


TI-Zusammenfassung

- Theorie
- Praxis

Theorie

- Implikant
 - Eins-Stelle im KV-Diagramm
- Primimplikant
 - Maximal großer Eins-Block
- Kernprimimplikant
 - Überdeckt einen Minterm, der von keinem anderen Primimplikanten überdeckt wird
- Minterme
 - Ein Implikant heißt Minterm, wenn jede Variable der Funktion nur einmal in ihm vorkommt
- DNF
 - disjunktive Normalform
 - Disjunktion von Konjunktionen in minimaler Form
- KMF
 - nicht eindeutig
 - minimale Konjunktion von disjunktiven Funktionsgliedern
- NAND/NOR
 - Induktiver Aufbau: Vorgänger negieren und mit neu hinzugekommener Variable verknüpfen (rotes Skript S.20)


- nMos-Transistor
 - leitet bei 1, sperrt bei 0
 - Source oben, Gate Mitte, Drain unten
 -  Oli sagt: „Wenn am nMos eine Eins anliegt, dann leitet er. Der Pfeil zeigt zum Gate wie eine liegende Eins.“
- pMos-Transistor
 - leitet bei 0, sperrt bei 1
- Ground
 - niedriger Spannungspegel (0 Volt)
 - wird mit GND bezeichnet
 - logische 0
- Versorgungsspannung

- hoher Spannungspegel (5 Volt)
- wird mit V_{dd} bezeichnet
- logische 1

- Hasardfehler
 - nicht monotone Folge: min. 2 Variablen ändern sich
 - statische
 - Anfangs- und Endwert sind gleich
 - statischer-0-Hasard, wenn der Wert 0 beträgt
 - statischer-1-Hasard, wenn der Wert 1 beträgt
 - dynamische
 - Anfangs- und Endwert sind unterschiedlich
 - dynamischer-01-Hasard, wenn der Anfangswert 0 und der Endwert 1 beträgt
 - dynamischer-10-Hasard, wenn der Anfangswert 1 und der Endwert 0 beträgt

- Multiplexer (MUX)
 - Stellt den Kanal durch, der oben ausgewählt wird

- Takt
 - synchronisiert die einzelnen Bausteine
- Schaltnetz
 - kombinatorische Schaltung
- Schaltwerk
 - sequentielle Schaltung
- Pegelsteuerung
 - Eingänge wirken sich nur dann auf den Zustand aus, wenn der Takt gerade 1 ist, ist er hingegen 0 dann werden die Eingangswerte gespeichert
 - Pegelgesteuerte Zustandspeicher werden als Latches bezeichnet
- Flankensteuerung
 - Man nutzt nur eine Taktflanke des gesamten Taktes
 - positive Taktflanke ($0 \rightarrow 1$)
 - wird als $\triangleright C1$ am Flipflop bezeichnet
 - negativ Taktflanke ($1 \rightarrow 0$)
 - wird als $\circ \triangleright C1$ am Flipflop bezeichnet

