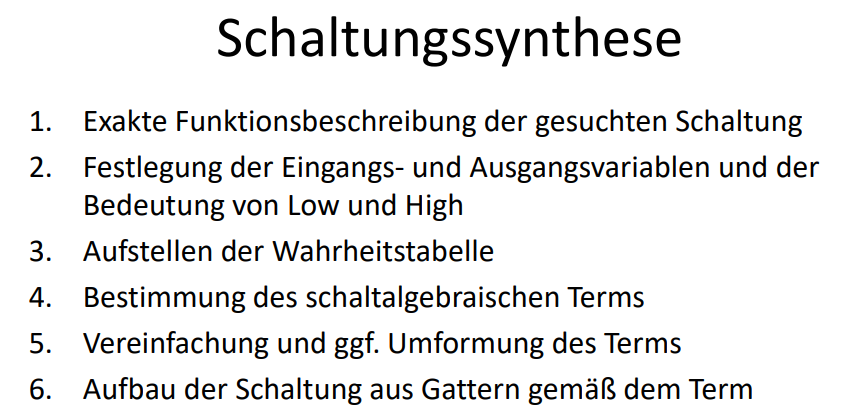
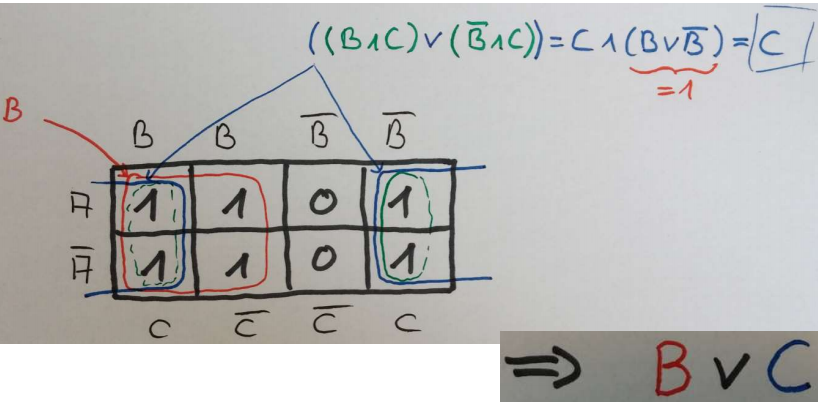
### Schaltungstechnische Grundlagen:

* Signale zum Übertragen von Informationen als Zahlenwerte: Sequenz von Symbolen
  + wertdiskret: Werte werden durch Norm auf bestimmte Werte „gerundet“
    - z. B. abrunden auf nächste ganze Zahl (Uin = [1V, 2V[ 🡪 Uout = 1V)
  + zeitdiskret: Takt, Werte werden zu bestimmten Zeiten ausgelesen, Übertragung 0 und 1
    - verhindert falsche Ausgaben, da nur zu festem Zeitpunkt gelesen wird
  + wert- und zeitdiskret: moderne Rechner
* Binäre Signale
  + digitale Signale**: TTL-Technik**: Low-Bereich = 0 – 0.8V, High-Bereich = 2-5V
  + H/L enthalten Information von 1 Bit: werden als 0 oder 1 interpretiert
    - manche Schaltungen interpretieren H als 0, andere H als 1
* Wahrheitswertetabelle
  + Aufführen aller Eingaben (a, a‘, b, b‘)
  + Aufteilen der Rechnung
  + Zusammenführen der Rechnungen
  + **Thautologie**: alle Ergebnisse = 1
* Elementare Gatter:
  + Konjunktion: AND-Gatter
  + Disjunktion: OR-Gatter
  + Negation: NOT-Gatter
* wenn die 3 Elementaren Gatter durch eine Operatormenge dargestellt werden können, ist sie **vollständig** und damit eine **Verknüpfungsbasis**

