Kapitel 0

# Einführung

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Einleitung

- Informatik beschäftigt sich mit Information, ihrer Analyse und Verarbeitung mittels Algorithmen.
- Algorithmen werden in der Regel als Software implementiert, die auf (programmierbarer) Hardware läuft.
- Was genau ist Hardware und wie interagiert sie mit Software? Wie sieht das „Gesamtsystem" aus?
  - Gesamtsystem :=
    Die Gesamtheit von Komponenten, die miteinander durch Beziehungen verbunden sind und gemeinsam eine Funktion erfüllen.

# Technische Informatik

- ...beschäftigt sich allgemein mit Hardware.
  - Programmierbare Universalrechner
  - Programmierbare Eingebettete Rechner
  - Spezialhardware
- Dabei sind wichtig
  - Theoretische Grundlagen zum Entwurf von Hardware,
  - die Methodik zum Entwurf von Hardware,
  - der Aufbau von Rechnern,
  - ...

# Beispiele

- Universalrechner (general purpose computer), PC, Arbeitsplatzrechner
- eingebetteter Rechner in Auto, Flugzeug, Waschmaschine, Fernseher, CD-Spieler, Spielkonsole, Navigator

# Marktanteile

Universalrechner $\approx$ 2%
Eingebettete Rechner $\approx$ 98%

# Verstehen / Entwerfen von (Rechen-)Systemen

- Wie versteht / entwirft man ein System, welches aus sehr vielen Komponenten besteht:
  - einem Rechner, der aus 100 Millionen Transistoren besteht
  - darauf laufender Software, Betriebssysteme, Übersetzer, Datenbanken, Kommunikationssysteme, Anwendungs-programme mit Millionen Zeilen Programmcode
- Erkennen der Struktur:
  - Aufbau aus Komponenten
  - Zusammenwirken dieser Komponenten zur Realisierung ihrer Funktion

# Inhalt der Vorlesung

**Teil 1:**
Entwurf und Analyse eines einfachen Rechners

- Mathematische Grundlagen
- Abstrakter Rechner „ReTI”
- Kodierung von Zeichen, Zahlen
- Kombinatorische Logik
- Sequentielle Logik
- Formale Spezifikation und Verifikation von Hardware
- Fehlertoleranz und -erkennung

# Inhalt der Vorlesung

**Teil 2:**
Architekturkonzepte und ihre Umsetzung

- Befehlssätze
- Pipelining
- Speicherbeschleunigung
- Parallelverarbeitung
- Spezialarchitekturen

# Literatur

- Bernd Becker, Paul Molitor
  „Technische Informatik: Eine einführende Darstellung"
  Oldenbourg Wissenschaftsverlag München,
  2008, ISBN 978-3-486-58650-3, 435 Seiten.

- Jörg Keller, Wolfgang J. Paul
  „Hardware-Design:
  Formaler Entwurf digitaler Schaltungen"
  Teubner, 2005.

# Ergänzende Literatur


- A. Tanenbaum
  Structured Computer Organization. Prentice Hall International, 2005.
- D. Patterson, J. Hennessy
  Computer Organization & Design - The Hardware/Software Interface. Morgan Kaufmann Publischers, 2008.
- J. Hennessy, D. Patterson
  Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, 2011.
- R. Drechsler, B. Becker
  Graphenbasierte Funktionsdarstellungen. Teubner, 1998.
- P. Molitor, C. Scholl
  Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen. Teubner, 1999.
- G. Hotz
  Einführung in die Informatik. Teubner, 1990.

# Rechnerhistorie

- Historische Entwicklung
  - von den Anfängen zum Multi-Core
  - Chips mit 2 Milliarden Transistoren
  - Probleme und Herausforderungen

# Wirtschaftliche Treiber der Computerentwicklung

- **1930-60**: Wenige, sehr teure Rechner.
  - Wettervorhersagen, militärische Anwendungen
- **1960-80**: EDV für die Infrastruktur.
  - Banken, Verkehrs-, Energiesysteme
- **1980-2000**: Rechnerbasierte Konsumgüter.
  - PCs, Unterhaltungselektronik, Autos, Handys
- **Heute**: Ubiquitous Computing.
  - Smartphones/Internet Devices, Geräte zur medizinischen Überwachung

# Technischer Fortschritt als Basis der Entwicklung

- Hardware-Fertigungstechnologie
  - Moore's Law: Verdoppelung der Anzahl Transistoren pro Flächeneinheit alle 18 Monate
- Neue Algorithmenklassen
  - Bilderkennung, schnelle Kommunikation, Data Mining / Suchmaschinen, WEB 2.0
- Neue Entwurfsverfahren
  - Entwurf deutlich komplexerer Systeme in gleicher / unterproportional erhöhter Zeit

# Zuse Z1 / Z3 (1934-1941)

- Bis zu 10 Hz Taktfrequenz
- 64 Speicherzellen der Breite 22
- 4000 Watt Leistung
- Gewicht: 1000 kg
- Programmierung über gelochte Filmstreifen

# Historische Entwicklung: ENIAC (1946)

- 30.000 kg schwer, 3 m hoch, 24 m breit
- 17.648 Vakuumröhren, 70.000 Widerstände, 1.500 Relais, 10.000 Kondensatoren
- 150.000 Watt Leistung
- Multiplikationszeit: 3 ms
- Programmierung über Schalttafeln

# Eniac

# Historische Entwicklung: Zitate

- Thomas Watson (IBM, 1943):

  *„I think there is a world market for maybe five computers.”*

- Popular Mechanics (1949):

  *„Computers in the future may weigh no more than 1.5 tons.”*

- Ken Olsen (DEC, 1977):

  *„There is no reason for any individual to have a computer in their home.”*

# Heutiger Stand: Mikroprozessoren

- 2000: Pentium 4
  - 42 Millionen Transistoren
  - Taktfrequenz $> 1,5\,\text{GHz}$
- 2011: Intel Sandy Bridge (i7)
  - $\approx 2$ Milliarden Transistoren
  - Taktfrequenz $\approx 4\,\text{GHz}$
- „Multi-Core” (bis zu 8 Kerne)
  - Parallelausführung mehrerer Programme
  - Parallelausführung von Teilen desselben Programms

# Fortschritt der Halbleiterfertigung: Moore's Law

- Verdoppelung der Transistor-Dichte alle 18 Monate.
  (Gordon Moore, Mitbegründer von Intel, 1965)

# Moore's Law

Kein Gesetz, sondern Voraussage, was technologisch möglich sein wird!

# Transistor

Transistor

Grippevirus

- Intel arbeitet inzwischen mit 22 nm Technologie (Ivy Bridge)

# Halbleiterfertigung im Überblick

- Siliziumkristalle werden gezüchtet und in dünne runde Scheiben (Wafers) geschnitten.
- Transistoren werden auf einen Wafer gefertigt (Frontend).
  - Transistoren bilden Logikgatter
- Metallische Verbindungsleitungen (Interconnects) werden über den Transistoren gefertigt (Backend).
  - Interconnects verbinden Logikgatter miteinander und stellen Stromversorgung bereit.

# Einige Fakten über Halbleiterfertigung

- Eine Intel-Fab neuester Generation kostet insgesamt ca. 3 Milliarden US-Dollar.
  - Die teuersten jemals gebauten Fabriken
  - Über 1.000 Maschinen, die mehr als 1 Mio $ kosten

- Über 500 Fertigungsschritte.
- Die Anzahl von Transistoren in einem Halbleiter-Chip übersteigt die Anzahl von Sternen in der Milchstraße.

# Fortschritt bei neuen Algorithmen

- Anwendungen, die auf der Basis früherer Hardware unmöglich wären.
- Beispiele:
  - Intelligente autonome Rettungsroboter
  - Identifikation von Sehenwürdigkeiten über Handy (lädt automatisch den Reiseführer)
  - Verteilter Zugriff auf größte Datenmengen
  - Wikipedia, Telemedizin, Routenplaner
- Mehr dazu in anderen Vorlesungen.

# Fortschritt bei Entwurfsverfahren

- Systeme bestehen aus Hardware, Software und gegebenenfalls Zwischenschichten (Betriebssystem und ähnliches).
- Für alle diese Bereiche gibt es effiziente Entwurfsabläufe („Flows”).
- Sie sind notwendig, um die Komplexität heutiger Systeme zu beherrschen und die Milliarden Transistoren auch zu nutzen.

# Vorgehen

- Es sind Entwurfsziele zu erreichen, die oft einander widersprechen und gegeneinander abgewogen werden müssen (Trade-Offs).
  - Systemabmessungen (physikalische Größe)
  - Geschwindigkeit
  - Energieverbrauch
  - Entwicklungszeit (Time to Market)
  - Kosten (Entwicklung / Fertigung)

# Allgemeine Entwurfsprinzipien

- Hierarchie: Partitionierung des Systems in kleinere Blöcke; Entwurf der Blöcke und ihre spätere Zusammensetzung.
- Entwurfsautomatisierung: Verwende automatische Verfahren für möglichst viele Entwurfsschritte.
- Wiederverwendung von existierenden Designs (früher entworfen oder von Außen als „intellectual property core" zugekauft).

# Design und Verification Gap

# Einige berühmte Beispiele von Entwurfsfehlern

- Pentium-Bug *(Fehler in der Hardware)*.
  http://www.intel.com/support/processors/pentium/fdiv/wp
- Mars-Pathfinder-Mission *(Fehler auf Systemebene)*.
  http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html
- Ariane-Trägerraketen-Explosion *(Software der Vorgängerversion übernommen)*.
  http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
- Therac-25 Strahlentherapiemaschine *(Synchronisation von HW-SW)*.
  http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=274940

# Pentium Bug (1/2)

$x = 4195835$

$y = 3145727$

$z = x - (x/y) \times y$

$z = 256!$

# Pentium Bug (2/2)

- Fehler in Floating-Point-Einheit

  ⇒ Imageverlust für Intel
  ⇒ 480 Mio $ zusätzliche Kosten für Intel im 4. Quartal 1994

# Herausforderungen (1/2)

- Wie entwerfen? Entwurfsproblematik - „Design-Gap”
  $\rightarrow$ VLSI-Entwurf, Entwurfsmethodik
  - Anforderungen
  - Entwurfszeit (Time to market)
    - Arbeitszeit der Designer
    - Laufzeit der Algorithmen
  - Energieverbrauch
  - Entwurfskorrektheit (siehe z.B. Pentium-Bug)
  - Effizienzanforderungen / Performanz
- Was tun mit all den Transistoren?
  Sinnvolle Ausnutzung der verfügbaren Ressourcen
  $\rightarrow$ Architekturkonzepte, Rechnerarchitektur

# Herausforderungen (2/2)

**VLSI-Entwurf, Entwurfsmethodik**

- Wird bei ReTI angedeutet
- Rechnerarchitektur, erster Teil
- Spezialvorlesungen
  - Testen
  - Verifikation
  - Eingebettete Systeme
  - ...
- Projekte, Praktikum FPGA-Design

**Architekturkonzepte**

- Kapitel 8
- Hardwarepraktikum
- Vorlesung Rechnerarchitektur

# Kapitel 1 – Grundlagen

## 1. Mathematische Grundlagen

2. Beispielrechner ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Mathematische Grundlagen

- Verständigung auf gemeinsame Basis
- Die meisten Begriffe sollten bekannt sein, bzw. werden in anderen Vorlesungen noch formal und im Detail eingeführt.
- Hier: Informale, möglichst intuitive Einführung
  - Mengen, Funktionen, Relationen
  - Boolsche Algebra $(\{0,1\},\wedge,\vee,\neg)$
  - Graphen, O-Notation
  - Beweistechniken

# „Philosophie” der Mathematik

- Gegeben gewisse Aussagen (Axiome), welche andere Aussagen lassen sich aus ihnen herleiten?
- Sind die Axiome wahr und existiert eine solche Herleitung (Beweis), so sind die Folgerungen unumstößlich und indiskutabel wahr!
- Beschreiben die Axiome etwa ein physikalisches System, so gelten die hergeleiteten Folgerungen für dieses System.
- Die Frage, ob Axiome Realitätsbezug haben, ist aber außerhalb der (reinen) Mathematik!

# Menge (Naive Definition)

## Definition

Eine Menge ist eine Zusammenfassung von wohldefinierten, paarweise verschiedenen Objekten zu einem Ganzen.

- Die Objekte nennt man Elemente der Menge.
  (Für eine formal vollständige Definition der Menge bräuchte man mehrere Vorlesungsstunden.)

- Notation: Sind $a_1, a_2, \ldots, a_n$ paarweise verschieden, so schreibt man die Menge $M$, die aus ihnen besteht, als $M = \{a_1, a_2, \ldots, a_n\}$.
  - $a_i \in M$ bezeichnet, dass $a_i$ Element von $M$ ist.

# Beispiele für Mengen

- Leere Menge: $\varnothing$ (es gibt kein $a \in \varnothing$).
- Menge der natürlichen Zahlen: $\mathbb{N} = \{0, 1, 2, \dots\}$.
- Menge der booleschen Werte: $\mathbb{B} = \{0, 1\}$.
- Achtung: Die Anordnung von Elementen der Menge und gegebenenfalls Wiederholungen sind belanglos: $\{a, b, c\} = \{c, a, b\} = \{a, a, b, c, a, b\}$.
- Eine Menge kann Elemente enthalten, die selber Mengen sind, z.B. $\{a, b, \{a\}, \{a, b\}\}$.

# Spezifikation von Mengen

- Man kann eine Menge durch Angabe von Zusatzbedingungen spezifizieren.

  Beispiele:

  - Menge der ganzen Zahlen:
    $\mathbb{Z} = \{z, -z \mid z \in \mathbb{N}\}$.
  - Menge der rationalen Zahlen:
    $\mathbb{Q} = \{p/q \mid p \in \mathbb{N}, q \in \mathbb{Z}, q \neq 0, p, q \text{ teilerfremd}\}$.
  - Menge der endlichen Zeichenketten:
    $STRINGS = \{s_1 s_2 \ldots s_n \mid n \in \mathbb{N}, s_i \text{ ein Buchstabe}\}$.

# Untermengen, Potenzmenge, Mächtigkeit

- Menge $U$ ist Untermenge von $M$, wenn jedes Element von $U$ auch Element von $M$ ist.
  - Notation: $U \subset M$ bzw. $M \supset U$
  - Achtung: $\{a\} \subset \{a,b,c\}$, aber $a \in \{a,b,c\}$
- Potenzmenge von $M$ : $Pot(M) = \{m \mid m \subset M\}$.
  - $Pot(\{a,b,c\})$
    $= \{\varnothing, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$
- Die Anzahl $|M|$ der Elemente einer Menge $M$ heißt Mächtigkeit oder Kardinalität von $M$.

# Operationen auf Mengen 1/2

- Mengendifferenz: $M_1 \setminus M_2 = \{m \mid m \in M_1 \text{ und } m \notin M_2\}$

- Mengenschnitt: $M_1 \cap M_2 = \{m \mid m \in M_1 \text{ und } m \in M_2\}$

# Operationen auf Mengen 2/2

- Mengenvereinigung: $M_1 \cup M_2 = \{m \mid m \in M_1 \text{ oder } m \in M_2\}$

- Kartesisches Produkt:

$$M_1 \times M_2 = \{(m_1, m_2) \mid m_1 \in M_1 \text{ und } m_2 \in M_2\}$$

  - $(m_1, m_2)$ ist ein Tupel, bei dem es, im Gegensatz zu einer Menge $\{m_1, m_2\}$, auf die Reihenfolge ankommt!
  - Notation: $M^n = M \times \cdots \times M$ ($n$ mal).

# Relationen

## Definition

Eine Relation $R$ zwischen den Mengen $X$ und $Y$ ist eine Teilmenge von $X \times Y$.

- Notation: Statt $(x,y) \in R$ schreibt man $xRy$.
- Beispiele:
  - Relation $<$ zwischen $\mathbb{N}$ und $\mathbb{N}$.
    $<= \{(0,1),(0,2),\ldots,(1,2),(1,3),\ldots\}$
  - $R = \{(a,b) \mid a,b \in \mathbb{N}, a+b \text{ ungerade } \}$

# Funktionen

## Definition

Seien $X$ und $Y$ Mengen. Eine Funktion $f : X \to Y$ ist eine Relation zwischen den Mengen $X$ und $Y$, wobei für jedes $x \in X$ genau ein $y \in Y$ existiert, so dass $(x,y) \in f$.

- $X$ heißt Definitionsbereich, $Y$ Wertebereich von $f$.

- Notation: Statt $(x,y) \in f$ schreibt man $y = f(x)$.

- Beispiele:
  - Quadratfunktion $f : \mathbb{N} \to \mathbb{N}, f(x) = x^2$.
    $f = \{(0,0),(1,1),(2,4),(3,9),(4,16),(5,25),\dots\}$

  - Kardinalitätsfunktion $f : Pot(\{a,b,c\}) \to \mathbb{N}$.
    $f = \{(\varnothing,0),(\{a\},1),(\{b\},1),(\{c\},1),(\{a,b\},2),$
    $(\{a,c\},2),(\{b,c\},2),(\{a,b,c\},3)\}$

# Beispiele: Relationen, Funktionen

- Jede Funktion ist auch eine Relation.
- Aber es gibt natürlich Relationen, die keine Funktionen sind.
- Beispiel:
  - $\sin^{-1}(x) = \{(\sin(x), x) \mid x \in \mathbb{R}\}$ ist eine Relation, aber keine Funktion!

# Summen und Produkte (Notation)

- Wir schreiben für $f : \mathbb{N} \to \mathbb{R}$

$$\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + \cdots + f(n-1) + f(n)$$
$$\prod_{i=m}^{n} f(i) = f(m) \cdot f(m+1) \cdot \cdots \cdot f(n-1) \cdot f(n)$$

- Beispiel:

$$\sum_{i=0}^{5} i^2 = 0^2 + 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 55$$

- Schreibweise mit beliebigen Bedingungen:

$$\sum_{i,j>0, i+2j\leq 5} (i^2/j) = (1^2/1) + (1^2/2) + (2^2/1) + (3^2/1) = 14,5$$

# Boolesche Algebra $(\{0,1\},\wedge,\vee,\neg)$ 1/4

## Definition

- $\mathbb{B} := \{0,1\}$
- Konjunktion (UND-Verknüpfung) $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
  $0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1$
- Disjunktion (ODER-Verknüpfung) $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$
  $0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1$
- Negation $\neg : \mathbb{B} \to \mathbb{B}$
  $\neg 0 = 1, \quad \neg 1 = 0$
- Boolescher Ausdruck
  - Die Elemente aus $\mathbb{B}$ sind boolesche Ausdrücke.
  - Seien $A$ und $B$ boolesche Ausdrücke, dann sind $(A \wedge B)$, $(A \vee B)$, $(\neg A)$ wieder boolesche Ausdrücke.

# Boolesche Algebra $(\{0,1\}, \wedge, \vee, \neg)$ 2/4

## Konventionen

- Man schreibt auch $\cdot$ statt $\wedge$ und $+$ statt $\vee$.
- Für $\neg x$ sind viele Notationen üblich: $\sim x$, $x'$ oder $\overline{x}$.
- Zur Vereinfachung der Notation bei booleschen Ausdrücken vereinbaren wir:
  Negation $\sim$ bindet stärker als Konjunktion $\cdot$, Konjunktion $\cdot$ bindet stärker als Disjunktion $+$.

# Boolesche Algebra $(\{0,1\},\wedge,\vee,\neg)$ 3/4

## Axiome der booleschen Algebra

| | |
|---|---|
| Kommutativität: | $x+y=y+x$ |
| | $x\cdot y=y\cdot x$ |
| Assoziativität: | $x+(y+z)=(x+y)+z$ |
| | $x\cdot(y\cdot z)=(x\cdot y)\cdot z$ |
| Absorption: | $x+(x\cdot y)=x$ |
| | $x\cdot(x+y)=x$ |
| Distributivität: | $x+(y\cdot z)=(x+y)\cdot(x+z)$ |
| | $x\cdot(y+z)=(x\cdot y)+(x\cdot z)$ |
| Komplement: | $x+(y\cdot\neg y)=x$ |
| | $x\cdot(y+\neg y)=x$ |

# Boolesche Algebra $(\{0,1\},\wedge,\vee,\neg)$ 4/4

- Neben der vorgestellten gibt es weitere boolesche Algebren, in denen diese Axiome gelten.
- Die folgenden Regeln sind aus den Axiomen ableitbar:

## Weitere Regeln für boolesche Algebren

| | |
|---|---|
| Doppeltes Komplement: | $\neg(\neg x) = x$ |
| Idempotenz: | $x + x = x \cdot x = x$ |
| De-Morgan-Regel: | $\neg(x+y) = (\neg x)\cdot(\neg y)$ |
| | $\neg(x\cdot y) = (\neg x)+(\neg y)$ |
| Consensus-Regel: | $(x\cdot y)+((\neg x)\cdot z)$ |
| | $= (x\cdot y)+((\neg x)\cdot z)+(y\cdot z)$ |
| | $(x+y)\cdot((\neg x)+z)$ |
| | $= (x+y)\cdot((\neg x)+z)\cdot(y+z)$ |

# Boolesche Funktion

## Definition

Eine boolesche Funktion $f$ in $n$ Variablen und mit $m$ Ausgängen ist eine Funktion

$$f : \mathbb{B}^n \to \mathbb{B}^m (n, m \in \mathbb{N}).$$

- Die Menge aller booleschen Funktionen in $n$ Variablen mit $m$ Ausgängen ist

$$\mathbb{B}_{n,m} := \{f \mid f : \mathbb{B}^n \to \mathbb{B}^m\}.$$

- Wir schreiben abkürzend $\mathbb{B}_n$ statt $\mathbb{B}_{n,1}$.
- Ein digitaler Schaltkreis ohne Speicherelemente, mit $n$ Eingängen und $m$ Ausgängen realisiert eine solche Funktion! (Details später)

# Gerichteter Graph

## Definition

$G = (V, E)$ ist ein gerichteter Graph, wenn folgendes gilt:

- $V$ endliche, nichtleere Menge (Knoten)
- $E$ endliche Menge (Kanten)
- Abbildungen $Q : E \to V$ und $Z : E \to V$
  $Q(e)$ ist Quelle, $Z(e)$ Ziel einer Kante $e$
- Abbildungen $indeg : V \to \mathbb{N}$ und $outdeg : V \to \mathbb{N}$
  $indeg(v) = \mid \{e \mid Z(e) = v\} \mid$ ist der Eingangsgrad,
  $outdeg(v) = \mid \{e \mid Q(e) = v\} \mid$ der Ausgangsgrad von $v$.

# Pfade in gerichteten Graphen

- Ein Knoten mit
  - $indeg(v) = 0$ heißt Wurzel.
  - $outdeg(v) = 0$ heißt Blatt.
  - $outdeg(v) > 0$ heißt innerer Knoten.

- Ein Pfad (der Länge $k$) in $G$ ist eine Folge von $k$ Kanten $e_1, e_2, \ldots, e_k$ $(k \geq 0)$ mit $Z(e_i) = Q(e_{i+1})$ für alle $i$ $(k-1 \geq i \geq 1)$

- Ein Zyklus in $G$ ist ein Pfad der Länge $\geq 1$ in $G$, bei dem Ziel und Quelle identisch sind ($G$ heißt azyklisch, falls kein Zyklus in $G$ existiert).

- Die Graph-Tiefe eines azyklischen Graphen ist definiert als die Länge des längsten Pfades in $G$.

# Bäume, Binäre Bäume

## Definition

Ein Baum ist ein gerichteter, azyklischer Graph mit genau einer Wurzel $w$ ($indeg(w) = 0$) und $indeg(v) = 1$ für alle andere Knoten $v$. Ein Baum heißt binär (bzw. Binärbaum), wenn für seine innere Knoten $v$ $outdeg(v) \leq 2$ gilt.

Beispiele:

Baum:

Binärbaum:

# Groß-O-Notation (1/2)

- Seien $f, g : \mathbb{R}_0^+ \to \mathbb{R}_0^+$.
  Man schreibt $f(x) \in O(g(x))$, wenn es $c \in \mathbb{R}_0^+, x_0 \in \mathbb{R}_0^+$ gibt, so dass $f(x) \leq c \cdot g(x)$ für alle $x > x_0$ gilt.
  - Beispiel: $5x + 2 \in O(x^2)$
    Beweis: Setze $c = 5, x_0 = 1$
    $5x + 2 \leq 5 \cdot x^2$, für $x > 1$.
- Groß-O-Notation wird verwendet, um Größe von parametrisierten Objekten (z.B. Graphen), Laufzeit von Algorithmen (Anzahl von Rechenschritten in Abhängigkeit von der Eingabe) usw. asymptotisch, d.h. bis auf eine multiplikative Konstante, abzuschätzen.
- Die Notation $f(x) = O(g(x))$ ist weit verbreitet, aber eigentlich falsch, da $O(g(x))$ eine Menge ist. So folgt aus $f(x) = O(g(x))$ und $h(x) = O(g(x))$ keinesfalls $f(x) = h(x)$!

# Groß-O-Notation (2/2)

# Beweistechniken

- Sukzessive Folgerungen bzw. Direkter Beweis
- Indirekter Beweis bzw. Beweis durch Widerspruch
- Vollständige Induktion

# Sukzessive Folgerungen

Gegeben Aussage $A$, es soll Aussage $B$ bewiesen werden.

- Sukzessive Folgerungen:
  Aus $A$ folgt $C$, aus $C$ folgt $D$, aus $D$ folgt $B$, also gilt $B$.

# Beispiel: Sukzessive Folgerungen

- Gegeben $f, g, h, \quad f(x) \in O(g(x)),\ g(x) \in O(h(x))$.
  Dann gilt $f(x) \in O(h(x))$.

**Beweis:**

1. Aus $f(x) \in O(g(x))$ folgt die Existenz von $c_f, x_{0f} : f(x) \leq c_f \cdot g(x)$ für $x > x_{0f}$. Aus $g(x) \in O(h(x))$ folgt die Existenz von $c_g, x_{0g} : g(x) \leq c_g \cdot h(x)$ für $x > x_{0g}$.
2. Man setze $x_0 := max\{x_{0f}, x_{0g}\}$. Dann gilt für $x > x_0$ sowohl $f(x) \leq c_f \cdot g(x)$ als auch $g(x) \leq c_g \cdot h(x)$.
3. Man setze $c := c_f \cdot c_g$. Dann gilt für $x > x_0$ :
   $f(x) \leq c_f \cdot g(x) \leq c_f(c_g \cdot h(x)) = c \cdot h(x)$.
   Dies bedeutet aber gerade $f(x) \in O(h(x))$

# Indirekter Beweis 1/2

Es soll Aussage $S$ bewiesen werden.

- Indirekter Beweis: Man nimmt an, $\neg S$ (also die Umkehrung von $S$) würde gelten. Daraus leitet man einen Widerspruch her (z.B. “es gilt $C$ **und** $\neg C$”, “$31 = 42$”, ...).
- Da der Widerspruch schrittweise aus $\neg S$ logisch hergeleitet wurde, kann $\neg S$ nicht gelten und somit muss $S$ gelten.

# Indirekter Beweis 2/2

- Betrachte den Spezialfall $S = A \Rightarrow B$.
  - Dann ist $\neg S = A \wedge \neg B$. Man nimmt also an, dass $A$ gilt, aber $\neg B$.
  - Ergibt sich aus der Annahme ein Widerspruch, dann muss aus der Gültigkeit von $A$ die Gültigkeit von $B$ folgen.

  - Ergibt sich der Widerspruch speziell durch Herleitung von $\neg A$ aus $\neg B$, dann reduziert sich der Widerspruchsbeweis auf den Spezialfall Beweis der "Kontraposition" $\neg B \Rightarrow \neg A$.
  - $A \Rightarrow B$ und $\neg B \Rightarrow \neg A$ sind logisch äquivalent.

  - Implizit setzt man immer die Gültigkeit sämtlicher Axiome voraus. Sei $Ax$ die Aussage "Sämtliche Axiome gelten".
  - Dann ist $S' = (A \wedge Ax) \Rightarrow B$ zu beweisen.
  - Annahme ist dann also: $\neg S' = A \wedge Ax \wedge \neg B$ gilt.

# Beispiel: Indirekter Beweis

- Zu zeigen: $x^2 \notin O(x)$

## Beweis:

- Wir nehmen an, dass $x^2 \in O(x)$ wäre. Dann gibt es $c$ und $x_0$, so dass für $x > x_0$ gilt:
$$x^2 \leq c \cdot x \tag{1}$$
- Nun suchen wir ein $x_1$, für das $x_1^2 = c \cdot x_1$. Dies ist für $x_1 = c$ der Fall.
- Für alle $x > x_1 = c$ ist $x^2 > c \cdot x$. Man wähle ein $x_2 > max\{x_0, x_1\}$. Dann gilt auch für $x_2$:
$$x_2^2 > c \cdot x_2 \tag{2}$$
- Andererseits muss für $x_2$ auch (1) gelten. Widerspruch! Somit kann die Annahme nicht stimmen.

# Vollständige Induktion

- Die vollständige Induktion ist eine Beweismethode für Aussagen, die für alle natürlichen Zahlen $n$ gelten sollen.
- Zuerst wird die Aussage für den Basisfall $n = 0$ beweisen (manchmal auch $n = 1$ oder höher).
- Dann wird der Induktionsschritt durchgeführt:
  Unter der Annahme, dass die Aussage für $n$ gilt (Induktionsvoraussetzung) wird bewiesen, dass die Aussage auch für $n + 1$ gilt.
- Daraus folgt die Gültigkeit der Aussage für alle natürlichen Zahlen.

# Vollständige Induktion: Beispiel (1/2)

- **Behauptung:**

  $\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}$ gilt für alle $n \in \mathbb{N}$.

- **Induktionsanfang:**
  Zeige die Behauptung für $n = 1$.

  $\sum_{k=1}^{1} \frac{1}{k(k+1)} = \frac{1}{1(1+1)} = \frac{1}{2} = \frac{1}{1+1}$

# Vollständige Induktion: Beispiel (2/2)

- **Induktionsvoraussetzung** (IV)**:**
  Nehme an, die Behauptung gilt für *ein* $n \in \mathbb{N}$.
  Also: Es gibt ein $n$ für das gilt: $\sum_{k=1}^{n} \frac{1}{k(k+1)} = \frac{n}{n+1}$

- **Induktionsschritt:**
  Zeige die Behauptung für $n+1$.

$$\sum_{k=1}^{n+1} \frac{1}{k(k+1)} = \sum_{k=1}^{n} \frac{1}{k(k+1)} + \frac{1}{(n+1)(n+2)} \overset{\text{IV}}{=} \frac{n}{n+1} + \frac{1}{(n+1)(n+2)}$$

$$= \frac{n(n+2)+1}{(n+1)(n+2)} = \frac{n^2+2n+1}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+1)(n+2)} = \frac{(n+1)}{(n+2)} = \frac{(n+1)}{(n+1)+1} \quad \square$$

# Kapitel 1 – Grundlagen

1. Mathematische Grundlagen
2. **Beispielrechner ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# ReTI (Rechner Technische Informatik)

- Ursprünglich eingeführt in [*Keller*, *Paul*] unter dem Namen ReSa.
- Hier wird ReTI zunächst abstrakt eingeführt.
  - Alle Speicher bestehen aus unendlich vielen Speicherzellen, die beliebig große ganze Zahlen aufnehmen können.
- Später wird die tatsächliche Implementierung von ReTI unter realistischen Annahmen thematisiert.

# Abstrakte ReTI-Maschine

- Zwei unendlich große Speicher
  - Datenspeicher $S$ für Daten (beliebig große Zahlen).
  - $S(i) =$ Inhalt von Zelle $i$ des Datenspeichers, $i \in N$ Adresse.
  - Programmspeicher $P$ für Maschinenbefehle.
  - Lade-/Speicher-, Rechen-, Sprungbefehle - siehe später.
  - $P(i) =$ Inhalt von Zelle $i$ des Programmspeichers.
- Zentraleinheit CPU (Central Processing Unit)
  - Vier für Benutzer sichtbare Register.
  - $PC =$ Befehlszähler (Program Counter).
  - $ACC =$ Akkumulator.
  - $IN1$, $IN2 =$ Indexregister 1 und 2.

# Aufbau von ReTI

# Programmablauf

- Programme bzw. Daten stehen beim Start der Maschine in $P$ bzw. $S$.
- Programm beginnt bei Zelle 0 von $P$.
- Inhalt von $P$ wird nicht geändert.
- Maschine arbeitet in Schritten $t = 1, 2, \ldots$
  In jedem Schritt $t$:
  - Ausführung eines Befehls: $P(PC)$ wird als Befehl interpretiert und in Schritt $t$ ausgeführt.
  - $PC$ erhält neuen Wert (abhängig von Befehl).
- Bei Programmstart ist $PC = 0$.

# ReTI-Befehle und ihre Wirkung

- Load/Store: Laden von Werten aus dem Datenspeicher $S$ bzw. Schreiben von Werten in $S$.
- Compute: Berechnungen (hier zunächst Addition und Subtraktion).
  - Mit Werten im Datenspeicher $S$.
  - Mit Absolutwerten (Immediate).
- Indexregister: Indirekte Speicheradressierung (siehe unten).
- Sprungbefehle: Bedingte und unbedingte Sprünge.

# Load/Store

Transport von Daten zwischen *ACC* und Datenspeicher.

- LOAD $i$:
  Lädt Inhalt $S(i)$ von Speicherzelle $i$ in Akkumulator *ACC* und erhöht *PC* um 1.

- STORE $i$:
  Speichert den Inhalt von *ACC* in $S(i)$ und erhöht *PC* um 1.

# Load/Store: Übersicht

| Befehl | Wirkung |
|---|---|
| LOAD $i$ | $ACC := S(i) \quad PC := PC + 1$ |
| STORE $i$ | $S(i) := ACC \quad PC := PC + 1$ |

# Beispielprogramm

Ein Programm, das Inhalte von Speicherzelle $S(0)$ $(= x)$ und $S(1)$ $(= y)$ vertauscht.

| | | |
|---|---|---|
| 0 | LOAD 0; | $ACC := S(0) = x$ |
| 1 | STORE 2; | $S(2) := ACC = x$ |
| 2 | LOAD 1; | $ACC := S(1) = y$ |
| 3 | STORE 0; | $S(0) := ACC = y$ |
| 4 | LOAD 2; | $ACC := S(2) = x$ |
| 5 | STORE 1; | $S(1) := ACC = x$ |

# Compute-Befehle

Verknüpfe den Inhalt von $ACC$ mit $S(i)$ oder mit einer Konstante und speichere das Ergebnis in $ACC$ ab.

- $ADD, SUB =$ Compute *memory*-Befehle
- $ADDI, SUBI =$ Compute *immediate*-Befehle
- Beides zusammen ergibt die Compute-Befehle.

Bei Compute memory: Interpretiere Parameter $i$ direkt als Speicheradresse.

| Befehl | Wirkung |
|---|---|
| ADD $i$ | $ACC := ACC + S(i) \quad PC := PC + 1$ |
| SUB $i$ | $ACC := ACC - S(i) \quad PC := PC + 1$ |

# Immediate-Befehle

Interpretiere Parameter $i$ direkt als Konstante.

| Befehl | Wirkung | |
|---|---|---|
| LOADI $i$ | $ACC := i$ | $PC := PC + 1$ |
| ADDI $i$ | $ACC := ACC + i$ | $PC := PC + 1$ |
| SUBI $i$ | $ACC := ACC - i$ | $PC := PC + 1$ |

- Anmerkung: ADDI und SUBI sind Compute Befehle. LOADI ist den Load-/Store-Befehlen zuzuordnen.

# Indexregister-Befehle

| Befehl | Wirkung | |
|---|---|---|
| LOADINj $i$ | $ACC := S(INj + i)$ | $PC := PC + 1$ |
| | $(j \in \{1, 2\})$ | |
| STOREINj $i$ | $S(INj + i) := ACC$ | $PC := PC + 1$ |
| | $(j \in \{1, 2\})$ | |
| MOVE $S$ $D$ | $D := S$ | $PC := PC + 1$ |
| | $(D \in \{ACC, IN1, IN2\},$ | |
| | $S \in \{ACC, IN1, IN2, PC\})$ | |
| MOVE $S$ $PC$ | $PC := S$ | |
| | $(S \in \{ACC, IN1, IN2\})$ | |

# Beispielprogramm für Indexregister-Befehle

$S(0) = x$, $S(1) = y$
Kopiere $y$ in Zelle $S(x)$:

| | | |
|---|---|---|
| 0 | LOAD 0; | $ACC := S(0) = x$ |
| 1 | MOVE $ACC$ $IN1$; | $IN1 := ACC = x$ |
| 2 | LOAD 1; | $ACC := S(1) = y$ |
| 3 | STOREIN1 0; | $S(x) = S(IN1 + 0) := ACC = y$ |

# Sprung-Befehle

Manipulation des Befehlszählers.

- JUMP für *unbedingte* Sprünge,
- $\text{JUMP}_c$ mit $c \in \{<, =, >, \leq, \neq, \geq\}$ für *bedingte* Sprünge.
- Mit bedingten Sprüngen kann man *Programmschleifen* und *bedingte Anweisungen* realisieren!

| Befehl | Wirkung |
|---|---|
| JUMP $i$ | $PC := PC + i \quad (i \in \mathbb{Z})$ |
| $\text{JUMP}_c\ i$ | $PC := \begin{cases} PC + i, & \text{falls } ACC\ c\ 0 \\ PC + 1, & \text{sonst} \end{cases}$ <br> $(i \in \mathbb{Z}, c \in \{<, =, >, \leq, \neq, \geq\})$ |

# Beispielprogramm

$S(0) = x;\ S(1) = y,\ y \geq 0$

| | | |
|---|---|---|
| 0 | LOADI 0; | $ACC := 0$ |
| 1 | STORE 2; | $S(2) := 0$ |
| 2 | LOAD 1; | $ACC := S(1)$ |
| 3 | SUBI 1; | $ACC := ACC - 1$ |
| 4 | STORE 1; | $S(1) := ACC$ |
| 5 | JUMP $< 5$; | $PC := PC + 5$, falls $ACC < 0$ |
| 6 | LOAD 2; | $ACC := S(2)$ |
| 7 | ADD 0; | $ACC := ACC + S(0)$ |
| 8 | STORE 2; | $S(2) := ACC$ |
| 9 | JUMP $-7$; | $PC := PC - 7$ |

# Zusammenfassung

- Mathematik erlaubt es uns, reale Zusammenhänge formal zu fassen und allgemeingültige Folgerungen aus ihnen herzuleiten.
- Rechner ReTI wird uns im weiteren Verlauf der Vorlesung als Illustrator und Anwendungsbeispiel für die vorgestellten Konzepte dienen.

# Motivation

- Ein Rechner speichert, verarbeitet und produziert Informationen.
- Alle Ergebnisse müssen als Funktion der Anfangswerte exakt reproduzierbar sein.

→ Informationsspeicherung und Verarbeitung müssen exakt sein.

- Probleme: Noise, Crosstalk, Abschwächung

→ Es gibt keine exakte Datenübertragung oder Datenspeicherung.

→ Ziel: Quantisierung der Informationsspeicherung mit Signal groß gegenüber maximaler Störung

- Binär-Codierung (nur zwei Zustände) ist die einfachste (und sicherste) Signal-Quantisierung.
- BIT (0, 1) als grundlegende Informationseinheit

# Motivation

- Ein Rechner kann üblicherweise
  - Zeichen verarbeiten (Textverarbeitung)
  - mit Zahlen rechnen
  - Bilder, Audio- und Videoinformationen verarbeiten und darstellen ...
- Ein Algorithmus kann zwar prinzipiell mit abstrakten Objekten verschiedener Art operieren, aber diese müssen im Rechner letztendlich als Folgen von Bits repräsentiert werden.

→ Kodierung!

# Kapitel 2.1 - Kodierung von Zeichen

- Wie werden im Rechner Zeichen dargestellt ?
- Codes fester Länge
- “Längenoptimale Kodierungen” von Zeichen: Häufigkeitscodes (Bsp.: Huffman-Code)

# Kapitel 2 – Kodierung

1. **Kodierung von Zeichen**
2. Kodierung von Zahlen
3. Anwendung: ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Alphabete und Wörter

## Definition

Eine nichtleere Menge $A = \{a_1, \ldots, a_m\}$ heißt (endliches) Alphabet der Größe $m$.
$a_1, \ldots, a_m$ heißen Zeichen des Alphabets.

- $A^* = \{w \mid w = b_1 \ldots b_n \textit{ mit } n \in \mathbb{N}, \forall i \textit{ mit } 1 \leq i \leq n : b_i \in A\}$ ist die Menge aller Wörter über dem Alphabet $A$.
- $|b_1 \ldots b_n| := n$ heißt Länge des Wortes $b_1 \ldots b_n$.
- Das Wort der Länge 0 wird mit $\varepsilon$ bezeichnet.

Beispiel:
Sei $A = \{a, b, c, d\}$.
Dann ist *bcada* ein Wort der Länge 5 über $A$.

# Code

Sei $A = \{a_1, \ldots, a_m\}$ ein endliches Alphabet der Größe $m$.

- Eine Abbildung $c : A \to \{0,1\}^*$ oder $c : A \to \{0,1\}^n$ heißt Code, falls $c$ injektiv ist.
- Die Menge $c(A) := \{w \in \{0,1\}^* \mid \exists a \in A : c(a) = w\}$ heißt Menge der Codewörter.
- Ein Code $c : A \to \{0,1\}^n$ heißt Code fester Länge.
- Für einen Code $c : A \to \{0,1\}^n$ fester Länge gilt: $n \geq \lceil \log_2 m \rceil$.
  - Ist $n = \lceil \log_2 m \rceil + r$ mit $r > 0$, so können die $r$ zusätzlichen Bits zum Test auf Übertragungsfehler verwendet werden (siehe Kap. 6).

# Codes fester Länge

- Die Kodierung eines jeden Zeichens besteht aus $n$ Bits.
  - ASCII (American Standard Code for Information Interchange): 7 Bits (es gibt Erweiterungen mit 8 Bits)
  - EBCDIC: 8 Bits
  - Unicode: 16 Bits
  - Vgl. „Rechnerarchitektur”
- Diese Kodierungen sind recht einfach zu behandeln. Unter Umständen wird für sie aber mehr Speicherplatz gebraucht als unbedingt nötig.