- Flipflop
 - Verzögerungsglied im Totzeitmodell
 - speichert eine Variable zwischen, wenn Nullen anliegen
 - Speichert eine 1, wenn vorher ein set durchgeführt wurde
 - Durch den Takteingang wird das Verhalten bestimmt
 - Zustandsspeicher durch Rückkopplung
- RS-Flipflop
 - hat 2 Eingänge: r (reset), s (set)
 - $q^t=0 \wedge s=1 \Rightarrow q^{(t+1)}=1$ Ist die Zustandsvariable auf 0, dann wird sie nun auf 1 gesetzt (set)
 - $q^t=1 \wedge r=1 \Rightarrow q^{(t+1)}=0$ Ist die Zustandsvariable auf 1 gesetzt (set), wird sie wieder auf 0 zurückgesetzt (reset)
 - Nebenbedingung: $r \wedge s=0$
- D-Flipflop
 - Verzichtet auf 2 Eingangsvariablen, da nach der Nebenbedingung r und s verschieden sind
 - Eine Eingangsvariable ist d die andere $\neg d$
 -  Oli sagt: „Direct' Delay, also $q^{t+1}=d^t$!“
- JK-Flipflop
 - Erweitert um $r=s=1$, was als Wechsel (toggle) genutzt wird
 - Eingangsvariablen werden als j (set) und k (reset) bezeichnet
 - charakteristische Gleichung des JK-Flipflops $q^{t+1}=(j\bar{q} \vee \bar{k}q)^t$
- T-Flipflop
 - hat nur einen Eingang t (toggle)
- MSB (Most Significant Bit) = Vorzeichenbit (erste Stelle von links beim Zweierkomplement und IEEE)
- Gleitkommazahl: MSB|Charakteristik,Mantisse
- Normalisierung
 - $\frac{1}{b} \leq 0, \text{Mantisse} < 1$ In dualer Darstellung ist die erste Zahl nach dem Komma gleich 1
 - 0,5 dezimal = 0,1 dual (binär)
- maxreal
 - größte darstellbare normalisierte Gleitkommazahl
- minreal
 - kleinste darstellbare normalisierte Gleitkommazahl

- smallreal
 - kleinste Gleitkommazahl, die man zu 1 addieren kann, um einen von 1 verschiedenen Wert zu erhalten
- IEEE
 - Normalisierung: 1 vor dem Komma implizit
 - Nur, wenn die Charakteristik nicht nur Nullen enthält
- MIMA
 - Lesephase:
 1. Takt: IAR --> SAR; IAR --> X; R=1
 2. Takt: Eins --> Y; R = 1
 3. Takt: ALU auf Addieren; R=1
 4. Takt: Z --> IAR
 5. Takt: SDR --> IR
 - Dekodierphase:
 6. Takt: D = 1
 - Ausführungsphase
 - Es gibt verschiedene Arten von Befehlen: Welche mit Adressen oder Konstanten als Parameter oder parameterlose Befehle
 - Mikroprogramme für Befehle mit Konstante als Parameter (LDC)
 - Konstante liegt nach Dekodierphase in IR
 - Erste 4 Bit im IR Befehlsopcode, restliche 20 Bit enthalten den Parameter
 - Da der Bus zum SAR nur 20 Bit beträgt, werden genau die ersten 4 Bit OpCode abgeschnitten, übrig bleibt die Parameter-Adresse
 - Beim Lesen/Schreiben aus/in dem/den Speicher werden immer 3 Takte benötigt, da Speicher prinzipiell langsamer ist
 - Befehle ohne Parameter beginnen mit F
- MIPS-Assembler:
 - Format: Befehl normal: dest, source1, source2; die mit i endenden Befehle greifen nicht auf ein Register zu, sondern direkt auf eine Zahl (i=immediate)
 - Logische Befehle: and, andi, nor, not, xor, xori, or, ori, rol(rotate left), ror, sll(shift left logical, rd,rs,imm=distance)), sllv(~variable), sra(shift right arithmetic, rd, rs,imm=distance), srlv(shift right logical)
 - Unterschied von sra, sll
 - a=arithmetic, l=logical; a erhält das MSB=Vorzeichen