*Termerstellung:*

* Alle Eingangswertekombinationen in Wertetabelle aufstellen
* DNF: **disjunktive Normalform**
  + alle **Schaltfunktionen mit Ergebnis 1** disjunktiv (OR) verknüpfen
  + einzelne Schaltfkt.: Eingangsvariablen mit AND verknüpfen
  + wenn in Wahrheitswertetabelle mehr 0 als 1
* KNF: **konjunktive Normalform**
  + alle **Schaltfunktionen mit Ergebnis 0** konjunktiv (AND) verknüpfen
  + einzelne Schaltfkt.: Eingansvariablen mit OR verknüpfen (Erg. muss 0 werden)
  + wenn in Wahrheitswertetabelle mehr 1 als 0
* Kürzung von DNF und KNF führt zu Minimalformen: Aufbau der Schaltung
  + Karnaugh-Veitch-Diagramm:
    - Eintragen von Schaltfunktionen in Tabelle
    - Zusammenfassen von nebeneinanderliegenden Paaren
    - abgedeckte Felder pro Schleife sind 2er-Potenzen

### Schaltnetze

* Entwurfsziele:
  + minimale Durchlaufzeit
  + minimaler Ressourcenverbrauch
  + Aufbau mit einzelnen Verknüpfungsgliedern
  + Aufbau mit bestimmtem Gattertyp (z. B. NOR oder NAND)
  + Aufbau mit addressierenden Bauelementen (Mutlitplexer, PROM, ...)
* Bauteile:
  + Multiplexer: wählt Eingangssignal aus, zusätzliches Eingagssignal aktiviert gewünschten Eingang
  + 2-Bit Komparator: Ausgabe Größer, Kleiner & Gleich, vergleicht einzelne Bits
* Don’t Care’s
  + zum Verkleinern von Tabellen (z.B. WE = 1, alle anderen Eingänge irrelevant = X)
* Enable-Signal
  + Schaltungen lieferen immer einen Wert, Problem bei Masseverbindung 🡪 Kurzschluss
  + Enable-Signal koppelt Signale ab, meist active low (0 aktiviert, 1 deaktiviert)

*Rechenschaltungen:*

* Addition
  + **Halbaddierer**: Addierer, die keinen eingehenden Übertrag berücksichtigen
  + **Volladdierer**: Addierer mit Übertrag, kann aus 2 Halb-Addierern + OR-Gatter gebaut werden
  + **Ripple-Carry-Addierer**: Aneinanderreihung von Volladdierern, Übertrag immer nach Addition
    - lange Laufzeit: 2n für finalen Übertrag, 2n+1 für finales Ergebnis
    - langsamer als Paralleladdierer, aber deutlich weniger Gatter
  + **Paralleladdierer**: einsetzen der Übertragsberechnung in nachfolgende Formeln
    - stark steigende Anzahl Terme (2n-1) und Eingänge (n+1)
    - durch hierarchischen Aufbau, da nur 5 Eingänge/Gatter mögl.
    - Tiefe des Baums: d = log5(2n-1)
    - schneller als RCA, aber deutlich mehr Gatter
  + **Carry-Look-Ahead Addierer**: Aufteilung von Berechnungen, gleichzeitige Berechnung Übertrag & Summe
    - Übertrag generiert: Übertrag entsteht an Stelle k, wird bis Stelle n durchgereicht
      * ak \* bk = 1 k < i < n: ai + bi = 1; g(0, n-1) = 1
    - Übertrag propagiert: Übertrag besteht schon bei Stelle 0, wird bis n durchgereicht
      * C0 = 1 0 ≤ i < n: ai + bi = 1; p(0, n-1) = 1
    - Cn = g(0, n-1) + (p(0, n-1) \* C0) 🡪 entweder an Stelle 0 generiert oder propagiert & davor vorhanden
    - Ablauf:
      * zuerst werden g’s & p’s berechnet, immer größere Intervalle
      * danach Übertrag von Anfang bestimmt
      * von größtem zu niedrigster Stelle Überträge bestimmen
      * Bestimmung Summe
  + **Carry-Select-Adder**: Additionen werden doppelt durchgeführt 🡪 mit/ohne Übertrag
    - Auswahl durch Multiplexer, welches Ergebnis genommen wird
    - Ergebnisse werden in Blöcken berechnet, pro Block 1 Bit mehr, Übertrag an nächsten Block
    - k Addiererblöcke, n bits, n1 Bit’s des ersten Addierers:
* Subtrahierer: Addition des 2er-Komplements
* Multiplizerer: wie in Schule, verrücken der Addition um jeweils eine Stelle
* Division: durch Subtraktion
  + wenn Ergebnis einer Subtraktion negativ ist, wird im Folgeschritt addiert 🡪 Ergebnis stimmt wieder