# Beispiel: ASCII-Tabelle

erste 3 Bits

| letzte 4 Bits | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000 | nul | dle | | 0 | @ | P | ` | p |
| 0001 | soh | dc1 | ! | 1 | A | Q | a | q |
| 0010 | sfx | dc2 | " | 2 | B | R | b | r |
| 0011 | etx | dc3 | # | 3 | C | S | c | s |
| 0100 | eot | dc4 | $ | 4 | D | T | d | t |
| 0101 | enq | nak | % | 5 | E | U | e | u |
| 0110 | ack | syn | & | 6 | F | V | f | v |
| 0111 | bel | etb | ' | 7 | G | W | g | w |
| 1000 | bs | can | ( | 8 | H | X | h | x |
| 1001 | ht | em | ) | 9 | I | Y | i | y |
| 1010 | lf | sub | * | : | J | Z | j | z |
| 1011 | vt | esc | + | ; | K | [ | k | { |
| 1100 | ff | fs | , | < | L | \ | l | \| |
| 1101 | cr | qs | - | = | M | ] | m | } |
| 1110 | so | rs | . | > | N | ^ | n | * |
| 1111 | si | us | / | ? | O | _ | o | del |

Steuerzeichen Schriftzeichen

# Häufigkeitsabhängige Codes

- **Ziel**: Reduktion der Länge einer Nachricht durch Wahl verschieden langer Codewörter für die verschiedenen Zeichen eines Alphabets (also kein Code fester Länge!)
- **Idee**:

| | | |
|---|---|---|
| Häufiges Zeichen | $\rightarrow$ | kurzer Code |
| Seltenes Zeichen | $\rightarrow$ | langer Code |

- **Voraussetzungen**:
  - Häufigkeitsverteilung ist bekannt $\rightarrow$ statische Kompression
  - Häufigkeitsverteilung ist nicht bekannt $\rightarrow$ dynamische Kompression

# Huffman-Code

- Der Huffman-Code ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.
- Beispiel:

| Zeichen | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| Häufigkeit [%] | 20 | 25 | 15 | 8 | 7 | 6 | 5 | 5 | 5 | 4 |

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

Markiere nun die linken Kanten mit 0 und die rechten Kanten mit 1, fertig ist der Huffman-Code!

# Erzeugte Huffman-Kodierung

| Zeichen | a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|---|
| Häufigkeit [%] | 20 | 25 | 15 | 8 | 7 | 6 | 5 | 5 | 5 | 4 |

Erzeugte Kodierung:

| a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|
| 00 | 10 | 110 | 1110 | 0100 | 0101 | 0110 | 0111 | 11110 | 11111 |

# Huffman-Code: Dekodierung

Erzeugte Kodierung:

| a | b | c | d | e | f | g | h | i | j |
|---|---|---|---|---|---|---|---|---|---|
| 00 | 10 | 110 | 1110 | 0100 | 0101 | 0110 | 0111 | 11110 | 11111 |

1. Lesen des Bitstromes bis Symbol erkannt wurde.
2. Erkanntes Symbol ausgeben und weiter mit 1.

# Präfixcodes

## Definition

Sei $A$ ein Alphabet der Größe $m$.

- $a_1 \dots a_p \in A^*$ heißt Präfix von $b_1 \dots b_l \in A^*$, falls $p \leq l$ und $a_i = b_i \; \forall i, \; 1 \leq i \leq p$.
- Ein Code $c : A \to \{0,1\}^*$ heißt Präfixcode, falls es kein Paar $i,j \in \{1,\dots,m\}$ gibt, so dass $c(a_i)$ Präfix von $c(a_j)$.
  - Der Huffman-Code ist ein Präfixcode.
  - Bei Präfixcodes können Wörter über $\{0,1\}$ eindeutig dekodiert werden. (Sie entsprechen Binärbäumen mit Codewörtern an den Blättern.)
  - Huffman-Code ist ein bzgl. mittlerer Codelänge optimaler Präfixcode (unter Voraussetzung einer bekannten Häufigkeitsverteilung) - ohne Beweis.

# Weitere Verfahren

- Es gibt zahlreiche Ansätze zur Datenkompression. (Beispiel: Lempel-Ziv-Welch.)
- In Programmtexten gibt es häufig viele Leerzeichen, gleiche Schlüsselwörter und so weiter.

→ Kodiere Folgen von Leerzeichen bzw. Schlüsselwörter durch kurze Codes.

- Das wird z.B. bei GIF und TIFF genutzt.
- Das soll auch funktionieren, wenn man noch nicht weiß, welche Zeichenketten häufig vorkommen.

# Kapitel 2 – Kodierung

1. Kodierung von Zeichen
2. **Kodierung von Zahlen**
3. Anwendung: ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Zahlensysteme

## Definition

Ein Zahlensystem ist ein Tripel $S = (b, Z, \delta)$ mit den folgenden Eigenschaften:

- $b \geq 2$ ist eine natürliche Zahl, die Basis des Stellenwertsystems.
- $Z$ ist eine $b$-elementige Menge von Symbolen, den Ziffern.
- $\delta : Z \to \{0, 1, \ldots, b-1\}$ ist eine Abbildung, die jeder Ziffer umkehrbar eindeutig eine natürliche Zahl zwischen 0 und $b-1$ zuordnet.

# Beispiele für Zahlensysteme

- Dualsystem:
  $b = 2 \quad Z = \{0, 1\}$
- Oktalsystem:
  $b = 8 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- Dezimalsystem:
  $b = 10 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Hexadezimalsystem:
  $b = 16 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

# Festkommazahlen

## Definition

Eine Festkommazahl ist eine endliche Folge von Ziffern aus einem Zahlensystem zur Basis $b$ mit Ziffernmenge $Z$.

- Sie besteht aus $n+1$ Vorkommastellen ($n \geq 0$) und $k \geq 0$ Nachkommastellen.
- Der Wert $\langle d \rangle$ einer nicht-negativen Festkommazahl $d = d_n d_{n-1} \ldots d_1 d_0 d_{-1} \ldots d_{-k}$ mit $d_i \in Z$ ist gegeben durch

$$\langle d \rangle = \sum_{i=-k}^{n} b^i \cdot \delta(d_i)$$

# Festkommazahlen: Schreibweise

- Vorkomma- und Nachkommastellen werden zur Verdeutlichung durch ein Komma oder einen Punkt getrennt:
  $d = d_n d_{n-1} \ldots d_1 d_0 . d_{-1} \ldots d_{-k}$
- Um anzudeuten, welches Zahlensystem zu Grunde liegt, wird gelegentlich die Basis als Index an die Ziffernfolge angehängt.
  - Beispiel $(n = 3, k = 0)$ :

$$
\begin{array}{lcr}
0110_2 & = & 6 \\
0110_8 & = & 72 \\
0110_{10} & = & 110 \\
0110_{16} & = & 272
\end{array}
$$

# Negative Festkommazahlen

*(Im Folgenden wird Basis 2 angenommen.)*

- Bei der Darstellung negativer Festkommazahlen nimmt die höchstwertigste Stelle $d_n$ eine Sonderrolle ein:
  - Ist $d_n = 0$, so handelt es sich um eine nichtnegative Zahl.
- Bei der Darstellung negativer Zahlen gibt es folgende Alternativen:
  - Darstellung durch Betrag und Vorzeichen:

    $$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k}^{n-1} d_i \, 2^i$$
  - Einer-Komplement-Darstellung:

    $$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_1 := \sum_{i=-k}^{n-1} d_i \, 2^i - d_n(2^n - 2^{-k})$$
  - Zweier-Komplement-Darstellung:

    $$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_2 := \sum_{i=-k}^{n-1} d_i \, 2^i - d_n \, 2^n$$

# Betrag und Vorzeichen

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k}^{n-1} d_i \, 2^i$$

**Beispiel**: $n = 2, k = 0$

| $a$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $[a]_{BV}$ | 0 | 1 | 2 | 3 | 0 | -1 | -2 | -3 |

- Der Zahlenbereich ist symmetrisch:
  - Kleinste Zahl: $-(2^n - 2^{-k})$, größte Zahl: $2^n - 2^{-k}$
- Man erhält zu $a$ die inverse Zahl, indem man das erste Bit komplementiert.
- Zwei Darstellungen für die Null (000 und 100 im Beispiel).
- „Benachbarte Zahlen" haben gleichen Abstand $2^{-k}$.

# Einer-Komplement

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_1 := \sum_{i=-k}^{n-1} d_i \, 2^i - d_n(2^n - 2^{-k})$$

**Beispiel**: $n = 2, k = 0$

| $a$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $[a]_1$ | 0 | 1 | 2 | 3 | -3 | -2 | -1 | 0 |

- Der Zahlenbereich ist symmetrisch: $-(2^n - 2^{-k}) \ldots 2^n - 2^{-k}$
- Zwei Darstellungen für die Null (000 und 111 im Beispiel).
- „Benachbarte Zahlen" haben gleichen Abstand $2^{-k}$.
- Man erhält zu $a$ die inverse Zahl, indem man alle Bits komplementiert (siehe Lemma nächste Folie).

# Einer-Komplement: Inversion

## Lemma

Sei $a$ eine Festkommazahl, $a'$ die Festkommazahl, die aus $a$ durch Komplementieren aller Bits ($0 \to 1, 1 \to 0$) hervorgeht. Dann gilt $[a']_1 = -[a]_1$.

**Beweisidee**: Addiert man Bits $n-1 \cdots -k$ von $a$ und $a'$, erhält man $111\ldots11$. Das ist aber gerade $(2^n - 2^{-k}) = (a_n + a'_n) \cdot (2^n - 2^{-k})$.

$$\begin{array}{ll} (a_{n-1}, \ldots, a_{-k}) & 011011 \\ (a'_{n-1}, \ldots, a'_{-k}) & 100100 \\ \hline & 111111 \end{array}$$

mit $< 111111 > = < (a_{n-1}, \ldots, a_{-k}) > + < (a'_{n-1}, \ldots, a'_{-k}) >$

# Zweier-Komplement

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_2 := \sum_{i=-k}^{n-1} d_i \, 2^i - d_n \, 2^n$$

**Beispiel**: $n = 2, k = 0$

| $a$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $[a]_2$ | 0 | 1 | 2 | 3 | -4 | -3 | -2 | -1 |

- Der Zahlenbereich ist asymmetrisch: $-2^n \ldots 2^n - 2^{-k}$
- Die Zahlendarstellung ist eindeutig, auch für die Null.
- „Benachbarte Zahlen" haben gleichen Abstand $2^{-k}$.
- Man erhält zu $a$ die inverse Zahl, indem man alle Bits komplementiert und an der niederwertigsten Stelle 1 addiert (siehe Lemma nächste Folie).

# Zweier-Komplement: Inversion

## Lemma

Sei $a$ eine Festkommazahl, $a'$ die Festkommazahl, die aus $a$ durch Komplementieren aller Bits ($0 \to 1, 1 \to 0$) hervorgeht. Dann gilt $[a']_2 + 2^{-k} = -[a]_2$.

**Beweisidee**: Addiert man Bits $n-1 \ldots 0$ von $a$ und $a'$, erhält man $111 \ldots 11$. Addiert man noch $000 \ldots 01$ hinzu, so erhält man gerade $2^n = (a_n + a'_n) \cdot 2^n$.

$$\begin{array}{ll} (a_{n-1}, \ldots, a_{-k}) & 011011 \\ (a'_{n-1}, \ldots, a'_{-k}) & 100100 \\ \text{“}2^{-k}\text{”} & 000001 \\ \hline & 1000000 \end{array}$$

mit

$$< 1000000 > = < (a_{n-1}, \ldots, a_{-k}) > + < (a'_{n-1}, \ldots, a'_{-k}) > + 2^{-k}$$

# Vorteil von Zweier-Komplement

- Wir werden später Schaltungen betrachten, die zwei Zahlen als Eingaben nehmen und ihre Summe oder Differenz an den Ausgängen bereit stellen (Addierer, Subtrahierer).
- Es stellt sich heraus, dass diese Schaltungen besonders einfach sind, wenn Negativzahlen im Zweier-Komplement dargestellt werden.
- Daher wird in der Praxis oft die Zweier-Komplement-Darstellung verwendet.

# Festkommazahlen - Übersicht

**Betrag mit Vorzeichen**

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k}^{n-1} d_i \, 2^i$$

**Einerkomplement**

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots, d_{-k}]_1 := \sum_{i=-k}^{n-1} d_i \, 2^i - d_n(2^n - 2^{-k})$$

**Zweierkomplement**

$$[d_n, d_{n-1}, \ldots, d_0, d_{-1}, \ldots$$
$$:= \sum_{i=-k}^{n-1} d_i \, 2^i - d_n$$

$n = 2, \; k = 0$

| $a$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $[a]_{BV}$ | 0 | 1 | 2 | 3 | 0 | -1 | -2 | -3 |
| $[a]_1$ | 0 | 1 | 2 | 3 | -3 | -2 | -1 | 0 |
| $[a]_2$ | 0 | 1 | 2 | 3 | -4 | -3 | -2 | -1 |

| | | | |
|---|---|---|---|
| | symmetrisch | symmetrisch | asymmetrisch |
| kleinste Zahl | $-(2^n - 2^{-k})$ | $-(2^n - 2^{-k})$ | $-2^n$ |
| größte Zahl | $2^n - 2^{-k}$ | $2^n - 2^{-k}$ | $2^n - 2^{-k}$ |
| Inverses durch | kompl. 1. Bit | kompl. alle Bits | kompl. alle Bits, add. 1 |
| Null | 2 Darstellungen | 2 Darstellungen | 1 Darstellung |
| Abstand | $2^{-k}$ | $2^{-k}$ | $2^{-k}$ |

# Probleme von Festkommazahlen

- Betrachte die Menge aller Zahlen, die eine Zweier-Komplement-Darstellung mit $n$ Vor- und $k$ Nachkommastellen haben.
  - Keine ganz großen bzw. kleinen Zahlen darstellbar!
    - Zahlen mit größtem Absolutbetrag: $-2^n$ und $2^n - 2^{-k}$
    - Zahlen mit kleinstem Absolutbetrag: $-2^{-k}$ und $2^{-k}$
- Operationen sind nicht abgeschlossen!
  - $2^{n-1} + 2^{n-1}$ ist nicht darstellbar, obwohl die Operanden darstellbar sind.
- Assoziativ- und Distributivgesetz gelten nicht, da bei ihrer Anwendung evtl. der darstellbare Zahlenbereich verlassen wird!
  - Beispiel: $(2^{n-1} + 2^{n-1}) - 2^{n-1} \neq 2^{n-1} + (2^{n-1} - 2^{n-1})$

# Gleitkomma-Zahlen

- Die verfügbaren Bits werden in Vorzeichen $S$, Exponent $E$ und Mantisse $M$ unterteilt.
- $a = (-1)^S \cdot M \cdot 2^E$.
- Einfache Genauigkeit (insg. 32 Bit)

| 31 | 30 29 28 27 26 25 24 23 | 22 21 20 19 … 3 2 1 0 |
|---|---|---|
| $S$ | Exponent $E$ | Mantisse $M$ |

- Doppelte Genauigkeit (insg. 64 Bit)

| 63 | 62 61 60 59 … 54 53 52 | 51 50 49 48 … 3 2 1 0 |
|---|---|---|
| $S$ | Exponent $E$ | Mantisse $M$ |

  - Implementierungsdetails: siehe z.B. IEEE754-Standard

# Kapitel 2 – Kodierung

1. Kodierung von Zeichen
2. Kodierung von Zahlen
3. **Anwendung: ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Realisierung von ReTI

# Unterschiede abstrakter/realer ReTI

- Bei realer Maschine nur ein Speicher $M$ für Daten und Befehle.
  - $M$ ist endlich (Größe $2^{32}$). Für $i \in \{0, \ldots, 2^{32} - 1\}$ ist $M(i)$ Inhalt der $i$-ten Speicherzelle.
  - Speicherzellen können Elemente aus $\mathbb{B}^{32}$ aufnehmen.
- CPU-Register $PC, ACC, IN1$ und $IN2$ können nur Elemente $w \in \mathbb{B}^{32}$ aufnehmen. $w$ heißt Wort.
  - Ein Wort kann als Binärzahl (z.B. Adresse im $M$), Zweierkomplementzahl oder Bitstring interpretiert werden.
- Befehle sind ebenfalls Wörter aus $\mathbb{B}^{32}$.

# Notation

- $b^j = \underbrace{(b, \ldots, b)}_{j \text{ mal}}$ für $b \in \{0, 1\}$

- $\langle A \rangle := B$

  ($A$ Register oder Speicherzelle, $B \in \{0, ..., 2^{32} - 1\}$)
  bedeute $A := bin_{32}(B)$
  - Beispiel: $\langle PC \rangle := \langle PC \rangle + 1$

- $[A] := B$

  ($A$ Register oder Speicherzelle, $B \in \{-2^{31}, ..., 2^{31} - 1\}$)
  bedeute $A := twoc_{32}(B)$
  $B$ wird als Zweierkomplement-Zahl interpretiert.

# Befehlsformate

- Zur Erinnerung: Der Befehlssatz von ReTI besteht aus Load-/Store-, Compute-, Indexregister- und Sprungbefehlen.
- Sie werden als Wörter aus $\mathbb{B}^{32}$ kodiert. Etwaige Parameter sind in der Kodierung enthalten.
  - Notation: Sei $I = i_{31}, \ldots, i_0 \in \mathbb{B}^{32}$.
    $I[y,x] := i_y, i_{y-1}, \ldots, i_x$ für $0 \leq x \leq y \leq 31$.
- Allgemeines Instruktionsformat:

| 31 30 | 29 … 24 | 23 … 0 |
|---|---|---|
| Typ | Spezifikation | Parameter i |

# Typ einer Instruktion

| I[31, 30] | Typ |
|---|---|
| 0 0 | Compute |
| 0 1 | Load |
| 1 0 | Store, Move |
| 1 1 | Jump |

| 31 30 | 29 … 24 | 23 … 0 |
|---|---|---|
| Typ | Spezifikation | Parameter i |

# Load-Befehle: Kodierungsprinzip

| 31 30 | 29 28 | 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|
| 0 1 | M | * | D | i |

- M: Modus
- D: Vorerst irrelevant

# Load-Befehle: Kodierung

| Typ | Modus | Befehl | Wirkung | |
|---|---|---|---|---|
| 0 1 | 0 0 | LOAD $i$ | $ACC := M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 0 1 | 0 1 | LOADIN1 $i$ | $ACC := M(\langle IN1 \rangle + [i])$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 0 1 | 1 0 | LOADIN2 $i$ | $ACC := M(\langle IN2 \rangle + [i])$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 0 1 | 1 1 | LOADI $i$ | $ACC := 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |

Durchführung von Rechnungen $\langle x \rangle + [y]$ später.

# Store-, Move-Befehle: Prinzip

| 31 30 | 29 28 | 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|
| 1 0 | M | S | D | i |

- M: Modus
- S: Source
- D: Destination

Kodierung S, D

| S, D | Register |
|---|---|
| 0 0 | PC |
| 0 1 | IN1 |
| 1 0 | IN2 |
| 1 1 | ACC |

# Store-, Move-Befehle: Kodierung

| Typ | Modus | Befehl | Wirkung | |
|---|---|---|---|---|
| 1 0 | 0 0 | STORE $i$ | $M(\langle i \rangle) := ACC$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 1 0 | 0 1 | STOREIN1 $i$ | $M(\langle IN1 \rangle + [i]) := ACC$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 1 0 | 1 0 | STOREIN2 $i$ | $M(\langle IN2 \rangle + [i]) := ACC$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 1 0 | 1 1 | MOVE $S$ $D$ | $D := S$ | $\langle PC \rangle := \langle PC \rangle + 1$ |

außer bei $D = 00$ ($PC$)

# Compute-Befehle: Prinzip

| 31 30 | 29 | 28 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|
| 0 0 | MI | F | D | i |

- MI: „compute **m**emory”/„compute **i**mmediate”
- F: Funktionsfeld
- D: Vorerst irrelevant

# Compute-Befehle: Kodierung

| Typ | MI | F | Befehl | Wirkung | |
|---|---|---|---|---|---|
| 0 0 | 0 | 0 1 0 | SUBI $i$ | $[ACC] := [ACC] - [i]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 0 1 1 | ADDI $i$ | $[ACC] := [ACC] + [i]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 0 | OPLUSI $i$ | $ACC := ACC \oplus 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 1 | ORI $i$ | $ACC := ACC \vee 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 1 0 | ANDI $i$ | $ACC := ACC \wedge 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 0 0 | 1 | 0 1 0 | SUB $i$ | $[ACC] := [ACC] - [M(\langle i \rangle)]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 0 1 1 | ADD $i$ | $[ACC] := [ACC] + [M(\langle i \rangle)]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 0 | OPLUS $i$ | $ACC := ACC \oplus M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 1 | OR $i$ | $ACC := ACC \vee M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 1 0 | AND $i$ | $ACC := ACC \wedge M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |

# Bitstring-Operationen am Beispiel von OPLUS

- $ACC := ACC \oplus 0^8 i_{23} \ldots i_0$
  $\cong (ACC_{31} \oplus 0, \ldots, ACC_{24} \oplus 0, ACC_{23} \oplus i_{23}, \ldots, ACC_0 \oplus i_0)$

# Sprungbefehle: Prinzip

| 31 30 | 29 28 27 | 26 25 24 | 23 … 0 |
|---|---|---|---|
| 1 1 | C | * | i |

- C: Condition

| C | Bedingung c |
|---|---|
| 0 0 0 | nie |
| 0 0 1 | $>$ |
| 0 1 0 | $=$ |
| 0 1 1 | $\geq$ |
| 1 0 0 | $<$ |
| 1 0 1 | $\neq$ |
| 1 1 0 | $\leq$ |
| 1 1 1 | immer |

# Bedingungskodierung nach Schema

| C | Bedingung c |
|---|---|
| 0 0 0 | nie |
| 0 0 1 | $>$ |
| 0 1 0 | = |
| 0 1 1 | $\geq$ |
| 1 0 0 | $<$ |
| 1 0 1 | $\neq$ |
| 1 1 0 | $\leq$ |
| 1 1 1 | immer |

nur $I[29] = 1 \Leftrightarrow <$ wird abgefragt
nur $I[28] = 1 \Leftrightarrow =$ wird abgefragt
nur $I[27] = 1 \Leftrightarrow >$ wird abgefragt

Andere Abfragen durch Kombinationen,
z.B. $C = 101 :< \ \underline{\text{oder}} \ >$, also $\neq$.

# Sprungbefehle: Kodierung

| Typ | Befehl | Wirkung |
|---|---|---|
| 1 1 | JUMP$_c$ $i$ | $\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC]\ c\ 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$ |

- Unbedingte Sprünge werden durch $C = 111$ ausgedrückt.
- Bei $C = 000$: Keine Wirkung des Befehls außer Inkrementieren des Befehlszählers
  $\Rightarrow$ NOP - Befehl (No Operation)

  (In Kap. 8 werden wir kurz auf die Notwendigkeit von NOP - Befehlen bei Piplining eingehen.)

# Zusätzliche Befehle

- Zusätzliche Befehle sind durchaus sinnvoll und bei anderen Architekturen evtl. schon als Grundbefehl vorhanden.
- Nicht vorhandene Befehle müssen hier durch Befehlsfolgen “simuliert” werden.
- Beispiel: Multiplikation, siehe Kapitel 1.2..

# Addition und Sign Extension

- Probleme bei Additionen:
  1. Addition verschieden langer Zahlen (z.B. $[ACC] + [i]$).
  2. Addition von Binärdarstellungen und Zweierkomplementzahlen (z.B. $M(\langle IN1 \rangle + [i]) := ACC$).

- Zu 1: Lösung durch Sign Extension
  - Sei $y \in \mathbb{B}^{24}$. $sext(y) := y_{23}^{8} y$ heißt sign extension von $y$.
  - Es gilt: $[y] = [sext(y)]$. (Beweis: Übung)
  - Dann wird $[ACC] + [i]$ zurückgeführt auf $[ACC] + [sext(i)]$.

- Zu 2: Siehe nächste Folie.

# Addition von Binär- und Zweierkomplementzahlen

## Lemma

Sei $x \in \mathbb{B}^{32}, y \in \mathbb{B}^{24}, 0 \leq \langle x \rangle + [y] < 2^{32}$ und sei
$\langle x \rangle + \langle sext(y) \rangle = \langle c, s \rangle$ mit $c \in \mathbb{B}, s \in \mathbb{B}^{32}$.
Dann gilt: $\langle x \rangle + [y] = \langle s \rangle$

- Bedeutung: Kommt es beim Addieren nicht zum Überlauf ($0 \leq \langle x \rangle + [y] < 2^{32}$, so kann man $x$ und $y$ als Binärzahlen interpretieren, addieren und Übertrag ignorieren!
- Zunächst ohne Beweis.
- Wir können somit Parameter $i$ bei den Befehlen ohne Fallunterscheidung nach $i$ postiv / negativ verwenden!

# Zusammenfassung

- Kodierungen von Zeichen: Codes fester Länge (z.B. ASCII) sind einfacher aber weniger effizient als Codes variabler Länge (z.B. Huffman).
- Zweier-Komplement-Kodierung von Festkomma-Zahlen erlaubt in Verbindung mit Sign Extension effiziente Umsetzung arithmetischer Operationen in Hardware eines Rechners (tatsächliche Implementierung siehe Kapitel 3.5).
- Der Befehlssatz von ReTI ist auf der nächsten Folie zusammengefasst.

| Load-Befehle | | $I[25:24] = D$ | |
|---|---|---|---|
| $I[31:28]$ | Befehl | Wirkung | |
| 0100 | LOAD $D$ $i$ | $D := M(\langle i \rangle)$ | |
| 0101 | LOADIN1 $D$ $i$ | $D := M(\langle IN1 \rangle + [i])$ | $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$ |
| 0110 | LOADIN2 $D$ $i$ | $D := M(\langle IN2 \rangle + [i])$ | |
| 0111 | LOADI $D$ $i$ | $D := 0^8 i$ | |

| Store-Befehle | | MOVE: $I[27:24] = SD$ | |
|---|---|---|---|
| $I[31:28]$ | Befehl | Wirkung | |
| 1000 | STORE $i$ | $M(\langle i \rangle) := ACC$ | |
| 1001 | STOREIN1 $i$ | $M(\langle IN1 \rangle + [i]) := ACC$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 1010 | STOREIN2 $i$ | $M(\langle IN2 \rangle + [i]) := ACC$ | |
| 1011 | MOVE $S$ $D$ | $D := S$ | $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$ |

| Compute-Befehle | | $I[25:24] = D$ | |
|---|---|---|---|
| $I[31:26]$ | Befehl | Wirkung | |
| 000010 | SUBI $D$ $i$ | $[D] := [D] - [i]$ | |
| 000011 | ADDI $D$ $i$ | $[D] := [D] + [i]$ | |
| 000100 | OPLUSI $D$ $i$ | $D := D \oplus 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$ |
| 000101 | ORI $D$ $i$ | $D := D \vee 0^8 i$ | |
| 000110 | ANDI $D$ $i$ | $D := D \wedge 0^8 i$ | |
| 001010 | SUB $D$ $i$ | $[D] := [D] - [M(\langle i \rangle)]$ | |
| 001011 | ADD $D$ $i$ | $[D] := [D] + [M(\langle i \rangle)]$ | |
| 001100 | OPLUS $D$ $i$ | $D := D \oplus M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$ |
| 001101 | OR $D$ $i$ | $D := D \vee M(\langle i \rangle)$ | |
| 001110 | AND $D$ $i$ | $D := D \wedge M(\langle i \rangle)$ | |

| Jump-Befehle | | |
|---|---|---|
| $I[31:27]$ | Befehl | Wirkung |
| 11000 | NOP | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 11001 | JUMP$_{>}$ $i$ | $\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC]\ c\ 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$ |
| 11010 | JUMP$_{=}$ $i$ | |
| 11010 | JUMP$_{\geq}$ $i$ | |
| 11011 | JUMP$_{<}$ $i$ | |
| 11100 | JUMP$_{\neq}$ $i$ | |
| 11110 | JUMP$_{\leq}$ $i$ | |
| 11111 | JUMP $i$ | $\langle PC \rangle := \langle PC \rangle + [i]$ |

Kodierung S,D

| S, D | Register |
|---|---|
| 0 0 | $PC$ |
| 0 1 | $IN1$ |
| 1 0 | $IN2$ |
| 1 1 | $ACC$ |

# Kapitel 3 – Kombinatorische Logik

1. **Kombinatorische Schaltkreise**
2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Kombinatorische Schaltkreise

## Definition

Kombinatorische Logik ist ein Modell von Hardware, die eine boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}^m (n, m \in \mathbb{N})$ implementiert.

- Ein kombinatorischer Schaltkreis (Schaltnetz) hat $n$ Eingänge und $m$ Ausgänge. Legt man an den Eingängen den Vektor $i \in \mathbb{B}^n$ an, berechnet der Schaltkreis den Vektor $f(i) \in \mathbb{B}^m$ und stellt ihn an den Ausgängen bereit.

- Es gibt weitere Arten von Hardware:
  - Sequentielle Logik mit speichernden Elementen (später).
  - Analog- und Mixed-Signal-Blöcke (nicht in TI).

# Kombinatorische Logiksynthese

- Kombinatorische Logiksynthese ist das Problem, zu einer gegebenen Booleschen Funktion einen möglichst effizienten kombinatorischen Schaltkreis, d. h. einen mit möglichst geringen Kosten, zu finden.

- Die Definition von Kosten hängen von der verwendeten Technologie ab und können sich auf die Größe, Verzögerung, Energieverbrauch des Schaltkreises beziehen und eventuell weitere Parameter (Zuverlässigkeit, Testbarkeit, ...) berücksichtigen.

# Technologien

- Wir konzentrieren uns hier auf zwei Arten von Technologien:

  1. Programmierbare Logikfelder (Programmable Logic Arrays, PLAs).
     - Implementieren sogenannter zweistufigen Realisierungen.
  2. Mehrstufige Realisierungen mit allgemeinen Bibliothekszellen (Logik-Gattern).

# Logikgatter

- Gatter sind kleine kombinatorische Blöcke, in der Regel mit bis zu 4 Eingängen und einem Ausgang.
- Gatter werden mit Transistoren realisiert.
- Gatter können zu größeren Schaltungen verbunden werden.
- Die Menge der verfügbaren Gattern ergibt eine Standardzellen-Bibliothek.

# Einige wichtige Gatter

| $i_1$ | $i_2$ | $AND_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $i_1$ | $i_2$ | $OR_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $i_1$ | $NOT_2$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

| $i_1$ | $i_2$ | $NAND_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $i_1$ | $i_2$ | $NOR_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

| $i_1$ | $i_2$ | $XOR_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Logikgatter - verschiedene Notationen

Es gibt verschiedene Notationen für Logikgatter.

| | IEC | ANSI | DIN |
|---|---|---|---|
| $OR_2$ | | | |
| $NOT$ | | | |

Wir werden in dieser Vorlesung die ANSI-Notation verwenden.

# Transistoren

- Einen Transistor kann man vereinfacht als spannungsgesteuerten Schalter sehen:
  - Leitung *g* (gate) regelt Leitfähigkeit zwischen Quelle und Senke.

p-Kanal Transistor

- Leitet, wenn an *g* eine 0 anliegt.
- Sperrt, wenn an *g* eine 1 anliegt.

n-Kanal Transistor

- Leitet, wenn an *g* eine 1 anliegt.
- Sperrt, wenn an *g* eine 0 anliegt.

# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.

# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).

# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).

# CMOS-NAND-Gatter

- Ausgang ist 0
  $\Leftrightarrow$ Es existiert ein leitender Pfad von 0 zum Ausgang
  $\Leftrightarrow$ beide n-Kanal-Transistoren leiten
  $\Leftrightarrow a = b = 1, a \wedge b = 1$
  $\Leftrightarrow NAND(a,b) = 0$

- Ausgang ist 1
  $\Leftrightarrow$ Es existiert ein leitender Pfad von 1 zum Ausgang
  $\Leftrightarrow$ einer der p-Kanal-Transistoren leitet
  $\Leftrightarrow a = 0$ oder $b = 0, \neg a \vee \neg b = 1$
  $\Leftrightarrow NAND(a,b) = 1$

# Weitere CMOS-Gatter

- Es gibt keine „direkte” Implementierung von AND- und OR-Gattern. Sie werden aus NAND-/NOR-Gattern plus Invertern zusammengesetzt.
- Zu jedem p-Kanal Transistor gibt es stets einen komplementären n-Kanal-Transistor, der genau dann sperrt, wenn der erste Transistor leitet und umgekehrt. Dadurch gibt es niemals einen leitenden Pfad von der Stromversorgung zur Masse. Dies reduziert Leistungsverluste.

# Schaltkreise

- Idee:
  „gerichteter Graph mit einigen zusätzlichen Eigenschaften”

# Gerichtete Graphen (Wiederholung)

- $G = (V, E)$ ist ein gerichteter Graph, genau dann, wenn
  - $V$ endliche nichtleere Menge („Knoten")
  - $E$ endliche Menge („Kanten")
  - Auf $E$ sind Abbildungen $Q, Z : E \to V$ definiert ($Q(e)$ heißt Quelle, $Z(e)$ Ziel einer Kante $e$)
- Die Abbildung $indeg : V \to \mathbb{N}, indeg(v) = |\{e \mid Z(e) = v\}|$ gibt den Eingangsgrad eines Knotens $v \in V$ an.
- Die Abbildung $outdeg : V \to \mathbb{N}, outdeg(v) = |\{e \mid Q(e) = v\}|$ gibt den Ausgangsgrad eines Knotens $v \in V$ an.
- Ein Pfad (der Länge k) in $G$ ist eine Folge von $k$ Kanten $e_1, e_2, \ldots, e_k (k \geq 0)$ mit $Z(e_i) = Q(e_{i+1})\ \forall i$ mit $k - 1 \geq i \geq 1$. $Q(e_1)$ heißt Quelle, $Z(e_k)$ Ziel des Pfades.
- Ein Zyklus in $G$ ist ein Pfad der Länge $\geq 1$ in $G$, bei dem Ziel und Quelle identisch sind.
- $G$ heißt azyklisch, falls kein Zyklus in $G$ existiert.
- Die Graph-Tiefe eines azyklischen Graphen ist definiert als die Länge des längsten Pfades in $G$.

# Modellierung durch Schaltkreise (1/2)

- Eine Zellenbibliothek $BIB \subset \bigcup_{n \in \mathbb{N}} \mathbb{B}_n$ enthält Basisoperatoren, die den Grundgattern entsprechen.
- Ein 5-Tupel $SK = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$ heißt Schaltkreis mit $n$ Eingängen und $m$ Ausgängen über der Zellenbibliothek $BIB$ genau dann wenn
  - $\vec{X}_n = (x_1, \ldots, x_n)$ ist eine endliche Folge von Eingängen.
  - $G = (V, E)$ ist ein azyklischer, gerichteter Graph mit $\{0,1\} \cup \{x_1, \ldots, x_n\} \subseteq V$.
  - Die Menge $I = V \setminus (\{0,1\} \cup \{x_1, \ldots, x_n\})$ heißt Menge der Gatter. Die Abbildung $typ : I \to BIB$ ordnet jedem Gatter $v \in I$ einen Zellentyp $typ(v) \in BIB$ zu.
  - ...

# Modellierung durch Schaltkreise (2/2)

- ...
- Für jedes Gatter $v \in I$ mit $typ(v) \in \mathbb{B}_k$ gilt $indeg(v) = k$.
- $indeg(v) = 0$ für $v \in \{0,1\} \cup \{x_1, \ldots, x_n\}$.
- Die Abbildung $IN : I \to E^*$ legt für jedes Gatter $v \in I$ eine Reihenfolge der eingehenden Kanten fest, d.h. falls $indeg(v) = k$, dann ist $IN(v) = (e_1, \ldots, e_k)$ mit $Z(e_i) = v \; \forall 1 \leq i \leq k$.
- Die Folge $\vec{Y}_m = (y_1, \ldots, y_m)$ zeichnet Knoten $y_i \in V$ als Ausgänge aus.

# Formale Semantikdefinition für Schaltkreise (1/2)

- Sei $SK = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$ ein Schaltkreis über einer Zellenbibliothek *BIB*.
- Sei eine Eingangsbelegung $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{B}^n$ gegeben.
- Eine Belegung $\Phi_{SK,\alpha} : V \to \mathbb{B}$ für alle Knoten $v \in V$ ist dann gegeben durch die folgenden Definitionen:
  - $\Phi_{SK,\alpha}(x_i) = \alpha_i \ \forall 1 \leq i \leq n$.
  - $\Phi_{SK,\alpha}(0) = 0$, $\Phi_{SK,\alpha}(1) = 1$.
  - falls $v \in I$ mit $typ(v) = g \in \mathbb{B}_k$, $IN(v) = (e_1, \ldots, e_k)$, dann ist $\Phi_{SK,\alpha}(v) = g(\Phi_{SK,\alpha}(Q(e_1)), \ldots, \Phi_{SK,\alpha}(Q(e_k)))$.

- Warum ist das wohldefiniert?
- Weil $G$ azyklisch!

# Formale Semantikdefinition für Schaltkreise (2/2)

- $(\Phi_{SK,\alpha}(y_1), \ldots, \Phi_{SK,\alpha}(y_m))$ ist dann die unter Eingangsbelegung $\alpha = (\alpha_1, \ldots, \alpha_n)$ berechnete Ausgangsbelegung des Schaltkreises $SK$.
- Die Berechnung von $\Phi_{SK,\alpha}$ bei Eingangsbelegung $\alpha$ heißt auch Simulation von SK für Belegung $\alpha$.

- Die an einem Knoten $v$ berechnete Boolesche Funktion $\Psi(v) : \mathbb{B}^n \to \mathbb{B}$ ist definiert durch

$$\Psi(v)(\alpha) := \Phi_{SK,\alpha}(v)$$

  für ein beliebiges $\alpha \in \mathbb{B}^n$.
- Die durch den Schaltkreis berechnete Funktion ist

$$f_{SK} := (\Psi(y_1), \ldots, \Psi(y_m)).$$

# Standardzellen-Bibliothek

- Eine Standardzellen-Bibliothek enthält eine Menge von Gattern und kleinen kombinatorischen Schaltungen (Standardzellen).
  - Z. B. einen 8-Bit-Addierer
- Für jedes Element der Bibliothek werden Parameter wie Fläche auf dem Chip, Schaltgeschwindigkeit, Leistungsaufnahme des Gatters bzw. der Standardzelle abgespeichert.
- Es sind oft z. B. mehrere Inverter unterschiedlicher Größe und Geschwindigkeit vorhanden.

# Kombinatorische Logiksynthese

- Allgemeine kombinatorische Logiksynthese optimiert mehrere Parameter gleichzeitig.
- Exakte Verfahren existieren, stoßen aber schon für kleinste Schaltkreise an ihre Grenzen.
- In der Praxis werden Heuristiken eingesetzt, die auf Ausschnitten eines großen Schaltkreises lokale Optimierungen durchführen.
- Hier beschränken wir uns auf eine wichtige Unterklasse von kombinatorischen Schaltkreisen:
  Die zweistufige Logik.
- Allgemeinere kombinatorische Schaltkreise betrachten wir später bei der Einführung arithmetischer Schaltkreise.

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. **Normalformen, zweistufige Synthese**
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Kombinatorische Schaltkreise

- **Ziel:**
  Wir werden zeigen, dass sich jede boolesche Funktion als ein Polynom, also eine Disjunktion (ODER-Verknüpfung) von Monomen, die ihrerseits Konjunktionen (UND-Verknüpfungen) von Eingangsvariablen und negierten Eingangsvariablen sind, darstellen lässt.

- Wir werden für solche Darstellungen Kostenkriterien aufstellen und diese optimieren.

- Monome und Polynome sind spezielle boolesche Ausdrücke (s. auch Kap. 7)

# Beispielfunktion

- 3 Eingangsvariablen $x_1, x_2, x_3$
- Wird im Folgenden zur Illustration verwendet.

| $x_1$ | $x_2$ | $x_3$ | $f_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# Literale

## Definition

Als Literal einer booleschen Funktion $f \in \mathbb{B}_n$ wird der Ausdruck $x_i$ oder $x_i'$ bezeichnet, wobei $x_i, i \in 1, \ldots, n$, eine Eingangsvariable von $f$.

- $x_i$ (auch $x_i^1$ geschrieben) wird positives Literal,
  $x_i'$ (auch $x_i^0$ geschrieben) wird negatives Literal genannt.

- Literale werden zur kompakten Beschreibung von booleschen Funktionen verwendet. So bezeichnet das Literal $x_i$ die boolesche Funktion $g \in \mathbb{B}_n$ mit $g(\alpha_1, \ldots, \alpha_n) = 1$ genau dann, wenn $\alpha_i = 1$.
  Analog gilt: $x_i'$ bezeichnet die boolesche Funktion $h \in \mathbb{B}_n$ mit $h(\alpha_1, \ldots, \alpha_n) = 1$ genau dann, wenn $\alpha_i = 0$.
  Wir schreiben vereinfachend auch $g = x_i$ bzw. $h = x_i'$.

# Monome

## Definition

- Ein Monom ist eine Konjunktion von Literalen, in der kein Literal mehr als einmal vorkommt und zu keiner Variable sowohl das positive als auch das negative Literal vorkommt. Außerdem ist „1" ein Monom.
  - $x_1 x_2 x_3'$ und $x_1' x_3$ sind Monome, $x_2 x_3 x_3'$ ist kein Monom.
- Ein Monom heißt vollständig oder Minterm, wenn jede Variable entweder als positives oder als negatives Literal vorkommt.
  - Wenn drei Variablen $x_1, x_2, x_3$ betrachtet werden, ist $x_1 x_2 x_3'$ ein Minterm, $x_1' x_3$ ist kein Minterm.
- Für eine Eingabebelegung $\alpha \in \mathbb{B}^n$ heißt

$$m(\alpha) = \bigwedge_{i=1}^{n} x_i^{\alpha_i} \quad \text{(Notation: } x_i^1 := x_i, x_i^0 := x_i'\text{)}$$

der zu $\alpha$ gehörende Minterm.

# Monome als Beschreibung boolescher Funktionen

- Beispiel: Das Monom $m = x_i x_j'$ bezeichnet die boolesche Funktion $f \in \mathbb{B}_n$ mit $f(\alpha_1, \ldots, \alpha_n) = 1$ genau dann, wenn $\alpha_i = 1$ und $\alpha_j = 0$. Wir schreiben vereinfachend auch $f = x_i x_j'$.

# Polynome

- Eine Disjunktion von paarweise verschiedenen Monomen heißt Polynom. Sind alle Monome des Polynoms vollständig, so heißt das Polynom vollständig.