- Laden von Konstanten:
 - li rd,imm
 - lui rd,imm (load upper immediate) (load lower halfword von imm in upper halfword of rd, rest ist 0)
 - WORD=32bit DATUM, halfword=16bit
 - Vergleichsbefehle (von 2 Registern)
 - slt, sltu (u=unsigned) (set less than), slti, sltiu, seq, sge, sgeu, sgt, sgtu, sle, sleu, sne
 - Kontrollfluß
 - unbedingt
 - j target, b label, jal target (jump and link, speichern der nachfolgenden Adresse),
 - bedingt (Vergleich zweier Register)
 - bgt, bgtu, blt, bltu, bge, bgeu, ble, bleu, beq, bnq, bgtz, bltz (z=zero), beqz, bnez, bne
 - Lade & Speicherbefehle
 - la rd, adress (load adress)
 - lw rt, adress (load word)
 - sw rt, address
- Pipelining
 - Parallele Verarbeitung durch Teilprozesse
 - Für jeden Teilprozess gibt es eine spezielle Ausführungseinheit
 - IF
 - Instruction Fetch
 - Befehl holen
 - ID
 - Instruction Decode & Register Fetch
 - Befehl dekodieren
 - Operanden aus Universalregister laden (2. Takt Hälfte)
 - EX
 - Execution/Effective Address Calculation
 - Befehl ausführen
 - Bei Lade- und Speicherbefehlen oder Verzweigungen berechnet die ALU die effektive Adresse
 - MEM
 - Memory Access
 - ggf. in den Speicher schreiben oder lesen

- WB
 - Write Back
 - Ergebnis in Universal-Register schreiben (1. Takthälfte)
 - Befehle ohne Ergebnis durchlaufen diese Phase passiv)
- Datenabhängigkeiten
 - $Inst_2$ wird nach $Inst_1$ ausgeführt (Achtung: auch wenn zwischen beiden Instruktionen andere Instruktionen ausgeführt werden, ist es eine DA)
 - echte Datenabhängigkeit (true dependence)
 - Die Instruktion $Inst_1$ hat eine echte DA zu $Inst_2$, wenn $Inst_1$ seine Ausgabe in ein Register schreibt, das von $Inst_2$ gelesen wird
 - erzeugt Lese-nach-Schreibe Konflikt (read after write, RAW)
 - falsche Datenabhängigkeiten
 - Gegenabhängigkeit (antidependence)
 - $Inst_2$ überschreibt ein Register, das von $Inst_1$ gelesen wurde
 - erzeugt Schreibe-nach-Lese-Konflikt (write after read, WAR)
 - Ausgabeabhängigkeit (output dependence)
 - $Inst_1$ schreibt in ein Register, das anschließend von $Inst_2$ überschrieben wird
 - erzeugt Schreibe-nach-Schreibe-Konflikt (write after write, WAW)
 - WAR und WAW können in der DLX-Pipeline nicht auftreten
 - Lesen immer in der Stufe 2
 - Schreiben immer in Stufe 5
- Datenkonfliktlösung
 - Software-Lösungen
 - NOOP/NOP-Befehle einfügen
 - Befehlsumordnungen
 - Hardware-Lösungen
 - Pipeline-Sperrung (interlocking)
 - Pipeline-Leerlauf (stalling)
 - Forwarding
 - Anstatt aus dem Universal-Register wird das ALU-Ausgaberegister der vorherigen Operation in das Eingaberegister übertragen
 - Result Forwarding
 - Rückführung des ALU-Ausgaberegisters
 - Load Forwarding

- garantierte Konsistenz, aber langsame Zykluszeit und Belastung des Systembus'
- gepuffertes Durchschreibverfahren (buffered write through policy)
 - Milderung der Nachteile durch einen Schreib-Puffer
 - Übertragung der Daten parallel zu weiteren Operationen des Prozessors
- Rückschreib-Verfahren (write back policy)
 - Datum wird nur in den Cache geschrieben und mit einem speziellen Bit (altered bit, modified bit, dirty bit) gekennzeichnet
 - Dirty Bit wird gesetzt, wenn ein Datum verdrängt wurde, also der Cache und Speicher Unterschiede für diesen Tag aufweisen
 - Valid Bit zeigt an, dass ein Block im Cache einen Eintrag hat
 - Der AS ändert sich nur, wenn ein so gekennzeichnetes Datum aus dem Cache verdrängt wird
 - Schreibzugriffe können auch mit der schnellen Zykluszeit ausgeführt werden, jedoch entstehen Konsistenzprobleme zwischen Cache und AS
- Speicher
 - RISC-Prozessor
 - Prozessor mit einem reduzierten Befehlssatz.
 - Direkter Speicherzugriff (DMA)
 - Datentransfer ohne Beteiligung des Prozessors.
 - Translation-Lookaside-Buffer (TLB)
 - Ein schneller vollassoziativer Cache zur Beschleunigung der Umsetzung virtueller in physikalischen Adressen. Der TLB speichert die zuletzt benutzten Einträge aus dem Seitentabellenverzeichnis und den Seitentabellen. Im Trefferfall muss nicht auf die im Hauptspeicher liegenden Tabellen zugegriffen werden.

