigma_i \arctan(2^{-i}) \tag{3.7}$$

Das Vorgehen, bei dem man mit dem Vektor $\binom{1}{0}$ beginnt und die Hilfsvariable z_i auf null bringt, nennt man **Rotationsmodus**. Wenn man mit einem Vektor $\binom{x}{y}$ beginnt und diesen „auf die x-Achse dreht“ – formal bedeutet das, dass die y-Komponente auf null gebracht wird –, kann man so die Länge des Vektors in der x-Komponente und den Winkel zur x-Achse in der z-Komponente bestimmen. Dieser Ablauf wird **Vektormodus** genannt.

todo: Was über die Verallgemeinung auf lineares und hyperbolisches Koordinatensystem schreiben.

Die Bitalgorithmen stellen eine weitere Klasse von Konvergenzalgorithmen dar. Sie bauen auf der Erkenntnis auf, dass die Abarbeitung häufig auf einzelne Bits zugreift. Die grundlegende Strategie ist, geeignete Iterationsvorschriften für die Iterationswerte x_i und y_i zu finden, sodass eine ebenfalls zu definierende *charakteristische Funktion* Φ für alle Iterationspaare (x_i, y_i) den gleichen Funktionswert liefert:

$$\Phi(x_{i+1}, y_{i+1}) = \Phi(x_i, y_i)$$

Vergleich. Vorteilhaft beim `CORDIC`-Algorithmus ist die einheitliche Berechnungsvorschrift für eine Vielzahl von Funktionen, die eine reguläre und einfache Hardwarestruktur ermöglicht. Nachteilig ist, dass die Berechnung nicht-trigonometrischer Funktionen oftmals eine zweimalige

Anwendung des Algorithmus' benötigt. Der Vorteil von Bitalgorithmen ist, dass die Berechnungsvorschriften mit n-Iterationen auskommen und er ist somit bei der Berechnung von Exponential-, Logarithmus- oder Wurzelfunktion dem CORDIC–Algorithmus überlegen. Die Einheitlichkeit der Hardwarestruktur ist jedoch bei ihm nicht gegeben.

todo: Genauer erklären, warum Bitalgorithmen uneinheitlich sind. Eigentlich ist es auch immer die gleiche Iterationsvorschrift, bloß mit anderen Faktoren in der Iterationsvorschrift. Oder?

todo: Der CORDIC ist auch nicht so einheitlich, wenn man alle Modi berücksichtigt. Dann ergibt sich nämlich auch ein weiterer Nachteil, dass eine Multipikation mit dem Korrekturfaktor erforderlich ist.
CORDIRC ist für trigonometrische Funktionen gut. Bitalgorithmen für Division, Multiplikation, Wurzel- und Logarithmusfunktion.

3.3. Rekonfigurierbare Hardware

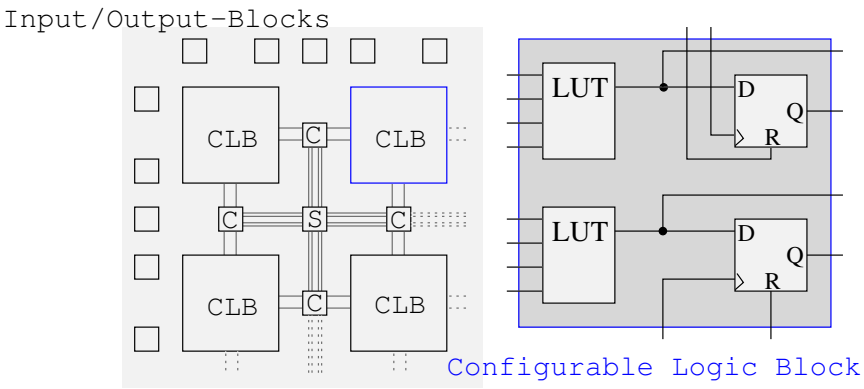
Der Begriff der *Rekonfigurierbarkeit* bedeutet, dass Hardware sowohl hinsichtlich der Verbindungen, als auch der Funktionalität *umprogrammierbar* ist. Die bekanntesteten Vertreter stellen die FPGA (*field programmable gate arrays*) dar. FPGAs sind zwischen den ASICs und den GPPs einzuordnen.

Vergleich. Während GPPs durch ein breites Sortiment unterschiedlicher Befehle gekennzeichnet sind, die nahezu jede logische und mathematische Operation ausführen können, so gilt für ASIC, dass sie nur bestimmte Probleme lösen. Dies erledigen sie jedoch schneller, kostengünstiger, leistungsärmer und auch mit geringerem Platzbedarf als jeder Universalprozessor. Die Herstellung und Veränderung besitzt bei ASICs *hohe Fixkosten* und der entsprechende Aufwand muß sich bereits bei geringen Stückzahlen amortisieren. Der FPGA nimmt zwar Instruktionen entgegen, jedoch nur zur Änderung seiner internen Konfiguration. Instruktionen sind somit *Konfigurationsdaten* und nach einer Änderung gilt für den FPGA der selbe Aufwand, wie für einen ASIC.

3.3.1. Aufbau von FPGAs

Obwohl es von Hersteller zu Hersteller Unterschiede im Design der FPGAs gibt, so läßt sich eine Grundstruktur festhalten. Ein FPGA besteht aus

1. konfigurierbaren Verdrahtungsressourcen,
2. regelmäßigen konfigurierbaren Logikblöcken (CLB) und
3. spezielle I/O-Blöcke für die Ein- und Ausgabe.



Die CLBs werden durch *Schaltmatrizen* gesteuert. Diese Matrizen können mittels *fuse/antifuse*-Techniken oder SRAM/EPROM-Speicherzellen geschalten werden. An den Leitungskreuzungen stehen also Schalter zur Verfügung und die konfigurierbaren Logikblöcke können *fein* oder grob-granular bezüglich der Funktionalität sein. Das Haupteinsatzgebiet von FPGAs besteht im *rapid prototyping*, also dem Test von ASICs.

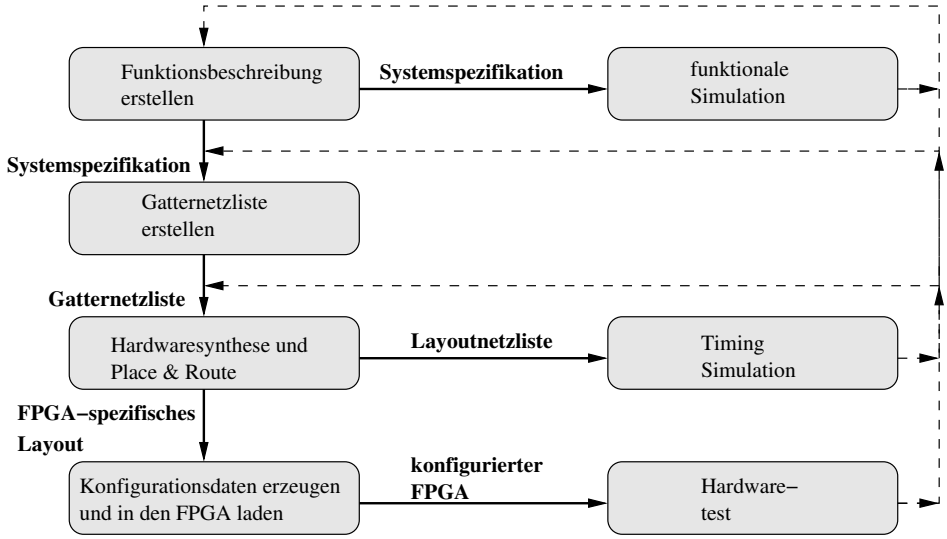
3.3.2. Konfiguration von Logikblöcken und Verbindungen

todo: Unmaskieren auf Folie 127, kap1/03. Vorteil der SRAM-Technik ist, dass mit ihr der Chip *beliebig oft* programmierbar ist, womit der Chip für viele Entwurfssituationen *wiederverwendbar* wird. Ein weiterer Pluspunkt ist, dass der Baustein während der Anwendung umprogrammierbar ist, also *dynamisch rekonfigurierbar* ist. Ein Nachteil dieser Technik ist die *Flüchtigkeit* des Speichers. Es muss nachjedem Neustart der Speicher neu geladen werden. Ausserdem werden mit der SRAM-Technik pro Speicherzelle 5 Transistoren benötigt, was man in der Folge als *flächenintensiv* bezeichnen kann.

Bei der fuse/antifuse-Technik werden hochohmige Kontakte durch eine überhöhte Programmierspannung *irreversibel* niederohmig gemacht. Die Irreversibilität stellt auch einen Nachteil neben den benötigten großen Transistoren zur Bereitstellung der hohen Programmierspannung dar. Positiv ist der dennoch geringe Platzbedarf und der niedrige Widerstand beim Schalten (nachdem die Isolationsschicht im Kreuzungsgebiet gezielt zerstört wird).

3.3.3. Entwurfszyklus von FPGAs

FPGAs werden in einem *top-down*-Entwurfszyklus designed.



Für die **Systemspezifikation** gibt es zwei Methoden. Zum einen eine hardwarenahe *Strukturbeschreibung* und zum anderen eine abstraktere, funktionalere *Verhaltensbeschreibung*. Bei der Strukturbeschreibung wird die Funktionalität in Form von einzelnen Modulen spezifiziert, die durch Signale verbunden sind. Diese Module werden vom Hersteller der Zielhardware in Bibliotheken bereitgestellt. Neben einfachen logischen Gattern, Speicherzellen werden auch komplexe Funktionen (arithmetische Funktionen, Speicherschnittstellen, I/O-Bus-Interfaces) zur Verfügung gestellt. Zur Beschreibung kann entweder ein Schaltplan-Editor verwendet werden um diese zu zeichnen, oder man erstellt die Strukturbeschreibung durch eine Systementwurfsprache wie `VHDL`, `Verilog` oder `SystemC`. Die Strukturbeschreibung ist *hardwarenah*, daher wird vom Entwickler ein hohes Maß an Hardwarekenntnissen der Zielarchitektur verlangt. Demhingegen wird bei der *Verhaltensbeschreibung* vom Entwickler keine spezielle Kenntnis der Zielhardware erwartet. Bei der **Hardwaresynthese** wird die Funktionspezifikation unter Verwendung von `EDA`−Werkzeugen (*electronic design automation*) in eine Gatternetzliste umgewandelt. Dabei wird wiederum auf Bibliothekselemente des Hardwareherstellers zurückgegriffen. Nun kann mit herstellerspezifischen Werkzeugen aus der Gatternetzliste eine Layoutnetzliste erstellt werden. Durch **Verifikation**, also durch Simulation und der Programmierung geeigneter Testumgebungen werden möglicherweise Fehler sichtbar und es müssen unter Umständen einzelne Syntheseschritte wiederholt werden.

3.3.4. Dynamische Rekonfigurierung

Zur Zeit wird dynamische Rekonfigurierung hauptsächlich in der Form des *rapid prototyping* angewendet. Allgemein ist die Möglichkeit der dynamischen Umprogrammierung von Hardware insofern interessant, als dass man versucht, die Funktionalität eines `GPP` mit beschränkten Hardwareressourcen zu erreichen. Dazu gibt es einen Forschungsbereich für sogenannte `DISC`-Prozessoren (*dynamic instruction set computers*).

4. HW/SW-Codesign

Unter HW/SW-Codesign versteht man den Entwurf eines digitalen Systems, in dem Teile in Software *und* Hardware spezifiziert und abgewickelt werden. Dieser Denkansatz ensteht aus dem Wunsch eine mit dem Softwareentwickler abgestimmte Entwicklung von Hardware zu gewährleisten und die sowohl die Hardware, als auch die Software *parallel* zu erstellen. Der Wert eines Rechensystems wird durch folgende Eigenschaften bestimmt:

1. Leistung
2. Kosten
3. Einfachheit der Programmierung
4. Möglichkeit der Reprogrammierung

Das HW/SW-Codesign zielt darauf ab diese Eigenschaften bereits auf der *Systemebene* zu verbessern. Dies soll durch *Interaktion* zu *Synergieeffekten* zwischen Hardware und Software führen. Das Codesign kann für *eingebettete Systeme*, *ISA-Architekturen* oder *rekonfigurierbare Hardware* angewendet werden, verläuft durch die verschiedenen Anforderungen jedoch unterschiedlich.

4.1. Anwendungsorte

Eingebettete Systeme verfolgen verschiedene Ansätze. Der *Null-Kosten*-Ansatz läßt kaum Spielraum für das Codesign. Der Null-Leistungsverbrauch-Ansatz für bspw. batteriebetriebene Anwendungen ist auch noch nicht durch Codesign vertreten. Die *Null-Verzögerung*-Anwendungen hingegen sind sehr interessant aus Sicht des HW/SW-Codesigns.

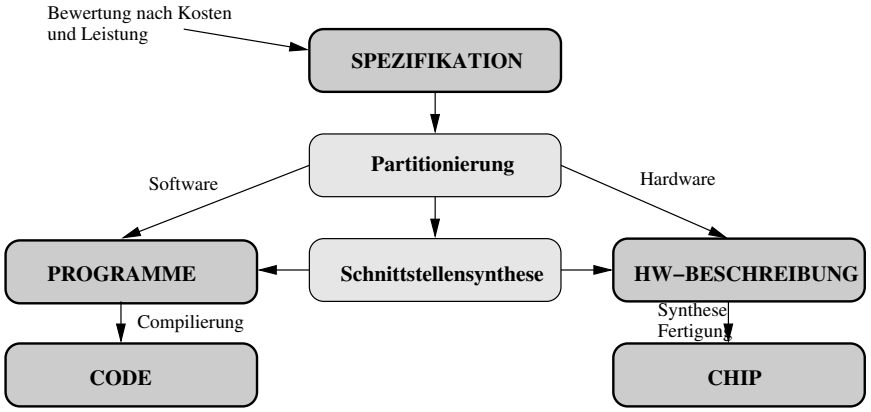
ISA-Architektur (*instruction set architecture*) ist für hohe Rechenleistung konzipiert. Es ist hier im Mittelpunkt der Betrachtung die *Wechselwirkung* zwischen Compilertechnologie und Instruktionssatz. Man könnte die Compilerentwicklung als eine frühe Form des Codesigns ansehen, jedoch stellt das Codesign höhere Anforderungen an den Compiler: Es muß eine Variation des Zielbefehls- und Registersatzes (*retargetable*) ermöglicht werden und weiterhin muß der Compiler *hochoptimierend* sein, um die Leistungszielsetzung zu erfüllen.