  Beispiel: Bei einer booleschen Funktion mit drei Variablen $x_1, x_2, x_3$ wäre:
  - $x_1' x_2 + x_2' x_3$ ein Polynom.
  - $x_1' x_2' x_3 + x_1 x_2' x_3$ ein vollständiges Polynom.
  - Das Polynom $x_1' x_2 + x_2' x_3$ beschreibt die boolesche Funktion $f \in \mathbb{B}_3$ mit $f(\alpha_1, \alpha_2, \alpha_3) = 1$ genau dann, wenn $\alpha_1 = 0, \alpha_2 = 1$ oder $\alpha_2 = 0, \alpha_3 = 1$. Wir schreiben vereinfachend auch $f = x_1' x_2 + x_2' x_3$.

# Disjunktive Normalform

- Eine disjunktive Normalform (DNF) von $f$ ist ein Polynom, das $f$ beschreibt. Eine kanonische disjunktive Normalform (KDNF) von $f$ ist ein vollständiges Polynom von $f$.
  - $f_1 = x_1' x_2' + x_2' x_3 + x_1 x_2$ ist in DNF, aber nicht in KDNF.

# Bestimmung der kanonischen disjunktiven Normalform

- Für $f : \mathbb{B}^n \to \mathbb{B}$ heißt $ON(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 1\}$ die Erfüllbarkeitsmenge von $f$.
- Die KDNF ist gegeben durch $f = \sum_{\alpha \in ON(f)} m(\alpha)$
- Die KDNF ist (bis auf Anordnung von Literalen in Mintermen und von Termen im Polynom) eindeutig.
- Beispiel: KDNF für $f_1 = x_1' x_2' + x_2' x_3 + x_1 x_2$

$$
\begin{aligned}
f_1 &= m(000) + m(001) \\
&\quad + m(101) + m(110) + m(111) \\
&= x_1' x_2' x_3' + x_1' x_2' x_3 \\
&\quad + x_1 x_2' x_3 + x_1 x_2 x_3' + x_1 x_2 x_3
\end{aligned}
$$

| $x_1$ | $x_2$ | $x_3$ | $f_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

- Anmerkung: Analog zur Erfüllbarkeitsmenge ist $OFF(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 0\}$ als die Unerfüllbarkeitsmenge definiert.

# Realisierungen von DNF

- Erste Möglichkeit: Benutze „gewöhnliche” UND- und ODER-Gatter.
- Zweite Möglichkeit: PLAs
  - Programmierbare logische Felder können nur Funktionen in DNF implementieren.
  - Sie benötigen dafür weniger Transistoren als eine Realisierung mit UND- und ODER-Gattern.

# Realisierung durch Logikgatter

- Bilde erst alle Monome durch UND-Gatter.
- Verbinde dann alle Monome mit ODER-Gattern.
  - Notation: Man verzichtet in der Regel auf die Abbildung von Invertern.

- Die Kosten ergeben sich dann aus allen benötigten UND- und ODER-Gattern.

# Programmierbare logische Felder (PLA)

Zweistufige Darstellung zur Realisierung von booleschen Polynomen.
$f_i = m_{i1} + m_{i2} + \cdots + m_{ik}$ mit $m_{iq}$ aus $\{m_1, \ldots, m_s\}$

Enthält Monom $m_j$ $k$ Literale, so werden $k$ Transistoren in der entsprechenden Zeile des **UND-Feldes** benötigt.

Besteht die Beschreibung von Funktion $f_t$ aus $p$ Monomen, so benötigt man $p$ Transistoren in der entsprechenden Spalte des **ODER-Feldes**.

Fläche: $\sim (m + 2n) \times$ (Anzahl der benötigten Monome)

# PLAs: Realisierungsdetails

## Beispiel

$f_1 = x_1' x_2' + x_2' x_3 + x_1 x_2$
$f_2 = x_2' x_3$

# Konkrete Belegung

## Beispiel

$f_1 = x_1' x_2' + x_2' x_3 + x_1 x_2$
$f_2 = x_2' x_3$

Bei Belegung von $x_1 = 1, x_2 = 1$ und $x_3 = 1$ liegt folgende Situation vor:

# Kosten von PLA-Realisierungen von Monomen und Polynomen

- Sei $q = q_1 \cdot \dots \cdot q_r$ ein Monom, dann sind die Kosten $|q|$ von $q$ gleich der Anzahl der zur Realisierung von $q$ benötigten Transistoren im PLA, also $|q| := r$.
- Seien $p_1, \dots, p_m$ Polynome, dann bezeichne $M(p_1, \dots, p_m)$ die Menge der in diesen Polynomen verwendeten Monome.
  - Die primären Kosten $cost_1(p_1, \dots, p_m)$ einer Menge $p_1, \dots, p_m$ von Polynomen sind gleich der Anzahl der benötigten Zeilen im PLA, um $p_1, \dots, p_m$ zu realisieren, also $cost_1(p_1, \dots, p_m) = |M(p_1, \dots, p_m)|$.
  - Die sekundären Kosten $cost_2(p_1, \dots, p_m)$ einer Menge $\{p_1, \dots, p_m\}$ von Polynomen sind gleich der Anzahl der benötigten Transistoren im PLA, also $cost_2(p_1, \dots, p_m) = \sum_{q \in M(p_1, \dots, p_m)} |q| + \sum_{i=1,\dots,m} |M(p_i)|$.

# Kombiniertes Kostenmaß

- Sei im Folgenden $cost = (cost_1, cost_2)$ die Kostenfunktion mit der Eigenschaft, dass für zwei Polynommengen $\{p_1, \ldots, p_m\}$ und $\{p'_1, \ldots, p'_m\}$ die Ungleichung

$$cost(p_1, \ldots, p_m) \leq cost(p'_1, \ldots, p'_m)$$

  gilt, wenn

  - entweder $cost_1(p_1, \ldots, p_m) < cost_1(p'_1, \ldots, p'_m)$
  - oder $cost_1(p_1, \ldots, p_m) = cost_1(p'_1, \ldots, p'_m)$ und $cost_2(p_1, \ldots, p_m) \leq cost_2(p'_1, \ldots, p'_m)$

# Das Problem der zweistufigen Logikminimierung

- **Gegeben:** Eine boolesche Funktion $f = (f_1, \ldots, f_m)$ in $n$ Variablen und $m$ Ausgängen in Form einer Tabelle der Dimension $(n+m) \cdot 2^n$ oder einer Menge von $m$ Polynomen $\{q_1, \ldots, q_m\}$ mit $f_i = q_i$.
- **Gesucht:** $m$ Polynome $p_1, \ldots p_m$, so dass $p_i$ für alle $i$ der Funktion $f_i$ entspricht und die Kosten $cost(p_1, \ldots, p_m)$ minimal sind.
- Ab sofort werden nur noch Funktionen mit einem Ausgang betrachtet.

# Veranschaulichung von Monomen und Polynomen

1-dim Würfel

2-dim Würfel

3-dim Würfel

4-dim Würfel

# Veranschaulichung durch Würfel

- Jede boolesche Funktion $f$ in $n$ Variablen und einem Ausgang kann über einen $n$-dimensionalen Würfel durch Markierung der $ON(f)$-Menge spezifiziert werden.
  - **Beispiel**:
    $f(x_1, x_2, x_3, x_4) = x_1 x_2 + x_1' x_2' x_3' + x_1 x_2' x_3' x_4$

# Monome und Polynome als Teilwürfel

- Monome der Länge $k$ entsprechen $(n-k)$-dimensionalen Teilwürfeln!
- Ein Polynom entspricht einer Vereinigung von Teilwürfeln.
- **Beispiel**:
  $f(x_1, x_2, x_3, x_4)$
  $= x_1 x_2 + x_1' x_2' x_3'$
  $+ x_1 x_2' x_3' x_4$

# Zweistufige Logikminimierung als Überdeckungsproblem auf dem Würfel

- **Gegeben**: Boolesche Funktion $f$ in $n$ Variablen und **einem** Ausgang, in Form eines markierten $n$-dimensionalen Würfels.

- **Gesucht**: Eine minimale Überdeckung der markierten Knoten durch maximale Teilwürfel im $n$-dimensionalen Würfel.

Entspricht einer Minimallösung:

$$x_1 x_2 + x_1' x_2' x_3' + x_2' x_3' x_4$$

# Implikanten und Primimplikanten

## Definition

Eine boolesche Funktion $f \in \mathbb{B}_n$ ist kleiner gleich einer anderen booleschen Funktion $g \in \mathbb{B}_n$ ($f \leq g$), wenn $\forall \alpha \in \mathbb{B}_n : f(\alpha) \leq g(\alpha)$.
(Das heißt, wenn $f$ an einer Stelle 1 ist, dann auch $g$.)

## Definition

Sei $f$ eine boolesche Funktion mit einem Ausgang. Ein Implikant von $f$ ist ein Monom $q$ mit $q \leq f$. Ein Primimplikant von $f$ ist ein maximaler Implikant $q$ von $f$, das heißt es gibt keinen Implikanten $s$ ($s \neq q$) von $f$ mit $q \leq s$.
Implikanten und Primimplikanten können durch $n$-dimensionale Würfel veranschaulicht werden.

# Veranschaulichung durch Würfel

- Ein Implikant von $f$ ist ein Teilwürfel, der nur markierte Knoten enthält.
- Ein Primimplikant von $f$ ist ein maximaler Teilwürfel mit dieser Eigenschaft.

# Illustration für konkrete Funktion

**Implikanten**

- alle markierten Knoten
- alle Kanten, deren Ecken alle markiert sind
- alle Flächen, deren Ecken alle markiert sind
- alle 3-dimensionalen Würfel, deren Ecken alle markiert sind

**Allgemein**

- Die Implikanten sind die Teilwürfel, deren Ecken alle markiert sind.

# Bestimmung von Primimplikanten

Es gibt 3 Primimplikanten, der durch unseren Würfel definierten Funktion:

- $x_2$

# Bestimmung von Primimplikanten

Es gibt 3 Primimplikanten, der durch unseren Würfel definierten Funktion:

- $x_2$
- $x_1' x_3'$

# Bestimmung von Primimplikanten

Es gibt 3 Primimplikanten, der durch unseren Würfel definierten Funktion:

- $x_2$
- $x_1' x_3'$
- $x_3' x_4$

# Polynome und Implikanten einer Funktion $f$

## Lemma

Die Monome eines Polynoms $p$ von $f$ sind alle Implikanten von $f$.

**Beweis:**

- Enthält ein Polynom $p$ von $f$ ein Monom $m$, welches nicht Implikant von $f$ ist, so gilt nicht: $m \leq f$
- Das heißt es gibt eine Belegung $(\alpha_1, \ldots, \alpha_n)$ der Variablen $(x_1, \ldots, x_n)$ mit
  - $f(\alpha_1, \ldots, a_n) = 0$
  - $m(\alpha_1, \ldots, \alpha_n) = 1$, also auch $p(\alpha_1, \ldots, \alpha_n) = 1$
- Das heißt Monom $m$ ist ein Teilwürfel, bei dem eine Ecke nicht markiert ist.

Demnach ist Polynom $p$ kein Polynom von $f$.

$\Rightarrow$ **Widerspruch!**

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
3. **Berechnung eines Minimalpolynoms**
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Billigste Überdeckung der markierten Ecken

Wir suchen ein sogenanntes Minimalpolynom, das heißt ein Polynom mit minimalen Kosten.

## Definition

Ein Minimalpolynom $p$ einer booleschen Funktion $f$ ist ein Polynom von $f$ mit minimalen Kosten, das heißt mit der Eigenschaft $cost(p) \leq cost(p')$ für jedes andere Polynom $p'$ von $f$.

# Quine's Primimplikantensatz

## Satz

Jedes Minimalpolynom $p$ einer booleschen Funktion $f$ besteht ausschließlich aus Primimplikanten von $f$.

**Beweis:**

- Nehme an, dass $p$ einen nicht primen Implikanten $m$ von $f$ enthält.
- $m$ wird durch einen Primimplikanten $m'$ von $f$ überdeckt, ist also in $m'$ enthalten.
- Es gilt demnach $cost(m') < cost(m)$.
- Ersetzt man in $p$ den Implikanten $m$ durch den Primimplikanten $m'$, so erhält man ein Polynom $p'$, das ein Polynom von $f$ ist mit $cost(p') < cost(p)$.
- **Widerspruch** dazu, dass $p$ ein Minimalpolynom ist.

# Berechnung von Implikanten

## Lemma 1

Ist $m$ ein Implikant von $f$, so auch $m \cdot x$ und $m \cdot x'$ für jede Variable $x$, die in $m$ weder als positives, noch als negatives Literal vorkommt.

**Beweis:**

- $m \cdot x$ und $m \cdot x'$ sind Teilwürfel des Würfels $m$.
- Sind also alle Ecken von $m$ markiert, so auch alle Ecken von $m \cdot x$ und $m \cdot x'$.

## Lemma 2

Sind $m \cdot x$ und $m \cdot x'$ Implikanten von $f$, so auch $m$.

**Beweis:**

- $f \geq m \cdot x + m \cdot x' = m \cdot (x + x') = m$

# Charakterisierung von Implikanten

## Satz

Ein Monom $m$ ist genau dann ein Implikant von $f$, wenn entweder

- $m$ ein Minterm von $f$ ist, oder
- $m \cdot x$ und $m \cdot x'$ Implikanten von $f$ sind für eine Variable $x$, die nicht in $m$ vorkommt.

- Äquivalente Schreibweise:

  $m \in Implikant(f)$

  $\Leftrightarrow (m \in Minterm(f)) \vee (m \cdot x, m \cdot x' \in Implikant(f))$

- Beweis folgt unmittelbar aus Lemma 1 und Lemma 2.

# Berechnung eines Minimalpolynoms

- Verfahren von Quine-McCluskey zur Berechnung aller Primimplikanten.
  - Idee: Berechne sogar alle Implikanten. Dann ist klar, welche Primimplikanten sind.

- Verfahren zur Lösung des „Überdeckungsproblems".
  - Treffe unter den Primimplikanten eine geeignete Auswahl, so dass die Disjunktion der ausgewählten Primimplikanten ein Polynom für $f$ ist und minimale Kosten hat.

# Verfahren von Quine: Der Algorithmus

Prime implicants function **Quine** ($f : \mathbb{B}^n \to \mathbb{B}$)

**begin**
  $L_0 := Minterm(f)$;
  $i := 0$;
  //$L_i$ enthält alle Implikanten von $f$ der Länge $n - i$.
  $Prim(f) := \emptyset$
  **while** ($L_i \neq \emptyset$) **and** ($i < n$)
  **loop** $L_{i+1} := \{m \mid m \cdot x$ und $m \cdot x'$ sind in $L_i$ für ein $x\}$;
    $Prim(f) := Prim(f) \cup$
    $\{m' \mid m' \in L_i$ und $m'$ wird von keinem $q \in L_{i+1}$ überdeckt$\}$;
    $i := i + 1$
  **end loop;**
  **return** $Prim(f) \cup L_i$;
**end;**

# Verbesserung durch McCluskey

- Vergleiche nur Monome untereinander
  - die die gleichen Variablen enthalten und
  - bei denen sich die Anzahl der positiven Literale nur um 1 unterscheidet.
- Dies wird erreicht durch:
  - Partitionierung von $L_i$ in Klassen $L_i^M$, mit $M \subseteq \{x_1, \ldots, x_n\}$ und $|M| = n - i$.
  - $L_i^M$ enthält die Implikanten aus $L_i$, deren Literale alle aus $M$ sind.
  - Anordnung der Monome in $L_i^M$ gemäß der Anzahl der positiven Literale.

# Beispiel Quine-McCluskey

$L_0^{\{x_1,x_2,x_3,x_4\}}$:

| |
|---|
| 0 0 0 0 |
| 0 0 0 1 |
| 0 1 0 0 |
| 1 0 0 0 |
| 0 0 1 1 |
| 0 1 0 1 |
| 1 0 0 1 |
| 1 0 1 0 |
| 1 1 0 0 |
| 0 1 1 1 |
| 1 1 0 1 |
| 1 1 1 0 |

steht für $x_1'x_2'x_3'x_4'$

steht für $x_1x_2'x_3'x_4$

Vergleiche im Folgenden nur Monome aus benachbarten Blöcken!

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (1/4)

$L_0^{\{x_1,x_2,x_3,x_4\}}$:

| |
|---|
| 0 0 0 0 |
| 0 0 0 1 |
| 0 1 0 0 |
| 1 0 0 0 |
| 0 0 1 1 |
| 0 1 0 1 |
| 1 0 0 1 |
| 1 0 1 0 |
| 1 1 0 0 |
| 0 1 1 1 |
| 1 1 0 1 |
| 1 1 1 0 |

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (2/4)

$L_0^{\{x_1,x_2,x_3,x_4\}}$:

0 0 0 0
0 0 0 1
0 1 0 0
1 0 0 0
0 0 1 1
0 1 0 1
1 0 0 1
1 0 1 0
1 1 0 0
0 1 1 1
1 1 0 1
1 1 1 0

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -

$L_1^{\{x_1,x_3,x_4\}}$:

0 - 0 0

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (3/4)

$L_0^{\{x_1,x_2,x_3,x_4\}}$:

| |
|---|
| 0 0 0 0 |
| 0 0 0 1 |
| 0 1 0 0 |
| 1 0 0 0 |
| 0 0 1 1 |
| 0 1 0 1 |
| 1 0 0 1 |
| 1 0 1 0 |
| 1 1 0 0 |
| 0 1 1 1 |
| 1 1 0 1 |
| 1 1 1 0 |

$L_1^{\{x_1,x_2,x_3\}}$:

| |
|---|
| 0 0 0 - |

$L_1^{\{x_1,x_3,x_4\}}$:

| |
|---|
| 0 - 0 0 |
| 0 - 1 1 |

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (4/4)

$L_0^{\{x_1,x_2,x_3,x_4\}}$:

| |
|---|
| 0 0 0 0 |
| 0 0 0 1 |
| 0 1 0 0 |
| 1 0 0 0 |
| 0 0 1 1 |
| 0 1 0 1 |
| 1 0 0 1 |
| 1 0 1 0 |
| 1 1 0 0 |
| 0 1 1 1 |
| 1 1 0 1 |
| 1 1 1 0 |

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -

$L_1^{\{x_1,x_3,x_4\}}$:

| |
|---|
| 0 - 0 0 |
| 0 - 1 1 |

Nicht kürzbar, da nicht Ecken der gleichen Kante. (Consensus existiert nicht!)

# Beispiel Quine-McCluskey: Alle bestimmten Mengen $L_1$

$L_1^{\{x_1,x_2,x_4\}}$:

0 0 - 1
1 0 - 0
0 1 - 1
1 1 - 0

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -
0 1 0 -
1 0 0 -
1 1 0 -

$L_1^{\{x_2,x_3,x_4\}}$:

- 0 0 0
- 0 0 1
- 1 0 0
- 1 0 1

$L_1^{\{x_1,x_3,x_4\}}$:

0 - 0 0
0 - 0 1
1 - 0 0
0 - 1 1
1 - 0 1
1 - 1 0

Alle Minterme von $f$ sind Eckpunkte von Kanten, die Implikanten sind: $Prim(f) = \emptyset$

# Beispiel Quine-McCluskey: Bestimmung von $L_2$ (1/2)

$L_1^{\{x_1,x_2,x_4\}}$:

0 0 - 1
1 0 - 0
0 1 - 1
1 1 - 0

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -
0 1 0 -
1 0 0 -
1 1 0 -

$L_1^{\{x_2,x_3,x_4\}}$:

- 0 0 0
- 0 0 1
- 1 0 0
- 1 0 1

$L_1^{\{x_1,x_3,x_4\}}$:

0 - 0 0
0 - 0 1
1 - 0 0
0 - 1 1
1 - 0 1
1 - 1 0

Alle Implikanten aus $L_1\{x_1,x_2,x_4\}$ sind Kanten von Flächen, die Implikanten sind: $Prim(f) = \emptyset$

# Beispiel Quine-McCluskey: Bestimmung von $L_2$ (2/2)

$L_1^{\{x_1,x_2,x_4\}}$:

0 0 - 1
1 0 - 0
---
0 1 - 1
1 1 - 0

$L_1^{\{x_1,x_2,x_3\}}$:

0 0 0 -
---
0 1 0 -
1 0 0 -
---
1 1 0 -

$L_1^{\{x_2,x_3,x_4\}}$:

- 0 0 0
---
- 0 0 1
- 1 0 0
---
- 1 0 1

$L_1^{\{x_1,x_3,x_4\}}$:

0 - 0 0
---
0 - 0 1
1 - 0 0
---
0 - 1 1
1 - 0 1
1 - 1 0

Alle Implikanten aus $L_1M$ sind Kanten von Flächen, die Implikanten sind: $Prim(f) = \emptyset$

# Beispiel Quine-McCluskey: Bestimmung von $L_3$ (1/2)

$L_2^{\{x_1,x_2\}}$:

$L_2^{\{x_1,x_4\}}$:

0 - - 1
1 - - 0

$L_2^{\{x_2,x_4\}}$:

$L_2^{\{x_1,x_3\}}$:

0 - 0 -
1 - 0 -

$L_2^{\{x_2,x_3\}}$:

- 0 0 -
- 1 0 -

$L_2^{\{x_3,x_4\}}$:

- - 0 0
- - 0 1

Die markierten Implikanten-Flächen sind nicht Rand eines 3-dim. Implikanten. Sie sind also prim! $\Rightarrow$ $Prim(f) = \{x_1'x_4, x_1x_4'\}$

# Beispiel Quine-McCluskey: Bestimmung von $L_3$ (2/2)

$L_2^{\{x_1,x_2\}}$:

$L_2^{\{x_1,x_4\}}$:

0 - - 1
1 - - 0

$L_2^{\{x_2,x_4\}}$:

$L_2^{\{x_1,x_3\}}$:

0 - 0 -
1 - 0 -

$L_2^{\{x_2,x_3\}}$:

- 0 0 -
- 1 0 -

$L_2^{\{x_3,x_4\}}$:

- - 0 0
- - 0 1

Die markierten Implikanten-Flächen sind Rand eines 3-dimensionalen Implikanten. Sie sind also nicht prim! $\Rightarrow$ $Prim(f) = \{x_1'x_4, x_1x_4'\}$

# Beispiel Quine-McCluskey: Ende

$L_3^{\{x_1\}}$:

$L_3^{\{x_2\}}$:

$L_3^{\{x_3\}}$:

- - 0 -

$L_3^{\{x_4\}}$:

$$Prim(f) = \{x_1' x_4, x_1 x_4'\}$$

$$\Rightarrow Prim(f) = \{x_1' x_4, x_1 x_4', x_3'\}$$

$$p_{complete}(f) = x_1' x_4 + x_1 x_4' + x_3'$$

# Korrektheit von Quine-McCluskey (1/2)

Prime implicants function **Quine** ($f : \mathbb{B}^n \to \mathbb{B}$)

**begin**
  $L_0 := Minterm(f)$;
  $i := 0$;
  $//L_i$ enthält alle Implikanten von $f$ der Länge $n - i$.
  $Prim(f) := \varnothing$
  **while** ($L_i \neq \varnothing$) **and** ($i < n$)
  **loop** $L_{i+1} := \{m \mid m \cdot x$ und $m \cdot x'$ sind in $L_i$ für ein $x\}$;
    $Prim(f) := Prim(f) \cup$
    $\{m' \mid m' \in L_i$ und $m'$ wird von keinem $q \in L_{i+1}$ überdeckt$\}$;
    $i := i + 1$
  **end loop;**
  **return** $Prim(f) \cup L_i$;
**end;**

# Korrektheit von Quine-McCluskey (2/2)

## Satz

Für alle $i = 0, 1, \ldots, n$ gilt:

- $L_i$ enthält nur Monome mit $n - i$ Literalen.
- $L_i$ enthält genau die Implikanten von $f$ mit $n - i$ Literalen.
- Nach Iteration $i$ enthält $Prim(f)$ genau die Primimplikanten von $f$ mit mindestens $n - i$ Literalen.

**Beweis:**
Induktion über $i$:

- Abbruchbedingung ($L_i = \emptyset$) oder ($i = n$):
- $L_i = \emptyset$ bedeutet, dass keine Implikanten bei der "Partnersuche" entstanden sind, d.h. $L_{i-1}$ ist vollständig in $Prim(f)$ aufgegangen.
- $i = n$ bedeutet, dass $L_n$ berechnet wurde, es gilt dann $L_n = \emptyset$ oder $L_n = \{1\}$, letzteres bedeutet $f$ ist die Eins-Funktion und $Prim(f) = \{1\}$.

# Kosten des Verfahrens

## Lemma

Es gibt $3^n$ verschiedene Monome in $n$ Variablen.

**Beweis:**
Für jedes Monom $m$ und jede der $n$ Variablen $x$ liegt genau eine der drei folgenden Situationen vor:

- $m$ enthält weder das positive noch das negative Literal von $x$.
- $m$ enthält das positive Literal $x$.
- $m$ enthält das negative Literal $x'$.

Jedes Monom ist durch diese Beschreibung auch eindeutig bestimmt.

# Komplexität des Verfahrens von Quine-McCluskey

## Satz

Die Laufzeit des Verfahrens liegt in $O(n^2 \cdot 3^n)$ beziehungsweise in $O(\log^2(N) \cdot N^{\log(3)})$, wobei $N = 2^n$ die Größe der Funktionstabelle ist.

**Beweisidee:**
Jedes der $3^n$ Monome wird im Verlauf des Verfahrens mit höchstens $n$ anderen Monomen verglichen.

- Gegeben sei ein Monom $mx$. Die Erzeugung von $mx'$ und die Suche nach $mx'$ in $L_i$ ist bei Verwendung geeigneter Datenstrukturen in $O(n)$ durchführbar.

$O(n^2 \cdot 3^n) = O(\log^2(N) \cdot N^{\log(3)})$ durch Nachrechnen:

$$3^n = (2^{\log(3)})^n = (2^n)^{\log(3)} = N^{\log(3)}$$

# Das Matrix-Überdeckungsproblem

- Wir haben nun durch das Verfahren von Quine-McCluskey alle Primimplikanten von $f$ bestimmt.
- Die Disjunktion aller Primimplikanten ist ein Polynom, das $f$ implementiert. Es ist aber im Allgemeinen kein Minimalpolynom von $f$.
- Für das Minimalpolynom benötigen wir eine kostenminimale Teilmenge $M$ von $Prim(f)$, so dass die Monome von $M$ $f$ überdecken.
- Diese Art von Problemen wird Matrix-Überdeckungsproblem genannt.

# SMILE - Das Matrix-Überdeckungsproblem: Einfaches Beispiel

- Für eine Expedition wird ein Fahrer, ein Messtechniker und ein Kameramann benötigt. Es stehen fünf Kandidaten mit unterschiedlichen Fähigkeiten und Gehaltsvorstellungen zur Auswahl. Welches ist das kostengünstigste Team?

| Kandidat | Fahrer? | Messtechniker? | Kameramann? | Gehalt |
|---|---|---|---|---|
| Alice | Ja | Nein | Ja | 4000 |
| Dilbert | Ja | Ja | Nein | 2000 |
| Dogbert | Ja | Ja | Ja | 5000 |
| Ted | Nein | Nein | Ja | 1000 |
| Wally | Nein | Ja | Ja | 1500 |

# Primimplikantentafel

- Definiere eine boolesche Matrix $PIT(f)$, die Primimplikantentafel von $f$:
  - Die Zeilen entsprechen eindeutig den Primimplikanten von $f$.
  - Die Spalten entsprechen eindeutig den Mintermen von $f$.
  - Sei $min(\alpha)$ ein beliebiger Minterm von $f$.
    Dann gilt für Primimplikant $m$ : $PIT(f)[m, min(\alpha)] = 1 \Leftrightarrow m(\alpha) = 1$.
- Der Eintrag an der Stelle $[m, min(\alpha)]$ ist also genau dann $1$, wenn $min(\alpha)$ eine Ecke des Würfels $m$ beschreibt.

**Gesucht:**
Eine kostenminimale Teilmenge $M$ von $Prim(f)$, so dass jede Spalte von $PIT(f)$ überdeckt ist,
d.h. $\forall \alpha \in ON(f) \quad \exists m \in M$ mit $PIT(f)[m, min(\alpha)] = 1$.

# Primimplikantentafel: Beispiel (1/2)

$$Prim(f) = \{x_1' x_4, x_1 x_4', x_3'\}$$

**Primimplikantentafel** $PIT(f)$:

| | 0 | 1 | 3 | 4 | 5 | 7 | 8 | 9 | 10 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1' x_4$ | | 1 | 1 | | 1 | 1 | | | | | | |
| $x_1 x_4'$ | | | | | | | 1 | | 1 | 1 | | 1 |
| $x_3'$ | 1 | 1 | | 1 | 1 | | 1 | 1 | | 1 | 1 | |

# Primimplikantentafel: Beispiel (2/2)

**Gesucht:**
Eine kostenminimale Teilmenge $M$ von $Prim(f)$, so dass jede Spalte von $PIT(f)$ überdeckt ist,
d.h. $\forall \alpha \in ON(f) \quad \exists m \in M$ mit $PIT(f)[m, min(\alpha)] = 1$.

$Prim(f) = \{x_1' x_4, x_1 x_4', x_3'\}$

**Primimplikantentafel** $PIT(f)$:

| | 0 | 1 | 3 | 4 | 5 | 7 | 8 | 9 | 10 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1' x_4$ | | 1 | 1 | | 1 | 1 | | | | | | |
| $x_1 x_4'$ | | | | | | | 1 | | 1 | 1 | | 1 |
| $x_3'$ | 1 | 1 | | 1 | 1 | | 1 | 1 | | 1 | 1 | |

# Erste Reduktionsregel - Wesentlicher Implikant

## Definition

Ein Primimplikant $m$ von $f$ heißt wesentlich, wenn es einen Minterm $min(\alpha)$ von $f$ gibt, der nur von diesem Primimplikanten überdeckt wird, also:

- $PIT(f)[m, min(\alpha)] = 1$
- $PIT(f)[m', min(\alpha)] = 0$

für jeden anderen Primimplikanten $m'$ von $f$.

## Lemma

Jedes Minimalpolynom von $f$ enthält alle wesentlichen Primimplikanten von $f$.

**1. Reduktionsregel:** Entferne aus der Primimplikantentafel $PIT(f)$ alle wesentlichen Primimplikanten und alle Minterme, die von diesen überdeckt werden.

# Erste Reduktionsregel: Beispiel (1/2)

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | 1 | | | | | | | | | | | | |
| 2 | | 1 | | | | 1 | | | | | | | | | | | |
| 3 | | | 1 | | | | 1 | | | | | | | | | | |
| 4 | | | | 1 | | | | 1 | | | | | | | | | |
| 5 | | | | | 1 | | | | 1 | | | | | | | | 1 |
| 6 | | | | | | 1 | | | | 1 | | | | | | | 1 |
| 7 | | | | | | | 1 | | | | 1 | | | | | | |
| 8 | | | | | | | | 1 | | | | 1 | | | | | |
| 9 | | | | | | | | | 1 | | | | 1 | | | | |
| 10 | | | | | | | | | | 1 | | | | 1 | | | 1 |
| 11 | | | | | | | | | | | 1 | | | | 1 | | |
| 12 | | | | | | | | | | | | 1 | | | | 1 | |
| 13 | | | | | | | | | | | | | 1 | 1 | 1 | 1 | |

# Erste Reduktionsregel: Beispiel (2/2)

wesentlich

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | 1 | | | | | | | | | | | | |
| 2 | | 1 | | | | 1 | | | | | | | | | | | |
| 3 | | | 1 | | | | 1 | | | | | | | | | | |
| 4 | | | | 1 | | | | 1 | | | | | | | | | |
| 5 | | | | | 1 | | | | 1 | | | | | | | | 1 |
| 6 | | | | | | 1 | | | | 1 | | | | | | | 1 |
| 7 | | | | | | | 1 | | | | 1 | | | | | | |
| 8 | | | | | | | | 1 | | | | 1 | | | | | |
| 9 | | | | | | | | | 1 | | | | 1 | | | | |
| 10 | | | | | | | | | | 1 | | | | 1 | | | 1 |
| 11 | | | | | | | | | | | 1 | | | | 1 | | |
| 12 | | | | | | | | | | | | 1 | | | | 1 | |
| 13 | | | | | | | | | | | | | 1 | 1 | 1 | 1 | |

# Nach Anwendung der 1. Reduktionsregel

| | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | | | | | | | | 1 |
| 6 | | 1 | | | | | | | 1 |
| 7 | | | 1 | | | | | | |
| 8 | | | | 1 | | | | | |
| 9 | 1 | | | | 1 | | | | |
| 10 | | 1 | | | | 1 | | | 1 |
| 11 | | | 1 | | | | 1 | | |
| 12 | | | | 1 | | | | 1 | |
| 13 | | | | | 1 | 1 | 1 | 1 | |

Die Matrix enthält keine wesentlichen Zeilen mehr!

# Zweite Reduktionsregel - Spaltendominanz

## Definition

Sei $A$ eine boolesche Matrix. Spalte $j$ von $A$ dominiert Spalte $i$ von $A$, wenn für jede Zeile $k$ gilt: $A[k,i] \leq A[k,j]$.

- Nutzen für unser Problem: Dominiert ein Minterm $w'$ von $f$ einen anderen Minterm $w$ von $f$, so braucht man $w'$ nicht weiter zu betrachten, da $w'$ auf jeden Fall überdeckt werden muss und hierdurch auch Minterm $w'$ überdeckt wird.

- Jeder in $PIT(f)$ vorhandene Primimplikant $p$, der $w$ überdeckt, überdeckt auch $w'$.

**2. Reduktionsregel:** Entferne aus der Primimplikantentafel $PIT(f)$ alle Minterme, die einen anderen Minterm in $PIT(f)$ dominieren.

# Zweite Reduktionsregel: Beispiel

| | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | | | | | | | | 1 |
| 6 | | 1 | | | | | | | 1 |
| 7 | | | 1 | | | | | | |
| 8 | | | | 1 | | | | | |
| 9 | 1 | | | | 1 | | | | |
| 10 | | 1 | | | | 1 | | | 1 |
| 11 | | | 1 | | | | 1 | | |
| 12 | | | | 1 | | | | 1 | |
| 13 | | | | | 1 | 1 | 1 | 1 | |

Spalte 17 dominiert Spalte 10
$\Rightarrow$ Spalte 17 kann gelöscht werden!

# Dritte Reduktionsregel - Zeilendominanz

## Definition

Sei $A$ eine boolesche Matrix. Zeile $i$ von $A$ dominiert Zeile $j$ von $A$, wenn für jede Spalte $k$ gilt: $A[i,k] \geq A[j,k]$.

- Nutzen für unser Problem: Dominiert ein Primimplikant $m$ einen Primimplikanten $m'$, so braucht man $m'$ nicht weiter zu betrachten, wenn $cost(m') \geq cost(m)$ gilt.
- Der Primimplikant $m$ überdeckt jeden noch nicht überdeckten Minterm von $f$, der von $m'$ überdeckt wird, obwohl er nicht teurer ist.

**3. Reduktionsregel:** Entferne aus der Primimplikantentafel $PIT(f)$ alle Primimplikanten, die durch einen anderen, nicht teureren Primimplikanten dominiert werden.

# Dritte Reduktionsregel: Beispiel

Nehme an, dass die Zeilen 5 bis 12 gleiche Kosten haben.

werden dominiert

| | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| ~~5~~ | ~~1~~ | | | | | | | |
| ~~6~~ | | ~~1~~ | | | | | | |
| ~~7~~ | | | ~~1~~ | | | | | |
| ~~8~~ | | | | ~~1~~ | | | | |
| 9 | 1 | | | | 1 | | | |
| 10 | | 1 | | | | 1 | | |
| 11 | | | 1 | | | | 1 | |
| 12 | | | | 1 | | | | 1 |
| 13 | | | | | 1 | 1 | 1 | 1 |

# Nach Anwendung der 3. Reduktionsregel

| | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| 9 | $\boxed{1}$ | | | | 1 | | | |
| 10 | | $\boxed{1}$ | | | | 1 | | |
| 11 | | | $\boxed{1}$ | | | | 1 | |
| 12 | | | | $\boxed{1}$ | | | | 1 |
| 13 | | | | | 1 | 1 | 1 | 1 |

- Offensichtlich kann nun wieder die erste Reduktionsregel angewendet werden, da die Zeilen 9, 10, 11, 12 wesentlich sind.
  - Die resultierende Matrix ist leer.
  - Das gefundene Minimalpolynom ist:
    1 + 2 + 3 + 4 + 9 + 10 + 11 + 12

# Zyklische Überdeckungsprobleme

## Definition

Eine Primimplikantentafel heißt reduziert, wenn keine der drei Reduktionsregeln anwendbar ist.

- Ist eine reduzierte Tafel nicht-leer, spricht man von einem zyklischen Überdeckungsproblem.
- In der Praxis werden solche Probleme heuristisch gelöst. Es gibt auch exakte Methoden (Petrick, Branch-and-Bound).

**Primimplikantentafel** $PIT(f)$:

| | 3 | 5 | 7 | 9 | 11 | 13 |
|---|---|---|---|---|---|---|
| $\{7,5\}$ | | 1 | 1 | | | |
| $\{5,13\}$ | | 1 | | | | 1 |
| $\{13,9\}$ | | | | 1 | | 1 |
| $\{9,11\}$ | | | | 1 | 1 | |
| $\{11,3\}$ | 1 | | | | 1 | |
| $\{3,7\}$ | 1 | | 1 | | | |

# Petrick's Methode

**Verfahren:**

- Übersetze die PIT in ein Produkt von Summen, d.h. in ein (OR, AND)-Polynom, das alle Möglichkeiten der Überdeckung enthält.
- Multipliziere das (OR, AND)-Polynom aus, so dass ein (AND-OR)-Polynom entsteht.
- Die gesuchte minimale Überdeckung ist gegeben durch das Monom, das einer PI-Auswahl mit minimalen Kosten entspricht.

| | 3 | 5 | 7 | 9 | 11 | 13 |
|---|---|---|---|---|---|---|
| $a:\{7,5\}$ | | 1 | 1 | | | |
| $b:\{5,13\}$ | | 1 | | | | 1 |
| $c:\{13,9\}$ | | | | 1 | | 1 |
| $d:\{9,11\}$ | | | | 1 | 1 | |
| $e:\{11,3\}$ | 1 | | | | 1 | |
| $f:\{3,7\}$ | 1 | | 1 | | | |

wird übersetzt in

$$(e+f)\cdot(a+b)\cdot(a+f)\cdot(c+d)\cdot(d+e)\cdot(b+c)$$

$$= (ea+eb+fa+fb)\cdot(ac+ad+fc+fd)$$
$$\cdot(db+dc+eb+ec)$$

$$\vdots$$

$$= ace + acde + abcde + abcd + \cdots + bdf$$

Bei gleichen Kosten für alle PIs sind *ace* und *bdf* minimal.

# „Greedy-Heuristik” zur Lösung von Überdeckungsproblemen

1. Wende alle möglichen Reduktionsregeln an.
2. Ist die Matrix $A$ leer, ist man fertig.
3. Sonst wähle die Zeile $i$, die die meisten Spalten überdeckt. Lösche diese Zeile und alle von ihr überdeckten Spalten und gehe zu 1.

- Dieser Algorithmus liefert nicht immer die optimale Lösung!
  - Hinweis: Bei der Ausgangs-Matrix aus unserem Beispiel überdeckt Zeile 13 die meisten Spalten.
    Diese ist nicht Teil der gefundenen Lösung!

# Zusammenfassung Schaltkreise

- Schaltkreise stellen boolesche Funktionen dar.
- Optimale boolesche Polynome können sehr viel größer sein, als entsprechende Schaltkreise.
  - exponentielle Unterschiede möglich
  - Rechtfertigung für Einsatz von Schaltkreisen statt PLAs
- Es gibt auch Algorithmen zur Berechnung optimaler (mehrstufiger) Schaltkreise.
  - anspruchsvoller als Optimierung von booleschen Polynomen
  - meist heuristisch (Näherungsverfahren)
  - nicht Gegenstand dieser Vorlesung
- Hier: Schaltkreise für spezielle Funktionen, insbesondere Arithmetik.

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
4. **Arithmetische Schaltungen**
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Arithmetische Schaltungen

- Addieren nach der Schulmethode: Carry-Ripple-Addierer.
- Effizienteres Addieren: Conditional-Sum-Addierer.
- Addition von Zweierkomplement-Zahlen.
- Subtrahierer.
- Exkurs: Multiplizierer.

# Kosten von Schaltkreisen

- Um unterschiedliche Schaltkreise, die eine Funktion (z.B. Addierer) implementieren, miteinander zu vergleichen, benötigt man ein Kostenmaß.

**Definition**

Die Kosten $C(SK)$ eines Schaltkreises $SK$ sind durch die Anzahl seiner Gatter gegeben.

- Deutet auf die Fläche und den Energieverbrauch der resultierenden Hardware-Blöcke hin.

**Definition**

Die Tiefe $depth(SK)$ eines Schaltkreises ist die maximale Anzahl von Gattern auf einem Pfad von einem beliebigen Eingang zu einem beliebigen Ausgang von $SK$.

- Deutet auf die Signallaufzeit durch $SK$ und somit die maximal mögliche Taktfrequenz (Geschwindigkeit) des Schaltkreises hin.

# Beispiel: Kosten und Tiefe

$$C(SK) = 8$$

$$Depth(SK) = 3$$

# Teilschaltkreise, hierarchischer Entwurf (informell)

- Illustration eines Teilschaltkreises.

# Hierarchische Schaltkreise

- In hierarchischen Schaltkreisen sind Teilschaltkreise durch Symbole ersetzt.
- Den zugehörigen („flachen") Schaltkreis erhält man, indem man die Symbole durch Einsetzen der Teilschaltkreise wieder entfernt.

# Wiederholung Zahlendarstellung

Sei $a = a_{n-1} \dots a_0$ eine Folge von Ziffern, $a_i \in \{0, 1\}$.

- Binärdarstellung: $\langle a \rangle = \sum_{i=0}^{n-1} a_i 2^i$
- Zweierkomplement: $[a_n a_{n-1} \dots a_0] = \sum_{i=0}^{n-1} a_i 2^i - a_n 2^n$
- Rechenregel: $-[a] = [\overline{a}] + 1$

mit $\overline{a} = \overline{a}_n \overline{a}_{n-1} \dots \overline{a}_0$.