Praxis

- Eins- und „don't care“-Stellen gegeben
 - Tragen Sie alle Primimplikanten ins KV-Diagramm
 - KV-Diagramm aufstellen
 - Mit 2 Feldern (Feld 0 und Feld 1) beginnen
 - Feld 0 ist \bar{a} , Feld 1 ist a
 - Nun wird an der unteren Außenkante des Diagramms gespiegelt
 - D.h. bei jeder Spiegelung kommt eine Variable hinzu, d.h. die neuen Felder sind b
 - Bei der ersten Spiegelung ist das logischerweise b
 - Die neuen Felder werden nun weiter nummeriert
 - Dort, wo die gespiegelte 0 landen würde, macht man nun mit der

2 weiter, entsprechend wird das Feld daneben Feld 3

- Die neu hinzugekommen Felder sind b , die alten sind \bar{b}
- Für c wird nun an der rechten Außenkante gespiegelt und entsprechend nummeriert, nicht vergessen, dass a nun verlängert werden muss, die neuen Felder sind wieder c
- Für d muss nun wieder an der unteren Außenkante gespiegelt werden usw...
- Einsen an die Eins-Stellen schreiben
- „-“ an die „don't care“-Stellen
- (Nullen an alle restlichen Stellen)
- Primimplikanten markieren
 - Maximal große Einsblöcke suchen („don't care“-Stellen werden hier als Einsen betrachtet, es muss mindestens eine Eins im Block enthalten sein)
 - **Wenn jede Variable nur einmal im Block vorkommt, d.h. keine ungerade Anzahl an Einsblöcken, Symmetrie beachten, Block markieren**
- Geben Sie eine disjunktive Minimalform (DMF) an
 - Primimplikanten raus schreiben
 - Spalten und Zeilen der Blöcke checken, welche Variable enthalten ist
 - Wenn b und \bar{b} eine 1 enthalten, spielt b keine Rolle
 - Die enthaltenen Variablen mit \wedge verknüpfen, es entsteht eine Klausel
 - Die einzelnen Klauseln mit \vee verknüpfen
 - Überflüssige Klauseln entfernen
- Tragen Sie alle Primimplikate ins KV-Diagramm ein
 - s.o.
 - Primimplikate markieren
 - Maximal große Nullblöcke suchen („don't care“-Stellen werden hier als Nullen betrachtet, es muss mindestens eine Null im Block enthalten sein)
 - **Wenn jede Variable nur einmal im Block vorkommt, d.h. keine ungerade Anzahl an Nullblöcken, Symmetrie beachten, Block markieren**
- Geben Sie eine konjunktive Minimalform (KMF) an
 - Primimplikate raus schreiben
 - Spalten und Zeilen der Blöcke checken, welche Variable enthalten ist
 - Wenn b und \bar{b} eine 0 enthalten, spielt b keine Rolle