Rekonfigurierbare Hardware hat ebenfalls hohe Rechenleistung im Sinn. Sie ist quasi prädestiniert für das HW/SW-Codesign und wurde vor allem dafür entworfen. Operative Eigenschaften rekonfigurierbarer Hardware wird unterschieden in *statische*, während einer Anwendung unabänderlicher Hardware und in *dynamische* Hardware, die zur Laufzeit rekonfigurierbar ist. Es kommen zwei Einsatzgebiete in Frage:

1. HW-Beschleuniger
2. Rapid-Prototyping

Für *HW-Beschleuniger* besteht das Codesign aus *Programmanalyse, Laufzeitmessungen,* Umsetzung *zeitkritischer* Berechnungen in Hardware. Zeitunkritische Teile verbleiben in der Software. Das Codesign selbst besteht aus viel Handarbeit, also dem Aufspüren der zeitkritischen Anteile. Der Geschwindigkeitsgewinn wird durch *sukkzessive* Verbesserung erreicht. Das *Rapid-Prototyping* ist meist rechnergestützt und dient zur *Validierung von Hardware*, der *prototypischen Realisierung* von Hardware oder besitzt seine Zielstellung im Bereich einer Einzelfallapplikation.

4.2. Durchführung von HW/SW-Codesign



Der Designprozess umfaßt mehrere Schritte. Zunächst wird während der *Modellierungs- und Partitionierungsphase* von einem Konzept ausgegangen und mit der permanenten Verfeinerung der Spezifikation eine *Hardware- und Softwaremodellbildung* geschaffen, die Grundlage für die folgende *Validierung* ist. Damit ist das Erreichen eines bestimmten Grades an *Vertrauen* gemeint, indem auf logische Korrektheit und Einhalten der Randbedingungen geprüft wird. Abschliessend erfolgt in der *Implementierungsphase* die technische Umsetzung des Designs in Form einer *Übersetzung* für den Softwareteil und der *Synthese* für den Hardwareteil.

4.2.1. Erfassen und Simulieren

Die grobe Struktur des zu entwerfenden Systems ist zu spezifizieren und dann zu *verfeinern*, von einer Blockstruktur zu Logikstruktur, bis hin zu einer Transistornetzliste. Während dieser Verfeinerung muss durch Simulation die Validierung gewährleistet werden. Auf der *Systemebene* wird eine noch höhere Abstraktion von HW/SW *gemeinsam* in einer Platform entworfen. Dazu sollte die Partitionierung am besten *automatisch* erfolgen.

4.2.2. Beschreiben und Synthetisieren

Bei der Logiksynthese wird eine *automatische* Generierung von funktionalen und Steuerungseinheiten aus Boolschen Gleichungen und endlichen Automaten vorgenommen. Verfahren

sind die aus RA I bekannten *Minimierung boolscher Gleichungen*, *Zustandsminimierungen* und *Technologieabbildung*.

Unter Architektursynthese versteht man die Synthese einer *Strukturbeschreibung* aus einer Verhaltensbeschreibung. Diese Strukturbeschreibung ist eine integrierte Schaltung und besteht aus Operationswerk (Datenpfad) und Steuerwerk (Kontrollpfad). Verfahren hierfür sind *Allokation, Ablaufplanung* und *Bindung*. Zum Einsatz kommen formale Beschreibungsmittel, wie *Daten- und Kontrollflussgraphen*.

Während der Softwaresynthese wird aus einem Quellprogramm ein Maschinencode. Für parallele Zielarchitekturen wird dann auch hier Allokation, Ablaufplanung und Bindung notwendig. Der Unterschied ist, dass hier die Zielarchitektur gegeben ist.

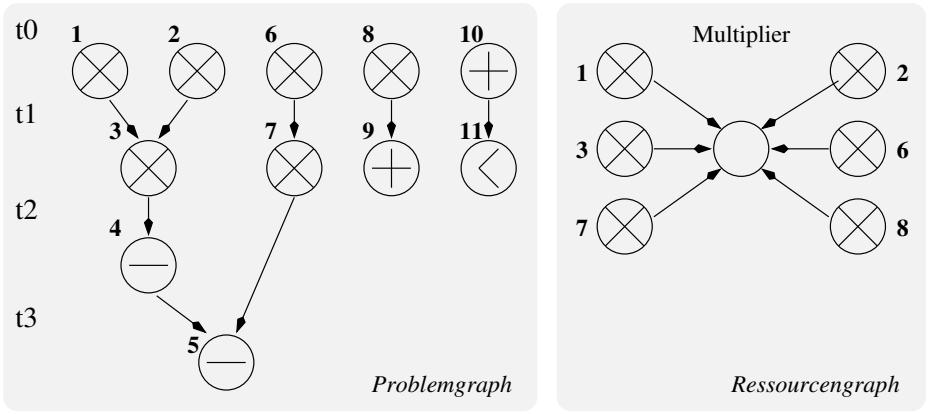
Die Systemsynthese ist schwieriger zu realisieren. Das Hauptproblem besteht in der Partitionierung und der Schnittstellensynthese. Gründe hierfür sind in der Frage der Aufteilung von HW/SW, der allgemeinen Heterogenität bei Hardware (unterschiedliche Prozessortypen) und das das Prozedere unter den Anforderungen Kosten, Zeit und Leistungsverbrauch betrachtet werden muss, zu finden. Folgende Definition werden nun benötigt:

Definition. Ein **Problemgraph** $G(V,E)$ ist *gerichtet* und *azyklisch*. Er besteht aus der Knotenmenge V und der Kantenmenge $E \subseteq V \times V$. Jeder Knoten $v_i \in V$ stellt eine Aufgabe (Prozess, Anweisung, Elementaroperation) dar. Jede Kante $e = (v_i, v_j)$ steht für eine Datenabhängigkeit zwischen zwei Aufgaben v_i, v_j.

Definition. Ein **Ressourcengraph** $G_R(V_R, E_R)$ ist *bipartit*. Die Knotenmenge $V_R = V \cup V_\gamma$ besteht aus der Knotenmenge V eines $\rightarrow$ Problemgraphen und der Menge von Knoten $r \in V_\gamma$, die Ressourcen darstellen (Speicher, Prozessor, Multiplizierer, Addierer). Für die Kantenmenge gilt: $E_R \subseteq V \times V_\gamma$. Dabei beschreibt eine Kante $e = (v_j, r_k) \in E_R$ die *Realisierbarkeit* einer Aufgabe v_j auf dem Ressourcentyp r_k. Eine *Kostenfunktion* $c : V_\gamma \rightarrow_0 +$ ordnet jedem Ressourcentyp r_k Kosten $c(r_k)$ zu. Eine *Gewichtsfunktion* $w : E_R \rightarrow_0 +$ ordnet jeder Kante (v_i, r_k) Berechnungszeit zu: $w(v_i, r_k) = d_i$, mit $w_{i,k} =_{def} d_i$.

Definition. Ein Paar $(G(V,E), G_R(V_R, E_R))$ heißt **Spezifikation**.

Hier ein Beispiel für eine *Spezifikation*. Links der Problemgraph mit der vertikalen Zeitachse, rechts der Ressourcengraph mit einem Beispiel für die Multiplikationsressource.



Allokation. Die *Allokation* dient zur Bestimmung der *Anzahl und Art* der Komponenten, etwa wie viele Registerbänke, Speicherbänke, Zahl und Art der internen Busse, Aussagen über die funktionalen Einheiten und sie dient zur *Kostenabschätzung*.

Ablaufplanung. Die *Ablaufplanung* legt die *Startzeiten* unter Berücksichtigung der Datenabhängigkeiten im Problemgraphen fest. Dabei werden die spezifizierten Aufgaben Zeitintervallen zugewiesen und geplant, wann welche Operation ausgeführt werden kann unter Beachtung der Datenabhängigkeiten.

Bindung. Die *Bindung* dient zur *Kopplung* von Ablaufplanung und Allokation. Sie spezifiziert, auf welcher Instanz und welchem Ressourcentyp eine Aufgabe implementiert wird. Jede Aufgabe auf einem Problemgraphen muss mit mindestens einem Ressourcentyp lösbar sein. Weiterhin erledigt die Bindung die *Zuordnung* von Daten zu Speicherzellen, von Operationen zu funktionalen Einheiten und von Kommunikationen zu Bussen.

Zielstellung. Die Ablaufplanung, die Allokation von Ressourcen und die Bindung von Aufgaben an Ressourcen soll so geschehen, dass eine spezielle *Zielfunktion*, etwa die Ausführungszeit, optimiert wird. Im folgenden wird etwas genauer auf die Ablaufplanung und Partitionierung eingegangen.

4.2.3. Ablaufplanung

Ablaufplanung ohne Ressourcenbeschränkung. Dies sind Probleme, die in polynomieller Zeit lösbar sind. Ziel der Betrachtung ist die *Minimierung der Latenz*. Formal ausgedrückt bedeutet dies:

$$\min\{L|\tau(v_i)-\tau(v_j)\geq d_i,\forall(v_i,v_j)\in E\}$$

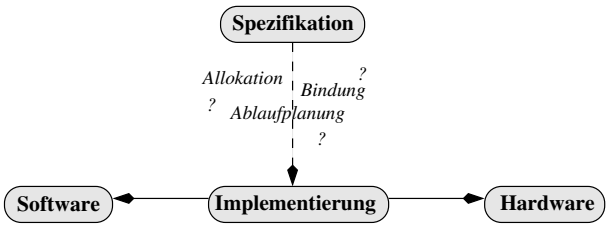
$$\text{mit (Latenz) } L=\max_{v_i\in V}\{\tau(v_i)-d_i\}-\min_{v_i\in V}\{\tau(v_i)\}$$

Es existieren zum einen der ASAP-Algorithmus und zum anderen der ALAP-Algorithmus. Die Idee des ASAP-A. ist jede Operation so *früh* wie möglich zu starten und liefert damit die gesuchte *minimale Latenz* L^S. Der ALAP- A. startet jede Operation so *spät* wie möglich und man erhält eine obere Latenzschranke. Man erhält ein Intervall für mögliche Startzeitpunkte aller Knoten, genannt *Mobilität* μ. Dies dient als Ausgangspunkt für Algorithmen *mit* Ressourcenbeschränkung.

Ablaufplanung mit Ressourcenbeschränkung. Hierbei handelt es sich um Optimierungsprobleme zur Ablaufplanung, die NP−hart sind. Es gibt den *heuristischen* Ansatz mit Listscheduling[1] und Kräftemodell[2]. Ein exaktes Verfahren dagegen ist das ILP (*integer linear programming*). Dabei wird aus den durch ASAP/ALAP-berechneten Werten ein Ungleichungssystem aufgestellt und gelöst.

4.2.4. Partitionierung

Die HW/SW-Partitionierung gehört zur Systemsynthese. Die *Systemsynthese* bedeutet die Abbildung *von* einer Verhaltensbeschreibung von funktionale Objekten mit der Granularität von Algorithmen, Prozeduren, Tasks und Prozessen *in* eine strukturelle Beschreibung mit Objekten aus Prozessoren, ASICs, Bussen und Speicherzellen. Die Systemsynthese trennt die Spezifikation bezüglich des Hardware und Softwareanteils. Folglich kann dann die Hardware durch Logik- und Architektursynthese verfeinert werden, während die Software der Softwaresynthese zugeführt wird.



Zur HW/SW-Partitionierung gibt es *konstruktive* und *iterative* Verfahren. Erste bestehen aus *bottom-up clustering*-Verfahren, wie dem *hierarchischen Clustering*. Dieses Verfahren dient Gruppierung von Objekten und anschliessender Berechnung von *closeness*-Funktionen. Eine andere Art eines konstruktiven Verfahrens ist die *Zufallsgruppierung*. Hierbei werden Objekte zufällig an Komponenten gebunden. Dies kostet $O(n)$ Aufwand und bildet eine Grundlage für *iterative* Verfahren. Darunter versteht man Verfahren, die eine iterative Verbesserung einer gegebenen Partition zur Berechnung einer Zielfunktion vornehmen. Vorraussetzung für beide Verfahrenstypen ist die bereits abgeschlossene Allokation. Nachteile von konstruktiven Verfahren bestehen in Schwierigkeiten von relativen closeness-Werten auf absolute Metriken schliessen zu können; in der fehlenden Überprüfbarkeit von globalen Entwurfsbeschränkungen. Weiterhin ist der Entwurf der closeness-Funktionen schwierig.

[1] Berücksichtigen von globalen Kriterien für die Knotenauswahl, etwa die Anzahl der Nachfolgeknoten oder das Gewicht des längsten Pfades.

[2] Berechnen von Kräften für Kanten im Ressourcengraph und Auswahl der kräftigsten.

Iterative Verfahren. Das *Kernigan-Lin*-Verfahren ist geeignet für Bipartitionen. Ausgehend von einer zufälligen Gruppierung wird versuchsweise die *Umgruppierung* eines Objektes von einer Partition in eine andere versucht. Schliesslich wird die Umgruppierung verwendet, die die *Kostenfunktion* am besten voranbringt. Ein anderes Verfahren, das *Simulated-Annealing*, ist dem Kernigan-Lin-Verfahren ähnlich. Der Ausgangspunkt hier ist ebenfalls die Bi-Partition $P = (P_{SW}, P_{HW})$. In Abhängigkeit der Ausgangssituation werden *komplexe* Funktionen in Software und performanzkritische Funktionenen in Hardware realisiert. Beim hardwareorientierten Ansatz gilt für die Anfangspartition $P_{HW} = O, P_{SW} = \{\}$. Analog gilt für den Softwareansatz $P_{SW} = O, P_{HW} = \{\}$. Desweiteren gibt es noch Greedy-, Gupta und evolutionäre Algorithmen. Letztere dienen dazu die optimale Implementierung (Allokation, Bindung, Ablaufplanung) zu finden.