# Addierer für nichtnegative Zahlen

- **Gegeben:**
  2 positive Binärzahlen $\langle a \rangle = \langle a_{n-1} \ldots a_0 \rangle, \langle b \rangle = \langle b_{n-1} \ldots b_0 \rangle$ mit Eingangsübertrag $c \in \{0, 1\}$.

- **Gesucht:**
  Schaltkreis, der Binärdarstellung $s$ von $\langle a \rangle + \langle b \rangle + c$ berechnet.

- Wegen $\langle a \rangle + \langle b \rangle + c \leq 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1$ genügen $n + 1$ Stellen für die Darstellung von $s$, d.h. der Schaltkreis hat $n + 1$ Ausgänge.

# Formale Definition $n$-Bit-Addierer

- Ein $n$-Bit-Addierer ist ein Schaltkreis, der die folgende boolesche Funktion berechnet:

$$+_n : \mathbb{B}^{2n+1} \to \mathbb{B}^{n+1},$$
$$+_n : (a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, c) = (s_n, \ldots, s_0)$$
$$\text{mit } \langle s \rangle = \langle s_n \ldots s_0 \rangle = \langle a_{n-1} \ldots a_0 \rangle + \langle b_{n-1} \ldots b_0 \rangle + c$$

# Addieren nach der Schulmethode (1/4)

- Wir werden im Folgenden den einfachsten Addierertypen einführen, der die „Schulmethode” umsetzt.
- Hierzu werden einige Grundschaltungen (Halb- und Volladierer) notwendig sein.
- Beispiel für die Schulmethode:

$$
\begin{array}{rr}
 & a \quad 1\,0\,1\,1 \\
+ & b \quad 0\,1\,1\,0 \\
 & \phantom{0}\,1\,1\,1\,0\,\phantom{0} \\
\hline
 & 1\,0\,0\,0\,1
\end{array}
$$

# Addieren nach der Schulmethode (2/4)

$$
\begin{array}{cccccc}
 & & & & & HA \\
 & a & & 1 & 0 & 1 & 1 \leftarrow a_0 \\
+ & b & & 0 & 1 & 1 & 0 \leftarrow b_0 \\
 & & 1 & 1 & 1 & 0 & \leftarrow ha_1 \\
\hline
 & & 1 & 0 & 0 & 0 & 1 \leftarrow ha_0
\end{array}
$$

| $a_0$ | $b_0$ | $ha_1$ | $ha_0$ |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

# Halbaddierer (Half Adder, *HA*)

- Ein Halbaddierer dient zur Addition zweier 1-Bit-Zahlen ohne Eingangsübertrag.
- Er berechnet die Funktion $ha : \mathbb{B}^2 \to \mathbb{B}^2$ mit $ha(a_0, b_0) = (ha_1, ha_0)$, wobei $2ha_1 + ha_0 = a_0 + b_0$.

| $a_0$ | $b_0$ | $ha_1$ | $ha_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$ha_0(a_0, b_0) = a_0 \oplus b_0$$
$$ha_1(a_0, b_0) = a_0 \wedge b_0$$

# Schaltkreis eines Halbaddierers

dabei gilt:

$C(HA) = 2, \quad depth(HA) = 1$

# Addieren nach der Schulmethode (3/4)

# Volladierer (Full Adder, *FA*)

- Ein Volladdierer dient zur Addition zweier 1-Bit-Zahlen mit Eingangsübertrag.
- Er berechnet die Funktion $fa : \mathbb{B}^3 \to \mathbb{B}^2$ mit $fa(a_0, b_0, c) = (fa_1, fa_0)$ wobei $2fa_1 + fa_0 = a_0 + b_0 + c$

| $a_0$ | $b_0$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Volladdierer als Funktion von *HA*s

Aus der Tabelle folgt:

$fa_0 = a_0 \oplus b_0 \oplus c$
$= ha_0(c, ha_0(a_0, b_0))$

$fa_1 = (a_0 \wedge b_0) \vee (c \wedge (a_0 \oplus b_0))$
$= ha_1(a_0, b_0) + ha_1(c, ha_0(a_0, b_0))$

Kosten und Tiefe eines *FA*:
$C(FA) = 5, \quad depth(FA) = 3$

| $a_0$ | $b_0$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Schaltkreis eines Volladdierers

# Addieren nach der Schulmethode (4/4)

# Aufbau eines Carry-Ripple-Addierers

# Induktive Definition des Carry-Ripple-Addierers *CR*

- Für $n = 1$ : $CR_1 = FA$
- Für $n > 1$ : Folgender Schaltkreis:

$c_{-1}$: Eingangsübertrag

$c_i$: Übertrag von Stelle $i$ nach $i+1$

# Zwei (identische) Darstellungen von *CR*

# Carry-Ripple-Addierer

## Satz

$CR_n$ ist ein $n$-Bit-Addierer.

**Beweis** (duch Induktion):

- $n = 1$ ($CR_1 = FA$) ✓
- $n-1 \to n$: Eingabe an $CR_n : (a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, c-1)$
  Zeige für Ausgabe $(s_n, \ldots, s_0)$ von $CR_n$:
  $\langle s \rangle = \langle s_n \ldots s_0 \rangle = \langle a_{n-1} \ldots a_0 \rangle + \langle b_{n-1} \ldots b_0 \rangle + c_{-1}$.
- Nach Induktionsvoraussetzung gilt für $CR_{n-1}$:
  $\langle s_n \ldots s_1 \rangle = \langle a_{n-1} \ldots a_1 \rangle + \langle b_{n-1} \ldots b_1 \rangle + c_0$. $(a)$
  Wegen FA-Eigenschaft gilt $\langle c_0, s_0 \rangle = a_0 + b_0 + c_{-1}$. $(b)$
- Insgesamt: $\langle s_n \ldots s_0 \rangle = 2 \cdot \langle s_n \ldots s_1 \rangle + s_0$
  $\overset{(a)}{=} 2 \cdot (\langle a_{n-1} \ldots a_1 \rangle + \langle b_{n-1} \ldots b_1 \rangle + c_0) + s_0$
  $= 2 \cdot (\langle a_{n-1} \ldots a_1 \rangle + \langle b_{n-1} \ldots b_1 \rangle) + 2 \cdot c_0 + s_0$
  $\overset{(b)}{=} 2 \cdot \langle a_{n-1} \ldots a_1 \rangle + a_0 + 2 \cdot \langle b_{n-1} \ldots b_1 \rangle + b_0 + c_{-1}$
  $= \langle a \rangle + \langle b \rangle + c_{-1}$

# Schaltbild und Komplexität von *CR*

- $C(CR_n) = n \cdot C(FA) = 5n$.
- $depth(CR_n) = ?$.

# Schaltbild und Komplexität von *CR*

- $C(CR_n) = n \cdot C(FA) = 5n$.
- $depth(CR_n) = 3 + 2(n-1)$.
- Sowohl die Kosten als auch die Tiefe von *CR* sind somit linear in $n$.

- Es gibt (asymptotisch) bessere Addierer. Wir werden hier den Conditional-Sum-Addierer kennen lernen, für den wir wieder eine Hilfsschaltung (Multiplexer) benötigen.
- Eine weitere wichtige Schaltung ist der Inkrementer.

# $n$-Bit-Inkrementer

## Definition

Ein $n$-Bit-Inkrementer $INC_n$ berechnet die Funktion
$inc_n : \mathbb{B}^{n+1} \to \mathbb{B}^{n+1}$
$inc_n(a_{n-1}, \ldots, a_0, c) = (s_n, \ldots, s_0)$ mit $\langle s_n \ldots s_0 \rangle = \langle a \rangle + c$

- Ein Inkrementer ist ein Addierer mit $b_i = 0$ für alle $i$.
  $\Rightarrow$ Ersetze in $CR_n$ die $FA$ durch $HA$.
- Kosten und Tiefe:
  - $C(INC_n) = n \cdot C(HA) = 2n$
  - $depth(INC_n) = n \cdot depth(HA) = n$

# $n$-Bit-Multiplexer

## Definition

Ein $n$-Bit-Multiplexer $MUX_n$ berechnet die Funktion

$$sel_n : \mathbb{B}^{2n+1} \to \mathbb{B}^n$$

$$sel_n(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0, s) = \begin{cases} (a_{n-1} \ldots a_0), & \text{falls } s = 1 \\ (b_{n-1} \ldots b_0), & \text{falls } s = 0 \end{cases}$$

- Es gilt: $(sel_n)_i = s \cdot a_i + \overline{s} \cdot b_i$

# Aufbau von $MUX_n$

# Schaltbild und Kosten $MUX_n$

Kosten und Tiefe:

$C(MUX_n) = 3n + 1.$
$depth(MUX_n) = 3.$

# Rückkehr zum Addierer

Gibt es billigere Addierer als $CR_n$?

**Untere Schranken:**
$C(+_n) \geq 2 \cdot n, \quad depth(+_n) \geq \log(n) + 1$

Sei $f \in \mathbb{B}_n$. Dann sind $C(f)$ und $depth(f)$ definiert durch
$C(f) = min\{C(SK) | f_{SK} = f\}$ und
$depth(f) = min\{depth(SK) | f_{SK} = f\}$.

Binäre Bäume mit $2n + 1$ Blättern haben $2n$ innere Knoten.
Binäre Bäume mit $n$ Blättern haben mindestens Tiefe $\lceil \log(n) \rceil$.

Im Folgenden sei $n = 2^k$.

# Conditional-Sum-Addierer ($CSA$)

- Idee: Nutze Parallelverarbeitung, um Tiefe zu reduzieren!

$$
\begin{array}{lrr}
 & a_{n-1} \dots a_{\frac{n}{2}} & a_{\frac{n}{2}-1} \dots a_0 \\
+ & b_{n-1} \dots b_{\frac{n}{2}} & b_{\frac{n}{2}-1} \dots b_0 \\
+ & c_{\frac{n}{2}-1} & c_{-1} \\
\hline
 & s_n s_{n-1} \dots s_{\frac{n}{2}} & s_{\frac{n}{2}-1} \dots s_0
\end{array}
$$

- $CSA_1 = FA$.
- $CSA_n$: Siehe nächste Folie.
- Im Folgenden sei $n = 2^k$.

# Aufbau von $CSA_n$

# Komplexität von $CSA_n$: Tiefe

## Satz

$CSA_n$ hat Tiefe $\leq 3\log(n) + 3$.

**Beweis:** Setze $n = 2^k$ voraus.

- $n = 1$: $depth(CSA_1) = depth(FA) = 3$.

- $n > 1$: $depth(CSA_n) \leq depth(CSA_{\frac{n}{2}}) + depth(MUX_{\frac{n}{2}+1})$
  $\leq depth(CSA_{\frac{n}{2}}) + 3$
  $\leq depth(CSA_{\frac{n}{4}}) + 3 + 3$
  $\leq depth(CSA_{\frac{n}{8}}) + 3 + 3 + 3$
  $\ldots$
  $\leq depth(CSA_{\frac{n}{2^k}}) + k \cdot 3$
  $= depth(CSA_1) + k \cdot 3$
  $\leq 3 \cdot (k+1) = 3\log(n) + 3.$

# Komplexität von $CSA_n$: Kosten

**Satz (ohne Beweis)**

$C(CSA_n) = 10n^{\log(3)} - 3n - 2.$

- Man kann den hier vorgestellten *CSA* in einfacher Weise modifizieren, so dass
  - Tiefe = $O(\log(n))$,
  - Kosten = $O(n \cdot \log(n))$.
- Es gibt auch Addierer mit linearen Kosten und logarithmischer Tiefe.
  - Carry-Lookahead-Addierer (*CLA*).
  - $C(CLA_n) \leq 11n$,
  - $depth(CLA_n) \leq 4 \cdot \log(n) + 2$.

# Addition von Zweierkomplementzahlen

- Auszurechnen ist:

  $$[a_n a_{n-1} \ldots a_0] + [b_n b_{n-1} \ldots b_0] = (-a_n 2^n) + (-b_n 2^n) + \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i$$

- Im Fall von $(n+1)$-Bit-Zweierkomplementzahlen können Ergebnisse im Bereich $R_n = \{-2^n, \ldots, 2^n - 1\}$ dargestellt werden; andernfalls kommt es zu einem Überlauf.

- Der Satz auf der nächsten Folie sagt aus:
  - Kommt es bei der Addition nicht zu einem Überlauf, so kann man den „gewöhnlichen" Binäraddierer zur Addition von Zweierkomplementzahlen benutzen.
  - Ob es zu einem Überlauf kommt, lässt sich anhand von Werten $a_n$, $b_n$ und $s_n$ im Binäraddierer entscheiden.

# Zweierkomplement-Addition formal

## Satz

Seien $a, b \in \mathbb{B}^{n+1}$, $c_{-1} \in \{0,1\}$ und $s \in \{0,1\}^{n+1}$,
so dass $\langle c_n, s\rangle = \langle a\rangle + \langle b\rangle + c_{-1}$.

Dann gilt:

- $[a] + [b] + c_{-1} \notin R_n \Leftrightarrow (a_n = b_n) \wedge (b_n \neq s_n)$
- $[a] + [b] + c_{-1} \in R_n \Rightarrow [a] + [b] + c_{-1} = [s]$

- Beweis durch Fallunterscheidung ($[a], [b]$ beide positiv, beide negativ, $[a]$ positiv / $[b]$ negativ) und Nachrechnen (Induktion).

- Man kann einen alternativen Überlauftest zeigen:

$$[a] + [b] + c_{-1} \notin R_n \Leftrightarrow c_n \neq c_{n-1}$$

# Addierer für $(n+1)$-Bit-Zweierkomplement-Zahlen

# Subtraktion

- Wegen $-[b] = [\overline{b}] + 1$ kann $[a] - [b]$ zurückgeführt werden auf $[a] + [\overline{b}] + 1$.
- **Beispiel:**

$$[a] = [0110] = 6_{10}, \quad [b] = [0111] = 7_{10}, \quad [\overline{b}] = [1000]$$

$$\begin{array}{rr} & 0\,1\,1\,0 \\ + & 1\,0\,0\,0 \\ + & 1 \\ \hline & 1\,1\,1\,1 \end{array} \qquad 1111 = (-1)_{10}$$

- Den Schaltkreis für Subtraktion gewinnt man aus einem Addierer.
- Kombinierter Addierer/Subtrahierer.

# Subtrahierer

$b_i \oplus 0 = b_i$ $\quad$ $sub = 0 : [a] + [b] + 0$

$b_i \oplus 1 = \overline{b_i}$ $\quad$ $sub = 1 : [a] + [\overline{b}] + 1 = [a] - [b]$

# Exkurs – Multiplizierer

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Multiplizierer

- **Gesucht**: Schaltkreis zur Multiplikation zweier Binärzahlen $< a_{n-1}, \ldots, a_0 >, < b_{n-1}, \ldots, b_0 >$.

- **Beispiel**:

$$\underbrace{(110)}_{6_{10}} \cdot \underbrace{(101)}_{5_{10}}$$

$$\begin{array}{cccccc}
 & & & 1 & 1 & 0 \\
+ & & 0 & 0 & 0 & \\
+ & 1 & 1 & 0 & & \\
\hline
 & 1 & 1 & 1 & 1 & 0
\end{array}$$

$$= 30_{10}$$

# Allgemeines zum Multiplizierer

- Wieviele Stellen werden für das Ergebnis benötigt?

$$
\begin{aligned}
< a > \cdot < b > \quad &\leq \quad (2^n - 1) \cdot (2^n - 1) \\
= \quad 2^{2n} - 2^{n+1} + 1 \quad &\leq \quad 2^{2n} - 1
\end{aligned}
$$

- **Also**:
  $2n$ Stellen zur Multiplikation von Binärzahlen.

# Vorgehen bei der Multiplikation

- Multipliziere die Beträge der Zahlen.
- Bestimme das Vorzeichen des Produkts.
- Setze das Endergebnis zusammen.

# n-Bit-Multiplizierer

## Definition

Ein n-Bit-Multiplizierer ist ein Schaltkreis, der die folgende Funtkion berechnet:

$mul_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$
$mul_n(a_{n-1}, \ldots, a_0, b_{n-1}, \ldots, b_0) = (p_{2n-1}, \ldots, p_0)$ mit
$< p_{2n-1}, \ldots, p_0 >=< a > \cdot < b >$

$$< a > \cdot < b >=< a > \cdot \sum_{i=0}^{n-1} b_i \cdot 2^i = \sum_{i=0}^{n-1} (< a > \cdot b_i \cdot 2^i)$$

# Die Multiplikationsmatrix

$$\begin{pmatrix} pp_0 \\ pp_1 \\ \vdots \\ pp_{n-1} \end{pmatrix} =$$

$$\begin{pmatrix} 0 & 0 & \dots & 0 & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \dots & a_1b_0 & a_0b_0 \\ 0 & 0 & \dots & 0 & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \dots & a_0b_1 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & a_{n-1}b_{n-1} & \dots & a_2b_{n-1} & a_1b_{n-1} & a_0b_{n-1} & 0 & \dots & 0 & 0 \end{pmatrix}$$

- Realisierung der Multiplikationsmatrix mit $n^2$ AND-Gattern (und $n^2$ Konstanten 0).

# Daraus entstehende Aufgabe:

- Schnelle Addition von $n$ Partialprodukten der Länge $2n$.
- Mit Carry-Lookahead-Addierern (CLAs) lösbar mit Kosten $O(n^2)$, Tiefe $O(n\ log(n))$ bei *linearem Aufsummieren* der Partialprodukte $((((pp_0 + pp_1) + pp_2) + \ldots) + pp_{n-1})$,

  Tiefe $O(log^2(n))$ bei *baumartigem Zusammenfassen* der Partialprodukte.

# Verbesserung

- Verwende Carry-Save-Addierer.

- Reduktion von 3 Eingabeworten $u$, $v$, $w$ zu zwei Ausgabeworten $s$, $c$ mit
  $< u > + < v > + < w > = < s > + < c >$.

$$
\begin{array}{cccccccc}
 & u_{n-1} & u_{n-2} & \dots & u_2 & u_1 & u_0 \\
 & v_{n-1} & v_{n-2} & \dots & v_2 & v_1 & v_0 \\
 & w_{n-1} & w_{n-2} & \dots & w_2 & w_1 & w_0 \\
\hline
c_{n-1} & c_{n-2} & c_{n-3} & \dots & c_1 & c_0 & 0 \\
0 & s_{n-1} & s_{n-2} & \dots & s_2 & s_1 & s_0
\end{array}
$$

- Gelöst durch Nebeneinandersetzen von Volladdierern (keine Carry-Chain!)

# Carry-Save-Addierer (CSavA)

# Bemerkung zum Aufbau des CSavA

- Speziell bei Partialprodukten:
  Reduziere 3 2n-Bit-Zahlen zu 2 2n-Bit-Zahlen.

$$
\begin{array}{cccc}
pp_{0,2n-1} & \cdots & pp_{0,1} & pp_{0,0} \\
pp_{1,2n-1} & \cdots & pp_{1,1} & pp_{1,0} \\
pp_{2,2n-1} & \cdots & pp_{2,1} & pp_{2,0} \\
\hline
c_{2n-2} & \cdots & c_0 & 0 \\
s_{2n-1} & \cdots & s_1 & s_0
\end{array}
$$

- ($c_{2n-1} = 0$: Carry-Ausgang des letzten FA nicht verwendet.)

# 1. Serielle Lösung

- Hintereinanderschalten von n-2 CSavA-Addierern der Länge 2n.
  - Fasse $n$ Partialprodukte zu 2 2n-Bit-Worten zusammen.
- Addiere die 2n-Bit-Worte mit CLA.
- *siehe Abb. einer Addierstufe*
- Kosten $O(n^2)$, Tiefe $O(n)$

# Addierstufe im Multiplizierer

# 2. Baumartige Lösung

- Neue Grundzelle zur Reduktion von 4 2n-Bit-Eingabeworten zu zwei Ausgabeworten, bestehend aus 2 CSavA-Addierern (*siehe Abb. zur Reduktionszelle*).
- Baumartiges Zusammenfassen der Partialprodukte mit 4-zu-2-Bausteinen zu 2 2n-Bit Worten.
- Addiere die 2n-Bit-Worte mit CLA.
- *siehe Abb. der Addierstufe mit log. Zeit*
- Kosten $O(n^2)$, Tiefe $O(logn)$

# 4-zu-2 Reduktions-Grundzelle

# Addierstufe des log-Zeit-Multiplizierers für 16 Bit

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. **Anwendung: ALU von ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Anwendung: ALU von ReTI

- Die ALU (Arithmetic Logic Unit, arithmetisch-logische Einheit) dient der Berechnung von arithmetischen und logischen Operationen.
- Sie wird von den Compute-Befehlen verwendet und übernimmt weitere Aufgaben, z.B. Berechnung von Speicheradressen oder Erhöhung des *PC*.
- Erinnerung: Der Befehlssatz von ReTI hat die folgenden Compute-Befehle (s. nächste Folie).

# Compute-Befehle: Kodierung

| Typ | MI | F | Befehl | Wirkung | |
|---|---|---|---|---|---|
| 0 0 | 0 | 0 1 0 | SUBI $i$ | $[ACC] := [ACC] - [i]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 0 1 1 | ADDI $i$ | $[ACC] := [ACC] + [i]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 0 | OPLUSI $i$ | $ACC := ACC \oplus 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 1 | ORI $i$ | $ACC := ACC \vee 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 1 0 | ANDI $i$ | $ACC := ACC \wedge 0^8 i$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| 0 0 | 1 | 0 1 0 | SUB $i$ | $[ACC] := [ACC] - [M(\langle i \rangle)]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 0 1 1 | ADD $i$ | $[ACC] := [ACC] + [M(\langle i \rangle)]$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 0 | OPLUS $i$ | $ACC := ACC \oplus M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 0 1 | OR $i$ | $ACC := ACC \vee M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |
| | | 1 1 0 | AND $i$ | $ACC := ACC \wedge M(\langle i \rangle)$ | $\langle PC \rangle := \langle PC \rangle + 1$ |

# Spezifikation der ALU für ReTI

- Eine $n$-Bit-ALU mit:
  - Zwei $n$-Bit-Operanden $a$, $b$, Eingangscarry $c$,
    - ReTI: $n = 32$.
  - Einem $m$-Bit select-Eingang, der ausgewählt, welche Funktion ausgeführt wird,
    - Hier: 8 Funktionen (s. nächste Folie), daher $m = 3$ Bits.
  - Einem $(n+1)$-Bit-Ausgang.
    - $n = 32$.

- Insgesamt 68 Ein- und 33 Ausgänge.

# Select-Eingang bei ReTI-ALU

| Funktionsnummer | | | ALU-Funktion |
|---|---|---|---|
| $s_2$ | $s_1$ | $s_0$ | |
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Mögliche Realisierungen der ALU (1/2)

- **Option 1**: Realisiere Funktionen $f_0, \ldots, f_{2^m-1}$ getrennt durch $SK_{f_i}$ für $f_i$, dann Auswahl durch einen verallgemeinerten Multiplexer.

# Realisierung durch einen verallgemeinerten Multiplexer

# Mögliche Realisierungen der ALU (2/2)

- **Option 1**: Realisiere Funktionen $f_0, \ldots, f_{2^m-1}$ getrennt durch $SK_{f_i}$ für $f_i$, dann Auswahl durch einen Verallgemeinerten Multiplexer.

- **Option 2**: Gemeinsame Behandlung ähnlicher Funktionen.
  - Komplexer, aber effizienter.

# Schaltrealisierung der ALU (1/2)

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Schaltrealisierung der ALU (2/2)

**Option 2:**

a b n n 1 AND OR EXOR n+1 n 0 n 0 0 n n+1 n+1 n+1 n+1 3 2 1 0 2 $s_1 s_0$ $s_1$ $s_0$ $s_0$ $s_1$ $a_{n-1}$ $a_0$ $b_{n-1}$ $b_0$ c ... ... n n $A_n$ 0 n 0 n+1 n+1 1 0 $s_0$ $s_1$ n+1 n+1 1 0 $s_2$ n+1

**Option 1:**

a b c $SKf_0$ $SKf_1$ ... $SKf_{2^m-1}$ n+1 n+1 n+1 verallgemeinerter Multiplexer m select n+1

# Zusammenfassung Kombinatorische Logik

- Kombinatorische Schaltkreise setzen boolesche Funktionen um.
- PLAs sind zweistufig, mehrstufige Schaltungen bestehen aus Gattern und diese aus Transistoren.
- Minimierung von PLAs mit Verfahren von Quine-McCluskey und Lösen des Überdeckungsproblems.
- Statt Minimierung allgemeiner mehrstufiger Schaltkreise wurde eine Klasse (Addierer für Binär- und Zweierkomplementzahlen) betrachtet und ihre Integration in der ALU von ReTI diskutiert.

# Kapitel 4 – Sequentielle Logik

1. **Speichernde Elemente**
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Schaltkreis (1/2)

$typ(v_1) = typ(v_2) = NAND$
$typ(v_3) = NOR$

Schaltkreis mit Eingabebelegung $a = (1, 1, 1, a_4)$.

# Schaltkreis (2/2)

# Schaltpläne (1/3)

- Analyse von Schaltplänen $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$ mit $G$ nicht notwendigerweise azyklisch.

# Schaltpläne (2/3)

- Eine Zellenbibliothek $BIB \subseteq \bigcup_{n \in \mathbb{N}} \mathbb{B}_n$ enthält Basisoperationen, die den Grundgattern entsprechen.
- Ein 5-Tupel $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$ heißt Schaltplan mit $n$ Eingängen und $m$ Ausgängen über der Zellenbibliothek $BIB$ genau dann, wenn
  - $\vec{X}_n = (x_1, \ldots, x_n)$ ist eine endliche Folge von Eingängen.
  - $G = (V, E)$ ist ein gerichteter Graph mit $\{0, 1\} \cup \{x_1, \ldots, x_n\} \subseteq V$.
  - Die Menge $I = V \setminus (\{0, 1\} \cup \{x_1, \ldots, x_n\})$ heißt Menge der Gatter.
    Die Abbildung $typ : I \to BIB$ ordnet jedem Gatter $v \in I$ einen Zellentyp $typ(v) \in BIB$ zu.
  - ...

# Schaltpläne (3/3)

- ...
- Für jedes Gatter $v \in I$ mit $typ(v) \in B_k$ gilt $indeg(v) = k$.
- $indeg(v) = 0$ für $v \in \{0, 1\} \cup \{x_1, \ldots, x_n\}$.
- Die Abbildung $IN : I \to E^*$ legt für jedes Gatter $v \in I$ eine Reihenfolge der eingehenden Kanten fest, d.h. falls $indeg(v) = k$, dann ist $IN(v) = (e_1, \ldots, e_k)$ mit $Z(e_i) = v \quad \forall 1 \leq i \leq k$.
- Die Folge $\vec{Y}_m = (y_1, \ldots, y_m)$ zeichnet Knoten $y_i \in V$ als Ausgänge aus.

# Belegungen von Schaltplänen (1/2)

Sei nun ein Schaltplan $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_n)$ gegeben.

- Eine Abbildung $\Phi_{SP,a} : V \to \{0,1\}$ für $a = (a_1, \ldots, a_n) \in \mathbb{B}^n$ heißt Belegung für Eingabebelegung $a$, falls
  - $\Phi_{SP,a}(x_i) = a_i \ \forall 1 \leq i \leq n$,
  - $\Phi_{SP,a}(0) = 0$, $\Phi_{SP,a}(1) = 1$.

# Belegung von Schaltplänen (2/2)

Interessant für uns sind stabile Belegungen.

- Eine Belegung $\Phi_{SP,a} : V \to \{0,1\}$ heißt stabil, wenn gilt:
  - Für alle $v \in I$ mit $typ(v) = g \in B_k, IN(v) = (e_1, \ldots, e_k)$ ist $\Phi_{SP,a}(v) = g(\Phi_{SP,a}(Q(e_1)), \ldots, \Phi_{SP,a}(Q(e_k)))$.
  - $(\Phi_{SP,a}(y_1), \Phi_{SP,a}(y_2), \ldots, \Phi_{SP,a}(y_n))$ heißt Ausgangsbelegung von $SP$ bei Eingangsbelegung $a$.

Es ist möglich, dass es zu einer Eingangsbelegung $a$

- keine stabile Signalbelegung $\Phi_{SP,a}$ gibt,
- mehrere stabile Signalbelegungen $\Phi_{SP,a}$ gibt.

# Ein weiterer Schaltkreis (1/2)

$$a = (a_1, a_2, a_3)$$

# Ein weiterer Schaltkreis (2/2)

# RS-Flipflop (RS-FF)

- Die vorherige Schaltung heißt RS-Flipflop.
- Sie hat mindestens zwei stabile Zustände.

Zustand $Q = 0$

Zustand $Q = 1$

# Übergang (1/2)

- Für das Umschalten von einem Zustand zum anderen in einer realen Implementierung eines RS-FFs ist es von entscheidender Bedeutung, dass reale Gatter Verzögerungszeiten haben.
- D.h.: Wenn sich die Eingangsbelegung eines Gatters ändert, dann erfolgt die daraus resultierende Änderung des Ausgangswertes nicht direkt, sondern mit einer gewissen Verzögerung.
- (Detailliertere Betrachtung in Kapitel 5, Physikalische Eigenschaften von Gattern.)

# Übergang (2/2)

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.
- Nach Zeit $t_{P/S/Q}$ ist $/Q = 0$.

# Weitere Bezeichnungen

- Umschalten des FF in Zustand $Q = 1$ heißt Setzen (set).
- Umschalten des FF in Zustand $Q = 0$ heißt Zurücksetzen (reset).
- $/S$ heißt Set-Signal.
- $/R = /C$ heißt Reset- oder Clear-Signal.
- Weil $/R, /S$ durch Absenken aktiviert werden, nennt man sie active low.
- Signalnamen von active-low-Signalen beginnen in der Regel mit /.

# “Zustand” $Q = 1$, $/Q = 1$

- Stabile Signalbelegung bei Eingangsbelegung $/S = 0$, $/Q = 0$
- Aber warum ist es trotzdem problematisch, $/S$ und $/R$ gleichzeitig zu aktivieren ($/S = 0$, $/R = 0$)?

# Flackern

Annahme:

- $/S$ und $/R$ werden nach ihrer Aktivierung beide gleichzeitig inaktiv ($00 \rightarrow 11$)
- $G_1$ und $G_2$ schalten exakt gleich schnell, d.h. haben exakt die gleiche Verzögerungszeit

- $\Rightarrow$ Es kommt es zum Flackern ("metastabiler" Zustand).
- In der Praxis wird in der Regel nach einer gewissen Zeit einer der beiden stabilen Zustände angenommen (weil Gatterverzögerungen leicht variieren).

# Schaltsymbol eines RS-FF

# Nachteil von RS-FF

Beim Speichern eines Wertes 0 oder 1 muss man den Wert kennen:

- 0 $\rightarrow$ Aktiviere $/R$
- 1 $\rightarrow$ Aktiviere $/S$

**Ziel:**

- Speichern unbekannter Werte.

# D-Latch (1/2)

?

| $W$ | $D$ | $/S$ | $/R$ |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

- $W$ ist active high.
  - $W = 0 \Rightarrow /S, /R$ inaktiv
  - $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv, falls } D = 1 \\ /R \text{ aktiv, falls } D = 0 \end{cases}$

# Ansteuerung: Schreibimpuls

- Die Daten müssen für eine gewisse Zeit $t_{SDW}$, genannt Setup-Zeit, an $D$ stabil anliegen.
- Dann geht $W$ von 0 auf 1, bleibt für eine Zeit $y$, genannt Pulsweite, auf 1 und geht auf 0 zurück.
- Anschließend müssen die Daten für eine Zeit $t_{HDW}$, genannt Hold-Zeit, an $D$ stabil gehalten werden.

# Timing-Diagramm

Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?

$\Rightarrow$ Siehe nächstes Kapitel (Timing).

# Weitere Eigenschaften eines D-Latches

- Bisher: Keine Datenänderungen während des Schreibpulses auf $W$.
- Man kann das D-Latch aber auch im transparenten Modus betreiben:
  - Das D-Latch heißt transparent, wenn das Schreibsignal aktiv ist.
  - Hält man $W$ lange aktiv und ändert $D$ zur Zeit $t$, dann ändert sich $Q$ zur Zeit $t + t_{PDQ}$.
  - Auch hier sind zeitliche Bedingungen zu beachten: Keine Datenänderungen kurz nach Beginn des Transparenzmodus bzw. kurz vor Ende des Transparenzmodus.

# Taktflankengesteuertes D-Flipflop (1/2)

- Taktflankengesteuerte Flipflops wie das D-Flipflop übernehmen Daten zu einem bestimmten Zeitpunkt (kein transparenter Modus!), nämlich bei der steigenden Flanke des Clocksignals.

| $D$ | $ck$ | $Q$ | $/Q$ |
|---|---|---|---|
| 0 | ⌐ | 0 | 1 |
| 1 | ⌐ | 1 | 0 |
| $X$ | 0 | $Q$ | $/Q$ |
| $X$ | 1 | $Q$ | $/Q$ |

- Vorteil: Daten müssen lediglich bei der steigenden Taktflanke stabil sein (zzgl. Setup- und Holdzeit).

# Taktflankengesteuertes D-Flipflop (2/2)

- Realisierung: Wesentlich komplexer als bei taktzustandsgesteuerten D-Latches
- Analyse des Schaltplanes (und entsprechende Timing-Analyse) wesentlich komplizierter.

# D-FF: Realisierung mit RS-Flipflops

# Einfache Bausteine mit Flipflops

- Register
- Schieberegister
- Zähler

# $n$-Bit Register

- $n$ D-Flipflops mit gemeinsamen Clocksignal.

- Entsprechend: $n$-Bit Latch = $n$ D-Latches mit gemeinsamem Schreibsignal $W$.

# Schieberegister

- In jedem Takt (bei jeder steigenden Flanke von *ck*) werden die Werte im Register um eine Position nach rechts verschoben.

# Zähler

Ein $n$-Bit-Zähler ist eine Schaltung mit folgenden Ein- und Ausgängen:

- Dateneingänge $X = (X_{n-1}, \ldots, X_0)$
- Datenausgänge $Y = (Y_{n-1}, \ldots, Y_0)$
- Dateneingang $C_{in}$ für Eingangsübertrag
- Datenausgang $C_{out}$ für Ausgangsübertrag
- Eingänge für Kontrollsignale:
  - $/C$ (Clear)
  - $/L$ (Load)
  - $ck$ (Clock)

# Aufbau eines Zählers

# $n$-Bit Zähler: Funktionalität

- Ein Zähler speichert ein $n$-Bit-Wort, das an den Ausgängen $Y$ erscheint (Zählerstand).
- Bei jeder steigenden Flanke von $ck$ wird ein neuer Zählerstand $Y_{neu}$ gespeichert. Für $Y_{neu}$ gilt :

$$Y_{neu} = \begin{cases} 0\ldots0, & \text{falls } /C = 0 \\ X, & \text{falls } /C = 1, /L = 0 \\ bin_n((\langle Y \rangle + C_{in}) mod 2^n), & \text{falls } /C = 1, /L = 1 \end{cases}$$

# $n$-Bit Zähler kaskadieren

- Ausgangsübertrag $C_{out}$ ermöglicht es, den Zähler zu kaskadieren, zum Beispiel aus $s$ $n$-Bit-Zählern einen $(s \cdot n)$-Bit-Zähler zu bauen.

# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. **Sequentielle Schaltkreise**
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Sequentielle Schaltkreise

- Im Folgenden werden keine allgemeinen Schaltpläne mehr analysiert, sondern sogenannte Schaltwerke (auch (synchrone) sequentielle Schaltkreise genannt).
- Diese bestehen aus einem Register und einem (kombinatorischen) Schaltkreis (auch kombinatorischer Kern genannt).
- Im Gegensatz zu (kombinatorischen) Schaltkreisen können Schaltwerke (= sequentielle Schaltkreise) Zyklen enthalten. Die Zyklen müssen aber durch Flipflops des Registers gehen.
- Der Zustand eines Schaltwerkes ist gegeben durch die im Register gespeicherten Werte.
- Schaltwerke (= sequentielle Schaltkreise) entsprechen endlichen Zustandsautomaten.

# Beispiel: Zähler als sequentieller Schaltkreis

# Endliche Zustandsautomaten

- Endliche Zustandsautomaten (Finite State Machines, FSMs) sind ein Formalismus, um sequentielles (zeitabhängiges) Verhalten zu spezifizieren.
  - Mealy- und Moore-Automaten
  - In der theoretischen Informatik werden endliche Automaten mit akzeptierenden Zustände betrachtet. Diese sind mit FSMs verwandt, aber nicht identisch.
- Aus einer FSM-Spezifikation kann der sequentielle Schaltkreis hergeleitet werden (Sequentielle Synthese).

# Halbautomat

## Definition

Das Quadrupel $H = (I, S, S_0, \delta)$ heißt deterministischer, endlicher Halbautomat. Dabei bezeichnet:

- $I$ eine endliche Menge von erlaubten Eingabesymbolen ("Eingabealphabet"),
- $S$ eine endliche Menge von Zuständen,
- $S_0 \subseteq S$ ist eine endliche Menge von erlaubten Anfangszuständen,
- $\delta : S \times I \to S$ eine Übergangsfunktion.

# Beispiel: Kaffeeautomat

Auswahl:
- Kaffee
- Tee
- Zucker

korrekter Münzeinwurf oder Fertig

# Darstellung als Zustandsdiagramm

- Knoten: Zustände des Automaten.
- Kanten: Zustandsübergänge.
- Kantenmarkierung: Eingabe (bzw. Ereignis).

# Mealy- und Moore-Automat

## Definition

Ein Mealy-Automat $M = (I, O, S, S_0, \delta, \lambda)$ ist ein endlicher deterministischer Halbautomat $H$ erweitert um:

- eine endliche Menge $O$ von Ausgabesymbolen („Ausgabealphabet"),
- eine Ausgabefunktion $\lambda : S \times I \to O$.

## Definition

Ein Moore-Automat $M = (I, O, S, S_0, \delta, \lambda)$ ist ein endlicher, deterministischer Halbautomat $H$ erweitert um:

- eine endliche Menge $O$ von Ausgabesymbolen,
- eine Ausgabefunktion $\lambda : S \to O$.

# Beispiel: Getränkeauswahl - Mealy-Automat

- Knoten: Zustände des Automaten.
- Kanten: Zustandsübergänge.
- Kantenmarkierung: Ein-/Ausgabe (Mealy-Automat).
- Beim Moore-Automaten werden die Zustände mit der Ausgabe beschriftet.
- $\varepsilon$ bedeutet keine Eingabe/Ausgabe.

# Mealy- vs. Moore-Automat (1/2)

- Beim Mealy-Automaten ist:
  - die Ausgabe abhängig vom aktuellen Zustand und der aktuellen Eingabe,
  - der Folgezustand abhängig vom aktuellen Zustand und der aktuellen Eingabe.
- Ein Moore-Automat ist ein spezieller Mealy-Automat, bei dem die Ausgabe nur vom aktuellen Zustand und nicht von der Eingabe abhängt.
- Moore- und Mealy-Automaten kann man ineinander überführen.

# Mealy- vs. Moore-Automat (2/2)

- Überführung Moore $\rightarrow$ Mealy: trivial
- Überführung Mealy $\rightarrow$ Moore:
  Grundidee:“Ziehe Ausgabe in den Zustand”

# Unterschiedliche Darstellungen von endlichen Zustandsautomaten

*a)* Zustands- und Ausgangstafel:

| $x$ | *state* | *next-state* | $y$ |
|---|---|---|---|
| 1 | $s_1$ | $s_1$ | 0 |
| 0 | $s_1$ | $s_2$ | 1 |
| - | $s_2$ | $s_1$ | 0 |

*b)* Flusstafel:

| | $x = 0$ | $x = 1$ |
|---|---|---|
| $s_1$ | $s_2, 1$ | $s_1, 0$ |
| $s_2$ | $s_1, 0$ | $s_1, 0$ |

*c)* Zustandsdiagramm:

*d)* Sequentieller Schaltkreis:

- Im Folgenden: Weg von *c)* zu *d)*

# Sequentielle Schaltkreise allgemein

$$y_i^t = \lambda_i(x_1^t, x_2^t, \ldots, x_k^t, s_1^t, s_2^t, \ldots, s_p^t)$$
$$s_i^{t+1} = \delta_i(x_1^t, x_2^t, \ldots, x_k^t, s_1^t, s_2^t, \ldots, s_p^t)$$

Die Belegung $s_t$ der Flipflops im Register heißt Zustand des sequentielle Schaltkreises zum Zeitpunkt $t$.

# Kombinatorischer Kern

- Der kombinatorische Kern hat vier Arten von Ein- und Ausgängen:
  - Primäre Eingänge bekommen Werte „von außen”.
  - Primäre Ausgänge liefern Werte „nach außen”.
  - Sekundäre Eingänge sind mit den Datenausgängen der Flipflops im Register verbunden. Auf diese Weise kann der aktuelle Zustand des Schaltkreises in Funktionen $\delta$ und $\lambda$ berücksichtigt werden.
  - Sekundäre Ausgänge sind mit den Dateneingängen der Flipflops verbunden. Durch sie wird der nächste Zustand des Schaltkreises spezifiziert.

# Primäre und sekundäre Ein- und Ausgänge

$$
\begin{aligned}
y_i^t &= \lambda_i(x_1^t, x_2^t, \ldots, x_k^t, s_1^t, s_2^t, \ldots, s_p^t) \\
s_i^{t+1} &= \delta_i(x_1^t, x_2^t, \ldots, x_k^t, s_1^t, s_2^t, \ldots, s_p^t)
\end{aligned}
$$

# Sequentielle Schaltung für einen FSM

## Mealy-Automat

## Moore-Automat

- Eingabevektor: $X = (x_1, x_2, \ldots, x_k)$
- Ausgabevektor: $Y = (y_1, y_2, \ldots, y_m)$
- Zustandsvektor: $S = (s_1, s_2, \ldots, s_p)$

- Ausgabefunktion (Mealy): $Y^t = \lambda(X^t, S^t)$
- Übergangsfunktion: $S^{t+1} = \delta(X^t, S^t)$
- Ausgabefunktion (Moore): $Y^t = \lambda(S^t)$

# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. **Entwurf sequentieller Schaltkreise**
4. SRAM
5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Entwurf sequentieller Schaltkreise

Zustandsdiagramm:

$\Rightarrow$

Zustands- und Ausgangstafel:

| $s^t$ | $x_1^t$ | $x_2^t$ | $s^{t+1}$ | $y^t$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

Sequentieller Schaltkreis:

$\Rightarrow$

# Entwurfsschritte

- Optimierung des Zustandsdiagramms: Zustandsminimierung
  - Identifikation der äquivalenten Zustände.
  - Ergebnis: Ein (evtl. kleineres) Zustandsdiagramm.