- Die enthaltenen Variablen mit \vee verknüpfen, es entsteht eine Klausel
 - Die einzelnen Klauseln mit \wedge verknüpfen
 - Überflüssige Klauseln entfernen
- Kernprimimplikanten gegeben
 - Welche der folgenden Produktterme können keine Primimplikanten der Funktion sein?
 - Kleines KV-Diagramm zur Hilfe zeichnen
 - Wird mehr als ein Minterm zweier Primimplikanten umfasst?
 - Bleibt jeder Kernprimimplikant noch Kernprimimplikant? „Umbruch“ an den Außenkanten beachten.
- Funktionen durch Minterme gegeben
 - Personalisieren Sie den PLA-Baustein
 - Kleines KV-Diagramm zur Hilfe zeichnen
 - Disjunktive Minimalform ablesen
 - (Funktionen checken, ob sie die gleichen Klauseln wie andere Funktionen enthalten, dann die Klauseln durch den Funktionsnamen ersetzen)
 - Danach entsprechend der Klauseln Leitungskreuzungen in der UND-Matrix und entsprechend der Funktionen Leitungskreuzungen in der ODER-Matrix einzeichnen
 - Taucht eine Funktion in einer anderen auf, einfach die gleichen Punkte erweitert um die restlichen Klausel-Leitungskreuzungen einzeichnen
 - Geben Sie eine DNF an
 - Entwicklung nach Shannon
 - $y = f(x_1, x_2, x_3, \dots)$
 - Man beginnt mit x_1 und klammert x_1 und $\neg x_1$ aus den entsprechenden Literalen aus
 - Hierbei beachten, dass sowohl x_1 , als auch $\neg x_1$ in Ausdrücken wie x_2 and x_3 ausgeklammert werden müssen
 - Danach in allen Klammern die nächsten Variable ausklammern, usw.
- Gegeben ist eine Überdeckungstabelle einer Schaltfunktion mit Mintermen, die Zeilenbezeichnungen entsprechen den Primimplikanten
 - Ist die Schaltfunktion vollständig oder unvollständig definiert?
 - mögliche Begründung
 - Primimplikanten bei vollständig definierten Funktionen überdecken 2^n Minterme.

- Die Zeilen checken, ob es Zeilen mit einer ungeraden Anzahl an Mintermen gibt
 - Beachte: $2^0=1$ Minterm-Anzahl ist zulässig
- Sind Primimplikanten in einem anderen Primimplikanten enthalten, wären beide bei einer vollständig definierten Schaltfunktion keine Primimplikanten, dies stellt ein Widerspruch dar
- Bestimmen Sie alle DMF
 - Schema
 - Kernprimimplikanten: Streichen der überdeckten Minterme dieser Kernprimimplikanten
 - Zeilendominanz: wird von dominiert und kann gestrichen werden.
 - Reduzierte Überdeckungstabelle
 - Spaltendominanz: Minterm dominiert Minterm und kann gestrichen werden.
 - Überdeckungsfunktion:
 - $\ddot{u}=(E \vee F)(A \vee C)....$
 - Disjunktive Minimalformen: $y=B \vee D \vee ...$
 - Kernprimimplikanten
 - Die Primimplikanten, die Minterme überdecken, die von keiner anderen Funktion überdeckt werden, streichen
 - d.h. wenn die Zeile ein Kreuz in einer Spalte als einzige hat, streichen
 - Zeilendominanz
 - Wenn alle Kreuze einer Zeile auch in einer anderen vorkommen, spricht man von einer Zeilendominanz, die Zeile mit weniger Kreuzen kann gestrichen werden
 - Reduzierte Überdeckungstabelle
 - Diese resultiert aus den ersten beiden Schritten
 - Spaltendominanz
 - Wenn alle Kreuze einer Spalte in einer anderen Spalte vorkommen, wird die Spalte mit mehr Kreuzen gestrichen
 - Überdeckungsfunktion
 - Klammer auf
 - Die erste Spalte anschauen, die Primimplikanten, in deren Zeile ein Kreuz steht, werden mit ODER verknüpft
 - Klammer zu und mit der nächsten Spalte fortfahren, die einzelnen Klammern mit UND verknüpfen
 - ausklammern und reduzieren