5. Intellectual Property

Der Produktivitätszuwachs des Schaltungsentwurfes liegt hinter dem Zuwachs an zur Verfügung stehender Halbleiterfläche zurück. In diesem Zusammenhang spricht man von der *Designkrise* und `IPs` (*intellectual properties*) werden als Lösung dafür betrachtet. Es werden bereits entworfene und verifizierte Schaltungsblöcke (*design reuse*) wiederverwendet. Dies gilt in zunehmenden Maß auch für hochkomplexe integrierte Schaltungen. Diese Schaltungsblöcke werden als IPs angeboten. Dabei werden Schaltungsblöcke, ein gesamtes Bauteil oder eine Funktion als Baustein standardisiert geboten. Entsprechend dem *Entwurfsgrades* werden diese eingeteilt in *Soft, Firm, Hard-Ips*.

Soft-IP. Diese werden in einer Hardwarebeschreibungssprache beschrieben und sind vollständig synthetisierbar, sehr flexibel und können unbegrenzt portiert werden. Ein Schutz des IP ist nicht möglich.

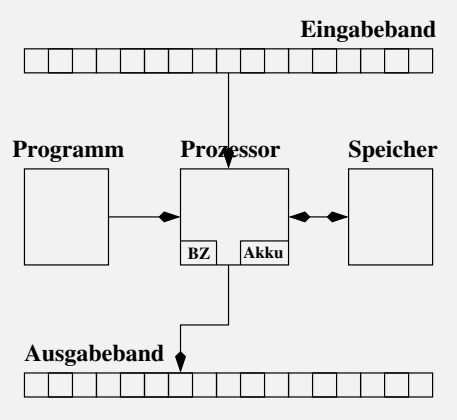
Firm-IP. Hier werden HDL-Beschreibungen und Netzlisten angeboten. Sie sind teilweise synthetisierbar und in einem gewissen Maß flexibel einsetzbar. Portierbar sind sie innerhalb einer Bibliothek. Auch hier gibt es keinen IP-Schutz.

Hard-IP. Damit werden komplette Layouts für eine Zielarchitektur angeboten. Synthetisierbarkeit gibt es nicht mehr, da keine HDL-Beschreibungen verfügbar sind. Dementsprechend gelten sie als unflexibel und die Portierbarkeit beschränkt sich auf eine Abbildung innerhalb eines Prozesses. Ein IP-Schutz ist hier gegeben und die Vorhersagbarkeit des Verhaltens ist (im Gegensatz zu soft und firm) klar definiert.

6. Parallelrechner

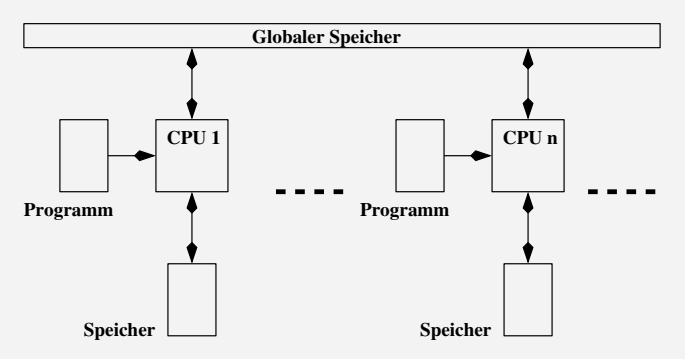
- Menge von Verarbeitungseinheiten, die in einer koordinierten Art und Weise teilweise zeitgleich zusammenarbeiten um einer Aufgabe zu lösen
- Prinzipien der Paralleltechnik in Mikroprozessoren und auch in der PC-Parallelverarbeitung
  - Eingebettete Systeme als spezialisierte Parallelrechner
  - Superskalare Prozessoren und da das Befehlspipelining
  - One-Chip Multiprozessoren
  - Mehrfädige Prozessoren
  - VLIW-Prozessoren
  - oder PC-Parallelverarbeitung: zeitliches Überlappen von Mikroprozessor und DMA-Einheit beispielsweise
- Wichtig jedoch: Klassifikation nach Flynn:
  - SISD: Universalrechenautomat
  - SIMD: Vektorrechner
  - MISD: spekulative Befehlsausführung
  - MIMD: Multiprozessorsysteme

6.1. Theoretische Grundlagen



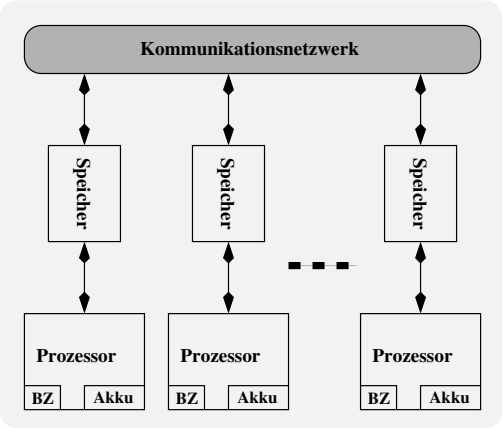
- RAM-Model für Einprozessorsysteme

- besteht aus Recheneinheit, Programm, Lese- Schreibspeicher, der abzählbar vielen Registern $L[0], \ldots$ entspricht, und Ein- und Ausgabeband
- Arbeitet auf den natürlichen Zahlen

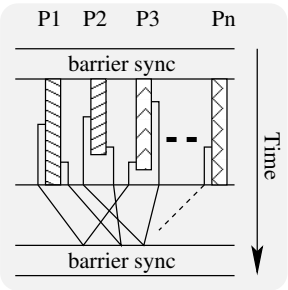


- PRAM-Modell: Modell eines idealisierten speichergekoppelten Parallelrechners
- bestehend aus identischen n-Prozessoren die alle auf einen gemeinsamen Speicher $G[0], \ldots$ zugreifen können
- gemeinsamer Takt, führen zu einem Zeitpunkt diesselben oder auch verschiedenartige Rechenoperationen aus
- es entstehen Zugriffskonflikte
  - Befehlszyklus aufsplitten in Speicher lesen, Operation Ausführen, Speicher Schreiben
  - Nach jedem Schritt Synchronisierung (lock-step)
  - d. h. Konflikte treten nur noch bei gemeinsamen Speicher lesen/schreiben auf
- EREW-PRAM: exklusiv Lesen und Schreiben
- CREW-PRAM: gleichzeitig Lesung und exklusiv Schreiben
- ERCW-PRAM, CRCW, analog
- Bei CW und Schreibkonflikten auflösung nach:
  - C-CRCW: nur erlauben wenn alle Prozessoren den gleichen Wert in die Speicherzelle schreiben wollen
  - A-CRCW: ein schreibender Prozessor gewinnt, die anderen werden ignoriert
  - P-CRCW: der Prozessor mit dem kleinsten Prioritätsindex darf schreiben

Die PRAM ist also ein *speichergekoppeltes*, idealisiertes Parallelrechnermodell, bei dem Speicherzugriff und Programmausführung ohne zusätzliche Kosten realisiert werden. Der Realität kommt das CREW-Modell am Nächsten.



- BSP - *bulk synchronous parallel computer*
- berücksichtig Kosten für Kommunikation
- beinhaltet ein nachrichtengekoppeltes Maschinenmodell
- Skalierbarkeit der Architektur, Lieferant für Entwicklung architekturunabhängiger, portabler Software für Parallelrechner
- Maschinenmodell, Programmiermodell
  - Maschinenmodell siehe Bild:
  - Prozessor-Speicher-Paare, Netzwerk für P2P-Kommunikationsverbindungen, Mechanismus zur Synchronisation aller bzw. einer Teilmenge der Prozessoren (*Barrierensynchronisierung*)



  - Programmiermodell:
  - Prinzip der supersteps (Superschritte)
  - gewisse Anzahl Superschritte wird parallel und unabhängig voneinander auf verschiedenen Prozessoren durchgeführt
  - Superschritt:
    * feste Anzahl Berechnungsschritte auf lokalen Variablen
    * Senden und Empfangen von Nachrichten

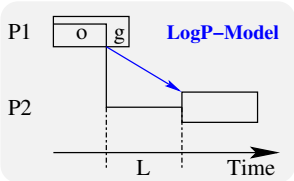
    - folgende Barrieresynchronisation, dann neuer Superschritt
  - Superschritte sorgen für Entkopplung von Berechnung und Kommunikation

Das BSP ist von Relevanz für *Architekturen mit Verteiltem Speicher*, *Multiprozessoren mit gemeinsamen Speicher* und Netzwerken von Workstations, etwa *Cluster-Computing*.

- Leistungsbewertung des BSP:
- Leistung bestimmt durch Zeit für Berechnung, Zeit für Kommunikation
- Leistung wird durch diese Parameter beeinflusst:
  - Anzahl Prozessoren p
  - Kosten für Daten beim Nachrichtenaustausch g [Schritte pro Wort]
  - Kosten für Barrieresynchronisierung l [Anzahl Schritte]
  - Kosten für einen Superschritt:
    - Prozessorindex i, Datenvolumen eines Prozessors h, Maß für die in einem Prozessor durchgeführten Berechnungen w
    - $Kosten(Superschritt) = MAX_i w_i + MAX_i h_i \cdot g + l$

Es gibt einen (standardisierten) Befehlssatz für BSP (aka oxford toolbox set). Merkmale:

- überschaubarer Befehlssatz (20 Befehle)
- frei verfügbar, entworfen von internationalem Team
- Kommunikationsbefehle:
  1. direct remote access $\rightarrow$ Prozessor kann direkt auf den Speicher eines anderen Prozessors zugreifen (ohne dessen Mithilfe)
  2. bulk synchronous message passing $\rightarrow$ Prozessor schickt seine Daten in die Warteschlange des anderen Prozessors, der diese zum nächsten Superschritt entgegennimmt und bearbeitet



- LogP-Model: nachrichtengekoppeltes Maschinenmodell
- Berücksichtigung von Asynchronizität
- Abstraktion von der konkreten Ausprägung des Netzwerkes

- **L**atenzzeit L ist die maximal benötigte Zeit um eine kleine Nachricht zu übertragen
- **o**verhead o ist der Zeitbedarf für das Senden/Empfangen (beim Senden/Empfangen ist ein Prozessor gesperrt für andere Operationen)
- **g**ap g ist untere Schranke für die Zeit, die jeder Prozessor bis zum Absenden einer neuen Nachricht abwarten muss
- **P**rozessoren P, jeder kann optional einen lokalen Speicher besitzen
- Zusammenfassung:
- PRAM: taktsynchrones, speichergekoppeltes Modell, gut programmierbar
- BSP: Synchronisierung mit Superschritten, nachrichten- und speichergekoppeltes Modell
- LogP: asynchron, nachrichtengekoppeltes Modell, geeignet für Laufzeitabschätzungen

6.2. Grundprinzipien

6.2.1. Funktionales Trennen

$\rightarrow$ für unterschiedliche Operationen stehen unterschiedliche ALUs zur Verfügung, wobei einzelne ALUs für ihre Aufgabe optimiert sind und zeitlich parallel arbeiten können.

6.2.2. Pipelining

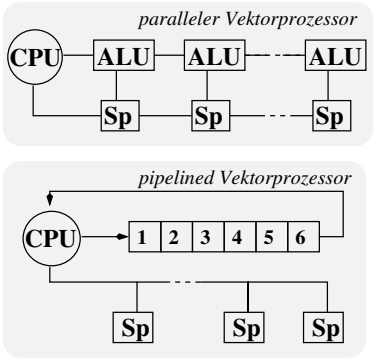
- meistverbreitetste Methode der Parallelverarbeitung
- Zerlegung einzelner Operationen in Elementaroperationen
- Aufbau einer Pipeline:
- linear (nacheinander werden alle Stufen durchlaufen, keine wird ausgelassen)
- nicht-linear (Rückkopplung und Überholvorgange möglich, höhere Komplexität, Lösung für Datenhazards)
- Funktionalitätsklassen von Pipelines:
- unifunktional (für spezielle Aufgaben- etwa Pipelines für Addition)
- multifunktional (flexible für unterschiedliche Aufgaben)
  - müssen Konfiguriert werden, entweder statisch oder dynamisch
  - statisch: zu Beginn einer Operation
  - dynamisch: zur Laufzeit

6.2.3. Datenparallelität

→ Anwendung gleicher Operationen auf Vektor-, Matrix- oder Gitterstrukturen. Entspricht dem klassischen SIMD-Prinzip.