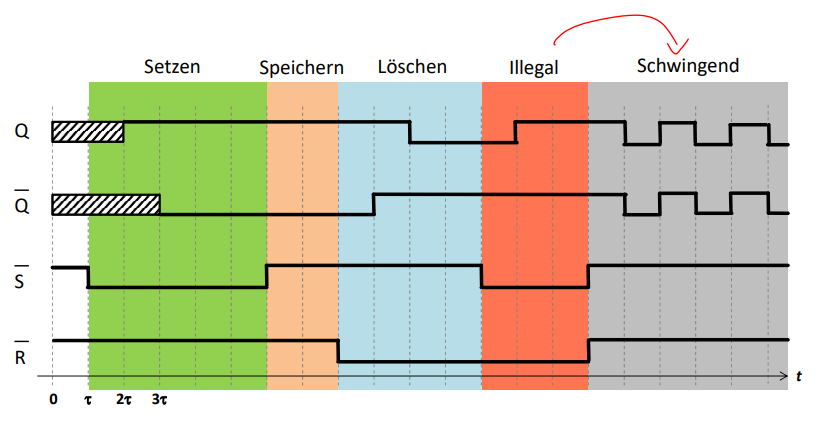
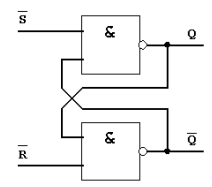
*Aufbau:*

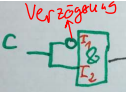
* diskreter Aufbau: nur für kleine Schaltnetze sinnvoll
* Speicher: Abbild der Wahrheitswertetabelle, sehr langsam & groß
* PLAs: Programmable Logic Arrays, komplexe Funktionen, extrem aufwändig

### Kippschaltungen

*Rückkopplung:*

* Informationsfluss nicht nach vorne gerichtet, sondern mit Rückführung
* FlipFlop: Kippglied, zum Setzen, Rücksetzen oder Speichern von Daten

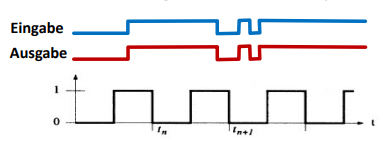
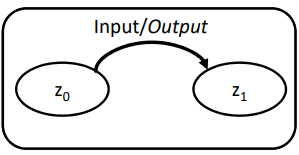


* + **RS-FlipFlop**: Eingaben S & R: 00 = speichern, 01 = löschen, 10 = setzen, 11 = illegal
    - mit Entprellschaltung möglich: beim Umschalten prellt Schalter, FlipFlop speichert
  + **Delay-FlipFlop:** getaktetes FlipFlop, nur eine Eingabe, ***Register***
  + **asynchroner Reset**: getaktetes FlipFlop, Takt wird überschreiben solange SD und RD != 1 1
* **Takt**: Zustandsänderung auf Speicher nur bei clk-Signal
  + spezifischer Zeitpunkt: Speicheränderung nur bei steigender Flanke, NOT verzögert 🡪

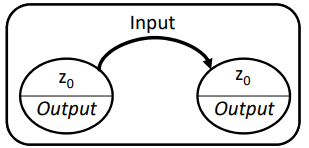
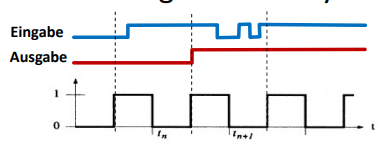
### Schaltwerke

*Automaten:*

* Mealy-Automat
  + Ausgabe reagiert sofort auf sich veränderte Eingabe, auch wenn Zustand noch nicht geändert



* Moore-Automat
  + Ausgabeverhalten hängt nur vom Zustand ab
  + reagiert auf Veränderung der Eingabe erst nach Aktualisierung des Zustands