- DMF
 - Man verODERT die Kernprimplikanen und die beiden Literale der Überdeckungsfunktion, wobei aus einem UND ein ODER wird
 - Verschiedene Gatter (NOR, NAND, NOT)
 - Realisieren sie die gegebene Schaltfunktion in CMOS
 - Jedes Gatter impliziert eine Funktion (negiertes ODER, negiertes UND und negieren)
 - Schaltfunktion mit Boolescher Algebra umwandeln bis man nur noch die implizierten Funktionen benötigt
 - NAND herleiten:
 - Funktionstabelle aufstellen
 - Merke: pMos nach oben, nMos nach unten
 - Wenn $a=0$ oder wenn $b=0$, dann ist $y=1$ (Ausgabe)
 - Wir zeichnen einen pMos für a und einen für b
 - V_{dd} an Source anschließen
 - Eingangsvariable an Gate
 - Ausgabe auf y (Leitungskreuzung nicht vergessen)
 - Der letzte Fall $a=1$ und $b=1$ wird durch zwei hintereinandergeschaltete nMos-Transistoren realisiert
 - GND an Drain anschließen
 - Source des a-nMos an Drain des b-nMos anschließen
 - Der erste nMos schaltet bei 1 durch, dann wird nur bei einer weiteren 1 auf y durchgeschaltet
 - a und b mit den nMos-Transistoren verbinden
 - NOR aufstellen:
 - Entsprechend wie NAND
 - Hintereinandergeschaltete Transistoren sind diesmal 2 pMos
 - NOT aufstellen:
 - Ein pMos und ein nMos
 - Gatter entsprechend der umgewandelten Schaltfunktion verbinden
 - Checken, ob genausoviele nMos- wie pMos-Transistoren in der CMOS-Schaltung verbaut wurden
- Gegeben ist ein 8:1-Multiplexer und ein Inverter
 - Realisieren sie die Funktion $p=odd(w, x, y, z)$, die die anliegenden Eingangsvariablen auf ungerade Parität überprüft
 - MUX zeichnen

- Wegen drei Eingangsvariablen $\begin{matrix} 0 & 0 \\ 1 & \Rightarrow G- \\ 2 & 7 \end{matrix}$ oben in den MUX eintragen
 - Ausgabe $odd(w, x, y, z)$
 - Bei einer Paritätsprüfung bildet man quasi die binäre Quersumme, sprich das Ergebnis kann entweder 0 oder 1 betragen
 - d.h. wenn wir drei Variablen als 0,1,2 -Steuereingänge anschließen, wissen wir welche Belegungen es gab
 - Beispiel: z an 0, y an 1 und x an 2, z=1, y=0, x=0, dies entspricht der Zahl $1*2^0+0*2^1+0*2^2=1$, folglich wissen wir, wenn der Eingang 1 durchgeschaltet wird, dass wir eine 1 hatten, hat die letzte Eingangsvariable w auch eine 1 dann hätte es gerade Parität ($p=0$), wäre w eine 0 sei, dann wäre es ungerade ($p=1$). Da wir an die Eingänge nur noch w und $\neg w$ anschließen können, müssen wir, damit wir aus $w=1$ ein $p=0$ bekommen den Eingang negieren
 - Wir schließen an alle Eingänge die durch eine gerade Anzahl (z, x, y) an Einsen geschaltet werden können w an
 - Also: 0, 3, 5, 6
 - Wir schließen an alle Eingänge die durch eine ungerade Anzahl (z, x, y) an Einsen (1 Eins oder 3 Einsen) geschaltet werden können $\neg w$ an
 - Also: 1, 2, 4, 7
 - x = 1, y = 0, z = 0
 - x = 0, y = 1, z = 0
 - x = 0, y = 0, z = 1
 - x = 1, y = 1, z = 1
- Gegeben ist eine Schaltfunktion
 - Geben Sie eine Realisierung der Funktion an, die frei von allen statischen Strukturhasards ist
 - Satz von Eichelberger: Disjunktion aller Primimplikanten oder Konjunktion aller Primimplikate ist frei von statischen Strukturhasards (und allen dynamischen SH, wenn nur eine Eingabevariable wechselt)
 - KV-Diagramm zeichnen
 - alle Primimplikate rauslesen und DNF aufstellen
 - Schaltnetz zeichnen
- Gegeben ist ein synchrones Schaltwerk mit flankengesteuerten JK-Flipflops
 - Wieviele Zustände kann das Schaltwerk haben?