6.3. Vektorrechner

- Prinzip des Pipelining hier besonders (→ *Pipelinerechner*)
- Arbeitet nicht auf skalarem, sondern gleichzeitig auf allen Elementen eines Vektors
- SIMD-Rechner



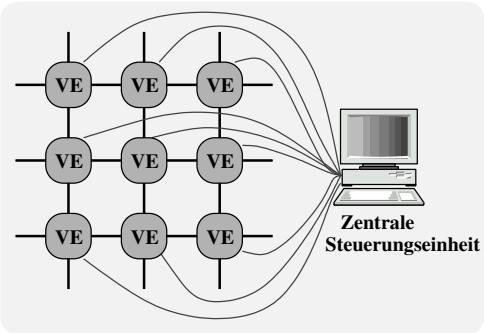
- Paralleler Vektorprozessor:
  - Ideal: Anzahl ALUs gleich Anzahl Vektorelemente
  - aus kostengründen anfangs nicht tragbar
- Pipelined Vektorprozessor:
  - mit arithmetisch/logischer Pipeline versehen

6.4. Feldrechner

Eigenschaften:

- Menge gleicher Recheneinheiten bilden einen Feldrechner
- Verbunden über Netzwerk, oder über einfache Leitungen ein- oder mehrdimensional verbunden
- Recheneinheiten besitzen lokalen Speicher
- Kontrolle einer zentralen Steuereinheit
  - RE führen gleichzeitig die selbe Operation auf verschiedenen Daten aus (**SIMD**)

- Ideal: Pro Zeitschritt bei n-RE gibt es n Rechenergebnisse
- Maskierung um einzelne RE auszuschliessen
- Aufbau:



- Möglich ist *lokal zugeordneter* und *gemeinsam genutzter* Speicher

Anwendungen:

- Signal- und Bildverarbeitung
- numerische Aufgabenstellungen
- Datenbank und Textretrival
- **datenparallele** Algorithmen

Programmierung:

- Automatisierung schwierig
- direkte Abbildung eines für einen seriellen Rechner vorliegenden Programmes:
  1. Erkennen und Neuprogrammierung offensichtlicher Parallelität (Bsp. Iterationen)
  2. Erkennen und Neuprogrammierung versteckter Parallelität (Bsp. Matrixbearbeitung)
  3. Betrachten des Lösungsalgorithmus (Änderungen am Algorithmus- nicht am seriellen Programm)

Beispielrechner:

- ILLIAC IV:
  - Superrechner von 72-81, für numerische Probleme
  - 64 parallele Verarbeitungseinheiten, Steuereinheit (SE)
  - 40ns Taktzyklus, lokaler Speicher $2K \cdot 64Bit$
  - NEWS-Netzwerk(North-East-West-South)

  - SE: Kompilierung Userprogramm, Verteilung Befehlskode, Ausführung Systemsoftware
  - Magnettrommel mit 122MB
- DAP:
  - 1-Bit Prozessorelemente, Volladdierer
  - gemeinsamer Speicher
  - Matrizenrechner, logische Ops auf Matrizen, Nutzung der Nachbarschaftsbeziehungen
- MasParI und II.
  - MP1/2 bis 90iger in betrieb
  - Vorrechner, Steuerfeldeinheit, Feld aus 16384 PEs
  - PE: 32Bit Architektur mit 4Bit Datenpfaden
  - MP2: 32Bit Datenpfade
  - lokales Netz: Verbindung der Nachbar-PEs
  - globales Netz: 3-stufiges CLOS-Netzwerk

Neue Entwicklung: On-Chip SIMD-Architekturen

- Klassischer Feldrechner hat ausgedient
- Aber: Renaissance des Prinzips für Spezielle Anwendungen (CMOS-Kamera)
- Parallele Bildaufnahme und -verarbeitung auf einem Chip
- Anwendung: Hochgeschwindigkeitskameras
- smart pixel technologie
  - jedes pe wendet gleiche operation an und arbeitet mit der lokalen nachbarschaft
- Berechnung der Erosion:

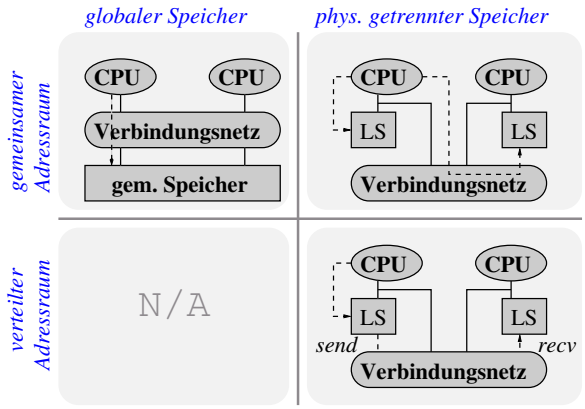
$$f_{erode}(x,y) = A(x,y) \cup A(x-1,y) \cup A(x,y-1) \cup A(x-1,y-1)$$

Beispiel III: Earth Simulator

- MIMD-Rechner, gemeinsamer, verteilter Speicher
- Prozessorknoten sind eng-gekoppelte Vektorprozessoren
- Speicher 10TB, 16GB pro Knoten

6.5. Multiprozessorsysteme

- nach Flynn: MIMD
- Unterschied zu Feldrechnern: unterschiedliche Befehle pro Prozessorelement, Strukturunterschiede bei Feldrechnern geringer
- Unterscheidung in speichergekoppelte und nachrichtengekoppelte Multiprozessorsysteme



- Speichergekoppelt:
  - gemeinsamer Adressraum für alle PE
  - Kommunikation und Synchronisation über gemeinsame Variablen
    - SMP: Symmetrischer Multiprozessor- ein globaler Speicher, gleiche Zugriffszeit für alle Prozessoren
    - DSM: Distributed Shared Memory System- physikalisch verteilter Speicher, aber gemeinsamer Adressraum
- Nachrichtengekoppelt:
  - nur physikalisch verteilte Speicher und prozessorlokale Adressräume
  - Kommunikation via Nachrichtenaustausch

6.5.1. Speichergekoppelte Multiprozessoren

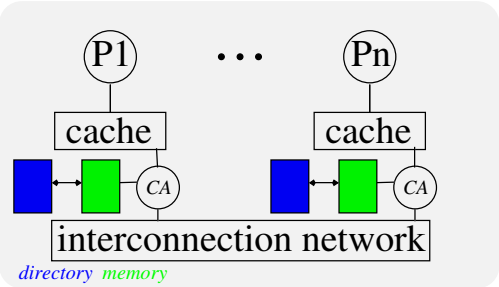
- Gemeinsamer Adressraum, Kommunikation/Synchronisation über gemeinsame Variablen
- UMA - *uniform memory access model*:
  - alle cpus greifen gleichermaßen auf gemeinsamen Speicher zu
  - Zugriffszeit aller Prozessoren auf gemeinsamen Speicher ist gleich
  - Lokaler Cache pro cpu möglich
  - typisch: Symmetrische Multiprozessoren

- NUMA - *non-uniform memory access model*:
  - Speichermodule des gemeinsamen Speichers auf die Prozessoren physikalisch aufgeteilt
  - $\rightarrow$ Zugriffszeit unterschiedlich (abhängig vom Ort des Zugriffs)
  - typisch: Distributed-Shared-Memory-Systeme
- Weiter Unterscheidung von NUMA bzgl Cache-Zugriff:
  - CC−NUMA: *cache coherent* für das gesamte System
  - NCC−NUMA: *non-cache coherent* - CC nur innerhalb eines Knotens
  - COMA: *cache only memory architecture*: Gesamter Speicher besteht nur aus Cache

6.5.2. Cache Kohärenz.

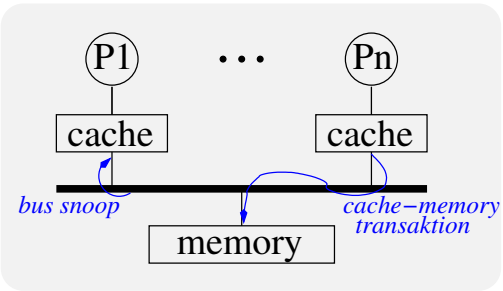
- Linderung der „memory gap“, Cache ist Standard in Mikroprozessoren!
- Inkonsistenz: Speicher vs. Cache $\rightarrow$ *noch kritischer* in MP-Systemen:
  - Kopien des gleichen Datums existieren mehrfach
- Monoprozessoren haben *write through* und *write back*-Strategien, MP-Systeme benötigen dagegen *Cache Coherence Strategies*, a. k. a. Cacheprotokolle
- CCS in Hardware und Software möglich
- CCS in Software:
  - via Compiler und OS
  - Zusatzaufwand auf Kosten der Compilezeit
  - komplexere Software
  - Prinzip: gemeinsame („neuralgische“) Variablen finden und als nicht-cachebar markieren
  - nicht besonders effizient
- CCS in Hardware:
  - erlauben dynamische Lösungen
  - transparent für den Programmierer
  - *Verzeichnis-, Snoopy*-Protokolle

directory cache coherence protocols



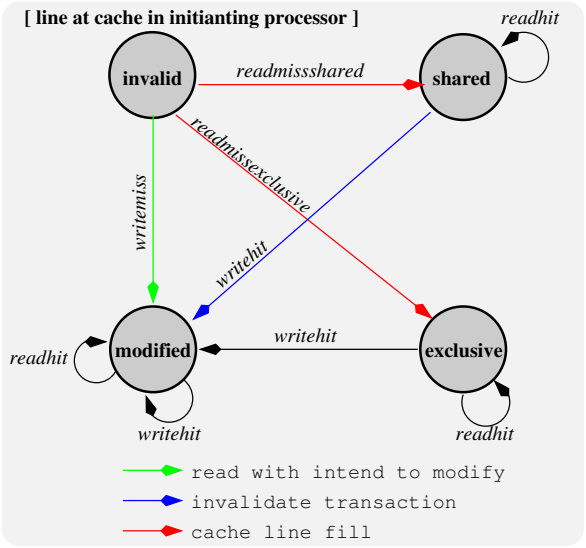
- Prinzip: Sammeln und Verwalten von Informationen zum Aufenthaltsort von Cacheinhalten
- Strukturen: zentraler Cachecontroller, lokale Cachecontroller, im Speicher abgelegtes Verzeichnis
- Traditional Directories:
  - Variante 1: p+1 Bits pro Block; drei Zustände pro Block (1. Invalid - no bit set, 2. shared read only - p bits set, extra bit not set, 3. exclusive - extra bit is set and *one* of the others)
  - Variante 2: 2 Bits invalidation; 1 bit für Gültigkeitsstatus, 1 Bit für exklusiv-Status; muss ungültigkeits nachrichten an alle versenden (traffic!)
  - Variante 3: n-pointers + rundruf; verbesserte variante 2: zusätzlicher Platz für n Zahlen der ProzessorIDs,
  - Variante 4: Verlinkte Liste: Basiszeiger für jeden Block, Verlinkte Liste auf alle Prozessoren, die sich den Speicher teilen; Ungültigkeitsnachrichten können dann an alle Prozessoren direkt verschickt werden (in der Liste)
- Fazit: Hohes Kommunikationsaufkommen, Skaliert gut bei großem Speicher,

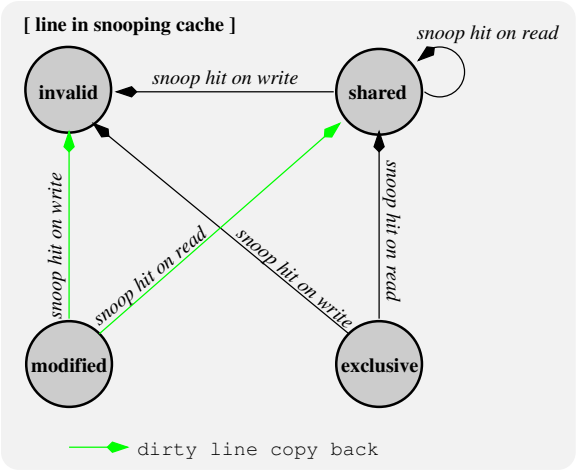
snoopy cache coherence protocols



- dezentrale Kontrolle auf alle lokalen Cachecontroller
- Cache muss erkennen, dass eine Cachezeile gemeinsam mit anderen Caches benutzt wird → Abhören der Kommunikation notwendig!

- Neuschreiben einer Zeile per Broadcast an alle lokalen Cachecontroller mitteilen
- Primär geeignet für Busgekoppelte MP-Systeme; alle Controller lauschen auf dem Bus
- ABER: Traffic auf dem Bus wird erhöht → wird unsinnig wenn der Bustraffic durch den Cachetraffic gemindert wird
- *write invalidate*
  - *schreib*aktion auf dem bus, die lokale cache kopie des eintrags betrifft → ungültig machen des Eintrags
- *write update/broadcast*
  - *schreib*aktion auf dem bus, die lokale cache kopie des eintrags betrifft → aktualisieren der kopien in den anderen caches
- für beide Varianten gilt: beim nächsten Zugriff des Prozessors auf den Block
  - entweder: cache miss durch Ungültigkeit → Zeile neulesen
  - oder: Block ist aktuell durch das vorhergehende update
- Beispiel: `MESI`-Protokoll



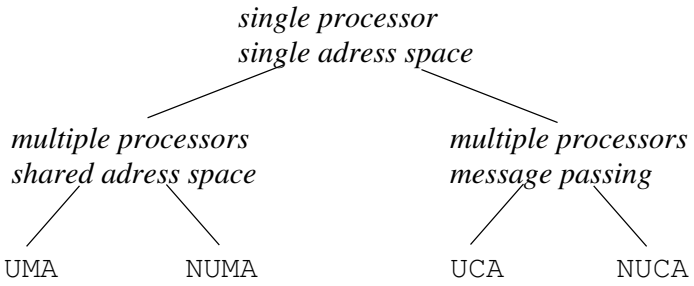


- write-invalidate-Variante
- jede Cachezeile kann 4 Zustände annehmen:
  - **M**odified: Zeile im Cache modifiziert, nur lokal verfügbar
  - **E**xclusive: Zeile identisch mit Hauptspeichereintrag, nur lokal verfügbar
  - **S**hared: Zeile identisch mit Hauptspeichereintrag, evtl. auf mehreren Caches verfügbar
  - **I**nvalidate: Zeile enthälte ungültige Daten

6.5.3. Nachrichtengekoppelte Multiprozessorsysteme

- NORMA: *no remote memory access model*
- UCA: *uniform communication architecture*
  - Zwischen allen Prozessoren können gleichlange Nachrichten mit einheitlicher Übertragungszeit gesendet werden.
- NUCA: *non uniform communication architecture*
  - Die Übertragungszeit des Nachrichtentransfers ist je nach Sender und Empfängerprozessor unterschiedlich.

6.5.4. Zusammenfassung Speicher- und Nachrichtenkopplung



- Speichergekoppelte Systeme gelten als leichter zu programmieren als die nachrichtegekoppelten
- Enge Kopplung bei SMP und DSM erlaubt effiziente Ausführung kommunikationsreicher Paralleler Programme
- Skalierung:
  - SMP: < 30 Knoten
  - DSM: < 256 Knoten
  - Message Passing: theoretisch unbegrenzt!
- Vorraussetzung für den Einsatz von nachrichtengekoppelten Multiprozessorsystemen:
  - Parallelisierbarkeit des Problems auf Prozessebene mit wenig Kommunikationsaufwand

6.6. Cluster Computing

Definition. Ein **Cluster** ist ein *paralleles, verteiltes* Rechensystem, dass

- aus einer Menge *für sich allein funktionierender Rechner* besteht, die
- miteinander über ein *Netzwerk verbunden* sind und die
- zusammen als eine *integrierte Rechenressource* arbeiten.