* Vorgehen
  + Definition Eingangs- und Ausgansvariablen, Festlegung Zustandsmenge und Anfangszustand
  + Erstellung des Zustandsgraphen (Automat)
  + Wahl einer Zustandskodierung
    - logarithmisch: Kodierung ist Binärzahl
    - One-Hot: jeder Zustand ist 1 Bit, bei 4 Zuständen: 1000, 0100, 0010, 0001
    - Gray-Code: zwischen benachbarten Werten ändert sich nur 1 Bit
  + Erstellung Übergangsfunktionen (Zustandsübergangstabelle)
  + Erstellen der Ausgabefunktion
  + DMF/KMF, Gatter-Implementierung

*Aufbau Schaltkreis:*

* **PROM**: Speicherbaustein, besteht aus Wahrheitswerte-Tabelle, **P**rogrammable **R**ead-**O**nly **M**emory
* Arithmetische Ausdrücke
  + an Übergangskanten im Automaten können **[relationale Ausdrücke]** übergeben werden
    - 1 oder 0 wird übergeben, je nachdem ob Ausdruck wahr oder falsch
  + werden in einem Register gespeichert 🡪 Berechnung ob relationaler Ausdruck wahr oder falsch
  + Vereinfachung
    - von Berechnungen durch RegisterFile und Multiplexer
    - RegisterFile-Größe wird auf 2er-Potenz gehoben zur Ansteuerung
    - Zustände automatisch hochzählen
    - Branch-Unit zur Berechnung von Flags
    - Benennung der Zustände, Konstanten & Register vereinfachen
* Assembler-Programm / PC gebaut

### Elementare Rechnerarchitektur

*Speicherbaustein:*

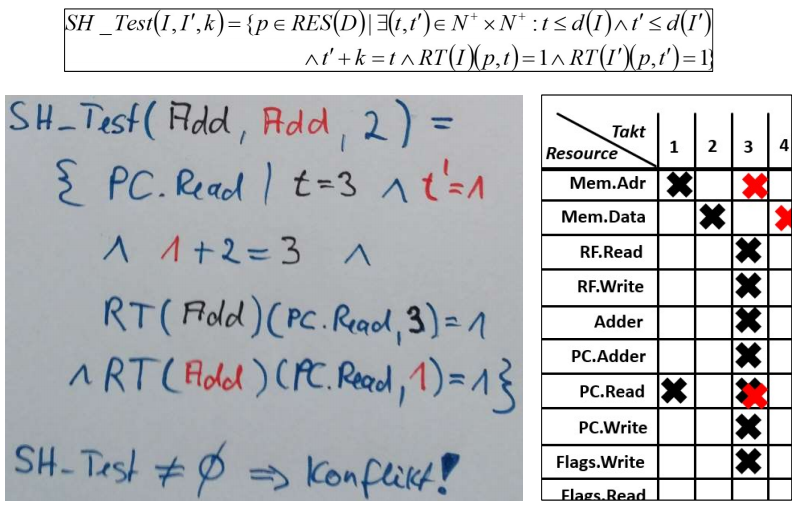
* Ansteuerung: 1. Zyklus übergibt Addresse, 2. Zyklus übergibt Datum
* während der Ansteuerung des Speicherbausteins wird PC blockiert
* Befehlsphasen werden aufgeteilt
  + Instruction Fetch: Befehl aus dem Speicher holen
    - Phase 1: Ansteuerung der Adresse
    - Phase 2: Daten aus der Adresse an Befehlsregister übergeben
  + Execute Phase: Befehl durchführen (1 Phase für Reg -> Reg, 2 Phasen für Reg -> Speicher)
    - Phase 1: Adresse berechnen und an Speicher geben
    - Phase 2: Daten berechnen und an Speicher geben

*Taktfrequenz:*

* maximale Taktfrequenz = Dauer des längsten Pfades von Speicherelement zu Speicherelement
* verschieden Kenngrößen:
  + Kennzahl = Σ |Anteil Befehle für Bsp.Programm| \* |Takte für Befehl| \* Taktzeit
    - durch Vergleich der Kennzahlen kann der Speedup für ein Beispielprogramm bestimmt werden
  + **CPI** (**C**ycles **P**er **I**nstruction): beschreibz Anzahl der Takte die ein Befehl benötigt
    - CPIP = Anzahl der Takte für Ablauf / Anzahl der abgearbeiteten Befehle des Ablaufes = NP / iP
    - CPIi: Anzahl der Takte eines Befehls i
    - CPIA: durchschnittlicher CPI einer Architektur A
  + MIPS / MFLOPS: Leistungsfähigkeit von Prozessoren
    - Bewertung in Benchmarks
    - MIPSA = mit = Taktfrequenz