- Wahl der Zustandskodierung.
  - Ergebnis: Anzahl der Flipflops im Register, Funktionen $\delta$ und $\lambda$ (Zustands- und Ausgangstafel).

- Implementierung von $\delta$ und $\lambda$.
  - Kombinatorische Logiksynthese, z.B. Quine-McCluskey.

# Zustandsminimierung

**Idee:**

Bestimme und verschmelze äquivalente Zustände.

- Zwei Zustände sind äquivalent, wenn der Automat von ihnen aus bei gleichen Eingabefolgen stets die gleichen Ausgabefolgen produziert.

# Weiteres Beispiel (1/4)

- Hinreichende Bedingung: Wenn bei zwei Zuständen bei gleicher Eingabe auch die gleiche Ausgabe erzeugt wird und der gleiche Folgezustand angenommen wird, dann sind die Zustände sicherlich äquivalent.
- Äquivalente Zustände können durch einen einzigen Zustand ersetzt werden (siehe nächste Folie).

# Weiteres Beispiel (2/4)

Zustand e und d sind äquivalent.

# Weiteres Beispiel (3/4)

Zustand e eliminiert.
Zustand b und c sind äquivalent.

# Weiteres Beispiel (4/4)

Zustand c eliminiert.

# Entwurfsschritte

- Optimierung des Zustandsdiagramms: Zustandsminimierung
  - Identifikation der äquivalenten Zustände.
  - Ergebnis: Ein (evtl. kleineres) Zustandsdiagramm.

- Wahl der Zustandskodierung.
  - Ergebnis: Anzahl der Flipflops im Register, Funktionen $\delta$ und $\lambda$ (Zustands- und Ausgangstafel).

- Implementierung von $\delta$ und $\lambda$.
  - Kombinatorische Logiksynthese, z.B. Quine-McCluskey.

# Zustandskodierung

Kodierung $S_1 \equiv 00, S_3 \equiv 10, S_4 \equiv 01$ : 9 Literale

$$\delta_1(s,i) = s_2 i + s_1 \bar{i}$$
$$\delta_2(s,i) = \overline{s_1}\,\overline{s_2} i$$
$$\lambda(s,i) = \overline{s_2} i$$

Kodierung $S_1 \equiv 01, S_3 \equiv 11, S_4 \equiv 10$ : 11 Literale

$$\delta_1(s,i) = s_1 s_2 \bar{i} + \overline{s_1} i + \overline{s_2} i$$
$$\delta_2(s,i) = s_1 + i$$
$$\lambda(s,i) = s_2 i$$

- **Ziel:** Wähle Zustandskodierung, die nachfolgende kombinatorische Synthese erleichtert.
- Dafür gibt es (heuristische) Verfahren.

# Entwurf eines einfachen sequentiellen Schaltkreises am Beispiel

- Aufgabenbeschreibung (Textspezifikation): Modulo-4 Vorwärts/Rückwärtszähler
  - Der Zähler soll von $0$ bis $3$ zählen können.
  - Ist der Steuereingang $x$ auf $1$ gesetzt, so soll vorwärts gezählt werden, d.h. die Zahlenfolge $0, 1, 2, 3$ durchlaufen werden.
  - Ist $x$ auf 0 gesetzt, so soll rückwärts gezählt werden, d.h. die Zahlenfolge $3, 2, 1, 0$ durchlaufen werden.
  - Am Ausgang ist der Zählerstand anzugeben (Ausgabevektor $y_0, y_1$). Der Zähler ist als Ringzähler zu realisieren.

# Von der Textspezifikation zum Zustandsdiagramm

- 4 Zustände erforderlich.
- Startzustand 0.

# Vom Zustandsdiagramm zur Zustands- und Ausgangstafel

- Zustandsminimierung $\Rightarrow$ Keine äquivalente Zustände.
- Zustandskodierung: $0 \rightarrow 00, 1 \rightarrow 01, 2 \rightarrow 10, 3 \rightarrow 11$.

| | $x$ | $z_1^t$ | $z_0^t$ | $z_1^{t+1}$ | $z_0^{t+1}$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| Vorwärts-zählen | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| Rückwärts-zählen | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Eingänge — Ausgänge

# Implementierung des kombinatorischen Kerns

| | $x$ | $z_1^t$ | $z_0^t$ | $z_1^{t+1}$ | $z_0^{t+1}$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| Vorwärts-zählen | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| Rückwärts-zählen | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Übergangsfunktion:

$$z_0^{t+1} = x\overline{z}_1^t\overline{z}_0^t + xz_1^t\overline{z}_0^t + \overline{x}z_1^t\overline{z}_0^t + \overline{x}\,\overline{z}_1^t\overline{z}_0^t$$

$$z_1^{t+1} = x\overline{z}_1^t z_0^t + xz_1^t\overline{z}_0^t + \overline{x}z_1^t z_0^t + \overline{x}\,\overline{z}_1^t\overline{z}_0^t$$

# Implementierung des komb. Kerns: Logikminimierung

Ausgangsfunktion: $y_0^t = z_0^t, \quad y_1^t = z_1^t$

Übergangsfunktion:

$$z_0^{t+1} = x\overline{z}_1^t\overline{z}_0^t + xz_1^t\overline{z}_0^t + \overline{x}z_1^t\overline{z}_0^t + \overline{x}\,\overline{z}_1^t\overline{z}_0^t$$

$$z_1^{t+1} = x\overline{z}_1^t z_0^t + xz_1^t\overline{z}_0^t + \overline{x}z_1^t z_0^t + \overline{x}\,\overline{z}_1^t\overline{z}_0^t$$

Minimierung:

$$z_0^{t+1} = \overline{z}_0^t$$

$$z_1^{t+1} = x\overline{z}_1^t z_0^t + xz_1^t\overline{z}_0^t + \overline{x}z_1^t z_0^t + \overline{x}\,\overline{z}_1^t\overline{z}_0^t$$

# Beispiel: Ergebnis

Resultierender Automat ist Moore-Automat.

Ausgabefunktion ist Identität auf den Zustandsbits.

# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. **SRAM**
5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# SRAM

- Static Random-Access Memory.
- Konzeptuell: Eine (sehr große) Anzahl $N$ von Speicherzellen sowie ein (sehr großer) Multiplexer, um auf die einzelnen Zellen zuzugreifen.
- Durch die Größe spielen Beschränkungen eine Rolle, die bei einer Realisierung beachtet werden müssen.
  - Fanout-Beschränkung: Eine Leitung kann nicht beliebig verzweigen $\Rightarrow$ Treiberbäume
  - Fanin-Beschränkung: Ein Gatter kann nicht beliebig viele Eingänge haben (Hier: $N$-faches ODER als Baum).

# SRAM: Ein-/Ausgänge und Zeichen

- Sei $n \in \mathbb{N}, N = 2^n$. Ein $N$-Bit statischer Speicher oder SRAM hat:
  - $n$ Eingänge $A = (A_{n-1}, \ldots, A_0)$ „Adresse",
  - Dateneingang $D_{in}$,
  - Datenausgang $D_{out}$,
  - Kontrollsignal $W$ „write"

# SRAM: Funktionalität

- Der Speicher enthält $N$ Speicherzellen $L_0, \ldots, L_{N-1}$, die je ein Bit speichern können.
- Zelle $L_{\langle A \rangle}$ wird mit Hilfe der Adresse $A$ ausgewählt.
  - An $D_{out}$ erscheint der Inhalt von $L_{\langle A \rangle}$.
  - Durch Schreibpuls an $W$ wird $D_{in}$ nach $L_{\langle A \rangle}$ übernommen.

# $N$-Bit-SRAM, $N \times s$-Bit-SRAM: Aufbau

- Ein $N \times s$-Bit-SRAM besteht aus $s$ $N$-Bit SRAMs mit gemeinsamen Adress- und Schreibsignalen.
  - $s$ heißt Bitbreite des $N \times s$-Bit-SRAMs.
- Ein $N$-Bit-SRAM besteht im Prinzip aus 3 Hilfsschaltkreisen:
  - mehrfaches ODER
  - Treiberbäume
  - Dekodierer

# SRAM: Schaltbild

# $O_N$: Mehrfaches ODER

- Ein N-faches ODER $O_N$ mit $N = 2^n$ ist ein Schaltkreis, der $N$-faches Oder berechnet.
- Balancierter Baum, um Verzögerungszeit zu minimieren (Tiefe $O(\log N) = O(n)$).

# Treiberbäume

- Ein Treiber ist ein Gatter mit einem Eingang $X$ und einem Ausgang $Y$, das die Identität $Y = X$ berechnet.
  - Eingesetzt, um Fanout-Beschränkung zu überwinden.
  - Beispiel: Fanout-Beschränkung von 3.

# $F_N$: Treiberbäume im SRAM

- Zur Erinnerung: Ein Baum ist ein azyklischer gerichteter Graph $G = (V, E)$ mit:
  - Genau einer Quelle $w$,
  - $indeg(v) = 1$ für alle $v \in V \setminus \{w\}$
  - Blätter = Knoten $v \in V$ mit $outdeg(v) = 0$
  - Innere Knoten = Knoten $v \in V$ mit $outdeg(v) \geq 1$.
- Im SRAM für Realisierung von $F_N$ eingesetzt.
- Fanout-Beschränkung von 10 $\rightarrow$ 10-äre Bäume.

# 10-äre Bäume (1/2)

$B_0$:

$B_1$:

$B_s$:

- Anzahl der Blätter von $B_s$: $L(B_s)$.
  - Entspricht dem erreichten Verzweigungsgrad.
- Anzahl der inneren Knoten von $B_s$: $I(B_s)$.
  - Entspricht der benötigten Anzahl von Treibern.

$$L(B_s) = 10^s$$

$$I(B_s) = \sum_{i=0}^{s-1} 10^i = \frac{10^s - 1}{10 - 1} < \frac{L(B_s)}{9}$$

# 10-äre Bäume (2/2)

- Benutze also $B_s$ zum $10^s$-fachen Vervielfältigen eines Signals
- Innere Knoten des Baumes werden durch Treiber ersetzt
- $\Rightarrow$ Treiberbaum mit Fanoutbeschränkung 10

# Treiberbäume: Allgemeiner Fall (1/2)

- Angenommen, ein Signal soll $N$-fach vervielfältigt werden mit $10^{s-1} < N < 10^s$ keine Zehnerpotenz.
- Ziel: Balancierte Treiberbäume, d.h. alle Pfade von der Wurzel zu einem Blatt haben gleiche Länge.

$N$ = 11, unbalanciert

$N$ = 11, balanciert

# Treiberbäume: Allgemeiner Fall (2/2)

- Idee: “Fülle Bäume von links her 10-är auf” und sorge zusätzlich für gleiche Tiefe der Blätter!
- Beispiel: ...

## Lemma

$\forall s \in \mathbb{N}$ und $N \in \{10^{s-1}+1, \ldots, 10^s\}$ gibt es einen Baum $T(N)$ mit Ausgangsgrad $\leq 10$ an jedem inneren Knoten und den folgenden Eigenschaften:

1. $T(N)$ hat $N$ Blätter.
2. $T(N)$ hat $\leq \frac{N}{9} + s$ innere Knoten.
3. Alle Pfade von der Wurzel zu einem Blatt haben genau die Länge $s = \lceil \log_{10} N \rceil$ mit $\lceil \log_{10} N \rceil < \frac{1}{3} \log_2 N + 1$.

- Beweis: Induktion über $s \Rightarrow$ Übung

# Notation: Invertierender Treiberbaum

- Invertierender Treiberbaum $/F_x$:
  Ersetze den Treiber an der Wurzel durch einen Inverter.

# $D_n$: Dekodierer

- Sei $n \in \mathbb{N}$, $N = 2^n$. Ein $n$-Bit-Dekodierer $D_n$ ist ein Schaltkreis, der die Funktion $d : \mathbb{B}^n \to \mathbb{B}^N$ berechnet, wobei gilt:

$$d_i(a) = \begin{cases} 1, & \text{falls } \langle a \rangle = i \\ 0, & \text{sonst} \end{cases} \qquad \forall i = 0, \ldots, N-1$$

  ($d_i(a)$ ist Bit $i$ des $N$-Tupels $d(a)$.)

- Induktive Konstruktion von $D_n$: Siehe nächste Folie.

# Dekodierer: Rekursiver Aufbau

# SRAM: Lesevorgang ($W = 0$)

- $D_n$ setzt $Y_i = 1$ für $i = \langle A \rangle$, $Y_j = 0$ für $j \neq i$.
- Der Inhalt der $i$-ten Zelle $L_i$ steht an $G_i$, für alle $j \neq i$ steht an $G_j$ der Wert 0.

# SRAM: Schreibvorgang (Puls auf $W$)

- $D_{in}$ an $D$-Eingänge sämtlicher Latches angelegt.
- Schreibpuls nur am $W$-Eingang von $L_i$ (da $Y_i = 1$).

# Tristate-Treiber und Busse

- Tristate-Treiber sind Treiber mit Eingangssignal $x$ und zusätzlichem Signal $/OE$, dem Output-Enable-Signal.

- Am Ausgang $y$ erscheint

$$y = \begin{cases} x, & \text{falls } /OE = 0 \\ Z, & \text{sonst } /OE = 1 \end{cases}$$

- $Z$ bezeichnet den Zustand hoher Impedanz (high-$Z$).

# Zustand hoher Impedanz

- Wir haben bisher Schaltungen betrachtet, die aus CMOS-Gattern bestehen. Dort ist jede Leitung zu jedem Zeitpunkt entweder mit $V_{DD}$ (logisch-1) oder Masse (logisch-0) verbunden.

- Eine Leitung im Zustand $Z$, also der Ausgang eines Treibers mit $/OE = 1$, ist weder mit $V_{DD}$ noch mit Masse verbunden. Man sagt, der Treiber ist disabled ($/OE = 0$: enabled).

# $n$-Bit-Treiber

- $n$-Bit-Treiber: $n$ Treiber mit gemeinsamen $/OE$.
- Im Gegensatz zu Ausgängen üblicher Gatter kann man Ausgänge von Tristate-Treibern zusammenschalten. Man muss dafür sorgen, dass zu jeder Zeit höchstens ein Treiber enabled ist.
- Ein $n$ Bit breiter Bus ist ein Bündel aus $n$ Leitungen, welches die Ausgänge von mehreren $n$-Bit-Treibern verbindet.

# Bus vs. Multiplexer

- $k$ Tristate-Treiber, die durch einen Bus verbunden sind, wirken ähnlich wie ein $k$-fach-Multiplexer.

- Vorteile Bus gegenüber Multiplexer:
  - Leicht erweiterbar.
  - Datentransport in verschiedene Richtungen zu verschiedenen Zeiten.

- Nachteil von Bus:
  - Man muss Bus Contention vermeiden, d.h. es darf nie mehr als ein Treiber auf einem Bus gleichzeitig enabled sein (sonst Folgen bis hin zur physikalischen Zerstörung der Schaltung)!

# Bus zur Kommunikation mit SRAM

- SRAM mit gemeinsamem Datenein- und -ausgang.

- Lesezugriff auf den Speicher: $/DOE_1$ enabled, alle anderen Treiber, z.B. $/DOE_2$, disabled.

- Schreibzugriff: $D_{in}$ nimmt den Wert vom Bus, $/DOE_1$ disabled.

# Kapitel 4

Sequentielle Logik:

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. **Anwendung: Datenpfade von ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Zur Erinnerung: ReTI bisher

# Zur Erinnerung: Datenpfade von ReTI

- ReTI besteht aus
  - 4 benutzersichtbaren Registern *PC*, *ACC*, *IN*1, *IN*2
    $\rightarrow$ Realisiert durch Zähler (*PC*) bzw. Register.
  - Einem $2^{32}$-Wort-Speicher (Wortbreite von 32 Bit), der Daten und Befehle enthält
    $\rightarrow$ Realisiert durch SRAM.
- ReTI unterstützt Load-/Store-, Compute-, Indexregister- und Sprungbefehle.

| 31 30 | 29 ... 24 | 23 ... 0 |
|---|---|---|
| Typ | Spezifikation | Parameter *i* |

# Zur Erinnerung: ReTI-Befehle im Überblick

| | 31 30 | 29 28 | 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|---|
| Load | 0 1 | M | * | D | i |

| | 31 30 | 29 28 | 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|---|
| Store | 1 0 | M | S | D | i |

| | 31 30 | 29 | 28 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|---|
| Compute | 0 0 | MI | F | D | i |

| | 31 30 | 29 28 27 | 26 25 24 | 23 … 0 |
|---|---|---|---|---|
| Jump | 1 1 | C | * | i |

M - Modus ; S - Source ; D - Destination ; MI - memory/immediate ;
F - Function ; C - Condition

# Umsetzung von ReTI: Externe Sicht

- 3-Bus-Architektur zur Ansteuerung von Speicher und I/O-Geräten:
  - 32 Bit breiter Datenbus $D = D[31:0]$,
  - 32 Bit breiter Adressbus $A = A[31:0]$,
  - Kontrollbus $C$ (Breite später festgelegt).

# Umsetzung von ReTI: Interne Sicht

- *CPU* besteht aus:
  - Zähler *PC*.
  - 3 für Benutzer sichtbaren Registern *ACC*, *IN*1, *IN*2,
  - Instruktionsregister *I*,
  - *ALU*,
  - *CPU*-internen Bussen:
    - *L*, *R* für linken bzw. rechten Operanden der *ALU*,
    - internem Datenbus *DI*.
- Register, *PC*, *ALU*, Busse und die zugehörigen Treiber sind 32 Bit breit.

# Interner Aufbau der ReTI-CPU

# Hinweise zum Schaltbild

- Namenskonvention für Treiber: Treiber zwischen Bus/Baustein $X$ und Bus/Baustein $Y$: $XYd$;
  - Output-Enable-Signal: $XYdoe$.
  - $0Ld$ und $0Rd$ können $0^{32}$ auf $L$ bzw. $R$ legen.
- Busse $A$ und $D$ sind an den Speicher (sowie I/O-Geräte) angeschlossen, s. nächste Folie.
- Das Bild enthält keine Steuerleitungen:
  - Output-Enable-Signale der Treiber,
  - Funktionsauswahl der ALU,
  - Clock-Signale und “Clock-Enable-Signale” der Register.

# Speicher SM von ReTI

- Datenein- und ausgänge mit Datenbus *D* der *CPU* verbunden.
- Adressleitungen mit Adressbus *A* der *CPU* verbunden.
- Treiber *ASMd* immer enabled.

# Verfeinerung des Schaltbilds

# Verarbeitung der Daten im Register *I*

- Im Instruktionsregister *I* steht der gerade verarbeitete Befehl.
  - $I[31:24]$: Befehlskodierung (für die im Schaltbild nicht dargestellten Steuersignale benötigt).
  - $I[23:0]$: Speicheradresse oder Konstante für die ALU.
    - Speicheradresse wird mit acht Nullen aufgefüllt. $0^8 I[23:0]$ wird auf Bus *A* gelegt (*IAd* enabled).
    - Natürliche 24-Bit-Konstante wird mit acht Nullen aufgefüllt. $0^8 I[23:0]$ wird auf *R* gelegt (*IRd* enabled).
    - Ganzzahlige 24-Bit-Konstante wird vorzeichenerweitert. $sext(I[23:0])$ wird auf *R* gelegt (*IRd* enabled).

# Verfeinerung

- Neues Kontrollsignal sext:
  - $sext = 0$: Ersetze $I[31:0]$ durch $0^8 I[23:0]$.
  - $sext = 1$: Ersetze $I[31:0]$ durch $sext(I[23:0])$.

# Weitere Verfeinerung für Register (1/3)

- Im Buch von Keller / Paul werden die Clockeingänge aller Register Reg mit einem Clocksignal Regck verbunden, das durch die Kontrolllogik berechnet wird.
- $\Rightarrow$ Datenübernahme zu ausgewählten Zeitpunkten

- Vorgehen bei Realisierung der ReTI auf einer Platine mit diskreten Bausteinen ok!
- Nicht ok bei heutigen Designs, die Prozessoren auf einem einzigen Chip integrieren.

# Weitere Verfeinerung für Register (2/3)

- Gründe für “Designrule”, die es verbietet, Clockeingänge mit berechneten Datensignalen zu verbinden:
  - Spezielle Methoden (“Clock–Tree–Synthese”) gewährleisten, dass die steigende Flanke der globalen Clock an allen Clockeingängen zum gleichen Zeitpunkt ankommt (z.B. durch Ausgleich des Effekts unterschiedlicher Leitungsverzögerungen m.H. von Treibern). Datensignale auf Clockeingängen verhindern Clock–Tree–Synthese.
  - Heutige Werkzeuge zur automatischen Timing–Analyse (und Berechnung der maximalen Clockfrequenz) sind nicht in der Lage, mit berechneten Clocksignalen umzugehen.
  - Spezielle Anforderungen an “Flankensteilheit” der Clocksignale (siehe Kapitel über physikalische Eigenschaften)

# Weitere Verfeinerung für Register (3/3)

- Transformation bzw. Verfeinerung:

# Programmabarbeitung durch ReTI

- Zwei sich abwechselnde Phasen der *CPU*:
  - Fetch-Phase: Lädt nächsten auszuführenden Befehl aus Memory ins Instruktionsregister *I* der *CPU*.
  - Execute-Phase: Befehl, der in *I* steht, wird ausgeführt.
- Vorgehen:
  1. Definition der Datenpfade, d.h. der benötigten Datenverbindungen zwischen den Komponenten der *CPU*.
  2. Herleitung der Kontrollsignale zur Ansteuerung der im Punkt 1 hergeleiteten Datenpfade.
     - Treiber-OE, *ALU*-Funktionsselektion, Register-Clock.
  3. Sequentielle Synthese.

# Datenpfade: Fetch-Phase

# Datenpfade: Fetch-Phase

# Bedeutung des Diagramms

- In der Fetch-Phase muss:
  - Bus *A* mit *PC* verbunden sein, d.h. *PCAd* enabled.
  - Register *I* den Wert von Bus *D* übernehmen, d.h. *Icken* muss enabled werden.
  - Alle anderen Treiber (außer denen, die stets enabled sind) sind disabled, um Bus Contentions zu vermeiden.
- Die Steuersignale müssen in der Fetch-Phase entsprechend gesetzt sein.
  - Z.B. $/PCAdoe = 0$, $/ALUAdoe = 1$, usw. (active low!)

# Fetch: Die durchgeschalteten Pfade

# Datenpfade in der Execute-Phase

- Betrachte unterschiedliche Befehlstypen.
  - *Compute Immediate*,
  - *Compute Memory*,
  - *JUMP*,
  - *LOAD*,
  - *LOADIN*1 (*LOADIN*2 analog),
  - *LOADI*,
  - *STORE*,
  - *STOREIN*1,
  - *MOVE*.

# Datenpfade: *Compute Immediate* (1/2)

# Datenpfade: *Compute Immediate* (2/2)

# Datenpfade: *Compute Memory* (1/2)

# Datenpfade: *Compute Memory* (2/2)

# Datenpfade: *JUMP* (1/2)

# Datenpfade: *JUMP* (2/2)

# Datenpfade: *LOAD i* (1/2)

# Datenpfade: *LOAD i* (2/2)

# Datenpfade: *LOADIN*1 *i* (1/2)

# Datenpfade: *LOADIN*1 *i* (2/2)

# Datenpfade: *LOADI i* (1/2)

# Datenpfade: *LOADI i* (2/2)

# Datenpfade: *STORE i* (1/2)

# Datenpfade: *STORE i* (2/2)

# Datenpfade: *STOREIN*1 *i* (1/2)

# Datenpfade: *STOREIN*1 *i* (2/2)

# Datenpfade: *MOVE ACC IN1* (1/2)

# Datenpfade: *MOVE ACC IN*1 (2/2)

# Zusätzliche Befehle

- Man kann weitere Befehle ohne zusätzliche Hardware realisieren!
- Load- und Compute-Befehle mit beliebigem Zielregister $r \in \{PC, IN1, IN2, ACC\}$.
  - Kein $\langle PC \rangle := \langle PC \rangle + 1$, wenn $r = PC$.
- Befehlsformat: *LOAD r i*; *ADD r i*; etc.
- Befehlskodierung:

| | 31 30 | 29 28 | 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|---|
| Load | 0 1 | M | * | **D** | i |

| | 31 30 | 29 | 28 27 26 | 25 24 | 23 … 0 |
|---|---|---|---|---|---|
| Compute | 0 0 | MI | F | **D** | i |

# Zur Illustration: ReTI-Simulator „Neumi”

- Verfügbar in Ilias (Zusatzmaterial)
  - Simulator der ReTI-Maschine.
  - Anleitungen und zwei Beispielprogramme.
- Für die Ausführung wird Java benötigt.
  - Für Windows: `http://www.java.com/de/download/manual.jsp`
  - Für Linux: Nicht den GNU-, sondern den SUN-Compiler verwenden (Neumi funktioniert nicht mit GCJ).
  - Für Solaris und MacOS sollte die richtige Java-Version vorliegen.
- Syntax geringfügig gegenüber Vorlesung abgewandelt.
  - Kommas zwischen Operanden: „ADDI ACC, 1” statt „ADDI ACC 1”.
  - Jump-Befehle anders geschrieben: „JUMP ge, 2” statt „JUMP$_{\geq}$ 2”.
  - Details: Anleitungen auf der Webseite.

# Implementierung von ReTI (1/2)

- Im Buch von Keller/Paul wird ReTI (bzw. ReSa) mit sogenannten diskreten FAST-Bausteinen realisiert.
  - Register, Zähler, ALU, Treiber, Speicher: Bausteine aus der FAST-Bibliothek, teilweise mehrere Bausteine, um Bitbreite 32 zu erreichen.
  - Kontrollsignale werden durch sog. PALs realisiert
    - Programmable Array Logic (Bausteine von AMD).
    - Spezielle Beschreibungssprache PALASM.
    - PALs werden heute nur noch selten verwendet.

# Implementierung von ReTI (2/2)

- Im Gegensatz dazu verwenden wir State-of-the-Art-Bausteine der NanGate-Bibliothek (`http://www.si2.org/openeda.si2.org/projects/nangatelib`) im Hinblick auf eine VLSI–Implementierung auf einem einzigen Chip.
- Auf Basis einer solchen Implementierung behandeln wir später wesentliche Konzepte für eine „Timing-Analyse". Wir beginnen zunächst mit der Realisierung der Kontrollsignale (Output–Enable, Clock–Enable ...).
- Kontrollsignale werden durch einen Endlichen Automaten generiert. Wir skizzieren hier das Vorgehen.

# Zu generierende Kontrollsignale

- Clock–Enable–Signale für alle Register $r$, Bez.: *rcken*.
- Output enable Signale (active low) für alle Treiber $XYd$, Bez.: $/XYdoe$.
- Funktions-Select-Signale $f[2:0]$ zum Selektieren der Funktion, die von ALU ausgeführt wird.
- Signale $/PCclear$, $/PCload$ für $PC$.
- *sext* zur Berechnung der Füllbits bei 24-Bit-Immediate-Konstanten.
- Für den Speicher benötigen wir die Kontrollsignale (active low) $/SMDdoe$, $/SMw$.

# Idealisierte Timing-Diagramme

- Grobe Ablaufplanung mit idealisierten Timing-Diagrammen.
- Vereinfachende Annahme: Verzögerungszeit aller Bausteine = 0 (exakte Analyse mit Verzögerungszeiten später).
- Befehlsabarbeitung ist unterteilt in Takte (= Folge von Taktsignalen *high*, *low*).
- Fragen:
  - Wie sollen Kontrollsignale zusammenspielen?
  - In welchem Takt sollen welche Treiber aktiviert, welche Registerclock enabled werden?

# Befehlsabarbeitung in Takten

- Sowohl Fetch- als auch Execute-Phase bestehen aus 4 Takten gleicher Länge.
- Kontrollsignal:
  - $E = 0$: Fetch-Phase,
  - $E = 1$: Execute-Phase.
- Signale $s_0, s_1$ = Binärkodierung der Nummer des Taktes (innerhalb Fetch, Execute), in dem ReTI sich befindet.
- Steigende Flanken (Anfang des Taktes) werden mit $P_i$, fallende (Mitte des Taktes) mit $N_i$ bezeichnet ($i = 0, \ldots, 3$).
- Clock $ck$, Signale $s_0, s_1$ und $E$ werden Phasensignale genannt. Weitere (Kontroll-)Signale werden aus den Phasensignalen erzeugt.

# Erzeugung der Phasensignale

3-Bit-Zähler, zählt bei jeder positiven Flanke von ck um 1 hoch.

# Idealisiertes Timing-Diagramm

# Zur Erinnerung: Datenpfade

# Entwurfsziele

- Nutze Busse möglichst lange (unter Vermeidung von Bus Contention).
- Clock-Enable-Signale der Register möglichst spät
  $\rightarrow$ viel Zeit für Berechnung neuer Daten.
- Nach dem Entwurfsende muss das Timing der *CPU* mit den konkreten Werten der eingesetzten Fertigungstechnologie überprüft werden. (“Wie schnell kann man maximal takten, um korrektes Funktionieren zu garantieren?”)
  - Siehe nächstes Kapitel.

# Aufbau der Kontrolllogik (1/2)

- Die eigentliche Kontrollogik wird realisiert durch einen Endlichen Automaten.
- Die Kontrollsignale sind als Ausgangssignale des Endlichen Automaten implementiert.
- Ist ein Kontrollsignal *active low*, dann bezeichnen wir es z.B. mit $/x$. Das Ausgangssignal $/x$ ergibt sich dann durch Negation des Ausgangssignals $x$ eines entsprechenden FFs mit Eingangssignal $x_{pre}$.
- Ist ein Kontrollsignal *active high*, dann bezeichnen wir es z.B. mit $x$. Das Ausgangssignal $x$ entspricht dem Ausgangssignal eines FFs mit Eingangssignal $x_{pre}$.

# Aufbau der Kontrolllogik (2/2)

# Berechnung von *lcken* als erstes Beispiel (1/2)

# Berechnung von *lcken* als erstes Beispiel (2/2)

- $lcken_{pre}$ hat steigende Flanke bei $P1$, fallende bei $P2$ von Fetch.
- Realisierung: $lck_{pre} = \overline{E} \cdot \overline{s_1} \cdot s_0$.

# Berechnung von *ACCcken*, *IN1cken*, *IN2cken*, *PCcken*

- Analog, unter Berücksichtigung der Tatsache, dass Register nur bei bestimmten Befehlen neu beschrieben werden dürfen

# Berechnung von *ACCcken* als Beispiel (1/3)

# Berechnung von *ACCcken* als Beispiel (2/3)

- $ACCcken_{pre}$ hat steigende Flanke bei $P1$, fallende bei $P2$ von Execute.
- Aber nur bei folgenden Befehlen:
  - Compute mit $D = ACC$
  - Load mit $D = ACC$
  - Move mit $D = ACC$

- Compute: $\overline{I_{31}} \cdot \overline{I_{30}}$
- Load: $\overline{I_{31}} \cdot I_{30}$
- Move: $I_{31} \cdot \overline{I_{30}} \cdot I_{29} \cdot I_{28}$
- $D = ACC$: $I_{25} \cdot I_{24}$

Kodierung S, D

| **S, D** | **Register** |
|---|---|
| 0 0 | PC |
| 0 1 | IN1 |
| 1 0 | IN2 |
| 1 1 | ACC |

# Berechnung von *ACCcken* als Beispiel (3/3)

$$
\begin{aligned}
ACCcken_{pre} = \quad & E \cdot \overline{s_1} \cdot s_0 \cdot && \text{// P1 von execute} \\
& I_{25} \cdot I_{24} \cdot && \text{// } D = ACC \\
& \left( \overline{I_{31}} \cdot \overline{I_{30}} + \overline{I_{31}} \cdot I_{30} + I_{31} \cdot \overline{I_{30}} \cdot I_{29} \cdot I_{28} \right) && \text{// Compute, Load oder Move}
\end{aligned}
$$

# Datenpfade und Treiber auf *L* und *R*

# Idealisiertes Timingdiagramm

# Treiber auf Bussen $L$ und $R$

- $0Ld, PCLd, IN1Ld, IN2Ld, ACCLd, 0Rd, IRd, DRd$.
- Enabled in der ganzen Execute-Phase.
- Beispiel für Realisierung: $/PCLdoe$.
  - Enabled für
    - JUMP ($I[31:30] = 11$)
    - Compute-Befehle ($I[31:30] = 00$) mit $D = PC$ ($I[25:24] = 00$)
    - MOVE ($I[31:28] = 1011$) mit $S = PC$ ($I[27:26] = 00$).

# Berechnung von /*PCLdoe* (1/2)

- *PCLdoe*$_{pre}$ hat steigende Flanke bei *P*3 von Fetch.

# Berechnung von $/PCLdoe$ (2/2)

$$
\begin{aligned}
PCLdoe_{pre} = \quad & \Big(\overline{E} \cdot s_1 \cdot s_0 \cdot && \text{// P3 von fetch} \\
& \big[I_{31} \cdot I_{30} + && \text{// JUMP} \\
& \overline{I_{31}} \cdot \overline{I_{30}} \cdot \overline{I_{25}} \cdot \overline{I_{24}} && \text{// Compute mit } D = PC \\
& I_{31} \cdot \overline{I_{30}} \cdot I_{29} \cdot I_{28} \cdot \overline{I_{27}} \cdot \overline{I_{26}}\big]\Big) && \text{// MOVE mit } S = PC \\
& + PCLdoe \cdot E \cdot \overline{s_1} \cdot \overline{s_0} && \text{// Halten in Takt 0 von execute} \\
& + PCLdoe \cdot E \cdot \overline{s_1} \cdot s_0 && \text{// Halten in Takt 1 von execute} \\
& + PCLdoe \cdot E \cdot s_1 \cdot \overline{s_0} && \text{// Halten in Takt 2 von execute}
\end{aligned}
$$

- Output-Enable-Signale für andere Treiber auf $L$ und $R$ analog.

# Datenpfade und Treiber auf Adressbus

# Treiber auf Adressbus

- *PCAd*: enabled (unabhängig vom Befehl) bei *N*0 der Fetch-Phase, disabled bei *N*3 der Fetch-Phase.
- *IAd*, *ALUAd*: enabled bei *N*0, disabled bei *N*3 von Execute (aber nicht bei allen Befehlen).
- Die D-FFs zu Output-Enable-Signalen auf dem Adressbus werden mit der invertierten Clock getaktet (Verschiebung um einen halben Takt!)

# Idealisiertes Timingdiagramm

# Weitere Kontrollsignale

- Treiber auf *D*, *DI*
- Signale */PCload*, */PCclear*
- Speicheransteuerung: s. nächste Folie.
- Kontrolle der ALU und Sign–Extension:
  - Funktions-Select-Signale f[2:0] der ALU
  - Eingangsübertrag $c_{in}$
  - sext und fill
  - All diese Signale werden durch den kombinatorischen Schaltkreis (ohne zusätzliche FFs) berechnet.

# Idealisiertes Timing-Diagramm

# Speicheransteuerung

- Output-Enable */SMDdoe* für SMDd (Treiber am Speicherausgang) aktiviert von P1 bis P0 bei Leseoperationen, d.h. bei
  - Fetch
  - Compute Memory
  - LOAD, LOADINj
- Schreibsignal für Speicher */SMw* (memory write) aktiviert von P2 bis P3 von execute bei Schreiboperationen, d.h. bei
  - STORE, STOREINj

# Zusammenfassung Sequentielle Logik

- Sequentielle Schaltkreise bestehen aus speichernden Elementen (Latches, Flipflops) und einem kombinatorischen Kern.
- Sie implementieren endliche Zustandsautomaten.
- Der Entwurf eines sequentiellen Schaltkreises besteht aus der Aufstellung des Zustandsdiagramms, der Zustandsminimierung, der Zustandskodierung und der Synthese der kombinatorischen Logik.
- Nun war es uns möglich, den Entwurf von ReTI zu vervollständigen (exakte Timing-Analyse folgt).

# Kapitel 5

Timing:

1. **Physikalische Eigenschaften**
2. Timing wichtiger Komponenten
3. Exaktes Timing von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Wiederholung: Übergang beim RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit $t_{P/SQ}$ ist $Q = 1$. Nach Zeit $t_{P/S/Q}$ ist $/Q = 0$.
- „Gatter brauchen Zeit zum Schalten!" Aber wie lange ist $t_{P/SQ}$, $t_{P/S/Q}$? Oder wie lange muss ein Puls mindestens dauern? (= Pulsweite).

# Wiederholung: Timing-Diagramm D-LATCH

- Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?
- D. h. Wie lange ist Setup–Zeit $t_{SDW}$, Hold–Zeit $t_{HDW}$, Pulsweite $y$?

# Physikalische Signale ↔ Logische Signale

$$I(U) = \begin{cases} 0, & \text{falls } U \leq V_L \\ 1, & \text{falls } U \geq V_H \\ \text{undef.}, & \text{sonst} \end{cases}$$

- In jeder Technologie gibt es eine Versorgunsspannung $VCC$ (z.B. 1.1 $V$ bei NanGate).
- Eine Spannung $U \in [0, VCC]$ wird als logischer Wert $I(U)$ interpretiert.
  - Am Eingang (Input) eines Gatters: $V_{IL}$, $V_{IH}$.
  - Am Ausgang (Output) eines Gatters: $V_{OL}$, $V_{OH}$.
- $V_{IL}, V_{IH}, V_{OL}, V_{OH}$ eines Bausteins sind gegeben.

# Zusammenschalten von Gattern

- Will man den Ausgang eines Gatters $u$ mit dem Eingang eines Gatters $v$ verbinden, dann sollte gelten:
  - $V_{OL}(u) \leq V_{IL}(v)$ und
  - $V_{OH}(u) \geq V_{IH}(v)$.
- Sonst werden Signale falsch interpretiert.

# Beispiel: NanGate

$V_{IL} = 30\% \cdot VCC = 0.33\ V$
$V_{IH} = 70\% \cdot VCC = 0.77\ V$

Entsprechend Output-Pegel $V_{OL}$, $V_{OH}$.

# Verzögerung

# Beispiel-Spannungsverlauf $x(t)$, $y(t)$

$x(t)$

Zwei Beispiele für $y(t)$

# Allgemeine Bemerkung zu Verzögerungszeiten

- Im Allgemeinen gilt nicht $y(t) = x(t - t_p)$, so dass man nicht einfach $t_p$ als Verzögerungszeit definieren kann.
  $y(t)$ wird verformt.

- Die Verzögerungszeit (Propagation Delay) wird definiert als $t_p := (t_2 - t_1)$ bezüglich einer festen „Referenzspannung" $M$ mit $V_L < M < V_H$
  (Bsp.: $M = 0.5VCC = 0.55\ V$ bei NanGate).

- Bestimme $t_1$, $t_2$ mit $x(t_1) = y(t_2) = M$.

# Angaben zur Verzögerungszeit

- In der Regel gibt es verschiedene Verzögerungszeiten für Übergänge am Ausgang:
  - $t_{PLH}$: Verzögerungszeit bei $0 \rightarrow 1$.
  - $t_{PHL}$: Verzögerungszeit bei $1 \rightarrow 0$.

# Modellierung der Verzögerungszeit

- **Problem** bei der Modellierung der Verzögerungszeit bezüglich fester Spannung $M$:
  - Keine Aussage darüber, wann logische Signale 0 oder 1 sind, d. h. physikalische Signale unterhalb $V_{OL}$ oder oberhalb $V_{OH}$ sind.

# Illustration des Problems

→ Ähnliches Problem am Gattereingang.

# Anstiegs- und Abfallzeiten

- Für jedes Signal braucht man also zusätzliche Informationen über:
  - Anstiegszeit (Rise Time) =
    Zeit, in der Signal von $V_L$ nach $V_H$ steigt.
  - Abfallzeit (Fall Time) =
    Zeit, in der Signal von $V_H$ nach $V_L$ fällt.
  - Bzw. noch genauer würde man eigentlich benötigen:
    - Anstiegszeit von $M$ nach $V_H$
    - Abfallzeit von $M$ nach $V_L$

# Beschränkung dieser Zeiten

- Die in unseren Analysen verwendeten Gatter haben die folgende angenehme Eigenschaft:

- $\exists\delta$ mit folgender Eigenschaft:

  Falls rise/fall time $\leq \delta$ am Gattereinang,
  dann rise/fall time $\leq \delta$ am Gatterausgang.

# Beispiel: NanGate

- $V_{IL} = 30\% \cdot VCC = 0.33$ V
  $V_{IH} = 70\% \cdot VCC = 0.77$ V
- NanGate für $M = 0.55$ V spezifiziert.
  Bausteine $NAND, NOT, AND, OR, EXOR$.
- $t_p$ zwischen 0.00 ns und 0.21 ns.
- $\delta = 0.13$ ns (1 ns = $10^{-9}$ s)
- Die Zeiten, an denen die entsprechenden Signale wohldefinierte logische Werte 0, 1 annehmen, unterscheiden sich von denen für $M$ um höchstens $\delta$.

# Bemerkung

- Eine rise/fall time $\leq \delta$ an den primären Eingängen einer Schaltung kann man garantieren, wenn man den Schaltvorgang zur Zeit $t_0$ beginnt und spätestens zur Zeit $t_0 + \delta$ abschließt.

# Analyse der Verzögerungszeit einer Kette von $n$ Gattern (1/3)

# Vereinbarungen

- Im Folgenden soll

  Signal $X$ wird zum Zeitpunkt $t_1$ abgesenkt/angehoben

  bedeuten

  $X$ wird abgesenkt/angehoben mit $X(t_1) = M$.

- Desweiteren sind alle Zeitangaben in *ns*.

# Einfluss auf Verzögerungszeiten

- Verzögerungszeiten von Gattern sind nicht konstant, sondern werden beeinflusst durch:
  - Betriebstemperatur
  - Fertigungsprozess des Chips
  - kapazitive Last am Gatterausgang (Fanout) (Gattereingänge, die mit einem Gatterausgang verbunden sind, verhalten sich wie Kondensatoren, d. h. sie werden beim Schalten ge- bzw. entladen.)

# Worst-case Timing-Analyse

- Wegen Abhängigkeit der Verzögerungszeit von Temperatur, Fertigungsprozess und Fanout werden vom Hersteller keine festen Zeiten $t_{PLH}/t_{PHL}$ angegeben, sondern 3 Werte:
  - $t^{min}$ = untere Schranke
  - $t^{max}$ = obere Schranke
  - $t^{typ}$ = *typischer* Wert (???)