Argumentation pro Cluster:

- PCs werden ständig leistungsstärker
- Zunahme der Bandbreite, Abnahme der Latenz bei der Kommunikation
- PCs sind einfacher in bestehende Strukturen zu integrieren als spezielle Parallelrechner
- geringe Auslastung von einzelnen PCs durch Besitzer
- Einzelne Entwicklungswerkzeuge sind häufig ausgereifter als spezielle Software für Parallelrechner
- Kostengünstiger!
- Einfache Ausbaufähigkeit

Architektur von Clustern

- mehrere Rechner
- Cluster-Betriebssystem
- Hochgeschwindigkeitsnetzverbindung, Switch
- Ethernet, SCI, Myrinet (Hardware)
- Active Messages, Fast Messages, VIA, Infiniband (überwiegend Software)

Vermittlungstechniken - switching

- *store and forward*
  - Paketvermittlung
  - Paketgröße begrenzt
  - Zwischenspeicherung und Auswertung der Zieladresse
- *cut through/wormhole switching*
  - Weiterleiten nach Auswerten des Headers
  - niedrige Latenz und kleine Puffer
  - Nachrichtengröße variable
  - ABER: schwierige Fehlererkennung

Wegewahl - routing

- Empfängeradresse im Header
- deterministisches und adaptives routing (*source-path-routing/table based routing*

6.6.1. Kommunikationsprotokolle in Clustern

- Schnelle Hardware? Flaschenhals bei der Software, den Protokollen!
- Ziel: Schnelle Protokolle
- Verringerung der Kopieroperationen auf dem Datenpfad
- Probleme von TCP/IP in Clustern:
  - Schichtenstruktur: große Anzahl an Speicher zu Speicher Kopieroperationen
  - in jeder Schicht Aufruf von Funktionen
  - Ergo: hoher Zeitaufwand und Performanzverschlechterung

- nach Eintreffen einer negativen Quittung → Neuanforderung der Dateneinheit und folgender Dateneinheiten
- *active messages*:
  - Sender kann immer Daten schicken unabhängig von der Aktivität des Empfängers
  - Trennung von Berechnung/Kommunikation
  - Schlanker Datenpfad (eliminiert Zwischenspeicherung, braucht spezielle NIC)
  - *receiver handler* (sofort nach Empfang der Nachricht Aufruf von benutzerdefinierter Funktion)
  - Alle Knoten haben den gleichen Adressraum (SPMD)
- *virtual interface architecture*
  - einheitliche Schnittstelle zwischen Userspace und Nic
  - hohe Bandbreite 1Gb/s, geringe Latenzzeit
  - große HW-Unterstützung
  - Einheitliches Programmiermodell durch FUnktionsspezifikation

6.6.2. Cluster Middle Ware - Betriebssysteme

- `SSI`: (*single systeme image*)
  - Darstellung eines *einheitlichen* Systemabbildes
- `SAI`; (*system availability infrastructur*)
  - Verfügbarkeit innerhalb unabhängiger aber miteinander über ein Netzwerk verbundene Rechner herstellen
  - Aufgaben: Fehlertoleranz, Recovery

Verfügbarkeitsfunktionen.

- Einheitlicher E/A-Raum → jeder Knoten hat auf beliebige Peripheriezugriff ohne zu wissen wo die ist
- Einheitlicher Prozessraum → jeder Prozess kann auf beliebigem Knoten ablaufen, Kommunikation der Prozess erfolgt über Signale,Pipes etc
- Check-Punkte und Prozess Migration → Prozesszustände und Zwischenergebnisse in den Speicher schreiben, dadurch Lastverteilung und Fehlertoleranz

6.7. Leistungsbewertung von Parallelrechnersystemen

→ Blickpunkt: Algorithmen und Architekturen

6.7.1. Leistungsmaße

- Speed-Up:

$$S = \frac{T_1}{T_P} = \frac{T_{seriell}}{T_{parallel}}$$

- Effizenz:

$$E = \frac{S}{P} = \frac{speed-up}{anzahl-prozessoren} \text{ und es gilt: } \frac{1}{P} \leq E \leq 1$$

- Speed-Up: *speziell*:

$$S(n,p) = \frac{T_{seriell}(n)+T_{parallel}(n)}{T_O(n,p)+T_{seriell}(n)+\frac{T_{parallel}(n)}{P}}$$

  - n: Problemgröße
  - P: Anzahl der Prozessoren
  - T_O: Zeitaufwand Overhead
  - $T_{seriell}$: serieller Zeitaufwand
  - $T_{parallel}$: paralleler Zeitaufwand

6.7.2. Gesetz von `Amdahl`

- Frage: Welcher Speed-Up ist maximal erreichbar für Problem mit fester Größe n bei P Prozessoren, wenn der serielle Programmanteil s % beträgt?
- $T_O(n,p) = 0$!

$$T_1(n) = T_{seriell}(n) + T_{parallel}(n) = s \cdot T_1(n) + (1-s) \cdot T_P(n)$$
$$T_P(n,p) = s \cdot T_1(n) + (1-s) \cdot T_p(n)/p$$

6.7.3. Gustaffson-Barsis

- geht vom parallelen Programm als Analyseursprung aus
- Wie hoch darf im parallelen Programm der Serielle Anteil sein?
- Zusatzaufwand vernachlässigt, wie Amdahl
- Lösung: *skalierter Speed-Up*:

$$S(n,p) \leq p + (1-p) \cdot s^* = p - (p-1) \cdot s^*$$

6.7.4. Karp-Flatt-Maß

- explizite Berücksichtigung des Overheads T_O
- Karp-Flatt-Maß e
  - experimentell bestimmter serieller Anteil

$$e = \frac{T_{ser}(n)+T_O(n,p)}{T_{ser}(n)\cdot T_{par}(n)}$$

  - dabei gilt:

$$e = \frac{(1/S)-(1/P)}{1-1/P}$$

6.8. Algorithmen für Parallelrechner

- Blockstreifenzerlegung
- Schachbrettzerlegung

7. Netzwerke

7.1. Interne Kommunikation

→ Betrachten Kommunikationsstrukturen für: CPU-CPU, CPU-LokalerSpeicher, CPU-GlobalerSpeicher

7.1.1. Direkte Kopplung- P2P

- *Registerkopplung*:
  - Register zwischen zwei Prozessoren sind verbunden
  - schnelle Übertragung
  - Unflexibel gegenüber Änderung der Topologie
- *Speicherkopplung*:
  - Zwei oder mehr Prozessoren sind über gemeinsamen Speicher entweder über *direkte* Leitungen oder Busse verbunden
  - Beide Prozessoren können lesen/schreiben → Zugriffskonflikte müssen beachtet werden!
  - langsamer als Registerkopplung, aber flexibler
- *Kanalkopplung*: `DMA`−Kopplung:
  - zusätzlicher DMA-Kontroller für Arbeitsspeicher
  - Prozessor stößt Kommunikation an (übermittelt Anfangsadresse und Datenmenge an Kontroller)
  - Gleichzeitige Bearbeitung des Prozessors der Daten und des Kontrollers der Kommunikation
  - Kontroller optimiert für Kommunikation → hohe Datenübertragungsraten möglich

7.1.2. Bussysteme

- Mehrere Prozessoren an einem Bussystem angeschlossen
- Langsamer aber wesentlich flexibler als Direktkopplung bei der Topologieänderung
- Strategien für Buszugriff:
- *time shared*:

  - Zeitscheibenmethode für jeden Prozessor

- *token*:
  - Token wandert von CPU zu CPU
  - Besitzer darf auf dem Bus senden

- *random access*
  - Aloha-Verfahren
  - Bus frei? → Einer darf senden
  - Notwendig: CSMACD zur Sicherung

- *Priorität*
  - Rangordnungen von Stationen mit gleichzeitigem Sendewunsch

- **FAZIT:** Flexibel, aber zu langsam und auch zu aufwendig für Parallelrechner → besser: Verbindungsnetzwerke, statisch oder dynamisch

7.1.3. Statische Verbindungsnetzwerke

- starre Pfade
- Basis: Permutationen

Exchange Permutation .

- k-tes bit der Permutation wird vertauscht

$$E_k(x) = (x_i, x_{i-1}, \ldots, \overline{x_k}, \ldots, x_1, x_0)$$

Perfect Shuffle Permutation

- Zyklische Linksverschiebung um ein bit

$$PS(x) = (x_{i-1}, x_{i-2}, \ldots, x_0, x_i)$$

Butterfly Permutation

- Für FFT angewendet
- Vertausch von MSB und LSB

$$BP(x) = (x_0, x_{i-1}, \ldots, x_1, x_i)$$

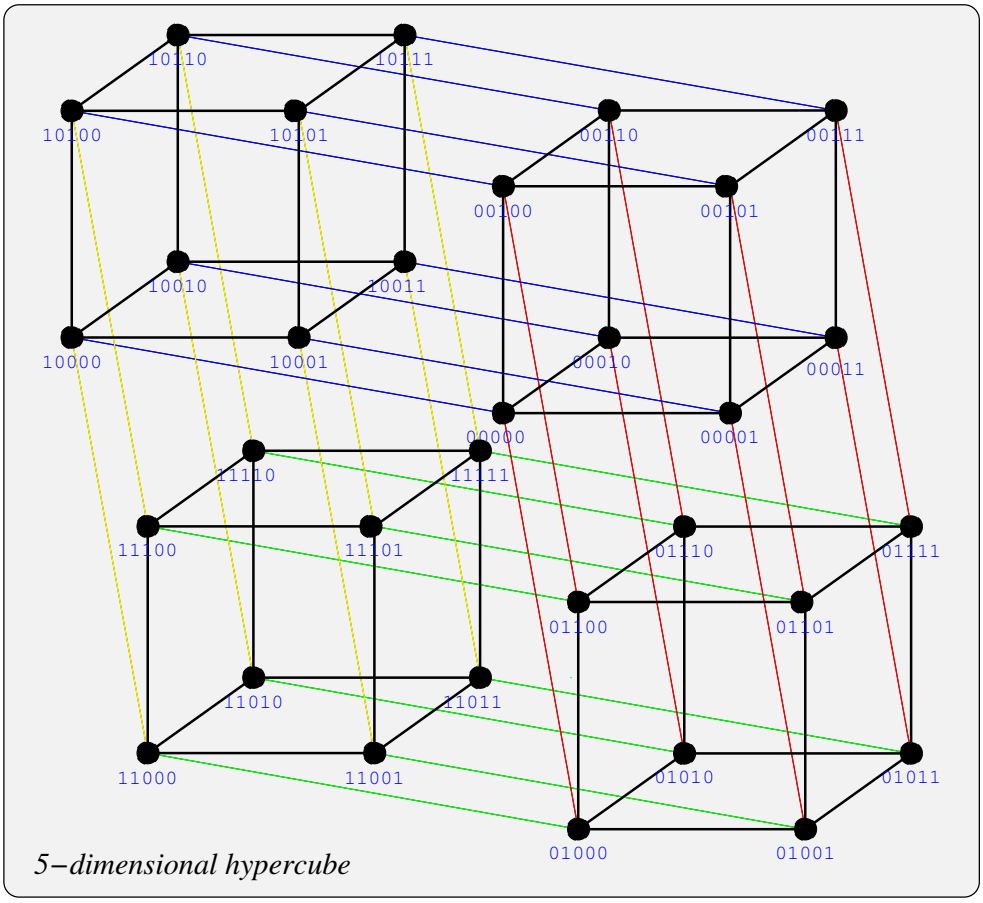
Bit Reversed Permutation

- Umkehrung der Bitreihenfolge

$$BR(x) = (x_0, x_1, \ldots, x_{i-1}, x_i)$$

- mit Hilfe dieser Permutationen → Aufbau von *statischen* Verbindungsnetzwerken
- Wesentliches Merkmal: Keine expliziten Schalterstufen!
- weitere statische Netzwerke: Feldstrukturen und Würfelstrukturen

Hyperwürfelstrukturen



5−dimensional hypercube

- Rekursives Konstruktionsprinzip:
  - HW mit Dimension $d = 0$: Einfacher Rechner
  - HW mit Dimension $d = i$:
    - Nimm Zwei Würfel der Dimension $i-1$ und verbinde äquivalente Ecken (Binärdarstellung der Adressen unterscheidet sich in genau einer Dimension)

- Anzahl Ecken $n = 2^d$
- Maximale Weglänge $d = \mathrm{ld}(n)$
- $\rightarrow$ viele Wegalternativen, hohe Redudanz, unterstützt Parallelität
- Routing im Hypercube
  - Startadresse $SA = 01010$, Zieladresse $ZA = 10101$
  - Maske $M = xor(SA, ZA) = 11111$

Maske	Zwischenadresse alt	Zwischenadresse neu
1111**1**	0101**0**	0101**1**
111**1**1	010**1**1	010**0**1
11**1**11	01**0**01	01**1**01
1**1**111	0**1**101	0**0**101
11111	**0**0101	**1**0101
		10101

Tabelle 7.1.: Routingprozess im Hypercubus

Weitere statische Verbindungsnetzwerke

- Cube-Connected Cycle
- Baumstrukturen

7.1.4. Dynamische Verbindungsnetzwerke

- zusätzliche zu statischen Verbindungen: Einbau von *Schaltern* $\rightarrow$ Schaltnetzwerke
- Abwechselnd: Schalterstufe, Permutationsstufe
- 1. Unterscheidung in *Einstufige, Mehrstufige* Schaltnetzwerke anhand der Anzahl der Schaltstufen
- 2. Unterscheidung anhand der Wegewahl in *selbst-konfigurierende, nicht-selbstkonfigurierende* Schaltnetzwerke

Einstufige Schaltnetzwerke.