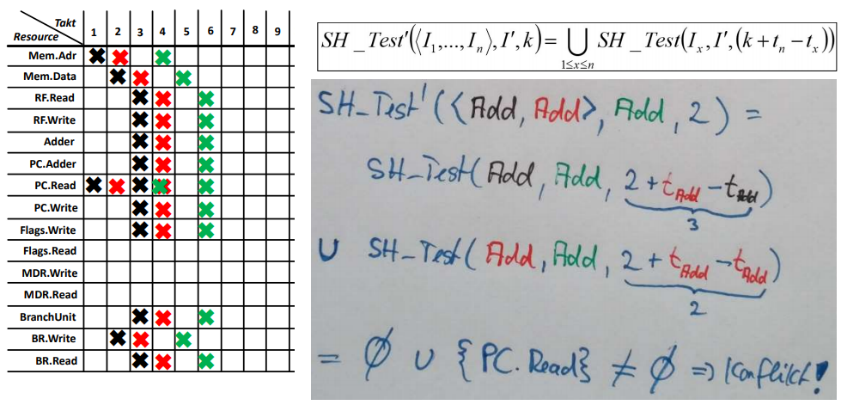
### Pipelining

Verschränkte Ausführung von Befehlen: statt sequenziell 🡪 überlappend

* Voraussetzung: keine überlappende Ausführung von Ressourcen
  + RES(D): Menge der Ressourcen
    - Schaltnetze, Speicher Lese- und Schreibeingänge sind einzeln, PROM
  + INST: Menge der Instruktionen, I ϵ INST, d(I) = Ausführungsdauer in Zyklen
* Reservierungstabelle:
  + einzelne Instruktionen: RT(I)(p, t) = 1 🡪 p = Ressource, t = Zyklus/Takt
  + Instruktionen werden in Tabelle zusammengefasst
  + zeigt in welchem Zyklus/Takt des Befehls welche Ressource belegt wird
* SH\_TEST: Test auf **s**trukturelle **H**azards
  + SH\_TEST(I, I‘, k) k = Abstand I zu I‘
  + t ≤ d(I) t‘ ≤ d(I‘) t‘ + k = t p ϵ RES(D)
  + SH\_TEST(I, I‘, k) = {p | t ꓥ t‘ ꓥ t‘ + k = t

ꓥ RT(I)(p, t) = 1

ꓥ RT(I‘)(p, t‘) = 1}

* + - RT-Funktionen aus Reservierungstabelle
    - wenn SH\_TEST nicht leer 🡪 Konflikt
    - oder für Befehle in Pipeline
  + MIPS gehen mit Pipelining nach oben 🡪 weniger Takte pro Befehl, gleiche Taktfrequenz
  + SH\_TEST‘(<I1, ..., In>, I‘, k) 🡪 kollisionsfreie Sequenz von Befehlen wird verglichen mit anderem Befehl
* Kollisionen / Hazards entstehen, wenn ein Befehl eine Ressource in mehr als einem Takt verwendet
  + Mehrfaches zugreifen verhindern, indem Befehle kombiniert werden 🡪 aktuelle Daten in Ressource für Nachfolgebefehle
* Beseitigung struktureller Hazards:
  + Replikation: Ressource wird mehrfach benötigt 🡪 jede Phase bekommt eigene Ressource
  + Harvard-Architektur: Mehrfachzugriff auf Speicher 🡪 trenne Daten- und Programmspeicher
  + Register: Daten müssen einen Takt aufbewahrt werden 🡪 lagern in neuem Register
    - z.B. bei Berechnung Adresse & Datum für Speicher: zuerst Adresse, Datum im nächsten Takt