# *min*, *max* und *typ* (1/2)

- Für die tatsächliche Verzögerungszeit $t_p$ gilt:

$$t^{min} \leq t_p \leq t^{max}$$

- Wir nehmen in den folgenden Analysen an, dass $t_p$ im Intervall $[t^{min}, t^{max}]$ liegt, falls
  - die Temperatur im Bereich $T$ liegt („kommerzieller Temperaturbereich“ 0 – 70 °C, „militärischer Temperaturbereich“ −55 – 125 °C)
  - und eine bestimmte kapazitive Last $C_0$ nicht überschritten wird.

- $C_0$ wird so gewählt, dass mit Einhalten einer Fanoutbeschränkung von 10 $C_0$ auf keinen Fall überschritten wird.

# *min*, *max* und *typ* (2/2)

- Für $t^{typ}$ gilt ebenfalls $t^{min} \leq t^{typ} \leq t^{max}$.
- Beim Rechnen mit $t^{typ}$ macht man aber einen *Fehler mit unbekannter Größe*.

$\rightarrow$ Kein Rechnen mit $t^{typ}$, sondern mit Intervallen $[t^{min}, t^{max}]$.

# Exkurs: Rechnen mit Intervallarithmetik (1/2)

## Definition

Ein Intervall $[a,b] := \{x \in \mathbb{R} \mid a \leq x \leq b\} \subset \mathbb{R}$ auf $\mathbb{R}$ ist eine zusammenhängende und abgeschlossene Teilmenge von $\mathbb{R}$. Man bezeichnet es auch als das abgeschlossene Intervall von $a$ bis $b$.

- Wir betrachten hier nur die Menge der abgeschlossenen Intervalle IR auf $\mathbb{R}$.
- Es gilt:
  - $\min[a,b] = a$
  - $\max[a,b] = b$
  - $a \in \mathbb{R} \simeq [a,a] \in \mathrm{IR}$
    (eine reelle Zahl $a$ kann aufgefasst werden als das Punktintervall von $a$ bis $a$)

# Exkurs: Rechnen mit Intervallarithmetik (2/2)

## Definition

Gegeben ein Operator $_{op} \in \{+,-,\cdot\}$ in $\mathbb{R}$. Der dazugehörige Operator $\textcircled{op}$ auf IR ist definiert als:
Für $a,b,c,d \in \mathbb{R}$:
$[a,b] \textcircled{op} [c,d] := \{x \ _{op} \ y \mid x \in [a,b], y \in [c,d]\}$

Beispiele:

- $[a,b] \textcircled{+} [c,d] = [a+c, b+d]$
- $[a,b] \textcircled{-} [c,d] = [a-d, b-c]$
- $[a,b] \textcircled{\cdot} [c,d] = [\min(a\cdot c, a\cdot d, b\cdot c, b\cdot d), \max(a\cdot c, a\cdot d, b\cdot c, b\cdot d)]$

# Bemerkungen

- Wir schreiben vereinfachend nur $_{op}$ statt $\textcircled{op}$.
- Für unsere Belange sind ausschließlich $+$, $-$ und $\cdot$ Operator von Bedeutung. ($\cdot$ mit Punktinvervallen)
- Ein Intervall bezeichnen wir mit $\tau = [t^{min}, t^{max}]$.

# Beispiel: AND-Gatter

AND

$\tau_{PLH} = [0.02, 0.12]$

$\tau_{PHL} = [0.02, 0.12]$

Bzw.:

| AND | $t^{\min}$ | $t^{\max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.02 | 0.12 |
| $\tau_{PHL}$ | 0.02 | 0.12 |

# Fall 1

| AND | $t^{\min}$ | $t^{\max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.02 | 0.12 |
| $\tau_{PHL}$ | 0.02 | 0.12 |

- $A$, $E$ fest auf 1.
- $B$ von 0 auf 1 zum Zeitpunkt $t_0$.

→ Änderung von $C$ zur Zeit

$$\begin{aligned}\tau_1 &= t_0 + \tau_{PLH}\,(\text{AND})\\ &= t_0 + [0.02, 0.12]\end{aligned}$$

→ Änderung von $D$ zur Zeit

$$\begin{aligned}\tau_2 &= \tau_1 + \tau_{PLH}\,(\text{AND})\\ &= t_0 + 2 \cdot \tau_{PLH}\,(\text{AND})\\ &= t_0 + 2 \cdot [0.02, 0.12]\\ &= t_0 + [0.04, 0.24]\end{aligned}$$

# Fall 1 - Timing-Diagramm

# Fall 2

- $A, B, E$ können sich zum Zeitpunkt $t_0$ ändern, sind vorher und nachher stabil.
- Es ist unbekannt, wieviele Signale sich ändern und wie sie sich ändern.

→ Gröbere Abschätzungen

# Gröbere Abschätzung

- Bestimmung von Zeitintervallen, zu denen Gatter überhaupt schalten können.
- Beispiel:

$\tau_{PLH} = [t_{PLH}^{min}, t_{PLH}^{max}]$
$\tau_{PHL} = [t_{PHL}^{min}, t_{PHL}^{max}]$

$u$ ist beliebiges Gatter oder eine Schaltung.

- Gesucht ist ein Intervall $\tau_p$ in dem die Verzögerungsintervalle aller möglichen Schaltvorgänge enthalten sind.
- Definiere $t_p^{min} := min(t_{PLH}^{min}, t_{PHL}^{min})$
  $t_p^{max} := max(t_{PLH}^{max}, t_{PHL}^{max})$
- Dann ist $\tau_p := [t_p^{min}, t_p^{max}]$ das gesuchte Intervall.

# SMILE – Verzögerungsintervalle über mehrere Gatter

$$\tau_p(u) = [a_1, b_1]$$
$$\tau_p(v) = [a_2, b_2]$$
$$\tau_p(w) = [a_3, b_3]$$

# Fall2

| AND | $t^{\min}$ | $t^{\max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.02 | 0.12 |
| $\tau_{PHL}$ | 0.02 | 0.12 |

- Wenn die Gatter schalten, dann in folgenden Intervallen:
  - A, B, E: $t_0 + [0.0, 0.0]$
  - C: $t_0 + [0.02, 0.12]$
  - D: $t_0 + [0.0, 0.12] + [0.02, 0.12] = t_0 + [0.02, 0.24]$

# Fall 2 - Timing-Diagramm

# Interpretation des Timing-Diagramms

- Was kann im grauen Bereich passieren?
- **Beispiel**:
  $t_0$: $A, B, E$ $110 \rightarrow 101$
- **Annahme**:
  AND-Gatter haben folgende Verzögerungszeiten.
- 1. AND-Gatter: $t_{PLH} = 0.12$, $t_{PHL} = 0.12$
  2. AND-Gatter: $t_{PLH} = 0.02$, $t_{PHL} = 0.02$

# Timing-Diagramm zum Beispiel

1. AND: $t_{PLH} = 0.12$ $t_{PHL} = 0.12$
2. AND: $t_{PLH} = 0.02$ $t_{PHL} = 0.02$

# Spikefreies Umschalten von Gattern

- **Ziel**:
  Übergang von $A = 1, B = 0$ zu $A = 0, B = 1$, ohne Spike am Ausgang.

- **Bemerkung**:
  Der Übergang $(0,1) \rightarrow (1,0)$ bzw. umgekehrt ist der einzige, bei dem an *AND*/*NAND*-Gattern ein Spike auftreten kann.

# AND-Gatter

# Sicherer Abstand für Senken von $A$ und Anheben von $B$

## Lemma

Man kann zeigen, dass Übergänge für $A$ und $B$ mit

$$0.12\,\text{ns} + 2\delta = 0.38\,\text{ns}$$

sicher sind, d. h. keine Spikes am Ausgang entstehen können.

# Zum Beweis - Timing im Gatter

1. Senke $A$ bei $t_0 = 0$.
   - $\rightarrow$ $C = 0$ wegen $A = 0$ spätestens bei $t_1 = t_0 + 0.12 + \delta$
   - Grund:
     - Bei tatsächlichem Schalten von $C = 0$ wegen $A = 0$ würde das Signal spätestens nach $t_{PHL}^{max} = 0.12$ ns den Wert $M$ durchlaufen und wäre 0 spätestens nach $0.12 + \delta$ ns.
     - Interner Umschaltvorgang „$C = 0$ wegen $A = 0$" muss also spätestens nach $0.12 + \delta$ ns beendet sein.
2. Hebe $B$ (bzgl. $M$!) zum Zeitpunkt $t_2 = t_1 + \delta$.
   - $\rightarrow$ Zum Zeitpunkt $t_1$ gilt auf jeden Fall noch $B = 0$.

- Also:
  Vor $t_1$: $B = 0 \Rightarrow C = 0$
  Nach $t_1$: $A = 0 \Rightarrow C = 0$

$\rightarrow$ Übergänge für $A$ und $B$ mit Abstand $t_2 - t_0 = 0.12 + 2\delta = 0.38 \quad (\delta = 0.13)$.

| AND | $t^{min}$ | $t^{max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.02 | 0.12 |
| $\tau_{PHL}$ | 0.02 | 0.12 |

# Regel für spikefreies Umschalten

- Wähle den Abstand für die Signaländerungen am Eingang eines Gattes so, dass
  - die maximiale Verzögerung des ersten Schaltvorganges *und*
  - $2\times$ die rise-/fall-time ($\delta$)

  zwischen den beiden Schaltvorgängen am Eingang liegt.

- Beispiel: NAND

| NAND | $t^{\min}$ | $t^{\max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.02 | 0.15 |
| $\tau_{PHL}$ | 0.02 | 0.12 |

- Kritischer Übergang: Zuerst $A : 1 \rightarrow 0$, dann $B : 0 \rightarrow 1$.
- Daraus ergibt sich der Abstand $\max(\tau_{PLH}) + 2\delta = 0.41$

# Kapitel 5

Timing:

1. Physikalische Eigenschaften
2. **Timing wichtiger Komponenten**
3. Exaktes Timing von ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Timing - Übersicht

- Timing für ein paar (bereits bekannte) Schaltpläne:
  - RS-Flipflop
  - D-Latch
  - D-Flipflop
- Timing weiterer Komponenten, die bei der Realisierung der ReTI genutzt werden:
  - Kontrolllogik
  - Register mit Clock-Enable
  - ALU
  - Speicher

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).

- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).

- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.
- Nach Zeit $t_{P/S/Q}$ ist $/Q = 0$.

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.
- Nach Zeit $t_{P/S/Q}$ ist $/Q = 0$.

# RS-Flipflop

- Zustand $Q = 0 \rightarrow$ Zustand $Q = 1$:

- Senke $/S$ zur Zeit $t_0$ ab und hebe zu $t_0 + x$ wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit $t_{P/SQ}$ ist $Q = 1$.
- Nach Zeit $t_{P/S/Q}$ ist $/Q = 0$.
- Wähle $x$ so, dass kein Spike entsteht.

# Übergang - graphisch

| NAND | $t^{min}$ | $t^{max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.01 | 0.15 |
| $\tau_{PHL}$ | 0.01 | 0.12 |

# Spikefreier Übergang

- Nach den Regeln des spikefreien Umschaltens von Gattern entsteht kein Spike, falls:

$$(t_0 + x) - (t_0 + 0.27) \geq 0.41 \Leftrightarrow x \geq 0.68ns$$

- Wechsel von Zustand $Q = 1$ zu Zustand $Q = 0$ aus Symmetriegründen analog.

# Symbole und Bezeichnungen

| **Symbol** | **Bezeichnung** | $t^{\min}$ | $t^{\max}$ |
|---|---|---|---|
| $x$ | Pulsweite | 0.68 | |
| $\tau_{P/SQ}$ | Verzögerungszeit von $/S$ bis $Q$ | 0.01 | 0.15 |
| $\tau_{P/S/Q}$ | Verzögerungszeit von $/S$ bis $/Q$ | 0.02 | 0.27 |
| $\tau_{P/RQ}$ | Verzögerungszeit von $/R$ bis $Q$ | 0.02 | 0.27 |
| $\tau_{P/R/Q}$ | Verzögerungszeit von $/R$ bis $/Q$ | 0.01 | 0.15 |

# D-Latch

- $W$ ist *active high*.
  - $W = 0 \Rightarrow /S, /R$ inaktiv
  - $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv}, & \text{falls } D = 1 \\ /R \text{ aktiv}, & \text{falls } D = 0 \end{cases}$

- Wie beim RS-Flipflop (minimale Pulsweite!) muss man auch beim D-Latch bestimmte Forderungen an den zeitlichen Verlauf der Signale stellen, um Spikefreiheit zu garantieren.

# Timing-Diagramm

# Timing-Bedingungen für das D-Latch

- $W$ muss beim Schreiben lange genug 1 sein, um minimale Pulsweite $x$ des RS-FFs zu garantieren.
- Vor $W : 0 \rightarrow 1$ werden Daten für Zeit $t_{SDW}$ stabil gehalten, um Spikes auf $/S$, $/R$ zu vermeiden (der kritischste Fall ist das Verhindern von Spikes auf $/R$ bei Schreiben von 1).
- Nach $W : 1 \rightarrow 0$ werden Daten für Zeit $t_{HWD}$ stabil gehalten, um Spikes auf $/S$, $/R$ zu vermeiden (der kritischste Fall ist das Verhindern von Spikes auf $/S$ beim Schreiben von 0).

# Man rechnet nach:

Der Schreibvorgang beim D-Latch funktioniert mit den Parameterwerten aus der Tabelle (siehe Übung).

| **Symbol** | **Bezeichnung** | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $y$ | Pulsweite des Schreibimpulses | 0.79 | |
| $t_{SDW}$ | Setupzeit von $D$ bis $W$ | 0.49 | |
| $t_{HDW}$ | Holdzeit von $W$ nach $D$ | 0.41 | |
| $\tau_{PWQ}$ | Verzögerungszeit von $W$ bis $Q$ | 0.02 | 0.39 |
| $(\tau_{PDQ}$ | Verzögerungszeit von $D$ bis $Q$ | 0.02 | 0.54) |

# Mögliche Realisierung: D-Flipflop

# Timing: D-Flipflop

- Vorgehen analog zu RS-Flipflop und D-Latch.
- Wir verzichten daher auf weitere Details.
- Die NanGate-Bibliothek enthält bereits ein D-FF mit folgenden charakteristischen Zeiten (in *ns*):

| **Symbol** | **Bezeichnung** | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $t_{SDC}$ | Setupzeit von *D* bis *ck* | 0.08 | |
| $t_{HCD}$ | Holdzeit von *D* nach *ck* | 0.14 | |
| $\tau_{PCQ}$ | Verzögerungszeit von *ck* bis *Q* | 0.12 | 0.26 |

# Aufbau der Kontrolllogik, zur Erinnerung

- Generierung der Kontrollsignale (OE von Treibern, ALU-Ansteuerung, ...).
- Ist ein Kontrollsignal *active low*, dann bezeichnen wir es z.B. mit $/x$. Das Ausgangssignal $/x$ ergibt sich dann durch Negation des Ausgangssignals $x$ eines entsprechenden FFs mit Eingangssignal $x_{pre}$.
- Ist ein Kontrollsignal *active high*, dann bezeichnen wir es z.B. mit $x$. Das Ausgangssignal $x$ entspricht dem Ausgangssignal eines FFs mit Eingangssignal $x_{pre}$.

# Kontrolllogik

| INV | $t^{min}$ | $t^{max}$ |
|---|---|---|
| $\tau_{PLH}$ | 0.01 | 0.15 |
| $\tau_{PHL}$ | 0.00 | 0.08 |

- Die Dauer eines Taktes bezeichnen wir als Zykluszeit $t_c$.
- Active-High-Ausgangssignale der Kontrolllogik, bei denen das FF mit $ck$ gesteuert ist, sind gegenüber der steigenden Flanke von $ck$ um Zeit $\tau^+_{p,ah}$ verzögert (resultiert aus D-FF-Verzögerung).
- Active-Low-Ausgangssignale der Kontrolllogik, bei denen das FF mit $ck$ gesteuert ist, sind gegenüber der steigenden Flanke von $ck$ um Zeit $\tau^+_{p,al}$ verzögert (resultiert aus D-FF-Verzögerung + Inverterverzögerung).
- Active-High-Ausgangssignale der Kontrolllogik, bei denen das FF mit $/ck$ gesteuert ist, sind gegenüber der letzten steigenden Flanke von $ck$ um Zeit $\tau^-_{p,ah} = \tau^+_{p,ah} + t_c/2 + \tau_{PLH,Inv}$ verzögert.
- Active-Low-Ausgangssignale der Kontrolllogik, bei denen das FF mit $/ck$ gesteuert ist, sind gegenüber der letzten steigenden Flanke von $ck$ um Zeit $\tau^-_{p,al} = \tau^+_{p,al} + t_c/2 + \tau_{PLH,Inv}$ verzögert.

# Timing: Kontrolllogik

- Mit geeigneter Implementierung des kombinatorischen Teiles erhält man folgende charakteristische Zeiten.

| **Symbol** | **Bezeichnung** | $t^{\min}$ | $t^{\max}$ |
|---|---|---|---|
| $\tau^{+}_{p,ah}$ | Verzögerungszeit *ck* bis *Q*, active high | 0.12 | 0.26 |
| $\tau^{+}_{p,al}$ | Verzögerungszeit *ck* bis *Q*, active low | 0.12 | 0.41 |
| $\tau^{-}_{p,ah}$ | Verzögerungszeit *ck* bis *Q* (von $/ck$ angesteuert, active high) | $t_c/2+0.13$ | $t_c/2+0.41$ |
| $\tau^{-}_{p,al}$ | Verzögerungszeit *ck* bis *Q* (von $/ck$ angesteuert, active high) | $t_c/2+0.13$ | $t_c/2+0.56$ |
| $t^{+}_{SDC}$ | Setupzeit von *D* bis *ck* | 0.88 | |
| $t^{-}_{SDC}$ | Setupzeit von *D* bis $/ck$ | 0.88 | |
| $t^{+}_{HCD}$ | Holdzeit von *D* nach *ck* | 0.06 | |
| $t^{-}_{HCD}$ | Holdzeit von *D* nach $/ck$ | 0.06 | |

# Register mit Clock-Enable

- Bei der Implementierung benötigen wir noch einen Treiberbaum der Tiefe 2, um *Regcken* auf 32 1–Bit–Multiplexer zu verteilen.

| Symbol | Bezeichnung | $t^{min}$ |
|---|---|---|
| $t_{SDC}$ | Setup-Zeit von *D* vor *ck* | 0.23 |
| $t_{HDC}$ | Hold-Zeit von *D* nach *ck* | 0.11 |
| $t_{SEC}$ | Setup-Zeit von *Regcken* vor *ck* | 0.46 |
| $t_{HEC}$ | Hold-Zeit von *Regcken* nach *ck* | 0.08 |

- $t_{SDC}$ ergibt sich aus Setupzeit D-FF + maximale Verzögerungszeit Multiplexer (Daten bis Ausgang) (0.08 + 0.15).
- $t_{HDC}$ ergibt sich aus Holdzeit D-FF - minimale Verzögerungszeit Multiplexer (Daten bis Ausgang) (0.14 - 0.03).
- $t_{SEC}$ ergibt sich aus Setupzeit D-FF + maximale Verzögerungszeit Multiplexer (Select bis Ausgang) + 2 x maximale Verzögerungszeit Treiber (0.08 + 0.16 + 2 x 0.11).
- $t_{HEC}$ ergibt sich aus Holdzeit D-FF - minimale Verzögerungszeit Multiplexer (Select bis Ausgang) - 2 x minimale Verzögerungszeit Treiber (0.14 - 0.02 - 2 x 0.02).

# Schaltrealisierung der ALU

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Timing: ALU

- Annahme: ALU mit 32-Bit-Addierer (Conditional Sum).
- Man zeigt:
  - Längster Pfad über ALU läuft durch den Addierer.
  - Annahme:
    - Die Funktion–Select–Bits sind mindestens $t_{select}$ = 0.28 ns vor den Operanden gültig.
    - Dann ist garantiert, dass der kritische Pfad nicht durch die select–Eingänge bestimmt wird.
  - Zeitverhalten der ALU:

| **Symbol** | **Bezeichnung** | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $t_{select}$ | | 0.28 | |
| $t_{ALU}$ | Verzögerungszeit von $a$, $b$ bzw. $c_{in}$ bis Ausgang | | 3.25 |

# SRAM

SRAM
A
n
$D_{out}$
W
$D_{in}$
W
$D_{in}$
$F_N$
$F_N$
N
N
W'
D'
D-Latch
D
Q
$Q_i$
$W_i$
W
$Y_0$
$Y_i$
$Y_{N-1}$
A
n
$D_n$
$G_0$
$G_i$
$G_{N-1}$
$O_N$
$D_{out}$

# Timing: SRAM

- Auch hier wäre das Vorgehen analog zu den bereits vorgestellten Analysen möglich. Zuvor muss man sich noch Gedanken machen um das Timing von:
  - Dekodierer
  - Treiberbäume
  - OR-Baum
- Eine detaillierte Timinganalyse ist aufwändig.
- Für die folgenden Timinganalysen orientieren wir uns an dem kommerziell angebotenen SRAM CY7C1079DV33 der Firma Cypress Semiconductor (siehe folgende Folien).

# Interface zu CY7C1079DV33

# Timing: CY7C1079DV33

- Aus dem Datenblatt entnimmt man:

| Symbol | Bezeichnung | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $t_{acc}$ | Lesezugriffszeit | | 12.0 |
| $t_{OED}$ | Zeit von $/SMDdoe = 0$ bis $D$ | | 7.0 |
| $t_{OEZ}$ | Zeit von $/SMDdoe = 1$ bis high-$Z$ | | 7.0 |
| $t_{wc}$ | Schreibzykluszeit | 12.0 | |
| $t_{SAW}$ | Setupzeit von $A$ bis $W$ | 0.0 | |
| $t_{SAEW}$ | Setupzeit von $A$ bis Ende $W$ | 9.0 | |
| $t_{HWA}$ | Holdzeit von $A$ nach $W$ | 0.0 | |
| $w$ | Schreibpulsweite | 9.0 | |
| $t_{SDEW}$ | Setupzeit von $D$ bis Ende $W$ | 7.0 | |
| $t_{HWD}$ | Holdzeit von $D$ nach $W$ | 0.0 | |

# Kapitel 5

Timing:
1. Physikalische Eigenschaften
2. Timing wichtiger Komponenten
3. **Exaktes Timing von ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Es gilt:

- Bei hinreichend langsamem Takt funktioniert der Rechner.

**Frage:**

- Wie schnell kann man den Rechner takten?
  Wie lange muss ein Takt mind. sein?

→ Ersetze idealisiertes Timing durch exakte Timinganalyse

→ Gesucht:
Untere Grenze für Zykluszeit $t_c$

# Schritte der Analyse

1. Einhalten von Setup- und Hold-Zeiten der Kontrolllogik
2. Vermeidung von Bus-Contention
3. PC-Inkrementierung
4. Compute-Befehle:
   OE: Compute Memory
5. *Fetch*, *Load*, *Store*, *Jump*

- Wir werden uns auf Compute-Befehle beschränken.

# Timing der Kontrolllogik (1/3)

| Symbol | Bezeichnung | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $\tau^{+}_{p,ah}$ | Verzögerungszeit *ck* bis *Q*, active high | 0.12 | 0.26 |
| $\tau^{+}_{p,al}$ | Verzögerungszeit *ck* bis *Q*, active low | 0.12 | 0.41 |
| $\tau^{-}_{p,ah}$ | Verzögerungszeit *ck* bis *Q* (von $/ck$ angesteuert, active high) | $t_c/2+0.13$ | $t_c/2+0.41$ |
| $\tau^{-}_{p,al}$ | Verzögerungszeit *ck* bis *Q* (von $/ck$ angesteuert, active high) | $t_c/2+0.13$ | $t_c/2+0.56$ |
| $t^{+}_{SDC}$ | Setupzeit von *D* bis *ck* | 0.88 | |
| $t^{-}_{SDC}$ | Setupzeit von *D* bis $/ck$ | 0.88 | |
| $t^{+}_{HCD}$ | Holdzeit von *D* nach *ck* | 0.06 | |
| $t^{-}_{HCD}$ | Holdzeit von *D* nach $/ck$ | 0.06 | |

- Setupzeit der Dateneingänge bis *ck* ist $t^{+}_{SDC} \geq 0.88$, Setupzeit der Dateneingänge bis $/ck$ ist $t^{-}_{SDC} \geq 0.88$.
- Dateneingänge $(s_0, s_1, E, I, ACC, reset)$ müssen rechtzeitig bereit sein.
- Alle Dateneingänge sind Ausgangssignale von FFs, die mit *ck* getaktet werden.

# Timing der Kontrolllogik (2/3)

- Wähle eine beliebige steigende Taktflanke *Pi* als zeitlichen Bezugspunkt.
- Die Dateneingänge sind also bereit zur Zeit $\tau_{PCQ} = [0.12, 0.26]$ (Verzögerung eines D-FF).
- Die nächste steigende Taktflanke von $/ck$ ist bei $\frac{t_c}{2} + [0.01, 0.15]$, die nächste steigende Taktflanke von *ck* bei $t_c$.

$$\Rightarrow \frac{t_c}{2} + \underbrace{0.01}_{\text{Inverterverzögerung}} \geq \underbrace{0.88}_{t^-_{SDC}} + \underbrace{0.26}_{\tau_{PCQ}\text{ für FFs}}, \text{ d.h. } t_c \geq 2.26.$$

$$\Rightarrow t_c \geq \underbrace{0.88}_{t^+_{SDC}} + \underbrace{0.26}_{\tau_{PCQ}\text{ für FFs}} = 1.14$$

# Constraints

- $t_c \geq 2.26$

# Compute-Befehle

- Am zeitkritischsten ist Compute memory!

# $[r] := [r] + [M(\langle i \rangle)]$; hier: $[PC] := [PC] + [M(\langle i \rangle)]$

# Analyse allgemein

- Beginn der Analyse bei *P3* von Fetch als zeitlicher Bezugspunkt.
- Bei *P3* von Fetch wird der Befehl ins Instruktionsregister übernommen.

# I-Ausgänge (1/2)

I-Ausgänge gültig bei

$$\tau_1 = \underbrace{[0.12, 0.26]}_{\tau_{PCQ} \text{ von Register } I}$$

| D-FF | Bezeichnung | $t^{min}$ | $t^{max}$ |
|---|---|---|---|
| $t_{SDC}$ | Setupzeit von *D* bis *ck* | 0.08 | |
| $t_{HCD}$ | Holdzeit von *D* nach *ck* | 0.14 | |
| $\tau_{PCQ}$ | Verzögerungszeit von *ck* bis *Q* | 0.12 | 0.26 |
| $\tau_{PDQ}$ | Verzögerungszeit von *D* bis *Q* | 0.10 | 0.21 |

- $0^8 I_{23} \ldots I_0$ wird über Treiber *IAd* auf Adressbus gegeben.
- $0^8$ ist eine Konstante und steht daher ebenfalls zu $\tau_1$ bereit.

# Constraints

- $\tau_1 = [0.12, 0.26]$

- $t_c \geq 2.26$

# I-Ausgänge (2/2)

*IAd* enabled bei *N*0 von Execute, d.h. /*IAdoe* aktiv zur Zeit

$$\begin{aligned}\tau_2 &= t_c + \tau^{-}_{p,al} \\ &= t_c + \frac{t_c}{2} + [0.13, 0.56] \\ &= \frac{3}{2} t_c + [0.13, 0.56]\end{aligned}$$

*I* schon gültig vor Aktivierung von *IAd* bei Punkt 2, falls

$$\begin{aligned}\max(\tau_1) &\leq \min(\tau_2) \Leftrightarrow \\ 0.26 &\leq \frac{3}{2} t_c + 0.13 \Leftrightarrow \\ \frac{3}{2} t_c &\geq 0.13 \Leftarrow \\ t_c &\geq 0.09\end{aligned}$$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$

# Gültiges $A$ (1/2)

$\rightarrow A$ gültig zur Zeit

$$\tau_3 = \tau_2 + \underbrace{[0.03, 0.11]}_{\text{Enable Zeit Treiber}}$$

$$= \frac{3}{2} t_c + [0.16, 0.67]$$

| | Tristate-Treiber | min | max |
|---|---|---|---|
| $\tau_{PZL}$ | Enable-Zeiten | 0.03 | 0.10 |
| $\tau_{PZH}$ | Enable-Zeiten | 0.03 | 0.11 |
| $\tau_{PLZ}$ | Disable-Zeiten | 0.03 | 0.11 |
| $\tau_{PHZ}$ | Disable-Zeiten | 0.03 | 0.10 |
| $\tau_{PLH}$ | Umschaltverzögerung bei $/OE = 0$ | 0.02 | 0.07 |
| $\tau_{PHL}$ | Umschaltverzögerung bei $/OE = 0$ | 0.03 | 0.10 |

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2} t_c + [0.16, 0.67]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$

# Gültiges $A$ (2/2)

- *ASMd* immer enabled

→ nur Treiber-Verzögerung berücksichtigt

→ *A* an *SM* bei

  - $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.18, 0.77]$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2} t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.18, 0.77]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$

# Daten am Speicherausgang (1/3)

- Lesezugriffszeit von *SRAM*: $[0.0, 12.0]$ (siehe Daten von CY7C1079DV33)

→ Gültige Daten am Speicherausgang bei

$$
\begin{aligned}
\tau_5 &= \tau_4 + [0.0, 12.0] \\
&= \frac{3}{2} t_c + [0.18, 0.77] + [0.0, 12.0] \\
&= \frac{3}{2} t_c + [0.18, 12.77]
\end{aligned}
$$

- Das ist aber nur korrekt, wenn der Ausgangstreiber durch $/$*SMDdoe* rechtzeitig enabled ist!

# Daten am Speicherausgang (2/3)

| *SRAM* CY7C1079DV33 | | | |
|---|---|---|---|
| **Symbol** | **Bezeichnung** | $t^{min}$ | $t^{max}$ |
| $t_{acc}$ | Lesezugriffszeit | | 12.0 |
| $t_{OED}$ | Zeit von $/SMDdoe = 0$ bis $D$ | | 7.0 |
| ... | | | |

- $/SMDdoe$ aktiviert zur Zeit $\tau' = 2 \cdot t_c + \tau^+_{p,al} = 2 \cdot t_c + [0.12, 0.41]$.
- Daten am Speicherausgang aufgrund Treiber-Enable gültig zur Zeit $\tau'' = \tau' + [0.0, 7.0] = 2 \cdot t_c + [0.12, 7.41]$.

$\Rightarrow$ Daten am Speicherausgang gültig spätestens zur Zeit $t''' = \max(\max(\tau_5), \max(\tau''))$.

# Daten am Speicherausgang (3/3)

- Daten am Speicherausgang gültig zur Zeit $t''' = \max(\max(\tau_5), \max(\tau''))$.
- Bedingung für $\max(\tau_5) \geq \max(\tau'')$:

$$\begin{aligned} \frac{3}{2}t_c + 12.77 &\geq 2 \cdot t_c + 7.41 \Leftrightarrow \\ \frac{1}{2}t_c &\leq 5.36 \Leftrightarrow \\ t_c &\leq 10.72 \end{aligned}$$

- Wir nehmen ab jetzt an, dass die Taktperiode $t_c \leq 10.72$ ist und rechnen mit $\max(\tau_5)$ weiter.
- (Es gilt auf jeden Fall $\min(\tau_5)$ $(= \frac{3}{2}t_c + 0.18) < \min(\tau'')$ $(= 2 \cdot t_c + 0.12)$
- Sollte sich später ergeben, dass die minimale Taktperiode $t_c > 10.72$, dann müssten wir die Rechnung nochmals korrigieren.

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2} t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2} t_c + [0.18, 12.77]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Daten auf *R*

*DRd* enabled bei *P*0 von Execute, also einen Takt vor Ausgangstreiber von *SM*
→ Enable nicht kritisch
→ Daten auf *R* spätestens bei

$$\tau_6 = \tau_5 + [0.02, 0.10] \text{ (Treiber-Verzögerung)}$$
$$= \frac{3}{2} t_c + [0.20, 12.87]$$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2} t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2} t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2} t_c + [0.20, 12.87]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Daten auf $L$

- Registerausgänge $r \in \{PC, ACC, IN1, IN2\}$ schon seit letzter Execute-Phase gültig.

$\rightarrow$ nicht kritisch

- Treiber *rLd* enabled bei $P0$ von Execute, d.h. wie auch bei *DRd* ist Zeit zum Enablen unkritisch im Vergleich zu $t_6$.

# $f[2:0], c_{in}$

- $f[2:0], c_{in}$ werden durch den kombinatorischen Schaltkreis der Kontrolllogik aus $l_{31}, \ldots, l_{24}$ berechnet.
- $l$-Ausgänge aber schon gültig bei $\tau_1 = [0.12, 0.26]$.
- Verzögerungszeit des kombinatorischen Schaltkreises $< t^+_{SDC} = 0.88$
- $f[2:0], c_{in}$ gültig spätestens bei $t_7 = 0.26 + 0.88 = 1.14$.

$\rightarrow$ völlig unkritisch verglichen mit $\max(\tau_6) = \frac{3}{2}t_c + 12.87$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2} t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2} t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2} t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2} t_c + [0.20, 12.87]$
- $t_7 = 1.14$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Voraussetzungen für die exakte Timinganalyse von *Compute memory*

- *ALU*
- Analyse der ALU (32-Bit mit Conditional Sum) unter folgender Annahme:
  - Funktionsselect-Signale liegen 0.28 ns vor den Daten an (unkritisch, da $t_7 + 0.28 = 1.14 + 0.28 = 1.42 < \min(\tau_6) = \frac{3}{2}t_c + 0.20$).
  - Resultatsausgänge gültig 3.25 ns nachdem die Daten anliegen.

| **Symbol** | **Bezeichnung** | $t^{\min}$ | $t^{\max}$ |
|---|---|---|---|
| $t_{select}$ | | 0.28 | |
| $t_{ALU}$ | Verzögerungszeit von $a$, $b$ bzw. $c_{in}$ bis Ausgang | | 3.25 |

# ALU-Ausgänge

- Spätestens gültig bei

$$t_8 = \max(\tau_6) + \underbrace{3.25}_{\text{Delay ALU}} = \frac{3}{2}t_c + 12.87 + 3.25$$

$$= \frac{3}{2}t_c + 16.12$$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# DI

- $/ALUDIdoe$ wird wie $/SMDdoe$ aktiviert bei $P1$ von Execute
- Daten kommen an *ALUDId* später an als an als Daten am internen *SRAM*–Treiber
- Enable–Zeit von *ALUDId* jedoch kürzer als bei *SRAM*
- Mit $t_c \leq 10.72$ ist auf jeden Fall auch für *ALUDId* gewährleistet, dass Treiber enabled, wenn Daten kommen.

→ Berücksichtige nur Treiberverzögerung.

→ Gültig spätestens bei

$$
\begin{aligned}
t_9 &= t_8 + 0.10 \\
&= \frac{3}{2} t_c + 16.12 + 0.10 \\
&= \frac{3}{2} t_c + 16.22
\end{aligned}
$$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Datenübernahme in Register $r$

- Clocksignale bei $P3$ von Execute
  - $\rightarrow$ steigende Flanke bei $\tau_{10} = 4t_c$
- Minimale Taktperiode wird aus Setup-Zeit von $r$ berechnet (Setup-Zeit am größten, wenn $r = PC$)

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$
- $\tau_{10} = 4 \cdot t_c$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Timing: Zähler

- Aus einer Analyse des Zählers in einer Implementierung gemäß Kapitel 4.1 (aber mit zusätzlichem Clock–Enable für das Register!) ergeben sich folgende Zeiten:

| **Symbol** | **Bezeichnung** | $t^{\min}$ | $t^{\max}$ |
|---|---|---|---|
| $t_{SDC}$ | Setup-Zeit von *D* vor *ck* | 0.53 | |
| $t_{HDC}$ | Hold-Zeit von *D* nach *ck* | 0.05 | |
| $t_{SLC}$ | Setup-Zeit von $/L$ vor *ck* | 0.76 | |
| $t_{HLC}$ | Hold-Zeit von $/L$ nach *ck* | 0.02 | |
| $t_{SEC}$ | Setup-Zeit von $/PCcken$ vor *ck* | 0.46 | |
| $t_{HEC}$ | Hold-Zeit von $/PCcken$ nach *ck* | 0.08 | |
| ... | ... | ... | ... |

# Setup-Zeit von Zähler

- Setup-Zeit: $t_{SDC} = 0.53\ ns$ (siehe Aufbau Zähler)

→ Bedingung:

- $t_9 + 0.53 \leq \min(\tau_{10})$
  $\Leftrightarrow \frac{3}{2}t_c + 16.75 \leq 4t_c$
  $\Leftrightarrow \frac{5}{2}t_c \geq 16.75$
  $\Leftrightarrow t_c \geq 6.70$

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau^{-}_{p,al} = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$
- $\tau_{10} = 4 \cdot t_c$

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$
- $t_c \geq 6.70$

# Es bleiben zu beachten:

- Hold-Zeit $t_{HCD}$
- Maximal bei $r \in \{ACC, IN1, IN2\}$, $t_{HCD} = 0.11$
  - Unproblematisch, da alle Treiber noch mindestens $\frac{1}{2}$ Takt nach *rck* enabled sind.

# Setup- und Hold-Zeiten von *rcken*

- *rcken* aktiv bei $P2$ von Execute, inaktiv bei $P3$ von Execute, d.h. aktiv von:
  - $\tau_{11} = 3t_c + \tau^+_{p,ah} = 3t_c + [0.12, 0.26]$ bis
  - $\tau_{12} = 4t_c + \tau^+_{p,ah} = 4t_c + [0.12, 0.26]$
- Setup-Zeit:
  - Für alle $r \in \{PC, ACC, IN1, IN2\}$: $t_{SEC} = 0.46$

  $\Rightarrow$ $\max(\tau_{11}) + 0.46 \leq 4t_c$, d.h. $3t_c + 0.26 + 0.46 \leq 4t_c$ bzw.

  $\Rightarrow$ $t_c \geq 0.72$ (unkritisch im Vergleich zu bisherigen Constraints)
- Hold-Zeit:
  - Für alle $r \in \{PC, ACC, IN1, IN2\}$: $t_{HEC} = 0.08$

  $\Rightarrow$ $\min(\tau_{12}) \geq 4t_c + 0.08$, d.h. $0.12 \leq 0.08$
- (Analog auch für alle anderen Takte.)

# Setup- und Hold-Zeiten $/PCload$ beim Zähler

- Setup $/L$ bis $ck$: $t_{SLC} = 0.76$
- Hold $/L$ nach $ck$: $t_{HLC} = 0.02$
- $/PCload$ (benötigt, wenn **neue** Werte in Zähler kommen) aktiv bei $P2$ von Execute, inaktiv bei $P0$ von Fetch, d.h. aktiv von:
  - $\tau_{13} = 3t_c + \tau^{+}_{p,al} = 3t_c + [0.12, 0.41]$ bis
  - $\tau_{14} = 5t_c + \tau^{+}_{p,al} = 5t_c + [0.12, 0.41]$

(* r = PC, IN1, IN2, ACC)

# Bedingungen für Setup- und Hold-Zeiten /*PCload* beim Zähler

- Setup:

  $\max(\tau_{13}) + 0.76 \leq \min(\tau_{10}) \Leftrightarrow$
  $3t_c + 0.41 + 0.76 \leq 4t_c \Leftrightarrow$
  $t_c \geq 1.18$

- Hold:

  $\min(\tau_{14}) \geq \max(\tau_{10} + 0.02) \Leftrightarrow$
  $5t_c + 0.12 \geq 4t_c + 0.02 \Leftrightarrow$
  $t_c \geq -0.10$

$\rightarrow$ Beide unkritisch im Vergleich zu bisherigen Constraints.

# Fazit: Zykluszeit und Befehlsrate

- Vorläufiges Ergebnis: Falls sich durch andere Befehle keine schärferen Bedingungen an die Zykluszeit ergeben, dann lautet sie:
  - $t_c \geq 6.70\ ns$
- Taktfrequenz:
  - $v = \frac{1}{6.70} \cdot 10^9\ Hz = 149.2\ MHz$
- 8 Takte pro Befehl $\rightarrow$ 18.6 Millionen Befehle pro Sekunde, d.h. Befehlsrate von 18.6 *MIPS* (= Million Instructions per Second)

# Anmerkungen zur Timing–Analyse

- Eine „echte” Timing–Analyse müsste noch Leitungslaufzeiten auf dem Chip berücksichtigen.
- Dazu muss dann aber schon das Layout des Chips bekannt sein, um die Leitungslängen und -kapazitäten zu berechnen.
- Leitungslaufzeiten waren früher bei einem Aufbau mit diskreten Bausteinen vernachlässigbar, sind es bei den heutigen Technologien aber nicht mehr.

$\Rightarrow$ Exakte Timing-Analysen sind heute daher kaum ohne maschinelle Unterstützung durchführbar.

- Synthesetools sind in der Lage, durch Optimierung der Treiberstärken von Grundgattern (verschiedene Versionen in der Bibliothek!) Laufzeiten zu minimieren.
- Wird das SRAM nicht auf dem Chip integriert (d.h. stattdessen ein kommerzielles externes SRAM angeschlossen), dann muss man noch Verzögerungszeiten für I/O–Pads des Chips mit eventueller Anpassung von Spannungspegeln berücksichtigen.

# Ausblick 1/2

- Beschleunigung
  - Schnellere Komponenten, z.B. ALU (siehe Übung)
  - Schnellere Adressberechnung (Treiber schon bei P0 öffnen)
  - Verkürzung des Fetch-Zyklus um 1 Takt
  - Pipelining
  - Schnellerer Speicher
  - ...

# Ausblick (2/2)

- Fehlertoleranz (Kapitel 6)
  - Wie umgehen mit Übertragungsfehlern auf den Leitungen?
  - Parity-Check, Hamming-Code
  - Andere Fehler ...
- Verifikation (Kapitel 7)
  - Ist der Entwurf der ReTI überhaupt korrekt?
  - Automatische Methoden für den Beweis der Korrektheit
- Architekturkonzepte (Kapitel 8)
  - Speicherhierarchie
  - Pipelining
  - Parallelität

Kapitel 6

# Fehlertoleranz

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Motivation

- Bisher: Rechnerarchitektur am Beispiel von ReTI
- Fehler auf verschiedenen Ebenen entdecken und beheben
  - → Fehler durch Fertigungsdefekte, Störungen: Kapitel 6
  - → Konzeptuelle Fehler, Entwurfsfehler: Kapitel 7
- Allgemeine Rechnerarchitektur und Entwurfskonzepte

# Physikalische Fehler

- Bei der Informationsverarbeitung und -Übermittlung können physikalische Störungen auftreten.
  - Elektrisches Rauschen, Radiation, Defekte.
  - Absichtliche Manipulation durch Angreifer.
- Die Störungen äußern sich darin, dass Wert 0 statt Wert 1 berechnet/übertragen wird und umgekehrt.
  - „Kippen" des Werts $0 \rightarrow 1$, $1 \rightarrow 0$.
- Hier konzentrieren wir aus auf Fehler bei der Datenübertragung.
  - Manche Fehler betreffen direkt die Hardware oder andere Systemteile. Ihre Behandlung ist komplexer.