- Bekannt: *Kreuzschienenverteiler*
- n Eingänge, n Ausgänge, n^2 Schalter, alle $n!$ Permutationen schaltbar
- Problem wenn n groß

- alle weiteren einstufigen Schaltnetzwerke sind Modifikationen des Kreuzschienenverteilers:
  - Anzahl Eingänge ungleich Anzahl Ausgänge
  - Konzentration, Expansion

Mehrstufige Schaltnetzwerke.

- Mehrere einfache (2x2) Kreuzschienenverteiler hintereinanderschalten
- Benes:
  - Ein n-E/A-Kreuzschienenverteiler ersetzbar durch $\frac{n}{2}$-E/A-Kreuzschienenverteile + 2 Exchange-Netzwerke
  - rekursiv fortsetzbar bis nur noch 2x2-Kreuzschienenverteiler übrig sind

Binäres Benes Netzwerk

- universell, alle $n!$-Permutationen schaltbar
- Anzahl Kreuzschinenverteiler: $n \cdot \mathrm{ld}(n) - \frac{n}{2}$
- Pro Stufe $n/2$ Verteiler $\rightarrow 2 \cdot \mathrm{ld}(n) - 1$ Stufen nötig

Omega Netzwerk

- Perfect Shuffle zwischen allen Stufen
- $\mathrm{ld}(n)$ Stufen erforderlich
- nicht alle Permutationen möglich $\rightarrow$ nicht universell
- *selbstkonfigurierend* - einfacher Routingalgo:
  - in Schaltstufe k wird der obere Ausgang geschalten, wenn quell xor ziel am kten bit ne 0 hat

Binäres n-Cube Netzwerk

- $\mathrm{ld}(n)$ Stufen
- Routing-Algo:
  - Unterscheiden sich quell und ziel an k-ter stelle so wird die k-te stufe kreuzgeschaltet

Banyan Netzwerk

- wie n-cube, aber:
- gespiegelt, Keine Kollisionen, falls die Adressen der Pakete in aufsteigender Reihenfolge anliegen

Batcher Sortier Netzwerk

- Anzahl Stufen: $\mathrm{ld}(n) \cdot (1 + \mathrm{ld}(n))/2$
- keine Binären Schalter!

7.2. Lokale Netze

7.2.1. Ethernet

- dezentrale Kontrolle, unempfindlich gegen Knotenausfälle (vgl. Token Ring)
- Zugriffsverfahren: CSMA/CD
- Aloha-Verfahren (CSMA): lauschen am Bus, wenn nachricht für einen knoten bestimmt, dann kann dieser empfangen

7.3. Optische Netzwerke

todo:

7.4. Drahtlose Netze

todo:

8. Verteilte Systeme

todo:

Teil IV.

Fragensammlung

9. Rechnerarchitektur Teil 1

9.1. Formale Entwurfsmethoden

Frage: *Wozu gibt es formale Entwurfsmethoden?*

- Unterstützung beim Entwurf von Rechnern (Hardware *und* Software)
- Automatische Ableitung von Hardware aus Beschreibungen (*Synthese*)
- Beherrschung der Komplexität durch *Modul- und Hierarchiebildung*
- Eindeutige Beschreibung der Spezifikation (Funktionalität)

Frage: *Welche formalen Entwurfsmethoden haben sie in der Vorlesung kennengelernt?*

- endliche Automaten, Statecharts, Petrinetze, SDF, CSP und natürlich Hardwarebeschreibungssprachen wie SystemC oder VHDL

Frage: *Nennen sie die Eigenschaften von Petrinetzen? Wofür eignen sie sich?*

- formales Beschreibungmodell speziell für Systeme mit *asynchronen* und *nebenläufigen* Prozessen
- Entwurf von sowohl (asynchroner) HW als auch SW
- Definition:
  - 6-Tupel $G = (P, T, F, K, W, M_0)$ mit
    - $P \cap T = \{\}$
    - und $F \subseteq (P \times T) \cup (T \times P)$ Flussrelation
    - wobei:
    - $P = \{p_1, p_2, \ldots, p_m\}$ Menge der Plätze
    - $T = \{t_1, t_2, \ldots, t_n\}$ Menge der Transitioenn
    - $K\colon P \longrightarrow N \cup \{\infty\}$ Kapazität der Stellen
    - $W\colon F \longrightarrow N$ Gewicht der Kanten in F
    - $M_0\colon P \longrightarrow N_0$ Anfangsmarkierung, es gilt: $\forall p \in P\colon M_0(p) \leq K(p)$
- *nicht statisch*, Veränderung, wenn sich Marken durch das Netz bewegen
- Zustand eines Netzes: $M\colon P \longrightarrow N_0$, sd. $M(p)$ Anzahl der Marken an der Stelle p

- Vor- und Nachbereich für Netzknoten x:
  $*x = \{y|(y,x) \in F\}$
  $x* = \{y|(x,y) \in F\}$
  - Definition: *Schaltbereitschaft* einer Transition $t \in T$ unter der Markierung M, also $M[t>$
    1. $\forall p \in *t \setminus t*: M(p) \geq W(p,t)$
       * *alle p aus dem Vorbereich von t haben mind. die benötigten Markierungen der Kante von p nach t*
    2. $\forall p \in t* \setminus *t: M(p) \leq K(p) - W(t,p)$
       * *in alle p aus dem Nachbereich von t passen mind. soviele Markierungen wie das Gewicht von (t,p)*
    3. $\forall p \in t* \cap *t: (M(p) \leq K(p) - W(t,p) + W(p,t)) \wedge (M(p) + W(t,p) - W(p,t) \geq 0)$
       * *für alle p, die sowohl Vor- als auch Nachbereich sind genügt der Platz für die Differenz von eingehenden und ausgehenden Marken* ***und*** *nach dem Schiessen ist die Markierungsanzahl größer/gleich 0*
  - Definition: *Durchschalten* einer Transition $t \in T$ von M nach M_{neu}: $M[t>M_{neu}$
    * $M_{neu}(p) =$
      1. $M(p) - W(p,t)$ falls $p \in *t \setminus t*$
         · falls p im Vorbereich von t, dann

         <++>
      2. $M(p) + W(t,p)$ falls $p \in t* \setminus *t$
      3. $M(p) + W(t,p) - W(p,t)$ falls $p \in *t \cap t*$
      4. $M(p)$ sonst.

- *Mächtigkeit von Petrinetzen*
  - Sequentialität, Synchronisation, nicht-deterministische Verzweigung, Ressourcenkonflikt, Nebenläufigkeit

Frage: *Geben sie die Definition eines endlichen Automaten an?*

- 6-Tupel $(X, Y, Z, \alpha, \delta, \gamma)$
- X, Y: Ein- und Ausgabealphabet
- Z: Menge von Zuständen
- $\alpha \in Z$: Anfangszustand
- $\delta: Z \times X \to Z$ Zustandsüberführungsfunktion

- γ: Ausgabefunktion - Unterscheidung in *Mealy, Moore*-Automat:
  - Mealy: γ: $Z \times X \rightarrow Y$
  - Moore: γ: $Z \rightarrow Y$

Frage: *Was können endliche Automaten und was nicht?*

- geeignet zur Modellierung von Steuerungen oder Schaltwerken
- Beschreibung von Systemen mit Gedächtnis
- endliche Menge von Zuständen $\rightarrow$ endlicher Automat
- Ungeeignet für:
- Modellierung von Nebenläufigkeit
- Hierarchiebildung
- Daher Lösung mittels Automatenerweiterungen: *SDL, CSP, Statecharts*

Frage: *Wie überführt man vom Mealy zum Moore Automaten und umgekehrt?*

- graphisch lösen!

Frage: *Wie überführt man einen Mealy-Automaten in ein Petrinetz?*

- jeder Zustand erhält eine Stelle
- jeder Zustandsübergang wird durch eine die entsprechenden Stellen verbindende Transition modelliert
- jeder einem Anfangszuständ repräsentierende Stelle wird eine Markierung gegeben

Frage: *Beschreiben sie CSP!*

- Ziel: Spezifikation, Entwurf, Verifikationi und Implementierung von Kommunikationsprotokollen
- Grundidee: Zerlegung des Systems in parallel arbeitende Teilsysteme, die miteinander und mit der Umgebung kommunizieren
- Korrektheit mit mathematischen Gesetzen beweisen

Frage: *Erläutern sie SDF!*

- Steuerung durch Verfügbarkeit von Daten, dadurch:
- geeignet für Systeme mit harten Realzeitanforderungen (DSP)
- Systeme mit Teilkomponenten mit unterschiedlichen Datenraten

- Prüfen auf Periodizität → keine unendlichen Datenmengen
- Periodizität gegeben, aber: Verklemmungen möglich
  - Algorithmus von LEE
- Definition: *SDF-Graph*
  - 5-Tupel $G = (V, E, cons, prod, d)$ mit
    - V: Anzahl Knoten
    - $E \subseteq V \times V$
    - $cons: E \longrightarrow N_0$ (konsumierte Marken beim Feuern)
    - $prod: E \longrightarrow N_0$ (produzierte Marken beim Feuern)
    - Anfangsmarkierung $d \in N_0^{card(E)}$
    - Topologiematrix $C = Z^{card(V) \times card(E)}$
      - verläuft e_j von v_i nach v_k, dann sind alle Einträge der Spalte c_j identisch null ausser:
      - $c_{i,j} = -prod(v_i, v_k)$ und
      - $c_{k,j} = cons(v_i, v_k)$
  - Dynamik der Markierungen: $d* = d - C^T \gamma$, wobei $\gamma \in N_0$
  - Notwendige Bedingung: $rang(C) = |V| - 1$
  - Lösen von $C^T \gamma = 0$, falls γ *trivial* → kein periodische Ablauf!
  - Für LEE:
    - minimaler Repetitionsvektor γ^*
      - kleinster,positiver, ganzzahliger Vektor, der wieder zur anfänglichen Tokenverteilung führt
    - zum Auflösen von Verklemmungen (System kann während Aktionen nicht mehr feuern)
    - konstruktives Verfahren:

Frage: *Wie kann man einen Automaten synthetisieren?*

- Zustände binär codieren
- Wertetabelle erstellen und DNF entnehmen
- Boolsche gleichung minimieren (KV-Tafel oder Algebraisch)
- Schaltung entwerfen

Frage: *Was sind die Grenzen von Automaten?*

- Sie können keine Nebenläufigkeit und Hierarchie modellieren.

Frage: *Mit welchen Methoden kann man die Einschränkungen von Automaten aufheben?*

- Statecharts
- Petrinetze

Frage: *Was leisten Statecharts?*

- liefern Zustandsdiagramme
- Hierarchie
- Nebenläufigkeit
- Broadcast-Kommunikation

Frage: *Wie funktionieren Statecharts?*

- Hierarchiebildung: Durch Super- und Subzustände
- Nebenläufigkeit: Durch AND-Dekomposition
- Ereignisse entweder elementar oder logisch verknüpft

Frage: *Wie kann man in Statecharts (nebenläufigen Prozessen) Synchronizität erreichen?*

- Anwendung gleicher Eingabealphabete (?)

Frage: *Wie synthetisiert man Statecharts?*

- Umweg über Automaten, der Produktautomat.

Frage: *Was ist der Unterschied zwischen System- und Rechnerentwurfssprachen?*

- Systementwurfssprachen: für HW+SW (Bsp. SystemC) ($\rightarrow$ HW/SW-Codesign)
  - Algorithmische Ebene
  - Processor-Memory-Switch-Ebene
  - Befehlsebene
  - Aber auch die folgenden Ebenen:
- Rechnerentwurfssprachen: HW (Bsp. VHDL)
  - Register-Transfer-Ebene
  - Logik-Ebene
  - Schaltkreisebene

Frage: *Erklären sie den Aufbau von VHDL!*

- Grobstruktur:

  - *use* - Bereitstellung von Bibliotheken
  - *entity* - Entwurfsobjekt und Schnittstellenbeschreibung
  - *architecture* - Implementierung der entity
  - *configuration* - Zuordnung von architekturen zu entitys

- Feinstruktur:
  - Kernelement ist die entity (spezifiziert die Schnittstelle)
    - Ports, Generics
  - dazu gehören zwei Hauparten der Architektur:
    1. *strukturell*
       - Definition der Schaltung der Unterkomponenten
    2. *verhaltensorientiert*
       - gleichzeitig (immer)
       - sequentiell (mit `process`-Modifizierer)
       - Datenflussbeschreibung (boolsche Ebene)
  - *design entity* = entity + architecture
  - configuration
    - Zuweisung von Architekturalternativen zu Entities
    - Unterstützen Parametrisierung und Flexibilität beim Entwurf
  - Variablen: kein Gegenstück in Hardware, Signale: Hardwareleitungen
  - **Fazit:**
    - mächtige Sprache $\rightarrow$ schwierige Synthese
    - viele Datentypen, starke Typisierung
    - Definition von Konstanten, Variablen, SIgnalen möglich (auch eigene Typen!)