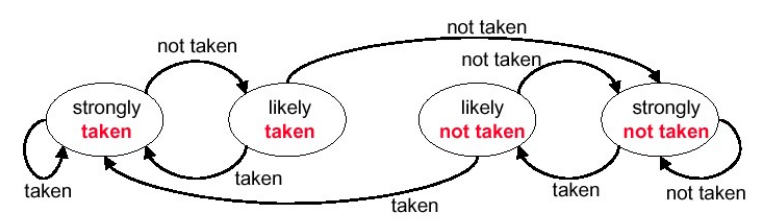
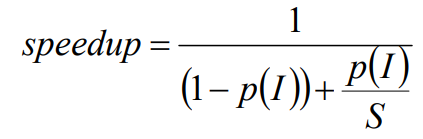
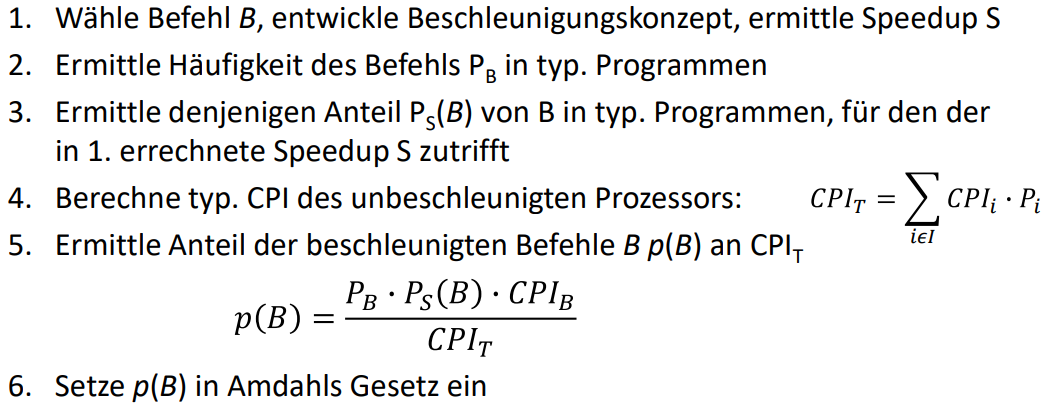
*Controller:*

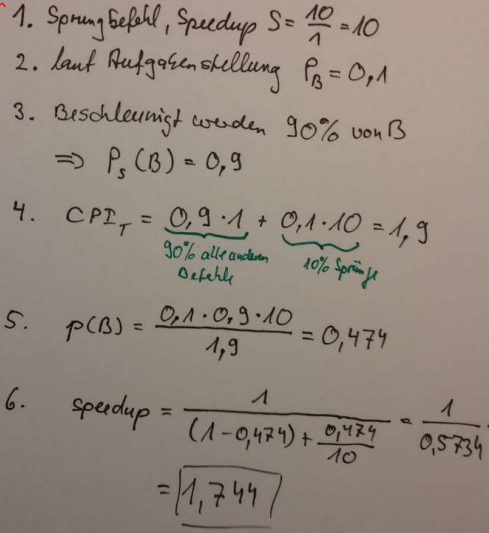
* Datenpfad wird in Stage’s unterteilt
  + auf jede Stage wird exklusiv zugegriffen
    - bei längerem Zugriff einer Instruktion auf Stage werden Folgeinstruktionen aufgehalten
  + jeder Stage werden Ressourcen zugeteilt
  + Stages bekommen Namen
  + Stage schreib Ergebnis in Register, nächste Stage greift auf Register zu
* Ansteuerung von Stages im Controller 🡪 verschieden Befehle in unterschiedlichen Stages überlappend
  + Ansteuerung ist Automat mit n Befehlen und k Pipelinestufen: **nk+1-1** Zustände
  + muss aufgeteilt werden, da bei vielen Befehlen Automat zu groß wird
    - Time Stationary: Controller in jeder Stage 🡪 Anweisung aus Befehlsreg. vor Stage
      * Nachteil: Dekodierung in jeder Phase, Zeitverlust
    - Data Stationary: Dekodierung in 2. Stage 🡪 über Register durchreichen zu entsprechender Stage
      * Nachteil: mehr Bauteile, aufwändig bei Konflikten, Interrupts und Multithreading
  + Speedup beim Pipelining: k Pipelinestufen (Stages), n Iterationen (verschachtelte Befehle)
* tiefe Pipelines sind egal, solange CPI = 1 🡪 Vorstellung wegen Speicherzugriff & Sprungbefehlen utopisch