# Wiederholung

Sei $A = \{a_1, \ldots, a_m\}$ ein endliches Alphabet der Größe $m$.

- Eine Abbildung $c : A \to \{0,1\}^n$ heißt Code fester Länge, falls $c$ injektiv ist.
- Die Menge $c(A) := \{w \in \{0,1\}^n \mid \exists a \in A : c(a) = w\}$ heißt Menge der Codewörter.
- Minimale Codelänge: Für einen Code $c : A \to \{0,1\}^n$ fester Länge gilt: $n \geq \lceil \log_2 m \rceil$.

# Fehler bei Datenübertragung (1/2)

**Annahme**:

- Sei $c$ ein Code minimaler fester Länge $n$.
- Ein Datum $a$ (z.B. ein Buchstabe, eine Zahl) wird, durch ein Codewort $w = c(a)$ repräsentiert übertragen.
- Sei $\tilde{w} \in \{0,1\}^n$ das empfangene Wort.
- Bei der Übertragung (z.B. über Internet) können einzelne Bits von $c(a)$ kippen. Dann ist $w \neq \tilde{w}$.

# Fehler bei Datenübertragung (2/2)

**Ziel**:

- Durch Verändern des Codes $c$ in einen Code $C$ fester Länge $n+r$ sollen diese Bits
  - erkannt

    $\rightarrow$ Fehlererkennende Codes
    Bsp. für 1-fehlererkennenden Code: Parity-Code
  - korrigiert

    $\rightarrow$ Fehlerkorrigierende Codes
    Bsp. für 1-fehlerkorrigierenden Code: Hamming-Code

  werden.

# Fehlererkennende Codes

**Idee:**

- Wähle Codewörter $w \in c(A)$ so, dass nach Kippen von Bits Wörter $\tilde{w} \in \{0,1\}^n$ entstehen, die keine Codewörter sind, d.h. $\tilde{w} \notin c(A)$.
- Wird ein Nicht-Codewort empfangen, so muss ein Übertragungsfehler aufgetreten sein.
- Benutze Codes mit $n = \lceil log_2 m \rceil + r$ mit $r > 0$.
- Benutze die $r$ zusätzlichen Bits zum Test auf Übertragungsfehler.
- Beispiel: Parity-Code

# Parity-Code

- Eine Bitfolge $w \in \{0,1\}^n$ besteht den Paritätstest (engl. Parity-Check), wenn die Anzahl der auf $1$ gesetzten Bitstellen gerade ist.

- Sei $c : A \to \{0,1\}^n$ ein Code fester Länge von $A$. Betrachte den Code $C : A \to \{0,1\}^{n+1}$, der aus Code $c$ entsteht, in dem eine Bitstelle an jedes Codewort $c(a)$ hinten angefügt wird und so gesetzt wird, dass der neue Code $C(a)$ den Paritätstest besteht.

# Fehlererkennender Code

- Ein Code $c : A \to \{0,1\}^n$ fester Länge heißt $k$-fehlererkennend, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort durch Kippen von bis zu $k$ Bits verfälscht wurde.

- Der Parity-Code $C$ ist 1-fehlererkennend.
  - **Beweis**: Kippt bei der Übertragung von $C(a)$ genau eine Bitstelle, so kommt eine Bitfolge an, die den Paritätstest nicht besteht und somit kein Codewort von $C$ darstellt. Überprüft der Empfänger die Parität der empfangenen Bitfolge, kann er auf einen Fehler schließen.

# Fehlererkennender Code

- Ein Code $c : A \to \{0,1\}^n$ fester Länge heißt $k$-fehlererkennend, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort durch Kippen von bis zu $k$ Bits verfälscht wurde.

- Der Parity-Code $C$ ist 1-fehlererkennend.
  - **Beweis**: Kippt bei der Übertragung von $C(a)$ genau eine Bitstelle, so kommt eine Bitfolge an, die den Paritätstest nicht besteht und somit kein Codewort von $C$ darstellt. Überprüft der Empfänger die Parität der empfangenen Bitfolge, kann er auf einen Fehler schließen.

# Hamming-Abstand (1/2)

## Definition

Der Hamming-Abstand $dist(v, w)$ zweier $n$-Bitfolgen $v$ und $w$ ist die Anzahl der Stellen, an denen $v$ und $w$ sich unterscheiden.

- $dist(00001101, 10001100) = 2$
- $dist(00001101, 00001101) = 0$
- Ist $v$ das übertragene und $w$ das empfangene Codewort, so liegt ein Übertragungsfehler genau dann vor, wenn $dist(v, w) \neq 0$.
  - Ein Übertragungsfehler heißt einfach, wenn $dist(v, w) = 1$.
- Der Hamming-Abstand eines Codes $c : A \to \{0, 1\}^n$ ist der kleinste Abstand zweier Codewörter von $c$:
  $dist(c) := min\{dist(c(a_i), c(a_j)); a_i, a_j \in A \text{ mit } a_i \neq a_j\}$.

# Hamming-Abstand (2/2)

**Lemma**

Ein Code $c$ fester Länge ist genau dann $k$-fehlererkennend, wenn $dist(c) \geq k+1$ gilt.

- **Beweisidee:** Durch das Kippen von bis zu $l \leq k$ Bits kann aus Codewort $a \in c(A)$ kein anderes Codewort $a^* \in c(A)$ entstehen, denn sonst wäre $dist(c(a), c(a^*)) = l$, was kleiner als der Hamming-Abstand des Codes wäre.

# Fehlerkorrigierende Codes

**Idee**:

- Benutze $r$ zusätzliche Bits, so dass das gesendete Codewort aus dem empfangenen Wort rekonstruiert werden kann.

- Beispiel: Hamming-Code

# Hamming-Code

- Benutze die Bitstellen $2^0, 2^1, \ldots, 2^{r-1}$ als Überprüfungsbits, wobei die Bitstelle $2^j$ die Bitstellen überprüft, deren Binärdarstellungen an der $j$-ten Stelle eine $1$ haben.
- Die Bitstelle $2^j$ wird so belegt, dass gerade viele Bitstellen, deren Binärdarstellungen an der $j$-ten Stelle eine $1$ haben, gesetzt sind. (vgl. Paritätstest)

# Hamming-Code an einem Beispiel

- Uncodiertes Wort: 0111 0101 0000 1111.
  $\rightarrow m = 16$.
- Konstruktion des Hamming-codierten Codeworts:
  - Das Wort wird unter Auslassung der „Zweierpotenz”-Bitstellen aufgeschrieben:
    01110 _ 1010000 _ 111 _ 1 _ _.
  - Dies ergibt insgesamt 21 Bitstellen ($r = 5$).
  - Die „Zweierpotenz”-Bitstellen werden als Überprüfungsbits benutzt (Nummerierung beginnt rechts mit der Stelle 1).
  - Zur Erinnerung: Die Bitstelle $2^j$ wird so belegt, dass gerade viele Bitstellen, deren Binärdarstellungen an der $j$-ten Stelle eine 1 haben, gesetzt sind.

# Hamming-Code-Beispiel (1/4)

| | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | zu codierende Bitfolge |
|---|---|---|---|---|---|---|
| 3 | | | | | | 1 |
| 5 | | | | | | 1 |
| 6 | | | | | | 1 |
| 7 | | | | | | 1 |
| 9 | | | | | | 0 |
| 10 | | | | | | 0 |
| 11 | | | | | | 0 |
| 12 | | | | | | 0 |
| 13 | | | | | | 1 |
| 14 | | | | | | 0 |
| 15 | | | | | | 1 |
| 17 | | | | | | 0 |
| 18 | | | | | | 1 |
| 19 | | | | | | 1 |
| 20 | | | | | | 1 |
| 21 | | | | | | 0 |

Das Überprüfungsbit $2^j$ überprüft die Bitstellen, die in ihrer Binärdarstellung an der $j$-ten Stelle eine 1 haben.

# Hamming-Code-Beispiel (2/4)

| | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | zu codierende Bitfolge |
|---|---|---|---|---|---|---|
| 3 | | | | _ | _ | 1 |
| 5 | | | _ | | _ | 1 |
| 6 | | | _ | _ | | 1 |
| 7 | | | _ | _ | _ | 1 |
| 9 | | _ | | | _ | 0 |
| 10 | | _ | | _ | | 0 |
| 11 | | _ | | _ | _ | 0 |
| 12 | | _ | _ | | | 0 |
| 13 | | _ | _ | | _ | 1 |
| 14 | | _ | _ | _ | | 0 |
| 15 | | _ | _ | _ | _ | 1 |
| 17 | _ | | | | _ | 0 |
| 18 | _ | | | _ | | 1 |
| 19 | _ | | | _ | _ | 1 |
| 20 | _ | | _ | | | 1 |
| 21 | _ | | _ | | _ | 0 |

Das Überprüfungsbit $2^j$ überprüft die Bitstellen, die in ihrer Binärdarstellung an der $j$-ten Stelle eine 1 haben.

# Hamming-Code-Beispiel (3/4)

| | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | zu codierende Bitfolge |
|---|---|---|---|---|---|---|
| 3 | | | | 1 | 1 | 1 |
| 5 | | | 1 | | 1 | 1 |
| 6 | | | 1 | 1 | | 1 |
| 7 | | | 1 | 1 | 1 | 1 |
| 9 | | 0 | | | 0 | 0 |
| 10 | | 0 | | 0 | | 0 |
| 11 | | 0 | | 0 | 0 | 0 |
| 12 | | 0 | 0 | | | 0 |
| 13 | | 1 | 1 | | 1 | 1 |
| 14 | | 0 | 0 | 0 | | 0 |
| 15 | | 1 | 1 | 1 | 1 | 1 |
| 17 | 0 | | | | 0 | 0 |
| 18 | 1 | | | 1 | | 1 |
| 19 | 1 | | | 1 | 1 | 1 |
| 20 | 1 | | 1 | | | 1 |
| 21 | 0 | | 0 | | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |

Der Prüfbitwert ergibt sich aus der Summe modulo 2 der jeweiligen Spalte.

Das Überprüfungsbit $2^j$ überprüft die Bitstellen, die in ihrer Binärdarstellung an der $j$-ten Stelle eine 1 haben.

# Hamming-Code Beispiel (4/4)

- Die Bitfolge 0111 0101 0000 1111
- wird mit dem Hamming-Code zum Codewort
  0 1110 1101 0000 0111 0100.

# Und wie findet man einen Fehler? (1/2)

- Nehme einen Übertragungsfehler an Position 13 des Codeworts an.

- Fehlerhaft empfangenes Wort:
  0 1110 1100 0000 0111 0100.

# Und wie findet man einen Fehler? (2/2)

| | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | empfangene Bitfolge |
|---|---|---|---|---|---|---|
| 3 | | | | 1 | 1 | 1 |
| 5 | | | 1 | | 1 | 1 |
| 6 | | | 1 | 1 | | 1 |
| 7 | | | 1 | 1 | 1 | 1 |
| 9 | | 0 | | | 0 | 0 |
| 10 | | 0 | | 0 | | 0 |
| 11 | | 0 | | 0 | 0 | 0 |
| 12 | | 0 | 0 | | | 0 |
| 13 | | 0 | 0 | | 0 | 0 |
| 14 | | 0 | 0 | 0 | | 0 |
| 15 | | 1 | 1 | 1 | 1 | 1 |
| 17 | 0 | | | | 0 | 0 |
| 18 | 1 | | | 1 | | 1 |
| 19 | 1 | | | 1 | 1 | 1 |
| 20 | 1 | | 1 | | | 1 |
| 21 | 0 | | 0 | | 0 | 0 |

$\Rightarrow$ Fehler muss sich in Zeile $8 + 4 + 1 = 13$ befinden!

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 |

Die Spalten 8,4 und 1 bestehen den Paritätstest nicht!

# Fehlerkorrigierende Codes

## Definition

Ein Code $c : A \to \{0,1\}^n$ fester Länge heißt $k$-fehlerkorrigierend, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort $w$ durch Kippen von bis zu $k$ Bits verfälscht wurde und daraufhin $w$ aus dem empfangenen Wort $\tilde{w}$ rekonstruiert werden kann.

- Der Hamming-Code ist 1-fehlerkorrigierend. Die Anzahl der Zusatzbits $r = 1 + \lfloor log_2 m \rfloor$ ist minimal (Korrektheit folgt aus noch folgendem Satz).

# Zusammenhang Hamming-Abstand und Fehlerkorrektur

## Lemma

Ein Code $c$ fester Länge ist genau dann $k$-fehlerkorrigierend, wenn $dist(c) \geq 2k+1$ gilt.

**Beweis**:

- Sei $M(c(a_i),k) := \{w \in \{0,1\}^n \mid dist(c(a_i),w) \leq k\}$ die Kugel um $c(a_i)$ mit Radius $k$.
- Dann gilt:
  $c$ ist $k$-fehlerkorrigierend $\Leftrightarrow \forall a_i, a_j i \neq j$ gilt: $M(c(a_i),k) \cap M(c(a_j),k) = \emptyset$.
- Für den Beweis ist also zu zeigen:
  $[\forall a_i, a_j\ i \neq j : M(c(a_i),k) \cap M(c(a_j),k) = \emptyset] \Leftrightarrow dist(c) \geq 2k+1$.

# Beweis der Hilfs-Behauptung

- $[\forall a_i, a_j \; i \neq j : M(c(a_i), k) \cap M(c(a_j), k) = \emptyset] \Leftrightarrow dist(c) \geq 2k+1$.
- Beweis „$\Rightarrow$”:
  - Annahme: $dist(c) < 2k+1$
  - D.h. $\exists \; a_i, a_j$ mit $dist(c(a_i), c(a_j)) = l$ und $l < 2k+1$;
  - also gibt es eine Folge: $c(a_i) = b_0, b_1, \ldots, b_{k-1}, b_k, b_{k+1}, \ldots, b_{2k} = c(a_j)$ mit $dist(b_i, b_{i+1}) = 0$ oder $dist(b_i, b_{i+1}) = 1$ (für alle $i = 0, \ldots, 2k-1$),
  - also $b_k \in M(c(a_i), k) \cap M(c(a_j), k)$.
- Beweis „$\Leftarrow$”:
  - Annahme: $M(c(a_i), k) \cap M(c(a_j), k) \neq \emptyset$.
  - Es gibt also $b$ im Durchschnitt mit:
    $dist(c) \leq dist(c(a_i), c(a_j)) \leq dist(c(a_i), b) + dist(b, c(a_j)) \leq k + k$.

# Anzahl Zusatzbits für Fehlerkorrigierende Codes

## Satz

Für einen 1-fehlerkorrigierenden Code $c : A \to \{0,1\}^{m+r}$ fester Länge über $A$ mit $|A| = 2^m$ gilt: $r \geq 1 + \lfloor log_2 m \rfloor$.

**Beweis**:

- $M_1(a) := \{b \in \{0,1\}^{m+r} :$
  $b$ entsteht aus $c(a)$ durch Kippen von bis zu 1 Bit$\}$.
- Nach Lemma muss gelten: $M_1(a_1) \cap M_1(a_2) = \emptyset$ für alle $a_1, a_2 \in A$,
- es gilt $|M_1(a)| = m + r + 1$ für alle $a \in A$.
- Also müssen $2^m$ überschneidungsfreie Kugeln, jede mit $(m+r+1)$ Elementen, im Raum $\mathbb{B}^{m+r}$ enthalten sein: $2^m(m+r+1) \leq 2^{m+r}$.
- Behauptung: Aus $2^m(m+r+1) \leq 2^{m+r}$ folgt $r \geq 1 + \lfloor log_2 m \rfloor$.
- Sei hierzu $k := \lfloor log_2 m \rfloor \Rightarrow 2^k \leq m$.
- $2^m(m+r+1) \leq 2^{m+r} \Leftrightarrow m+r+1 \leq 2^r \Rightarrow 2^k + r + 1 \leq 2^r \Rightarrow 2^k + 1 \leq 2^r \Rightarrow$
  $k < r \Leftrightarrow k + 1 \leq r \Leftrightarrow \lfloor log_2 m \rfloor + 1 \leq r$.

# Ausblick

- Wir haben bisher angenommen, dass Fehler auf Kommunikationskanälen auftreten.
- Es gibt auch Fehler in der Hardware selbst.
  - Permanente Fehler (Fertigungsdefekte)
    $\rightarrow$ Testmethoden (Rechnerarchitektur, Spezialvorlesung „Testen”)
  - Latente Fehler („Beinahe-Defekte”)
    $\rightarrow$ Stresstest (Spezialvorlesung „Testen”)
  - Transiente Fehler (Störungen während des Betriebs)
    $\rightarrow$ Fehlertoleranz, Redundanz (Spezialvorlesung „Testen”)
  - Absichtlich herbeigeführte Fehler (Angriffe)
    - Aus Vergleich des Systemverhaltens mit und ohne Fehler auf geschützte Daten schließen (Fault-Based Cryptanalysis). (*Seminar*)
- Vor allem bei sicherheitskritischen Systemen in neuesten Fertigungstechnologien sind Fehler problematisch.

# Kapitel 7

Formale Spezifikation von Hardware:

1. **Boolesche Ausdrücke**
2. Binäre Entscheidungsdiagramme (BDDs)
3. Anwendung: Formale Verifikation

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Motivation

- Der Entwurf von ReTI hat (hoffentlich) gezeigt, dass Hardware-Synthese komplex und fehleranfällig ist.
- Es gibt automatische Methoden, um Fehler zu finden oder ihre Abwesenheit nachweisen zu können.
- Für ihre Anwendbarkeit muss ein Schaltkreis formal vollständig spezifiziert werden.
- Wir schauen uns daher boolesche Funktionen nochmals (und genauer) an und lernen effiziente Algorithmen und Datenstrukturen zu ihrer Handhabung.

# Motivation

- Der Entwurf von ReTI hat (hoffentlich) gezeigt, dass Hardware-Synthese komplex und fehleranfällig ist.
- Es gibt automatische Methoden, um Fehler zu finden oder ihre Abwesenheit nachweisen zu können.
- Für ihre Anwendbarkeit muss ein Schaltkreis formal vollständig spezifiziert werden.
- Wir schauen uns daher boolesche Funktionen nochmals (und genauer) an und lernen effiziente Algorithmen und Datenstrukturen zu ihrer Handhabung.

# Boolesche Algebren - allgemein

- Es sei $M$ eine Menge auf der zwei binäre Operationen $\cdot$ und $+$ und eine unäre Option $\sim$ definiert sind.
- Das Tupel $(M, \cdot, +, \sim)$ heißt boolesche Algebra, falls $M$ eine nichtleere Menge ist und für alle $x, y, z \in M$ die folgenden Axiome gelten:

| | | |
|---|---|---|
| Kommutativität | $x+y=y+x$ | $x \cdot y = y \cdot x$ |
| Assoziativität | $x+(y+z)=(x+y)+z$ | $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ |
| Absorption | $x+(x \cdot y)=x$ | $x \cdot (x+y)=x$ |
| Distributivität | $x+(y \cdot z)=(x+y) \cdot (x+z)$ | $x \cdot (y+z)=(x \cdot y)+(x \cdot z)$ |
| Komplement | $x+(y \cdot \sim y)=x$ | $x \cdot (y+\sim y)=x$ |

# Beispiele boolescher Algebren

- $(\mathbb{B}, \wedge, \vee, \neg)$
- Boolesche Algebra der Teilmengen einer Menge $S$: $(Pot(S), \cap, \cup, ^C)$
- Boolesche Algebra der booleschen Funktionen in $n$ Variablen: $(\mathbb{B}_n, \cdot, +, \sim)$

$\Rightarrow$ **Allgemein**: Lässt sich eine Aussage direkt aus den Axiomen herleiten, dann gilt sie in allen booleschen Algebren!

- Man darf beim Beweis der Aussage aber auch wirklich nur die Axiome verwenden und keine Eigenschaften der konkreten booleschen Algebra.

# Boolesche Algebra der Teilmengen von $S$ $(Pot(S), \cap, \cup, ^C)$

- Menge: Potenzmenge von $S$
- $\cdot$: $Pot(S) \times Pot(S) \to Pot(S)$; $(M_1, M_2) \mapsto M_1 \cap M_2$
- $+$: $Pot(S) \times Pot(S) \to Pot(S)$; $(M_1, M_2) \mapsto M_1 \cup M_2$
- $^C$: $Pot(S) \to Pot(S)$; $M \mapsto M^C := S \backslash M$

## Satz

$(Pot(S), \cap, \cup, ^C)$ ist eine boolesche Algebra.

- **Beweis**: Nachrechnen, dass alle Axiome gelten.

  **Beispiel**: Absorption

  - Seien $M_1, M_2 \in Pot(S)$.
  - Dann ist $(M_1 + (M_1 \cdot M_2)) = (M_1 \cup (M_1 \cap M_2)) = M_1$ und $(M_1 \cdot (M_1 + M_2)) = (M_1 \cap (M_1 \cup M_2)) = M_1$.

# Boolesche Algebra der Funktionen in $n$ Variablen $(\mathbb{B}_n, \cdot, +, \sim)$

- Menge: $\mathbb{B}_n$ (Menge der booleschen Funktionen in $n$ Variablen)
- $\cdot\colon \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n$; $(f \cdot g)(\alpha) = f(\alpha) \cdot g(\alpha)$ für alle $\alpha \in \mathbb{B}^n$
- $+\colon \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n$; $(f + g)(\alpha) = f(\alpha) + g(\alpha)$ für alle $\alpha \in \mathbb{B}^n$
- $\sim\colon \mathbb{B}_n \to \mathbb{B}_n$; $(\sim f)(\alpha) = 1 \Leftrightarrow f(\alpha) = 0$ für alle $\alpha \in \mathbb{B}^n$

## Satz

$(\mathbb{B}_n, \cdot, +, \sim)$ ist eine boolesche Algebra.

- **Beweis**: Nachrechnen, dass alle Axiome gelten.

  **Beispiel**: Kommutativität
  - Seien $f, g \in \mathbb{B}_n$.
  - Für alle $\alpha \in \mathbb{B}^n$ gilt: $(f+g)(\alpha) = \underbrace{f(\alpha) + g(\alpha)}_{+\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}} = g(\alpha) + f(\alpha) = \underbrace{(g+f)(\alpha)}_{+\colon \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n}$.
  - Also $f + g = g + f$.

# Weitere, aus den Axiomen ableitbare Regeln:

- Existenz neutraler Elemente:
  $\exists \mathbf{0} : x + \mathbf{0} = x,\ x \cdot \mathbf{0} = \mathbf{0} \quad \exists \mathbf{1} : x \cdot \mathbf{1} = x,\ x + \mathbf{1} = \mathbf{1}$
- Doppeltes Komplement:
  $(\sim(\sim x)) = x$
- Eindeutigkeit des Komplements:
  $(x \cdot y = \mathbf{0} \text{ und } x + y = \mathbf{1}) \Rightarrow y = (\sim x)$
- Idempotenz:
  $x + x = x \quad x \cdot x = x$
- de Morgan-Regel:
  $\sim(x + y) = (\sim x) \cdot (\sim y) \quad \sim(x \cdot y) = (\sim x) + (\sim y)$
- Consensus-Regel:
  $(x \cdot y) + ((\sim x) \cdot z) = (x \cdot y) + ((\sim x) \cdot z) + (y \cdot z)$
  $(x + y) \cdot ((\sim x) + z) = (x + y) \cdot ((\sim x) + z) \cdot (y + z)$
- Diese Regeln gelten in allen booleschen Algebren!

# Dualitätsprinzip bei booleschen Algebren

## Prinzip der Dualität

Gilt eine aus den Gesetzen der booleschen Algebra abgeleitete Gleichung $p$, so gilt auch die zu $p$ duale Gleichung, die aus $p$ hervorgeht durch gleichzeitiges Vertauschen von $+$ und $\cdot$, sowie **0** und **1**.

- **Beispiel**:
  - $(x \cdot y) + ((\sim x) \cdot z) + (y \cdot z) = (x \cdot y) + ((\sim x) \cdot z)$
  - $(x + y) \cdot ((\sim x) + z) \cdot (y + z) = (x + y) \cdot ((\sim x) + z)$

# Boolesche Ausdrücke - allgemein

- Formal vollständige Definition boolescher Ausdrücke
  - Syntax (korrekte Schreibweise) $\rightarrow$ Def. boolescher Ausdrücke $BE(X_n)$
  - Semantik (Bedeutung) $\rightarrow$ Interpretationsfunktion $\Psi$ von $BE(X_n)$

$\rightarrow$ Zweck: Einem Rechner unzweifelhaft „beibringen", was und was nicht ein boolescher Ausdruck ist und was seine Funktion bezüglich einer booleschen Algebra ist.

$\rightarrow$ Zum Beispiel: Unterschied zwischen dem Ausdruck „$(x_1 \cdot (\sim x_2))$" und der Funktion $f = x_1 \wedge \neg x_2$.

# Syntax boolescher Ausdrücke

- Sei $X_n = \{x_1, \ldots, x_n\}$ eine endliche Menge von Symbolen/Variablen.
- Sei $A = X_n \cup \{0, 1, +, \cdot, \sim, (, )\}$ ein Alphabet.

## Definition

Die Menge $BE(X_n)$ der vollständig geklammerten booleschen Ausdrücke über $X_n$ ist eine Teilmenge von $A^*$, die folgendermaßen induktiv definiert ist:

- $0, 1$ und $x_i \in X_n$ $i = 1, \ldots, n$ sind boolesche Ausdrücke
- Sind $g$ und $h$ boolesche Ausdrücke, so auch
  - die Disjunktion $(g + h)$,
  - die Konjunktion $(g \cdot h)$,
  - die Negation $(\sim g)$.

# Schreibweise von $BE(X_n)$

- **Konvention**: Negation $\sim$ bindet stärker als Konjunktion $\cdot$, Konjunktion $\cdot$ bindet stärker als Disjunktion $+$.
  - $\rightarrow$ Klammern können weggelassen werden, ohne dass Mehrdeutigkeiten entstehen.
- Je nach Kontext (betrachtete boolesche Algebra) schreibt man auch
  - statt $0, 1$: Die entsprechenden neutralen Elemente,
  - statt $\cdot$: $\wedge, \cap$,
  - statt $+$: $\vee, \cup$,
  - statt $\sim x$: $\neg x, x^C, x', \overline{x}$.

$\rightarrow$ So „vereinfachte" Ausdrücke entsprechen zwar nicht genau der obigen Definition, es gibt aber für jeden solchen Ausdruck einen äquivalenten vollständig geklammerten Ausdruck im Sinne der Definition.

- **Beispiel**: Der äquivalente vollständige geklammerte Ausdruck für „$x_1 \wedge \neg x_2$" wäre „$(x_1 \cdot (\sim x_2))$".

# Semantik boolescher Ausdrücke

- Sei $\widetilde{M} = (M, \cdot, +, \sim)$ eine beliebige boolesche Algebra.
- Seien $0, 1 \in M$ die neutralen Elemente von $B$.

## Definition

Jedem booleschen Ausdruck $BE(X_n)$ kann durch eine Interpretationsfunktion $\Psi : BE(X_n) \to M_n$ eine boolesche Funktion $M_n : M^n \to M$ zugeordnet werden.

$\Psi$ wird folgendermaßen induktiv definiert:

- $\Psi(0) = \mathbf{0}$; $\Psi(1) = \mathbf{1}$;
  $\Psi(x_i)(\alpha_1, \ldots, \alpha_n) = \alpha_i$ für alle $\alpha \in M^n$ (Projektion)
- $\Psi((g+h)) = \Psi(g) + \Psi(h)$ (Disjunktion)
- $\Psi((g \cdot h)) = \Psi(g) \cdot \Psi(h)$ (Konjunktion)
- $\Psi((\sim g)) = \sim (\Psi(g))$ (Negation)

# Interpretation boolescher Ausdrücke

- Sei $e$ ein boolescher Ausdruck.
  - $\Psi(e)(\alpha)$ für ein $\alpha \in M^n$ ergibt sich durch Ersetzen von $x_i$ durch $\alpha_i$ in $e$, für alle $i$ und Rechnen in der booleschen Algebra $\widetilde{M}$.
  - Gilt $\Psi(e) = f$ für eine boolesche Funktion $f \in M_n$, so sagen wir, dass $e$ ein boolescher Ausdruck für $f$ ist, bzw. dass $e$ die boolesche Funktion $f$ beschreibt.
  - Zwei boolesche Ausdrücke $e_1$ und $e_2$ heißen äquivalent ($e_1 \equiv e_2$) genau dann, wenn $\Psi(e_1) = \Psi(e_2)$.
    Sie sind gleich, wenn $e_1 = e_2$.
- Wir betrachten folgend nur noch die Interpretation in $B = (\mathbb{B}, \wedge, \vee, \neg)$.

# Boolesche Ausdrücke $\leftrightarrow$ boolesche Funktionen

## Lemma 1

Zu jedem booleschen Ausdruck $e \in BE(X_n)$ existiert eine boolesche Funktion $f$, die durch $e$ beschrieben wird.

- **Beweis**: $f := \Psi(e)$

## Lemma 2

Zu jeder booleschen Funktion $f$ existiert ein boolescher Ausdruck, der $f$ beschreibt.

- **Beweis**: Es gilt: $f = \Psi(\sum_{\alpha \in ON(f)} m(\alpha))$.

  m.a.W. Die DNF ist ein Boolescher Ausdruck.

# Zusammenhang mit Schaltkreisen

**Lemma 3**

Zu jedem booleschen Ausdruck $e \in BE(X_n)$ gibt es einen kombinatorischen Schaltkreis, der $e$ implementiert.

- **Beweis**: Übung

- Zu jeder booleschen Funktion gibt es einen kombinatorischen Schaltkreis, der sie implementiert (zum Beispiel zweistufige Umsetzung der DNF/KDNF).
- Zu jedem kombinatorischen Schaltkreis gibt es sowohl eine boolesche Funktion, als auch einen boolescher Ausdruck.

# Kapitel 7

Formale Spezifikation von Hardware:

1. Boolesche Ausdrücke
2. **Binäre Entscheidungsdiagramme (BDDs)**
3. Anwendung: Formale Verifikation

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Motivation: Eindeutigkeit der Darstellung boolescher Funktionen

- Die Darstellung einer gegebenen booleschen Funktion als Schaltkreis oder als boolescher Ausdruck ist im Allgemeinen nicht eindeutig.
- **Beispiel**: „$x_1 \cdot \sim x_2 + x_3$" und „$x_1 \cdot \sim x_2 + x_3 + x_3$".
- Wir kennen eindeutige Darstellungen einer booleschen Funktion.
- **Zum Beispiel**:
  - Funktionstabelle
  - Kanonische disjunktive Normalform (alle Minterme)
- **Aber**: Diese eindeutigen Darstellungen sind „sehr sperrig".

  $\Rightarrow$ $2^n$ Einträge/alle Minterme für Funktionen in $n$ Variablen.
- **Ziel**: Finde eine eindeutige, aber kompakte Darstellung boolescher Funktionen.

# Wozu Eindeutigkeit?

- Zur schnellen Überprüfung, ob zwei boolesche Funktionen äquivalent sind.

- **Beispiel**:
  - Ein Designer hat Optimierungen an der Hardware vorgenommen und 15 Gatter gespart. Hat er dabei vielleicht die Funktionalität verändert?

  → Erzeuge eindeutige Darstellung für alten und neuen Block und vergleiche.

→ Dieses Vorgehen nennt man formalen Äquivalenzcheck.

# Illustration: Formaler Äquivalenztest

Eindeutige Darstellung der durch $C_1$ realisierten booleschen Funktion.

Eindeutige Darstellung der durch $C_2$ realisierten booleschen Funktion.

# Binäre Entscheidungsdiagramme (Binary Decision Diagrams)

- BDDs repräsentieren boolesche Funktionen eindeutig und (oft) kompakt zugleich.
  - → Enthält genau die gleiche Information, wie eine Funktionstabelle.
- Es gibt Verfahren, um schnell ein BDD aus einem Schaltkreis zu erzeugen.
  - → Man muss nicht erst die Funktionstabelle erzeugen.
- Es existieren Software-Packages, die Erzeugung und Manipulation von BDDs unterstützen.
  - CUDD von der Colorado University: `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`
  - Zahlreiche Optimierungen bei Cache-Ausnutzung, Garbage Collection, usw.

# Binary Decision Diagram – Syntax

- Sei $X_n = \{x_1, \ldots, x_n\}$ eine endliche Menge von Variablen.

## Definition

Ein BDD ist

- ein azyklischer gerichteter Graph $G = (V, E)$ mit
  - genau einer Wurzel,
  - $outdeg(v) = 2$ für alle inneren Knoten $v$.
- Jeder Knoten $v$ hat eine Markierung
  - $label(v) = \begin{cases} x_i \in X_n, & v \text{ ist innerer Knoten} \\ c \in \{0,1\}, & v \text{ ist Blatt} \end{cases}$
- Die Nachfolger (Kinder) eines inneren Knoten $v$ sind
  - das Low-Kind: $low(v)$,
  - das High-Kind: $high(v)$.

# BDDs: Semantik

- Gegeben sei ein BDD.

## Definition

- Sei $f_v$ die Funktion, die ein Teil-BDD ausgehend von Knoten $v$ beschreibt.
  - Ist $v$ ein Blatt, so beschreibt $v$ die konstante Funktion, die jede Variablenbelegung auf $label(v) \in \{0, 1\}$ abbildet.
  - Ist $v$ ein innerer Knoten mit $label(v) = x_j$, so gilt die Kompositionsregel: $f_v = (x_j \wedge f_{high(v)}) \vee (x'_j \wedge f_{low(v)})$.

# BDDs: Beispiel

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs: Beispiel

$$f_v = (x_j \wedge f_{high(v)}) \vee (x_j' \wedge f_{low(v)})$$

# BDDs von typischen Funktionen

Seien $x_1, x_2$ beliebige boolsche Variablen.

$x_1$:

# BDDs von typischen Funktionen

Seien $x_1, x_2$ beliebige boolsche Variablen.

$x_1$:

$x_1'$:

# BDDs von typischen Funktionen

Seien $x_1, x_2$ beliebige boolsche Variablen.

$x_1$:

$x_1'$:

$x_1 \cdot x_2$:

# BDDs von typischen Funktionen

Seien $x_1, x_2$ beliebige boolsche Variablen.

$x_1$:

$x_1'$:

$x_1 \cdot x_2$:

$x_1 + x_2$:

# BDDs von typischen Funktionen

Seien $x_1, x_2$ beliebige boolsche Variablen.

$x_1$:

$x_1'$:

$x_1 \cdot x_2$:

$x_1 + x_2$:

$x_1 \oplus x_2$:

# Variablenordnung

## Definition

- Ein BDD heißt geordnet, wenn auf jedem Pfad von der Wurzel zu einem Blatt jede Variable höchstens einmal (oder keinmal) als Markierung vorkommt und die Variablen stets in der selben Reihenfolge abgefragt werden.
- Diese Reihenfolge heißt Variablenordnung.

# Beispiel für unterschiedliche Variablenordnung

Variablenordnung

$x_1, x_2, x_3, x_4$

# Beispiel für unterschiedliche Variablenordnung

Variablenordnung

$x_1, x_2, x_3, x_4$

Variablenordnung

$x_1, x_4, x_2, x_3$

# Beispiel für unterschiedliche Variablenordnung

Variablenordnung

$x_1, x_2, x_3, x_4$

Variablenordnung

$x_1, x_4, x_2, x_3$

$\Rightarrow$ Beide BDDs beschreiben die boolesche Funktion $x_1x_2x_3 + x_2'x_4 + x_3'x_4$.

# Vollständiges BDD

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

- Ist ein geordnetes BDD, wobei alle Pfade von der Wurzel zu einem Blatt maximale Länge haben und $indeg(v) = 1$ für alle Knoten $v$.
- Entspricht genau der Funktionstabelle (Beweis klar)

# Zusammenhang: BDD $\leftrightarrow$ Boolesche Funktion

## Lemma 1

Jedes (geordnete) BDD stellt eine boolesche Funktion dar.

- **Beweis**: Ist $v$ die Wurzel des BDDs, dann erhält man mit der Kompositionsregel die dargestellte boolesche Funktion $f_v$.

## Lemma 2

Zu jeder booleschen Funktion existiert ein (geordnetes) BDD, das diese darstellt.

- **Beweis**: Konstruiere vollständiges BDD aus Funktionstabelle.

# Eindeutigkeit von BDDs

- Nicht geordnete BDDs sind nicht eindeutige Darstellungen, ebenso BDDs mit unterschiedlicher (nicht fester) Variablenordnung.
- Die vollständigen BDDs sind eindeutige Darstellungen (da die Funktionstabelle eindeutig ist).
- **Aber**: Ineffizient $\rightarrow$ Reduktion von BDDs

# Reduzierte BDDs

**Idee**: Stelle Information möglichst kompakt dar, „isomorphe Teil-BDDs" werden identifiziert.

## Definition

Ein BDD heißt reduziert, wenn es

- keinen nichtterminalen Knoten $v$ gibt, mit Label $x$ und $f_{high(v)} = f_{low(v)}$.
- keine verschiedenen Knoten $v$ und $w$ gibt, die gleich markiert sind und deren low- und high-Kind (falls existent) jeweils identisch sind.

# Beispiel für reduzierte und nicht reduzierte BDDs (1/2)

# Beispiel für reduzierte und nicht reduzierte BDDs (1/2)

# Beispiel für reduzierte und nicht reduzierte BDDs (1/2)

# Beispiel für reduzierte und nicht reduzierte BDDs (2/2)

# Beispiel für reduzierte und nicht reduzierte BDDs (2/2)

# Beispiel für reduzierte und nicht reduzierte BDDs (2/2)

# Beispiel: Reduktion eines BDDs (1/7)

# Beispiel: Reduktion eines BDDs (1/7)

# Beispiel: Reduktion eines BDDs (2/7)

# Beispiel: Reduktion eines BDDs (3/7)

# Beispiel: Reduktion eines BDDs (4/7)

# Beispiel: Reduktion eines BDDs (5/7)

# Beispiel: Reduktion eines BDDs (5/7)

# Beispiel: Reduktion eines BDDs (6/7)

# Beispiel: Reduktion eines BDDs (7/7)

# Beispiel: Reduktion eines BDDs (7/7)

# Satz von Bryant (1986)

**Satz**

ROBDDs (Reduced Ordered BDDs, geordnete, reduzierte BDDs) sind kanonische Darstellungen boolescher Funktionen.

- Ohne Beweis.
- **Bedeutung**: Für eine fixe Variablenordnung ist ein ROBDD eine eindeutige Darstellung für eine boolesche Funktion.
- **Notation**: Ab sofort bezeichen wir ROBDDs als BDDs.

# Kofaktoren und Shannon-Zerlegung

- Für eine boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ heißt $f_{x_i=1} : \mathbb{B}^n \to \mathbb{B}$ definiert durch $f_{x_i=1}(\alpha_1, \ldots, \alpha_{i-1}, \alpha_i, \alpha_{i+1}, \ldots, \alpha_n) = f(\alpha_1, \ldots, \alpha_{i-1}, 1, \alpha_{i+1}, \ldots, \alpha_n)$ für alle $\alpha \in \{0,1\}^n$ der Kofaktor von $f$ nach $x_i = 1$.

- Entsprechend heißt $f_{x_i=0} = f(\alpha_1, \ldots, \alpha_{i-1}, 0, \alpha_{i+1}, \ldots, \alpha_n)$ der Kofaktor von $f$ nach $x_i = 0$.

## Shannonscher Entwicklungssatz

Es gilt für beliebiges $f$:

$$f = (x_i \wedge f_{x_1=1}) \vee (x_i' \wedge f_{x_i=0})$$

- $f = (x_i \wedge f_{x_i=1}) \vee (x_i' \wedge f_{x_i=0})$ heißt Shannon-Zerlegung von $f$.

# Zusammenhang Kompositionsregel, Kofaktoren und Shannon-Zerlegung

- Sei $v$ ein innerer Knoten eines geordneten BDDs, der mit $x_i$ markiert ist.
- Dann kommt $x_i$ in Kindern von $v$ nicht vor (sonst mehrfaches Vorkommen auf dem gleichen Pfad).
- Also sind die Subfunktionen $f_{high(v)}$, $f_{low(v)}$ von $f_v$ von $x_i$ unabhängig, d.h. für alle $\alpha \in \{0,1\}^n$ gilt:
  - $f_{high(v)}(\alpha_1,\ldots,\alpha_{i-1},1,\alpha_{i+1},\ldots,\alpha_n) = f_{high(v)}(\alpha_1,\ldots,\alpha_{i-1},0,\alpha_{i+1},\ldots,\alpha_n)$
  - $f_{low(v)}(\alpha_1,\ldots,\alpha_{i-1},1,\alpha_{i+1},\ldots,\alpha_n) = f_{low(v)}(\alpha_1,\ldots,\alpha_{i-1},0,\alpha_{i+1},\ldots,\alpha_n)$
- Es folgt dann: $f_{high(v)} = f_{v,x_i=1}, \quad f_{low(v)} = f_{v,x_i=0}$.
- Somit wird an $v$ die Shannon-Zerlegung berechnet:
  $f_v = (x_i \wedge f_{high(v)}) \vee (x_i' \wedge f_{low(v)}) = (x_i \wedge f_{v,x_i=1}) \wedge (x_i' \wedge f_{v,x_i=0})$.

# Konstruktion von BDDs

- Über die Kofaktoren.
  - **Beispiel**: Um ein $bdd_f$ von $f = x_1x_2x_3 + x_2'x_4 + x_3'x_4$ bzgl. der Variablenordnung $x_1, x_2, x_3, x_4$ zu konstruieren, konstruiere BDDs von $f_{x_1=1} = x_2x_3 + x_2'x_4 + x_3'x_4$ und $f_{x_1=0} = x_2'x_4 + x_3'x_4$. Sie stellen die Söhne des Knotens $x_1$ dar. Wende dies rekursiv an.
- Aus einem booleschen Ausdruck über boolesche Operationen.
- Aus einem Schaltkreis über boolesche Operationen (Symbolische Simulation).