- Vorteile von VHDL:
  - Standardisiert!
  - Langfristigkeit wahrscheinlich
  - umfangreiche Modellierungsmöglichkeiten

Frage: *Erläutern sie den Unterschied von Struktur-, Datenfluss- und Verhaltensbeschreibung in VHDL!*
Frage: *Auf welchen Ebenen existiert die Synthese?*

- Architektursynthese
  - Input: Menge von Prozessen, die über Nachrichten kommunizieren
  - Output: Struktur von Prozessoren, Speichern, Interface-Komponenten

  - ist ein Beispiel für die Struktursynthese

- Register-Transfer-Synthese
  - Input: Beschreibung eines endlichen Automaten
  - Output: Datenpfad und Steuerwerk
  - Datenpfad bestimmt Ein- und Ausgänge
  - Steuerwerk berechnet Zustandsübergänge und speichert Zustände
- Logiksynthese
  - Input: Boolsche Netzliste
  - Output: Gatternetzliste von Bibliothekskomponenten
- Schaltungssynthese
  - Input: Spezifikation einer Schaltung
  - Output: Transistornetzliste (bzw. Layout)

Frage: *Welche Unterscheidung existiert bezüglich des Syntheseergebnisses?*

- *Schaltnetz* - kombinatorische Schaltung
- *Schaltwerk* - sequentielle Schaltung

Frage: *Beschreiben sie wie Technologieabbildung funktioniert!*

- **Aufgabe:**
  - Abbildung einer *technologieunabhängigen* Optimierung einer Boolschen Netzliste **auf** eine vom Hersteller der Zielarchitektur *technologieabhängigen* Zielstruktur
- **Strategie:**
  - Rückführung des Boolschen Netzes auf gemeinsame standardisierte Form
  - Versuch der Überdeckung der Schaltung mit Bibliothekselementen (mit Rücksicht auf die Kostenfunktion)

Frage: *Worin besteht der Unterschied zwischen SystemC und VHDL?*

- SystemC: Systementwurfssprache (HW+SW)
- VHDL: Rechnerentwurfssprache (HW)

Frage: *Wie mächtig sind Petrinetze?*

- Mächtigstes Werkzeug, bildet die Obermenge aller formalen Beschreibungen
- ermöglichen einen *asynchronen* Entwurf

- kann formulieren:
  - Sequentialität
  - Synchronisieren
  - nicht deterministische Verzweigung
  - Ressourcenkonflikt

Frage: *Für welche Fragestellungen eignen sich deterministische endliche Automaten?*

- Modellierung von Schaltwerken oder Steuerungen
- Beschreibung von Systemen mit „Gedächtnis“

Frage: *Wie lassen sich Nebenläufigkeit und Hierarchiebildung mit Statecharts darstellen?*

- mit Superzuständen und Subzuständen, sowie `AND`-Dekomposition
- mehr siehe oben ↑

Frage: *Was sind Produktautomaten (bezüglich Statecharts) und wozu kann man sie einsetzen?*

- enstehen bei Auflösung der `AND`-Dekomposition von Subzuständen
- Zustände des PA gleich der Anzahl aller möglichen Konfigurationen der Zustände (aber gemeinsame Ereignisse können zusammengefasst werden)
- Möglichkeit das Statechart synthesefähig zu machen TODO: stimmt das?

Frage: *Zählen sie die sechs Ebenen zur Rechnerbeschreibung auf!*

- Algorithmische Ebene (Spezifikation eines Algorithmus zur Lösung eines Problems)
- PMS-Ebene (Process, Memory, Switch - Beschreibung durch die Hauptelemente eines Rechners)
- Befehlsebene (Beschreibung durch die Struktur der Befehle)
- Register-Transfer-Ebene (Beschreibung durch Register und Operationen auf diesen Registern)
- Logik-Ebene (Logische Gatter + FlipFlops)
- Schaltkreisebene (Transistoren, Widerstände, Dioden)

Frage: *Welche Aufgaben haben Systementwurfssprachen?*

- Synthese
  - sollen Hilfsmittel für physikalische Beschreibung und automatische Ableitung eines Chiplayouts aus Abstrakter beschreibung ermöglichen
- Spezifikation schaffen
- Validierung eines Systems durch Verifikation
- Austauschbarkeit gewährleisten (→Intellectual Properties)

9.2. Halbleitertechnologie

Frage: *Was besagt Moores Gesetz und wie lange wird es noch gültig sein? Warum?*

- Moore: Verdopplung der Bauelemente auf einem Chip alle 18-24 Monate
- Schätzung: ca. 20 weitere Jahre gültig, aber dann:
  - natürliche Grenze: Leitungen können nicht geringer sein als die Elektronen, die sich darin bewegen

Frage: *Was bedeutet Skalierung?*

- Bauelementeparameter werden um Faktor α>1 skaliert
- Ziel: elektrische Feldstärke konstant halten (typisch $\alpha = 1,4$)
- jedoch: nicht alle Parameter werden skaliert (z.Bsp. Versorgungsspannung)

Frage: *Was bedeutet „interconnection crisis“?*

- zu langsame globale Verbindungen für schnelle Transistoren
  - RC-Konstante bei Leitungen trotz Skalierung konstant
- zu wenige externe Verbindungen für Kommunikation zwischen Baugruppen und integrierten Schaltkreisen
  - quadratischer Anstieg bei Anzahl Bauelemente *vs.* linearer Anstieg der externen Verbindungen

Frage: *Welche Bedeutung hat die Regel von Rent?*

- empirisch gewonnener Ausdruck (wie Moore)
- Antwort auf die Frage:

  „Wie hängt die Pinanzahl mit der Anzahl der Bauelemente zusammen?“

- benötigte externe Anschlüsse P
- Anzahl der Verwendeten Gatter N
- Schaltkreisspezifische Konstanten B und s

$$P = B * N^S$$

- für Prozessoren gilt der „Rent-Koeffizient“ s=0.7 $\rightarrow$ Bedarf an Pins steigt stärker als vorhanden $\rightarrow$ Flaschenhals bei den Chipexternen Verbindungen

Frage: *Welche Gegenmaßnahmen gibt es für die „interconnect crisis“?*

- Laufzeitverzögerung auf „langen Leitungen“
- dazu: Materialien mit geringeren Widerständen verwenden → Verringerung der *RC*-Konstante
- Andere Möglichkeit: System-On-Chip → Möglichst viele Bauelemente auf dem Chip integrieren

Frage: *Was beschreibt die RC-Konstante?*

- Für on-chip Leitung gilt: $t_{line} = RC * \frac{l^2}{2}$
- Bei der Skalierung bleibt *RC* konstant, warum:
  - α>1 und $l_n = \frac{l}{\alpha}, A_n = \frac{A}{\alpha^2}$
  - $R = s_{Materialkonstante} * \frac{l}{A} \longrightarrow R_n = s * \frac{l_n}{A_n} * \alpha \longrightarrow R_n = R * \alpha$
  - $C_n = \frac{C}{\alpha}$
  - Damit ist $R_n * C_n = R * C$, konstant!

9.3. Komponenten eines Rechners

Frage: *Nennen Sie die 7 Prinzipien des URA-Konzepts!*

1. Der Rechner besteht aus 4 Werken
   - Speicherwerk
   - RechenWerk
   - Leitwerk
   - E/A-Werk
2. Die Struktur des Rechners ist unabhängig vom Problem! (*Programmsteuerung*)
3. Programme sind Daten, die andere Daten verarbeiten!
4. Hauptspeicher ist in Zellen gleicher Größe eingeteilt, die fortlaufend adressiert sind!
5. Ein Programm besteht aus einer Folge von Befehlen!
   - *Prinzip der Sequentialität*
6. Es gibt bedingte und unbedingte Sprungbefehle!
7. Es wird das duale Zahlensystem verwendet!

Frage: *Vergleichen Sie CISC/RISC?*

- CISC:

  - Kosten für Software übersteigen die der HW
  - Ansteigende Komplexität durch Hochsprachen
  - Semantische Lücke
    * Die Unterschiede/Konflikte beim Umsetzen von einer höheren Abstraktionsebene auf eine niedrigere, konkretere Ebene.
  - Folge:
    * Anschwellen der Befehlssätze
    * Einbau von Hochsprachenkonstrukten in Hardware
    * Mehrere Adressierungsarten
  - Ziele:
    * Vereinfachung für Compilerbauer
    * Verbesserung der Ausführungseffizienz
    * Unterstützung für Hochsprachen

- RISC:
  - Patterson-Studie (82)
    * Untersuchung bestimmter Eigenschaften bei Ausführung von Hochsprachenprogrammen
      · Operationen, Operanden, Ausführungssequenzen ← wie oft werden diese benutzt
      · Ergebnis: Register für Variablen und Prozeduraufrufe optimieren/minimieren
  - Folge:
    * *Register*: speichern lokaler Variablen in Registern (Speicherzugriffe minimieren) → großen Registersatz schaffen
    * oder in Software: RISC-Compiler müssen Register optimal allokieren (→ Graphfärbungsalgorithmus)
    * *Prozeduren*: wenige Parameter, begrenzte Aufruftiefe, kleine Registermengen
    * Registerfenster für schnelles Umschalten der kleinen Registermengen
  - Eigenschaften:
    * pro Takt eine Instruktion
    * Operationen nur auf Registern
    * nur Lade und Speicherbefehle können auf den Speicher zugreifen
    * wenige, einfache Adressierungsarten
    * wenige, einfache Befehlsformate
    * fixes Befehlsformat

    - keine Mikroprogrammierung
    - erfordert mehr Compilezeit

- CISC: Vor allem auf Kompatibilität, RISC: eher Leistungsstark

Frage: *Was sagt die Patterson-Studie aus und welche Folge hatte sie?*

- Ausführungseigenschaften bezüglich Operanden, Operationen
- Weg vom CISC, hin zum RISC
- HW: Registeranzahl erhöhen
- SW: Registerallokation optimieren (Compiler)

Frage: *Beschreiben sie den Maschinenbefehlszyklus!*

- Befehlsholphase
  - Befehlszähler auslesen und entsprechende Instruktion ins Instruktionsregister laden
- Dekodierungsphase
  - Operationskode dekodieren und Steuersignale generieren
- Operandenholphase
  - holt die Operanden gemäß des Instruktionsbefehls (evtl. parallel zur Dekodierphase)
- Ausführungsphase
  - die Operanden werden in den Registern des Rechenwerksverknüpft
- Rückschreibephase
  - Ergebnisse der Ausführungsphase werden in die vorgesehenen Speicherstellen (Speicher,Register) zurückgeschrieben
- Adressierungsphase
  - evtl. parallel mit einer der vorigen Phasen, Adresse des nächsten Befehlsbestimmen und in den Befehlszähler laden

Frage: *Erklären Sie Pipelining, Superpipelining und superskalares Pipeling!*

- Pipelining
  - überlappende Abarbeitung des Maschinenbefehlszyklus
  - dazu: Maschinenbefehlszyklus aufteilen in Stufen
  - Dauer eines Befehls(Latenz): Gleichbleibend, ABER: Durchsatz wird erhöht (ideal: n-fach bei n-Stufen)
- Superpipelining

  - Pipelining in modernen Prozessoren → Anzahl der Stufen ist mindestens ein dutzend

- Superskalarität
  - Gruppierung von mehreren Befehlen, die gleichzeitig nach dem Pipelining abgearbeitet werden
  - notwendig: mehrere Rechenwerke, Befehlsgruppierer (d. h. Umordnung sequentiell einlaufender Befehle - *dynamische Parallelisierung* zur Laufzeit ← Allgemeines Prinzip bei Superskalaren Rechnern
  - Prinzip von Vektorrechnern entnommen: gleichzeitige Anwendung von Operationen auf einzelne Elemente eines Vektors
    * zwar nicht alle Operationen Vektorops, aber dennoch sind mehrere Rechenwerke besser

Frage: *Welche Leistungssteigerung ist durch Pipeling theoretisch möglich?*

- Anzahl der Pipelinestufen k
- Verzögerung bedingt durch Zwischenspeichern d
- Maximale Verzögerung aller Stufen $\tau_m = max(\tau_i)$ mit $1 \leq i \leq k$
- Zykluszeit $\tau = \tau_m * k + d$ (*bestimmt den Takt!*)
- Gesamtzeit zur Bearbeitung von n Instruktionen

$$T_k = (k + (n-1)) * \tau$$

- Erreichbarer Speed-Up:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k+(n-1)\tau}$$

Frage: *Welche Probleme können beim Pipelining auftreten?*

- Strukurhazards
  - Bsp. zuwenig Speicherports um mehrfachen Speicherzugriff zu ermöglichen
  - Gegenmaßnahmen:
    * stalls einfügen
    * zusätzliche Hardware einbauen
- Steuerhazards
  - Problem bei Sprungbefehlen, wenn die Folgeinstruktion nicht die sequentiell nächste ist

  - bedingte Sprunganweisung: muss die pipelinestufen bis zur auswertung durchlaufen, erst dann entscheidet sich ob „branch taken“ oder „branch not taken“ → auf jedenfall Strafzeit!
  - Gegenmaßnahme: *Spekulative Befehlsausführung*
    * mehrfache Befehlsausführung
    * prefetch branch target - Sprungziel vorher speichern (mit loop buffer)
    * static/dynamic branch prediction - Verzweigungsvorhersage
    * delayed branch - Sprungverzögerung

- Datenhazards
  - ergeben sich aus der Reihenfolge der Ausführung von Befehlen
  - RAW - I2 liest R1, bevor I1 es geschrieben hat
  - WAR - I2 schreibt R1, bevor I1 es gelesen hat
  - WAW - I2 schreibt R1, I1 überschreibt

Frage: *Nennen sie fünf Gegenmaßnahmen bei Steuerungshasards!*

- Spekulative Befehlsausführung (mehrfache Befehlsausführung)
  - Probleme:
    * könnten weitere Verzweigungen in einem Strang sein
    * könnten Strukturhazards auftreten
    * Kosten! Zusätzliche HW notwendig

- loop buffer
  - Idee: Einmal angesprungenes Ziel wird vermutlich häufiger angesprungen (Schleife, Unterprogramm)
  - Pufferspeicher für zuletzt angesprungene Adressen
  - gespeichert wird: angesprungener Befehl und folgende Befehle
  - findet in der Befehlsholephase statt
    * Aus dem instruction cache in den instruction buffer (fifo) und von da in das instruction register
    * Funktion ähnlich Cache (mit dessen Vorzügen), aber es werden nur aufeinanderfolgende BEfehle gespeichert
    * „if then else“ branches befinden sich im puffer → unterstützt schnellem cachezugriff

- delayed branching
  - Idee:

    - CPU soll was sinnvolles machen in der Zeit, wenn stalls vorkommen
    - dazu: die unmittelbare Instruktion wird auf jedenfall in die pipeline genommen und die ausführungsreihenfolge wird erst geändert, wenn „branch taken“