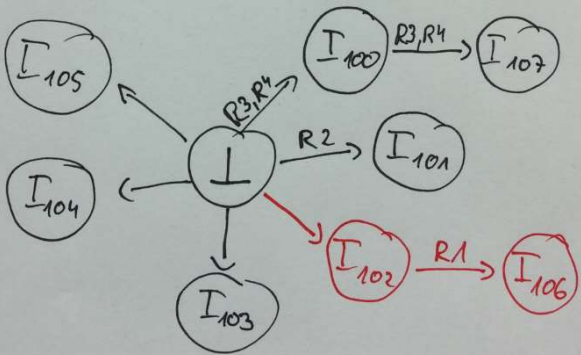
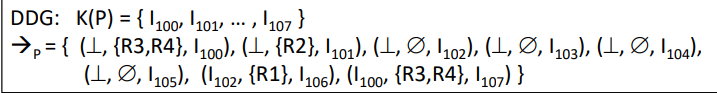
*Instruction Set Architecture (ISA):*

* definiert Sicht des Programmierers auf:
  + Programm-Register
  + Zugriff, Typ und Größe von Operanden
  + Befehlssatz und -codierung
  + Adressierungsarten
* Beispiele: RISC, CISC, DSP ...
  + CISC: einheitlicher Befehlssatz, 80% der Programme benutzen nur 5-10% des Befehlssatzes
  + RISC: für Pipelining und Speicherzugriff besser, wegen hoher Anzahl an Registern

*Hazards:*

* Structural Hazards: Mehrfachzugriff auf einzelne Ressourcen
* Data Hazards: Datenabhängigkeit eines Befehls von Vorgängern
  + Notbremsen-Technik: konsumierender Befehl wird verzögert bis Ergebnis vorhanden
  + Compiler-Optimierung: ordne Befehle um, so dass zwischen problem. Befehlen genug Takte liegen
  + Forwarding: Ergebnis zwischen Stages kommunizieren
    - Ergebnisse werden von allen Stages in OF Phase geladen, jüngster Befehl 🡪 höhere Priorität
* Control Hazards: Abhängigkeit von Sprungbefehlen von vorangegangen Befehlen
  + Delay-Slots: Hochzählen von Schleifen in Downtime (während Flag’s überprüft werden)
  + Brach-Prediction: Vorhersagen, ob Schleife wiederholt wird oder nicht, z.B. 2-Bit-Prädikation
* Ahmdals Gesetz: Berechnung des speedups



* Leistungsbewertung von Prozessoren abhängig von ausgeführten Programmen
  + soll aber vergleichbar sein: Benchmarks (typische Programme)
* Compiler-Optimierung zur Vermeidung von Data Hazards
  + Data Dependency Graph 🡪 Abhängigkeiten zwischen Befehlen feststellen
  + so umordnen, dass Befehle, die abhängig voneinander sind, genug Abstand haben

### Speicher

*DRAM:*

* elektrisches speichern von Informationen (Schaltung: Kondensator): Stack oder Trench Technologie
* Zellen Matrix angeordnet: Zeile wird ausgelesen & Spalte dann ausgewählt
  + Kondensator verliert bei Lesevorgang Ladung und muss neu aufgeladen werden, refresh in regelmäßigen Abständen
* Beschleunigungstechnologien:
  + Burst-Mode interner Zähler erhöht Spaltenadresse automatisch
  + Synchron (SDRAM) Synchron zu externem Takt, internes Interleaving
    - Signale bei steigender Taktflanke gültig
    - kann im Burst-Mode arbeiten
    - arbeitet mit Pipelining 🡪 in jedem Taktzyklus neue Spaltenadresse
    - Nomenklatur: PC-xxx CL a-b-c
      * xxx = Taktfrequenz
      * a = CAS-Latenz von fallender Flanke bis zur Ausgabe der Daten (tCL) (**Read**)
      * b = RAS-zu-CAS-Verzögerung minimale Zeit zwischen RAS und CAS (tRCD) (**Activate**)
      * c = RAS-Vorladezeit Zeit zum Beenden des Zugriffs und vorbereiten (tRP) (**Precharge**)
  + Double Date Rate (DDR) Daten bei fallender & steigender Flanke übertragen
    - Erweiterung SDRAM: jeder Zugriff liest 2 benachbarte Bits aus, 2 Datenworte / Takt
    - Nomenklatur: DDR-xxx
      * xxx = doppelte Taktfrequenz
      * **CL a-b-c** optional
* Bauformen: SIMM und DIMM
* Zugriff auf Speicherzellen hat hohe Verzögerung, hohe Taktfrequenz erst nach Prefetch spürbar