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $c$ | $f_{a=1,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $c$ | $f_{a=1,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $c$ | $f_{a=1,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $c$ | $f_{a=1,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (1/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=1}(b,c) = b \cdot f_{a=1,b=1}(c) + b' \cdot f_{a=1,b=0}(c)$.

| $b$ | $c$ | $f_{a=1}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $c$ | $f_{a=1,b=1}$ |
|---|---|
| 0 | 1 |
| 1 | 1 |

| $c$ | $f_{a=1,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $c$ | $f_{a=0,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $c$ | $f_{a=0,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $c$ | $f_{a=0,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $c$ | $f_{a=0,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (2/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.
- $f_{a=0}(b,c) = b \cdot f_{a=0,b=1}(c) + b' \cdot f_{a=0,b=0}(c)$.

| $b$ | $c$ | $f_{a=0}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $c$ | $f_{a=0,b=1}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

| $c$ | $f_{a=0,b=0}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (3/3)

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (3/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (3/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD des Carry-Bits $f := fa_1$ eines Volladierers aus Kofaktoren (3/3)

- $f(a,b,c) = a \cdot f_{a=1}(b,c) + a' \cdot f_{a=0}(b,c)$.

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# BDD aus einem booleschen Ausdruck

- Konstruiere BDD $bdd_f$ von $f = x_1x_2x_3 + x_2'x_4 + x_3'x_4$.
  - Generiere $bdd_{x_1}$, $bdd_{x_2}$, $bdd_{x_3}$, $bdd_{x_4}$.
  - Berechne $bdd_{x_1x_2} := AND(bdd_{x_1}, bdd_{x_2})$.
  - Berechne $bdd_{x_1x_2x_3} := AND(bdd_{x_1x_2}, bdd_{x_3})$.
  - Berechne $bdd_{x_2'} := NOT(bdd_{x_2})$.
  - Berechne $bdd_{x_2'x_4} := AND(bdd_{x_2'}, bdd_{x_4})$.
  - Berechne $bdd_{x_3'} := NOT(bdd_{x_3})$.
  - Berechne $bdd_{x_3'x_4} := AND(bdd_{x_3'}, bdd_{x_4})$.
  - Berechne $bdd_{x_1x_2x_3+x_2'x_4} := OR(bdd_{x_1x_2x_3}, bdd_{x_2'x_4})$.
  - Berechne $bdd_f := OR(bdd_{x_1x_2x_3+x_2'x_4}, bdd_{x_3'x_4})$.
- Berechnung von AND, OR, NOT: Später!

# BDD aus Schaltkreis: Symbolische Simulation

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,
$bdd_{out_1}$
$:= OR(bdd_{x_1}, bdd_{x_{11}})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,
$bdd_{out_1}$
$:= OR(bdd_{x_1}, bdd_{x_{11}})$,
$bdd_{12}$
$:= OR(bdd_{x_6}, bdd_{x_7})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,
$bdd_{out_1}$
$:= OR(bdd_{x_1}, bdd_{x_{11}})$,
$bdd_{12}$
$:= OR(bdd_{x_6}, bdd_{x_7})$,
$bdd_{13}$
$:= AND(bdd_{x_{10}}, NOT(bdd_{x_8}))$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,
$bdd_{out_1}$
$:= OR(bdd_{x_1}, bdd_{x_{11}})$,
$bdd_{12}$
$:= OR(bdd_{x_6}, bdd_{x_7})$,
$bdd_{13}$
$:= AND(bdd_{x_{10}}, NOT(bdd_{x_8}))$,
$bdd_{14}$
$:= AND(bdd_{x_{12}}, bdd_{x_8})$,

# BDD aus Schaltkreis: Symbolische Simulation

**BDD-Operationen**:

$bdd_{x_1}, bdd_{x_2}, \ldots, bdd_{x_8}$,
$bdd_9$
$:= XOR(bdd_{x_2}, bdd_{x_3})$,
$bdd_{10}$
$:= AND(bdd_{x_4}, bdd_{x_5})$,
$bdd_{11}$
$:= XOR(bdd_{x_9}, bdd_{x_{10}})$,
$bdd_{out_1}$
$:= OR(bdd_{x_1}, bdd_{x_{11}})$,
$bdd_{12}$
$:= OR(bdd_{x_6}, bdd_{x_7})$,
$bdd_{13}$
$:= AND(bdd_{x_{10}}, NOT(bdd_{x_8}))$,
$bdd_{14}$
$:= AND(bdd_{x_{12}}, bdd_{x_8})$,
$bdd_{out_2}$
$:= OR(bdd_{x_{13}}, bdd_{x_{14}})$.

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Beispiel: BDD des Carry-Bits $f := fa_1$ eines FAs aus Schaltkreis

# Übrigens …

- Wir haben soeben einen Äquivalenzcheck durchgeführt!

| $a$ | $b$ | $c$ | $fa_1$ | $fa_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

# Realisierung von booleschen Operationen zwischen BDDs

- Gegeben BDDs $a$ und $b$, berechne BDD $c = AND(a,b)$.
- Erste Option: Berechne die Funktionstabelle von $(a \cdot b)$, erzeuge daraus das BDD mit der Kofaktor-Methode $\rightarrow$ sehr ineffizient!
- Besserer Ansatz: If-Then-Else-Funktion ($ITE$).
  - $ITE(F,G,H) := FG + F'H$.
  - Alle binären Operationen sind auf $ITE$ zurückführbar.
    - $AND(F,G) = ITE(F,G,0)$
    - $OR(F,G) = ITE(F,1,G)$
    - $NOT(F) = ITE(F,0,1)$
    - $XOR(F,G) = ITE(F,G',G)$

# Realisierung von booleschen Operationen zwischen BDDs

- Gegeben BDDs $a$ und $b$, berechne BDD $c = AND(a,b)$
- Erste Option: Berechne die Funktionstabelle von $(a \cdot b)$, erzeuge daraus das BDD mit der Kofaktor-Methode $\rightarrow$ sehr ineffizient!
- Besserer Ansatz: If-Then-Else-Funktion ($ITE$).
  - $ITE(F,G,H) := FG + F'H$.
  - Alle binären Operationen sind auf $ITE$ zurückführbar.
    - $AND(F,G) = ITE(F,G,0)$
    - $OR(F,G) = ITE(F,1,G)$
    - $NOT(F) = ITE(F,0,1)$
    - $XOR(F,G) = ITE(F,G',G)$

# Berechnung von *ITE* auf BDDs-Prinzip

- Gegeben sind drei BDDs $F$, $G$ und $H$ mit der gleichen Variablenordnung. Gesucht ist BDD für $ITE(F,G,H)$.
- Es gilt für eine Variable $x$:
- $$\begin{aligned} & ITE(F,G,H) \\ & = FG + F'H \\ & = x(FG+F'H)_{x=1} + x'(FG+F'H)_{x=0} \\ & = x(F_{x=1}G_{x=1} + (F_{x=1})'H_{x=1}) + x'(F_{x=0}G_{x=0} + (F_{x=0})'H_{x=0}) \\ & = x \cdot ITE(F_{x=1},G_{x=1},H_{x=1}) + x' \cdot ITE(F_{x=0},G_{x=0},H_{x=0}). \end{aligned}$$
- *ITE*-Berechnung im vollständigen BDD:
  - Sei $x$ die oberste Variable in BDDs $F$, $G$ und $H$.
  - BDDs für $F_{x=1}, G_{x=1}, H_{x=1}, F_{x=0}, G_{x=0}, H_{x=0}$ sind Kinder von $x$.
  - Berechne BDDs $ITE(F_{x=1},G_{x=1},H_{x=1})$ und $ITE(F_{x=0},G_{x=0},H_{x=0})$ rekursiv. $x$ mit diesen BDDs als Kinder ist das gesuchte BDD.

# $ITE$ auf reduzierten BDDs

- $ITE(F,G,H)$
  $= x \cdot ITE(F_{x=1}, G_{x=1}, H_{x=1}) + x' \cdot ITE(F_{x=0}, G_{x=0}, H_{x=0})$.

- Rekursive Vorgehensweise:
  - Berechne BDDs $ITE(F_{x=1}, G_{x=1}, H_{x=1})$ und $ITE(F_{x=0}, G_{x=0}, H_{x=0})$.
  - Falls $ITE(F_{x=1}, G_{x=1}, H_{x=1}) = ITE(F_{x=0}, G_{x=0}, H_{x=0})$, so ist $ITE(F,G,H) = ITE(F_{x=1}, G_{x=1}, H_{x=1}) = ITE(F_{x=0}, G_{x=0}, H_{x=0})$.
  - Sonst generiere einen neuen Knoten $x$, dessen high-Kante auf $ITE(F_{x=1}, G_{x=1}, H_{x=1})$ und low-Kante auf $ITE(F_{x=0}, G_{x=0}, H_{x=0})$ zeigt.
  - **Terminalfälle**:
    $ITE(1,F,G) = ITE(0,G,F) = ITE(F,1,0) = ITE(G,F,F) = F$.

# *ITE*-Berechnung: Beispiel (1/2)

- Wir haben vorhin gerechnet: $R = OR(F, G)$
- Dies entspricht: $R = ITE(F, 1, G)$.
- Wir wollen nun die rekursive *ITE*-Berechnung nachvollziehen.
- Es liegt kein Terminalfall vor, obere Variable: $a$.

# *ITE*-Berechnung: Beispiel (2/2)

$F :=$


$G :=$


$R :=$


- Berechne zunächst rekursiv $ITE(F_{a=1}, 1_{a=1}, G_{a=1})$.

$F_{a=1} =$


$, 1_{a=1} = 1,$ $G_{a=1} =$


- Rechne rekursiv mit Variable $b$ weiter:
  $ITE(F_{a=1}, 1_{a=1}, G_{a=1}) = b \cdot ITE(F_{a=1,b=1}, 1_{a=1,b=1}, G_{a=1,b=1}) +$
  $b' \cdot ITE(F_{a=1,b=0}, 1_{a=1,b=0}, G_{a=1,b=0})$.

# Beispiel: $ITE(F_{a=1}, 1_{a=1}, G_{a=1})$

$F_{a=1} =$


$, 1_{a=1} = 1,$ $G_{a=1} =$


- $ITE(F_{a=1}, 1_{a=1}, G_{a=1}) = b \cdot ITE(F_{a=1,b=1}, 1_{a=1,b=1}, G_{a=1,b=1})$
  $+ b' \cdot ITE(F_{a=1,b=0}, 1_{a=1,b=0}, G_{a=1,b=0})$.
  - $F_{a=1,b=1} = 1_{a=1,b=1} = 1, G_{a=1,b=1} = 0,$
    $ITE(1,1,0) = 1$ **(Terminalfall)**.
  - $F_{a=1,b=0} = 0, 1_{a=1,b=0} = 1, G_{a=1,b=0} = c =$
    $ITE(0,1,c) = c$ **(Terminalfall)**.

- Insgesamt:
  $ITE(F_{a=1}, 1_{a=1}, G_{a=1}) =$

# Beispiel: $ITE(F_{a=0}, 1_{a=0}, G_{a=0})$

$F :=$

$G :=$

$R :=$

- Berechne zunächst rekursiv $ITE(F_{a=0}, 1_{a=0}, G_{a=0})$.

$F_{a=0} = 0, \quad 1_{a=0} = 1, \quad G_{a=0} =$

Ein **Terminalfall** liegt vor:
$ITE(0, 1, G_{a=0}) = G_{a=0} =$

# Beispiel - Ende

- $ITE(F,1,G) = a \cdot ITE(F_{a=1},1_{a=1},G_{a=1}) + a' \cdot ITE(F_{a=0},1_{a=0},G_{a=0})$.

$ITE(F_{a=1},1_{a=1},G_{a=1}) =$

$ITE(F_{a=0},1_{a=0},G_{a=0}) =$

- Durch die Verwendung der sogenannten Computed Tables (s.u.) sind die berechneten BDDs bereits reduziert:

- Somit muss *ITE* nur noch einen Knoten *a* einfügen und seine ausgehenden Kanten „richtig lenken”.

# Zur Komplexität von *ITE*

- Bei der rekursiven Konstruktion von *ITE* überprüft man, ob die BDDs $ITE(F_{x=1}, G_{x=1}, H_{x=1})$ und $ITE(F_{x=0}, G_{x=0}, H_{x=0})$ vielleicht bereits generiert wurden und in der sogenannten Computed Table (CT) vorliegen.
- Nur wenn dies nicht der Fall ist, werden die BDDs erzeugt und in die CT eingefügt.
- Implementiert man dies effizient, wird die Komplexität von *ITE* zu $O(|F| \cdot |G| \cdot |H|)$.
  - Ohne Beweis.
- Somit lässt sich etwa die Komplexität von $AND(F,G) = ITE(F,G,0)$ durch $O(|F| \cdot |G|)$ abschätzen (0 ist das BDD für die konstante 0-Funktion, das aus 1 Knoten besteht.)

# ITE-Algorithmus

$ITE(F, G, H)\{$

**if** $(F == 1)$ **return** $G$; // Überprüfe erst 4 Terminalfälle.
**if** $(F == 0)$ **return** $H$;
**if** $((G == 1)$ **and** $(H == 0))$ **return** $F$;
**if** $(G == H)$ **return** $G$;

$x = Top_Variable(F, G, H)$; // Gleiche Variablenordnung.
$high = ITE(F_{x=1}, G_{x=1}, H_{x=1})$;
$low = ITE(F_{x=0}, G_{x=0}, H_{x=0})$;
**if** $(high == low)$ **return** $low$;
$R = Find_Or_Add_in_Computed_Table(x, high, low)$;
**return** $R$;

$\}$

# Größe von BDDs

- Oft exponentiell.
- Hängt stark von der Variablenordnung ab.
  - Siehe Beispiel auf der nächsten Folie.
- Für viele in der Praxis vorkommende Funktionen sind BDDs zumindest für einige Variablenordnungen kompakt.
- BDD-Minimierung: Finden einer „guten” Variablenordnung (NP-vollständiges Problem).
  - Exakte und heuristische Verfahren.
  - Für einige Funktionen, u.a. Multiplizierer, ist BDD-Größe für jede Variablenordnung exponentiell.

# Einfluss der Variablenordnung

- **Beispiel**: $f^n(x_1, \ldots, x_{2n}) = x_1 x_{n+1} + x_2 x_{n+2} + \cdots + x_n x_{2n}$.

# Einfluss der Variablenordnung

- **Beispiel**: $f^n(x_1,\ldots,x_{2n}) = x_1x_{n+1} + x_2x_{n+2} + \cdots + x_nx_{2n}$.

exponentielle Größe

# Einfluss der Variablenordnung

- **Beispiel**: $f^n(x_1,\ldots,x_{2n}) = x_1x_{n+1} + x_2x_{n+2} + \cdots + x_nx_{2n}$.

exponentielle Größe

lineare Größe

# Schaltkreis direkt aus BDD

# Schaltkreis direkt aus BDD

# Schaltkreis direkt aus BDD

# Schaltkreis direkt aus BDD

# Schaltkreis direkt aus BDD

# Schaltkreis direkt aus BDD

… beschreibt die boolesche Funktion $x_1' x_2' x_3' + x_1' x_2 + x_1 x_2'$.

# Kapitel 7

Formale Spezifikation von Hardware:

1. Boolesche Ausdrücke
2. Binäre Entscheidungsdiagramme (BDDs)
3. **Anwendung: Formale Verifikation**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Formale Verifikation

- **Ziel**: Mit formalen Methoden weitgehend automatisch feststellen, ob der Entwurf korrekt ist.
  - **Hier**: Nur grobe Prinzipien, es gibt eine Vielzahl von Weiterentwicklungen, die auch im Einsatz sind.
- Äquivalenz-Check: Verleich der Spezifikation mit der Implementierung.
  - **Hier**: Komplexeres Beispiel (ein „Slice" der ReTI-CPU) mit Hilfe von BDDs.
- Eigenschafts-Check: Überprüfung, ob bestimmte Eigenschaften gelten.
  - **Hier**: Abwesenheit von Bus Contention auf einem Bus.

# Schaltrealisierung der ALU

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# „0-tes Slice der ALU”

- Das „0-te Slice” ist der Teil der kombinatorischen Logik, der den 0. Ausgang der ALU berechnet.
- Das Vorgehen für die gesamte CPU wäre identisch, die BDDs jedoch (sehr) viel größer.
- **Vorgehen**:
  - Erstelle die Funktionstabelle für den 0. Ausgang der ALU („Spezifikation”) und berechne das BDD dazu.
  - Bestimme die Hardware, welche den 0. Ausgang ansteuert („Implementierung”) und berechne das BDD dazu.
  - Sind die BDDs gleich, ist die Implementierung korrekt!

# Spezifikation für das 0-te Slice

| $s_2$ | $s_1$ | $s_0$ | 0. Ausgang |
|---|---|---|---|
| 0 | 0 | 0 | $(0\ldots0)_0 = 0$ |
| 0 | 0 | 1 | $([b]-[a])_0 = b_0 \oplus a'_0 \oplus 1 = a_0 \oplus b_0$ |
| 0 | 1 | 0 | $([a]-[b])_0 = a_0 \oplus b'_0 \oplus 1 = a_0 \oplus b_0$ |
| 0 | 1 | 1 | $([a]+[b]+c)_0 = a_0 \oplus b_0 \oplus c$ |
| 1 | 0 | 0 | $(a \oplus b)_0 = a_0 \oplus b_0$ |
| 1 | 0 | 1 | $(a \vee b)_0 = a_0 \vee b_0$ |
| 1 | 1 | 0 | $(a \wedge b)_0 = a_0 \wedge b_0$ |
| 1 | 1 | 1 | $(1\ldots1)_0 = 1$ |

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# BDD für die Spezifikation

- Variablenordnung: $s_2$, $s_1$, $s_0$, $c$, $a_0$, $b_0$.
- Reduziert bis auf Terminalknoten (wegen Übersichtlichkeit).

# Schaltrealisierung der ALU

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Hardware für das „0-te Slice”

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# BDD für die Schaltrealisierung

- Das BDD entspricht nicht dem für die Spezifikation!
- Irgendwo muss ein Entwurfsfehler sein!

# Im Entwurf ist ein Fehler – aber wo?

- Wir wollen nicht nur wissen, dass der Entwurf fehlerhaft ist, sondern den Fehler finden (und dann beheben)!
- Dies wird „Diagnose von Entwurfsfehlern" genannt.
- Ein mögliches Vorgehen: Finde eine Belegung der Eingänge, für die Spezifikation und Implementierung nicht übereinstimmen.
- Wir suchen also eine Belegung von ($s_2$, $s_1$, $s_0$, $c$, $a_0$, $b_0$), für welche gilt:
  $Implem.(s_2, s_1, s_0, c, a_0, b_0) \neq Spez.(s_2, s_1, s_0, c, a_0, b_0)$.
- Mit BDDs ist das ganz einfach: Berechne BDD für
  $Implem.(s_2, s_1, s_0, c, a_0, b_0) \oplus Spez.(s_2, s_1, s_0, c, a_0, b_0)$
  und betrachte einen beliebigen Pfad zu 1-Terminalknoten.

# Fehlerdiagnose

- Ergebnis der XOR-Verknüpfung der beiden BDDs:

- Ein Pfad zum 1-Terminal: $s_2 = 0$, $s_1 = 1$, $s_0 = 0$, $c = 0$.
- $a_0$, $b_0$ sind unbestimmt; setze z.B. $a_0 = 0$, $b_0 = 0$.
- Die ALU soll also $[a] - [b]$ berechnen (da $s_2 s_1 s_0 = 010$).

# Simulation der gefundenen Belegung (1/3)

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Simulation der gefundenen Belegung (2/3)

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Simulation der gefundenen Belegung (3/3)

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Analyse der Simulationsergebnisse

- Es werden zwei Zahlen subtrahiert, die mit „0” enden. Das Ergebnis soll mit „1” enden.
- Das kann nicht sein, die ALU subtrahiert falsch.
- Die Dateneingänge des Addierers sind richtig.
- Der Übertragseingang des Addierers ist aber falsch, er hätte bei Subtraktion 1 sein müsen!
- Der Fehler muss in der Logik liegen, die den Übertragseingang steuert. Das sind 4 Gatter.
- Tatsächlich sind ein AND- und ein XOR-Gatter vertauscht. Vgl. die (richtige) Folie aus Kap. 3.5.

# Original-Folie aus Kapitel 3

| $s_2$ | $s_1$ | $s_0$ | |
|---|---|---|---|
| 0 | 0 | 0 | $0\ldots0$ |
| 0 | 0 | 1 | $[b]-[a]$ |
| 0 | 1 | 0 | $[a]-[b]$ |
| 0 | 1 | 1 | $[a]+[b]+c$ |
| 1 | 0 | 0 | $a \oplus b = (a_{n-1} \oplus b_{n-1}, \ldots, a_0 \oplus b_0)$ |
| 1 | 0 | 1 | $a \vee b = (a_{n-1} \vee b_{n-1}, \ldots, a_0 \vee b_0)$ |
| 1 | 1 | 0 | $a \wedge b = (a_{n-1} \wedge b_{n-1}, \ldots, a_0 \wedge b_0)$ |
| 1 | 1 | 1 | $1\ldots1$ |

# Was hat die formale Analyse gebracht?

- Wir mussten den Fehler nach wie vor manuell suchen, ...
- ... allerdings in den 4 Gattern und nicht in der gesamten Schaltung!
- Es gibt (komplexere) Diagnose-Ansätze, die voll-automatisch (oder fast voll-automatisch) arbeiten.
- Außerdem würden wir, wenn wir die Analyse mit der korrigierten Schaltung wiederholen, wissen, dass die Implementierung nun korrekt ist und wir keine weiteren Fehler suchen müssen!

# Ist die Analyse den Aufwand wert?

- Es ist sehr kompliziert, BDDs aufzuschreiben und ihre Verknüpfungen auszurechnen.
- Dadurch ist es auch fehleranfällig.
  - Woher weiß man, dass man bei den BDD-Operationen keinen Rechen- oder Schreibfehler gemacht hat?
- **Antwort**: Software-Werkzeuge, die Manipulation von BDDs übernehmen!
  - Diese können BDDs mit Millionen Knoten repräsentieren und in vertretbarer Zeit verarbeiten.
  - Spezifikation und Implementierung werden aus Dateien eingelesen und BDDs automatisch erzeugt/verglichen.
  - Standard in modernen Entwurfsabläufen!

# Eigenschaftsprüfung

- **Ziel**: Nachweis, dass eine bestimmte Eigenschaft immer gilt, manchmal gilt oder nie gelten kann.
- Es gibt spezielle Sprachen, um solche Eigenschaften zu formulieren.
- An dieser Stelle soll das Vorgehen am Beispiel von Bus Contention auf Bus L von ReTI illustriert werden.
- Konkret wollen wir zeigen, dass Treiber *PCLd* und *0LD* nie gleichzeitig enabled sind.

# Zur Erinnerung: ReTI-Aufbau

# Eigenschaftsprüfung: Bus Contention (1/2)

- Weise nach, dass es keine Eingangsbelegung gibt, für die $/PCLdoe$ und $/0Ldoe$ beide aktiv sind.
  - $/PCLdoe$ enabled in der Execute-Phase ($E = 1$) für Befehle JUMP ($I[31:30] = 11$), Compute-Befehle ($I[31:30] = 00$) mit $D = PC$ ($I[25:24] = 00$) und für MOVE-Befehl ($I[31:28] = 1011$) mit $S = PC$ ($I[27:26] = 00$).
  - $/0Ldoe$ enabled in der Execute-Phase ($E = 1$) für Befehl LOADI ($I[31:28] = 0111$).

# Eigenschaftsprüfung: Bus Contention (2/2)

- **Eigenschaft**: Für alle Belegungen der relevanten Signale $(E, I_{31}, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})$ gilt stets:

  $(/PCLdoe(E, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})$
  $\vee\ /0Ldoe(E, I_{30}, I_{29}, I_{28}, I_{27}, I_{26}, I_{25}, I_{24})) = 1.$

  - **Bedeutung**: Zu jedem Zeitpunkt muss mindestens eines der Signale 1 sein, sie sind nie gleichzeitig 0 (aktiv).

- **Vorgehen**: Berechne BDD für $/PCLdoe$, BDD für $/0Ldoe$ und die OR-Verknüpfung der beiden BDDs. Diese muss 1 sein, d.h. aus nur einem 1-Terminalknoten bestehen.

  - Falls nicht, liefern die Pfade zum 0-Terminalknoten die Belegungen, für welche die Eigenschaft verletzt ist!

# Eigenschaftsprüfung: Diskussion

- Wir haben eine selbstverständliche Aussage bewiesen (Signale bei unterschiedlichen Befehlen aktiv).
  - Der Entwerfer hätte aber, etwa bei der Logik-Optimierung, einen Fehler machen können, die zur Verletzung der Eigenschaft geführt hätte.
- Wir sind noch lange nicht fertig.
  - Auf dem L-Bus und auf anderen Bussen gibt es noch viele Möglichkeiten, wie Bus Contention auftreten kann.
  - Selbst wenn wir sämtliche Quellen für Bus Contention ausschließen, wissen wir noch nicht, dass der Prozessor korrekt implementiert ist. Wir wissen nur, dass keine Bus Contention auftritt.

# Formale Verifikation in der Praxis

- Eigene Produktklasse der Entwurfsautomatisierungsindustrie (auch in Deutschland).
- Äquivalenzprüfung ist Routine-Bestandteil beim Entwurf von komplexen Hardware-Produkten.
- Eigenschaftsprüfung spielt im Hardware- und im Software-Bereich eine immer größere Rolle, der Einsatz ist jedoch mit Schwierigkeiten verbunden.
  - Formulierung von Eigenschaften erfordert hochqualifizierte (und teure) Mitarbeiter.
  - Es ist oft nicht klar, wann genug Eigenschaften erstellt wurden und das Produkt als korrekt anzusehen ist.

# Zusammenfassung: Formale Spezifikation

- Boolesche Funktionen, boolesche Ausdrücke, kombinatorische Schaltkreise und BDDs sind ineinander transformierbar.
- Formale Verifikation kann Entwurfsqualität enorm steigern und die Entwurfszeiten reduzieren.
- Ein Großteil des Aufwandes kann von einem Rechner geleistet werden.
- Voraussetzung dafür ist eine formal-mathematische Beschreibung, die der Rechner ohne menschliche Hilfe interpretieren kann.

# Kapitel 8 – Entwurfs- und Architekturkonzepte

1. **VLSI-Entwurf**
2. Rechnerarchitektur

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# VLSI-Entwurf

- Im Schnelldurchgang ...
- VLSI: Very Large-Scale Integration
- Entwurfs-Flow kann für Mikroprozessoren oder beliebige andere Schaltungen verwendet werden!

# Übersicht (1/6)

# Übersicht (1/6)

Textbeschreibung, formale Beschreibungssprachen

Gatter-Netzliste

Korrekter Entwurf

Zum späteren Testen nötige Daten und On-Chip-Blöcke

Maskendaten für Transistoren und Metallisierung

# Chip-Entwurf

# Übersicht (2/6)

# Spezifikation

- Ein digitales System kann auf verschiedene Weise beschrieben werden:
  - Beschreibung als Grafik (z.B. als Schaltplaneingabe),
  - Textuelle Beschreibung als Netzliste (z.B. EDIF),
  - Textuelle Beschreibung auf Basis einer Hardwarebeschreibungssprache (z.B. VHDL, Verilog)
  - ... oder auf noch höherer Abstraktionsebene.

# Übersicht (3/6)

# Synthese

- Aufgabe der Synthese ist das Umsetzen einer Spezifikation in eine (Schaltkreis-)Realisierung.
- Syntheseschritte:
  - High-Level-Synthese (Rechnerarchitektur)
  - Logiksynthese, meist noch technologieunabhängig (siehe auch Kap. 3.3 - Zweistufige Logiksynthese mit Quine/McCluskey für PLAs)
  - Technologiemapping: Abbildung einer technologieunabhängigen Netzliste auf eine Bauteilebibliothek eines Halbleiterherstellers.

# Übersicht (4/6)

# Simulation

- Aufgabe von Simulation und Verifikation ist der Nachweis, dass Spezifikation und Realisierung übereinstimmen.
- Validierung durch Simulation (unvollständig!)

# Verifikation

- Formale Verifikation ist der Beweis von Eigenschaften der Realisierung, durch:
  - Äquivalenztests (z.B. mit BDDs, siehe Kap. 7.3)
  - Model Checking (Nachweis von Eigenschaften, einfaches Beispiel siehe Kap. 7.3)
  - Automatentraversierung (z.B. endlicher Zustandsautomat, siehe Kap. 4.3)
- Diese und weitere Formalismen werden behandelt in der Vorlesung Rechnerarchitektur, sowie in diversen Spezialvorlesungen (Verifikation eingebetteter Systeme, Verifikation probabilistischer und hybrider Systeme, ...)

# Übersicht (5/6)

# Testen (1/2)

- Aufgabe des Testens ist die Überprüfung eines Chips auf Fehler nach der Fertigung des Chips.
- **Motivation**: Es ist nicht ungewöhnlich, dass $60 - 70\%$ der gefertigten Chips fehlerhaft sind.
- **Ziel**: Aussortieren der fehlerhaften Chips.

# Testen (2/2)

- Aufgabe des Testens ist die Überprüfung eines Chips auf Fehler nach der Fertigung des Chips.
- Daher vor der Fertigung der Chips: Maßnahmen zur Vorbereitung bzw. Erleichterung des Testens, z.B.
  - Berechnung von Testmustern für den späteren Fertigungstest
    - Testen der Funktionalität
    - Testen der Geschwindigkeit
    - ATPG = „Automatic Test Pattern Generation"
    - Fehlersimulation: Bewertung von gegebenen Testmustern
  - Designänderungen, um „Testbarkeit" und „Zuverlässigkeit" zu verbessern. (DFT = „Design for testability")
  - Zusatzhardware zum Testen (BIST = „Built-in Self-Test").
- (Ausführlich in Rechnerarchitektur und Spezialvorlesung Test und Zuverlässigkeit).

# Übersicht (6/6)

# Placement, Routing

- Die erzeugte Netzliste wird verwendet, um die Elemente der Schaltung zu platzieren und zu verdrahten.
- 1. Schritt:
  Geometrisches Platzieren vorgefertigter Standardzellen.
- 2. Schritt:
  Verdrahtung der Zellen, d.h. Herstellen leitender Verbindungen zwischen Standardzellen gemäß der Netzliste.
- Danach:
  Erzeugen von Maskendaten, verwendet zum Belichten der Chipherstellung.

# Beispiel: Placement, Routing

# Zusammenfassung

- Herausforderungen:
  - Chips mit Millionen/Milliarden von Transistoren in annehmbarer Zeit entwerfen,
  - gleichzeitig Entwurfskorrektheit garantieren,
  - neue Problemstellungen durch immer kleinere Strukturgrößen („nano scale”-Entwürfe),
  - ganze Systeme auf einem Chip.
- Designaufgabe bleibt beherrschbar bei
  - hierarchischem Vorgehen und Wiederverwenden vorhandener Entwürfe (Design Re-Use),
  - Entwurfsautomatisierung,
  - ausreichendem Aufwand für Entwurfskorrektheit.

# Kapitel 8 – Entwurfs- und Architekturkonzepte

1. VLSI-Entwurf
2. **Rechnerarchitektur**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur

WS 2016/17

UNI FREIBURG

# Rechnerarchitektur

- Designprinzipien
- Pipelining, Parallelverarbeitung
- Speicheroganisation

- Details s. Vorlesung Rechnerarchitektur

# Designprinzipien für Rechner

- Was sind die Schlüsseloperationen für diesen speziellen Rechnertyp?
- Entwurf des Datenpfades (Rechenwerk mit Verbindungsstruktur) für diese Schlüsseloperationen.
- Entwurf der Maschinenbefehle mit dem Ziel, dass möglichst jeder Befehl in einem Datenpfadszyklus (Fetch-Execute) ausführbar ist.
- Erweiterung des Befehlssatzes nur dann, wenn dadurch die Maschine nicht langsamer wird; ggfs anwendungsspezifische Zusatzhardware bereitstellen.
- Schnittstelle nach außen (Speicher, Input/Output)

# Pipelining

- Performanz von Rechnern lässt sich durch Pipelining steigern.
  - Prinzip der Fließbandverarbeitung
  - Probleme der Fließbandverarbeitung

# Das Prinzip an einem alltäglichen Beispiel

- Personen A, B, C und D kommen aus dem Urlaub; es ist viel schmutzige Wäsche zu waschen!
- Zur Verfügung stehen:
  - eine Waschmaschine (1/2 Stunde Laufzeit)
  - ein Trockner (1/2 Stunde Laufzeit)
  - ein Bügeleisen (1/2 Stunde Arbeit zum Bügeln)
  - ein Wäscheschrank (1/2 Stunde Arbeit zum Einräumen)
- jeder der Personen A, B, C, D aus dem Haushalt wäscht seine Wäsche selbst.
- Es gibt grundsätzlich zwei Möglichkeiten, die vier Waschvorgänge auszuführen!

# Das Prinzip an einem Beispiel

Dauer der Arbeiten:
8 Stunden

Mit Pipelining:

Dauer der Arbeiten:
$3\frac{1}{2}$ Stunden

# Aufteilung der Befehlsabarbeitung in Phasen

- Um Pipelining im Datenpfad ausnutzen zu können, muss die Abarbeitung eines Maschinenbefehls in mehrere Phasen mit möglichst gleicher Dauer aufgeteilt werden.
- Eine sinnvolle Aufteilung ist abhängig vom Befehlssatz und der verwendeten Hardware.
- Beispiel: Abarbeitung in 4 Schritten:
  - Befehls-Holphase (instruction fetch)
  - Dekodierphase / Lesen von Operanden aus Registern
  - Ausführung / Adressberechnung
  - Abspeicherphase (result write back phase)
- Pipelining ist schwierig, wenn die Dauer der Dekodier- und Ausführungsphase bei denen verschiedenen Maschinenbefehlen sehr unterschiedlich ist.

# Pipelining: Illustration

- **Annahme**: Aufteilung der Befehlsabarbeitung in 4 gleichlange Phasen.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Befehl 1: | P1 | P2 | P3 | P4 | | | | | | |
| Befehl 2: | | P1 | P2 | P3 | P4 | | | | | |
| Befehl 3: | | | P1 | P2 | P3 | P4 | | | | |
| Befehl 4: | | | | P1 | P2 | P3 | P4 | | | |
| Befehl 5: | | | | | P1 | P2 | P3 | P4 | | |
| Befehl 6: | | | | | | P1 | P2 | P3 | P4 | |
| Befehl 8: | | | | | | | P1 | P2 | P3 | P4 |
| Zeitschritt: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

# Pipelining: Speedup (1/2)

- **Annahmen**:
  - Abarbeitungszeit eines Befehls ohne Pipelining: $t$
  - $k$ Pipelinestufen, gleiche Laufzeit der Stufen

  $\Rightarrow$ Laufzeit einer Stufe der Pipeline: $\frac{t}{k}$

- Beschleunigung bei $m$ auszuführenden Instruktionen:

  $m = 1$: Laufzeit mit Pipeline $= k \cdot \frac{t}{k} = t$
  $\Rightarrow$ keine Beschleunigung

  $m = 2$: Laufzeit mit Pipeline $= t + \frac{t}{k} = (k+1)\frac{t}{k}$
  $\Rightarrow$ Beschleunigung um Faktor $\frac{2k}{k+1}$

# Pipelining: Speedup (2/2)

- $m \geq 1$: Laufzeit mit Pipeline $= t + (m-1)\frac{t}{k}$,
  Laufzeit ohne Pipeline $= m \cdot t$.

  $\Rightarrow$ Beschleunigung um $\frac{mt}{t+(m-1)\cdot\frac{t}{k}} = \frac{mk}{m+k-1} = k - \frac{k(k-1)}{m+k-1}$

- **Ergebnis**:

  Für $m >> k$ nähert sich der Speedup also der Anzahl $k$ der Pipelinestufen.

- Es wurde vorausgesetzt, dass sich die Ausführung der Befehle ohne weiteres „verzahnen" läßt. Dies ist in der Praxis nicht immer der Fall
  $\Rightarrow$ Pipelinehemmnisse, sog. Hazards!

# Pipelining: Zusammenfassung

- Beschleunigung um bis zu Faktor $k$ durch Einsatz einer Pipeline
  ($k$ = Anzahl der Pipeline-Stufen).
- Hazards verringern die Beschleunigung.
- Viele Möglichkeiten Hazards zu vermeiden, sowohl durch Software- als auch Hardwaremaßnahmen

# Übersicht: Formen der Parallelität

- Parallelität auf Bitebene: bis etwa 1985
  - Kombinatorische Addierer und Multiplizierer, etc.
  - wachsende Wortbreite auf 64 Bit
- Parallelität auf Instruktionsebene: 1985 bis heute
  - Pipelining der Instruktionsverarbeitung
  - Mehrere Funktionseinheiten (superskalare Prozessoren) bei mehr als 4 Funktionseinheiten werden Datenabhängigkeiten oft zum Hindernis für eine effiziente Ausnutzung.
  - Vektorprozessoren führen eine Operation parallel auf vielen Daten durch
- Parallelität auf Prozessor-/Rechnerebene
  - Nur so scheinen Beschleunigungen um Faktor 50 und mehr möglich.

# Speicherorganisation im Überblick

- **Fragen**:
  - Wie kann verhindert werden, dass ein Rechner durch Hauptspeicherzugriffe zu sehr verlangsamt wird?
  - Was passiert, wenn nicht alle Daten in den Hauptspeicher passen?
  - Welche sonstigen Speichermedien gibt es?
- **Begriffe**:
  - Speicherhierarchie
  - Cache
  - Virtueller Speicher, Verdrängungsstrategien
  - Festplatte, Flash, CDROM / DVD / Blu-Ray, Magnetband

# Gründe für komplexe Speicherorganisation

- Ein Zugriff auf eine Hauptspeicherzelle ist langsamer, als ein Zugriff auf ein Register.
  - Hauptspeicherzellen sind DRAM-Zellen (dynamische Speicherzellen), während Register in der Regel SRAM-Zellen (statische Speicherzellen) sind!
  - Bei einem Registerzugriff kommt man ohne Bus-Operation aus!
- **Idee**: Man stellt dem Prozessor einfach einige Mbyte Register zur Verfügung.
  - Aber: SRAM-Zellen sind wesentlich größ er als DRAM-Zellen (Faktor 3-4).
  - So abwegig ist die Idee nicht!
  - Mit der weiteren Technologieentwicklung (noch kleinere Strukturen) wird die verfügbare Chip-Fläche vorwiegend dazu benutzt werden, um schnellen Speicher zu integrieren.

# Speicherorganisation heute

Programm
1-8 bytes

Cache
8-128 bytes

OS
512-4K bytes

Benutzer
MByte, GByte, TByte

# Speicherhierarchie (1/3)

| | | |
|---|---|---|
| **Geschwindigkeit:** | Am schnellsten | Am langsamsten |
| **Größe:** | Am kleinsten | Am größten |
| **Kosten:** | Am teuersten | Am billigsten |

# Speicherhierarchie (2/3)

| | | |
|---|---|---|
| **Geschwindigkeit:** | Am schnellsten | Am langsamsten |
| **Größe:** | Am kleinsten | Am größten |
| **Kosten:** | Am teuersten | Am billigsten |

# Speicherhierarchie (3/3)

- Grundprinzip:
  - Wenig schneller und teurer Speicher
  - Viel langsamer und billiger Speicher
  - Speicherhierarchie
- Wieso funktioniert das?
  - Wegen Prinzip der Lokalität!

# Prinzip der Lokalität (1/2)

Typische Programme verbringen 80% der Ausführungszeit in etwa 10% des Gesamtprogramms.

Anteil der Instruktionen des Gesamtcodes, die 80% (gelb) bzw. 90% (rot) der Ausführungszeit ausmachen.

Patterson, Hennessey Computer Architecture a Quantitative Approach

# Prinzip der Lokalität (2/2)

- Es gibt zwei Arten von Lokalität:
  - Zeitliche Lokalität (Locality in Time): Wenn ein Datum verwendet wird, wird es auch bald wieder verwendet werden.
  - Räumliche Lokalität (Locality in Space): Wenn ein Datum verwendet wird, werden auch die Adressen in dessen Nähe bald verwendet werden.
  - Siehe z.B. Schleifen …
- Wegen Lokalität kommt man auf höheren Ebenen der Speicherhierarchie mit weniger Speicher aus.

# Speicherorganisation heute

Programm
1-8 bytes

Cache
8-128 bytes

OS
512-4K bytes

Benutzer
MByte, GByte, TByte

# Überblick

| | | | |
|---|---|---|---|
| Kap. 0 Einführung/Umfeld | 8 | | |
| | 7 | Formale Spezifikation v. Hardware: Boolesche Ausdrücke, BDDs | Formale Verifikation, ReTI ALU |
| | 6 | Fehlertoleranz fehlererkenn./fehlerkorrig. Codes | Parity, Hamming Code |
| | 5 | Physikalische Eigenschaften Timing | ReTI: Exaktes Timing |
| | 4 | Sequentielle Logik | ReTi: Datenpfade, Idealisiertes Timing |
| | 3 | Komb. Logik Minimalpolynome Arithmetik | ReTI: ALU |
| | 2 | Kodierung von Zeichen/Zahlen | ReTI: Befehls-Kodierung |
| | 1 | ReTI abstrakt | Kap. 12 |
| | | Mathem. Grundlagen | Kap. 11 |



UNI FREIBURG

# Überblick

| | | | |
|---|---|---|---|
| Kap. 0 Einführung/Umfeld | 8 | Entwurfs- und Architekturkonzepte | |
| | 7 | Formale Spezifikation v. Hardware: Boolesche Ausdrücke, BDDs | Formale Verifikation, ReTI ALU |
| | 6 | Fehlertoleranz fehlererkenn./fehlerkorrig. Codes | Parity, Hamming Code |
| | 5 | Physikalische Eigenschaften Timing | ReTI: Exaktes Timing |
| | 4 | Sequentielle Logik | ReTi: Datenpfade, Idealisiertes Timing |
| | 3 | Komb. Logik Minimalpolynome Arithmetik | ReTI: ALU |
| | 2 | Kodierung von Zeichen/Zahlen | ReTI: Befehls-Kodierung |
| | 1 | ReTI abstrakt | Kap. 12 |
| | | Mathem. Grundlagen | Kap. 11 |