- branch prediction
  - **statisch**:
    - führen keine history über die vergangenen branches
    - „branch always taken“ - immer Sprungzielbefehl laden
    - „branch never taken“ - immer Folgebefehl laden
  - **dynamisch**:
    - „one-bit-prediction“ - in Abhängigkeit des letzten Sprungbefehls agieren
    - „two-bit-prediction“ - Guthabenmethode, Automaten zeichen!
    - branch history table:
      - Verzweigungsadresse (oder gar den kompletten Befehl - speicheraufwendig!) in der tabelle speichern
      - 1-bit/2-bit Verfahren müssen Verzweigungsadresse erst dekodieren, falls „branch taken“ - hier nicht mehr nötig

Frage: *Erläutern sie Datenhazards unter Verwendung eines Beispiels!*

- RAW - I2 liest Operanden, den I1 noch nicht geschrieben hat (Annahme: DIV dauert länger als ADD)

```
DIV R1,R2,R3
ADD R3,R1,R1
```

- WAR - I2 schreibt Operanden, den I1 noch nicht gelesen hat (erst durch Befehlsumordnungen möglich → Superskalarität)
- Beispiel: ADDD kann F0 nicht lesen und auch F8 nicht, aber der SUBD überschreibt F8, bevor ADD es lesen kann

```
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14
```

- WAW - I1 überschreibt Operanden, den I2 früher schon geschrieben hat
- d. h. die letzte Änderung geht verloren, Beispiel: ADDD wird später fertig als LOAD, in folge steht ein „falscher“ Wert in F1

```
ADDD F1,F2,F3
LOAD F1,F4
```

Frage: *Wie funktioniert das Forwarding? Wozu ist es gut?*

- by-pass, load forwarding
- *by-pass*: zusätzliche Rückkopplung von ALU-Ergebnissen an die Eingänge der ALU zur direkten Verwendung in den Folgebefehlen
- *load-forward*: Speicher direkt an die ALU anschliessen (ohne Registerübergang)
- Wozu? hilft bei Datenhasards, neben ($\rightarrow$ Umordnung der Befehlsreihenfolge und $\rightarrow$ dynamischer Ablaufplanung (scoreboards,tomasolu))

Frage: *Erläutern sie Scoreboard genauer!*

- Grundprinzip: *Verwalten von Instruktionsfolgen mit dem Ziel ihre Ausführungsreihenfolge zu ändern*
- Ziel: Anhalten der Pipeline verhindern
- Ablauf: (4-stufig)
  1. Instruktion dekodieren, Strukturhasards erkennen und ggfs. stalls einfügen
  2. Operanden lesen, zuvor jedoch warten bis keine Datenhasards vorliegen
     - Quelloperanden werden nutzbar, wenn eine auf ihn schreibende Instuktion endet, oder wenn das Operandenregister gar nicht benutzt wird
     - Dadurch: Beseitung des RAW-Hasards durch Umstrukturierung der Befehlsreihenfolge
  3. Ausführungsphase
     - bei Multizyklusops an dessen Ende das Scoreboard benachrichtigen
  4. Rückschreibephase
     - Test auf WAR-Hasards, evtl. stalls einfügen

Frage: *Worin unterscheiden sich Scoreboard und Tomasolu-Algorithmus?*

- Scoreboard unternimmt *out-of-order-commit* und erzeugt damit stärkere Tendenz zu WAR und WAW
- RAW-Konflikte werden ausgeschlossen
- Tomasulo nimmt Scoreboard-Prinzip auf und erweitert dies um das Prinzip der *Registerumbenennung*
- durch gemeinsamen Datenbus ist das Ergebnis einen Takt früher verfügbar als beim Scoreboard (forwarding)

Frage: *Wie funktioniert die Tomasulo-Architektur?*

- Wesentlich: Steuerung und Zwischenspeichereinheiten sind verteil
- Mittel: Reservierungsstationen, gemeinsamer Datenbus
- Ablauf: (3-stufig)
  1. Befehl installieren - (BH+DE)
     - Reservierungsstation frei (kein Struktureller Hazard) → Operanden übertragen (*Registerumbenennung*)
  2. Ausführung (BA)
     - beide Operanden nutzbar → starte Ausführung
     - falls nein: CDB (*common data bus*) beobachten
  3. Rückschreiben (RS)
     - Ergebnis über CDB an alle wartenden Funktionseinheiten verteilen
     - busy status für Reservierungsstation entfernen

Frage: *Was bedeutet Registerumbenennung bei der Tomasulo-Architektur?*

- Mechanismus, bei dem Registerreferenzen ersetzt werden durch
  - Zeiger auf Reservierungstationen oder
  - konkrete Daten
- dadurch: Ausschluss von WAW und WAR-Konflikte

Frage: *Was ist ein Cache?*

- Zwischenspeicher, in CPU-Nähe schneller/kleiner
- dient zur Zwischenspeicherung von Befehlen/Daten um schnelleren Zugriff zu gewährleisten
- Zwei Probleme:
  1. *Platzierungsproblem*
  2. *Identifikationsproblem*

Frage: *Beschreiben Sie eine typische Speicher-Hierarchie!*

- CPU-Register, L1-Cache, L2-Cache, Hauptspeicher, Peripherer Speicher
- Speicher der niedrigeren Hierarchiestufe enthält Ausschnitt des nächstgrößeren

Frage: *Nennen sie die drei Organisationsformen von Caches und beschreiben sie kurz!*

- **1-fach-Assoziativ (direct mapping)**

  - jeder Adresse A eines Hauptspeicherblocks wird direkt ein Block B aus dem Cache zugewiesen
  - z. B. $B = A\%N$, wobei N ... Anzahl der Cacheblöcke
- **n-fach assoziative Abbildung**
  - S Cache-Blöcke in s n-Mengen teilen:

$$s = N/n$$

  - n=1: Direkte Abbildung
  - n=N: Vollassoziativ
    * jede Hauptspeicheradresse kann in jeden Cacheblock abgebildet werden

Frage: *Wie wird eine Hauptspeicheradresse im Cache identifiziert?*

- Identifikationsproblem
- Caches haben für jeden Block ein Adress-Tag
- dies enthält die Blockadresse des Eintrags im Hauptspeicher
- Aufbau:
  - `| 24 Bit Tag | 6 Bit Index | 2 Bit Offset |`
  - Zusätzlich noch ein Valid-Bit für die Gültigkeit
  - Offset: Auswahl der Daten im Block
  - Index: bezeichner der Menge im Cache
  - Tag: dient zum Vergleich (Eintrag vorhanden? jalnein)
- n-fach ass. $\rightarrow$ n Komparatoren
- Verglichen wird nur der Adresstag, Index/Block sind ja eindeutig im Cache

Frage: *Was bedeuten Ersetzungs- und Aktualisierungsstrategien bei Caches?*

- Inkonsistenz zwischen Hauptspeicher und Cache $\rightarrow$ Aktualisierung
- *write through*
  - Änderung sofort im Hauptspeicher aktualisieren
  - Konsistenz immer da, aber CPU-Speicherbusbelastung
- *write back*
  - Erst bei Verdrängung findet eine Aktualisierung statt
  - dirty bit flag um unnötiges rückschreiben zu verhindern
- Wann müssen Werte im Cache verändert werden? $\rightarrow$ Ersetzungsstrategie

  - Cache miss tritt auf → aus HS nachladen, CPU anhalten

- Welche Werte werden verändert? (*replacement problem*)
  - Welche Zeile im Cacheblock wird ersetzt?
    * LFU,LRU,FIFO,RANDOM

Frage: *Was können sie über das Verhältnis von Cachegröße zu Cache-miss, also Fehlzugriffen sagen?*

- Zunahme der Cachegröße → Abnahme der Cache misses

Frage: *Wie wird die Leistung von Caches gemessen?*

- mittlere effektive Speicherzugriffszeit

$$T_a = T_h + m * Tm$$

- m . . . miss rate
- T_m . . . miss penalty (Nachladezeit)
- T_h . . . hit time (Cache-Zugriffszeit)
- Optimierung des Caches besteht in der Minimierung von m, der miss rate
  - durch Erhöhung der Cache-Kapazität oder des Grades an Assoziativität
- Andere Optimierung: *prefetching*, vorausgreifendes Laden von Cacheeinträgen

Frage: *Geben sie die Klassifikation von Fehlzugriffen bei Caches an!*

- *compulsory*
  - Kaltstart-miss, erstmaliges Laden vom Block
- *capacity*
  - Cache zu klein um komplette Befehlsfolge zu cachen
  - neben compulsory einzige miss-quelle für vollassoziative caches
- *conflict*
  - in nicht voll-assoziativen caches
  - durch Adresskonflikte werden Blöcke überschrieben

Frage: *Beschreiben sie das Verhaltnis der Cachegrößen bei 2-Ebenen Speicher! (Folie)*
Frage: *Welche Merkmale besitzt die VLIW-Architektur?*

- *very long instruction word*-Architektur
- ebenso wie → Multithreading: Ziel ist die Beseitigung der Beschränkung des Parallelismus bei Superskalararchitekturen

- VLIW: Optimierung zur Compilephase, Erzeugen langer Instruktionen um jeder ALU einen Maschinenbefehl zuzuordnen
- Bsp. Implementierung, EPiC-Architektur:
  - spekulative BA, spekulatives Laden

Frage: *Was bedeutet Multithreading?*

- wie VLIW: Versuch Beschränkungen der Parallelisierung bei Superskalararchitekturen beizukommen ($\rightarrow$ VLIW)
- Programm in unabhängige Teile (*threads*) zerlegen (durch Programmierer oder Compiler)
- pro Prozessor ein thread, lokaler Speicher pro Thread, Globaler Speicher zum Syncen, einfacher Prozesskontext (um Schalten nicht zu verteuern)
- Bsp. Implementierung: Intels Hyperthreading:
  - mehrere logische Prozessoren auf einem physikalischen Prozessor

9.4. Leistungsbewertung

Frage: *Welche Methoden der Leistungsbewertung gibt es?*

- analytische Methoden
  - mathematisch, Warteschlangenmodelle
  - zur Berechnung quantitativer Leistungsmasse wie Mittelwerte, Durchsatz, Verweilzeit
- Simulation
  - Verkehrsverhalten eines Rechners auf einem anderen nachbilden
  - sinnvoll zum Entwurf von Rechnern, oder Auswahl von Konfigurationsänderungen
- Messung
  - beobachtende Leistungsbewertung von einzelnen Komponenten oder ganzen Rechenanlagen
  - monitoring
    - Aufzeichnung von interessanten Sachen mit Messgeräten (ereignis-(un)abhängig)
    - HW/SW-Monitoring (SW schlecht, da Ergebnis verfälscht)
  - benchmarking
    - Reale Programme, Kernels, Spielzeuge
    - synthetische Benchmarks zum Testen spezieller Sets von Instruktionen (wheatstone, dhrystone, linpack, specint, specfp)

9.5. Fehlertoleranz

Frage: *Faseln sie was zur Fehlertoleranz zusammen!*

- Zuverlässigkeit R eines Systems (bedingte Wahrscheinlichkeit, dass es in dem Zeitintervall überlebt)
- Ausfallrate γ
- mittlere Ausfallrate $1/\gamma$
- Verbesserung durch Redundanztechniken
  - Fehlerdiagnose und Behandlung
  - RAID - Fehlertolerante Architektur

10. Rechnerarchitektur Teil 2

10.1. Spezialprozessoren

Frage: *Welche Arten von Prozessoren kennen sie?* **Antwort:** Universalprozessoren, ...

10.2. HW/SW-Codesign

Frage: *Was versteht man unter HW/SW-Codesign* **Frage:** *Erläutern Sie die Ablaufplanung beim Codesig*
Frage: *Welche Algorithmen haben wir bei der Ablaufplanung besprochen?* **Frage:** *In welchen Phasen lä*
Frage: *Erklären sie Logik-, Software-, Hardware- und Systemsynthese!* **Frage:** *Erläutern Sie Allokation,*
Frage: *Was sind nicht-ressourcenbeschränkte Ablaufplanungsalgorithmen?* **Frage:** *Was sind ressourcenb*
Frage: *Wie funktionieren ASAP, ALAP und ILP? Was sind die Unterschiede?*

10.3. Intellectual Property

Frage: *Erläutern sie kurz IP* **Frage:** *Welche Arten von IP gibt es? Welche Unterschiede haben diese?*
Antwort: hard, firm, soft

10.4. Parallelrechner

Frage: *Welche Parallelrechnerstrukturen haben wir kennengelernt?* **Frage:** *Ordnen Sie die Strukturen na*
Antwort: SISD= normaler Rechner, SIMD=Feldrechner (Vektorrechner), MIMD=Multiprozessorsysteme, MISD=indirekt bei spekulativer Befehlsausführung beim Pipelining

Frage: *Was ist ein Feldrechner? Welche Vor- und Nachteile hat er?* **Frage:** *Was ist SIMD/MIMD? Was i*
Frage: *Wie sind Multiprozessorsysteme verbunden?* **Antwort:** Speicher und Nachrichtenkopplung.

Frage: *Wie sieht das Leitwerk bei Vektorrechnern aus?* **Frage:** *Was sind Vor- und Nachteile von Speiche*
Frage: *Warum skaliert Speicherkopplung irgendwann nicht mehr?* **Frage:** *Wie werden Parallelrechner ge*

10.5. Netzwerke

Frage: *Erklären Sie Aufbau, Nachteil und Lösung bei mehrstufigen Netzen!* **Frage:** *Wie funktioniert ein Kreuzschiene*
Frage: *Erläutern sie die Arbeitsweise des Benes-Netzwerks!* **Frage:** *Was ist ein 2x2-Verteiler?*
Frage: *Leiten sie den n-dimensionalen Hypercube grafisch her!* **Frage:** *Was ist der Vorteil bei der Hypercube-Vernetzu*
Antwort: Vorteil: max. Weglänge d, mehrfache Wege zwischen zwei Knoten; Nachteil: Knotengrad = d - (d - #dimensionen)

Frage: *Beschreiben sie den Sunshine-Switch!* **Antwort:** Batcher-Sortiernetzwerk wichtig für Banyon, weil es dann keine Konflikte mehr gibt.

Frage: *Was ist Wellenlängenmultiplexing?*

10.6. Verteilte Systeme