*Speicherhierarchie:*

* Hierarchie:
  + Register: schnelle Speicher im CPU, geringe Anzahl, teuer: min. 28 Transistoren / Bit
  + Cache: auf CPU oder extern oder beides, mittelschnell, 6 Transistoren / Bit
  + RAM: extern verbaut, besondere Sequenzen von Ansteuerung, 1 Transistor + 1 Kondensator / Bit
  + Hintergrundspeicher: Festplatte, hohe Informationsdichte, nicht flüchtig
* Lokalität:
  + zeitlich: falls Datum oder Instruktion referenziert 🡪 bald wieder
  + örtlich: falls Datum oder Instruktion referenziert 🡪 nahegelegene Daten & Adressen auch
* Zugriffsverhalten:
  + Register und SRAM schnell wahlfrei, DRAM und Festplatten schnell blockweiße

*Cache:*

* kleine Speichermenge, verschieden Level: L1, L2, (L3) 🡪 höhere Stufe = größer, aber langsamer
* Assoziativ-Speicher: Content Addressable Memory = CAM
  + k-Bit Schlüssel: Suchmaske (relevante Bits) + Suchschlüssel für die Suche nach Datum
  + m-Bit Datenfeld: Inhalt der Speicherzelle im Hauptspeicher
  + Tag zur Auswahl: Größe t = w – d Bit, niedrige d Bit nicht im Tag enthalten 🡪 d für Auswahl Datum
  + Bei 1. Schleifendurchlauf wird Cache befüllt, bei 2. Durchlauf sind alle Instruktionen und Daten vorhanden & Speedup ist sehr groß
  + nur für kleine Caches, sonst Aufwand zu hoch und Trefferbestimmung langsam
* Direct-mapped-Cache:
  + Einteilung des Hauptspeichers in gleich große Segmente (Segmentgröße = Cachegröße)
  + jede Hauptspeicherzeile kann nur direkt in bestimmte Cachezeile geladen werden
    - bei Konflikt wird Zeile erst freigegeben, dann belegt
  + Tag zur Auswahl: Größe t = w - k - d Bit, k = Zeilenanzahl Cache & Indexgröße
  + Index: Zur Auwahl der Cachezeile Tag: Auswahl des Segments
  + wenn mitten in Zeile eingegriffen wird 🡪 Critical Word First, dann Round Robin
  + Ladeabbruch wenn Daten nicht benötigt werden kann implementiert werden
  + bei 1. Schleifendurchlauf befüllt, bei 2. Durchlauf nur Neubefüllung der überschriebenen Zeilen
* n-Wege assoziativer Cache
  + Mischfrom: direct-mapped Cache mit n Partitionen 🡪 beliebige Partition, feste Cachezeile
    - bei Kollision wird andere Partition ausgewählt
  + Berechnung Tag, Index und Byteauwahl wie bei direct-mapped Cache
  + 2. Durchlauf nur Neubefüllung benötigter Zeilen
* Verdrängungsstrategien: welche Line soll überschrieben werden (n-Wege ass., assoziativ)
  + Random zufällig 🡪 pseudo-random
  + RoundRobin letzte überschrieben line merken, nächste in zyklischer Reihenfolge
  + **LeastRecentlyUsed** je länger line nicht benutzt, desto älter 🡪 älteste wird überschrieben
* Verhalten bei Cache-Hit (ändern der Daten im Cache)
  + Write-Through Schreibzugriff in Cache und darunter (RAM, Festplatte)
  + Write-Back Schreibzugriff im Cache, erst bei Verdrängung wird darunter geändert
* Verhalten bei Cache-Miss (Datum nicht in Cache)
  + Write-Allocate Line wird in Cache geladen und Datum wird geschrieben
  + No-Write-Allocate Datum wird in darunter liegenden Hierarchieebene beschrieben, Line nicht in Cache
* Kombinationen:
  + Write-Throug mit No-Write-Allocate
  + Write-Back mit Write-Allocate