- 2 pro Flipflop, 2 Flipflops $\Rightarrow 2^2=4$ Zustände
- Stellen sie die kodierte Ablaufabelle schrittweise dar
 - Ansteuerfunktionen der Flipflops bestimmen
 - Ansteuerfunktionen aus dem Schaltbild ablesen
 - z.B.: $j_B^t = (\bar{A}^t \neq x^t)$
 - Ausgabefunktion aus dem Schaltbild ablesen
 - Zustandsübergangsgleichungen nach der charakteristischen Gleichung des JK-Flipflops $q^{t+1} = (j\bar{q} \vee \bar{k}q)^t$ entwickeln
 - q^{t+1} bezeichnet den nächsten Zustand, q den aktuellen
 - Entsprechend des Schaltbilds q durch B und j durch j_B^t ersetzen usw.
- Kodierte Ablaufabelle erstellen

Zustand			Eingabe		Folgezustand		Ausgabe
A^t	B^t	x^t	A^{t+1}	B^{t+1}			y^t
0	0	0	0	1			1
usw.							

- Alle Möglichkeiten von Zustands- und Eingangsbelegung durchspielen und dementsprechend nach oben entwickelnden Funktionen Folgezustände und Ausgabe bestimmen
- Automatengraph eines synchronen Schaltwerks gegeben
 - Vervollständigen Sie die die Tabelle der Zustands-, Eingabe- und Ausgabefolgen des Schaltwerks
 - Gemäß der Eingabe (e) und Ausgabe (a) sind die Übergänge eindeutig bestimmbar
- Schaltwerk mit Flipflops gegeben
 - Die Erklärung ist implizit in den vorangegangenen Aufgabenschemata gegeben
 - Anmerkungen
 - Taktflanken beachten (Wann ändert sich etwas?)
 - ggf. gegebene Verzögerungen beachten und die Verlaufsänderungen dementsprechend dem Takt nachgestellt eintragen
 - prinzipiell nach links schauen, da im Moment der relevanten Taktflankenänderung der Wert, der vorher anlag relevant ist
- Dezimalzahl (Basis 10) in Zahl anderer Basis umwandeln
 - Euklidischer Algorithmus
 - Die höchste Potenz der Basis suchen, die noch kleiner ist als die Zahl

- Dann die Dezimalzahl durch Basiszahl hoch dieser Potenz teilen
- Ganzzahlanteil als erste Ziffer notieren
- Rest durch die nächsthöhere Potenz teilen, Ganzzahlanteil als nächste Ziffer notieren usw.
- Bei Ganzzahlanteil von Null, mit negativem Exponenten (-1, -2...) weiterfahren, bis eine genügend hohe Genauigkeit erlangt wurde
- Abgewandeltes Homerschema
 - Ganzzahlanteil durch Basiszahl teilen, Rest notieren
 - Was ist die größte Potenz der Basiszahl, die noch kleiner als der Ganzzahlanteil der Dezimalzahl ist?
 - Wie oft passt die Basiszahl hoch dieser Potenz in den Ganzzahlanteil?
 - Was ist die größte Potenz der Basis für den Rest und wie oft passt sie in ihn? usw.
 - Faktoren als Ziffern des Ganzzahlanteils des Ergebnisses notieren
 - Beachte: „Lücken“ in den Potenzen tauchen als 0 im Ergebnis auf (Faktor 0)
 - Ganzzahlanteil der Division durch Basiszahl teilen, Rest notieren usw. bis der Rest kleiner als die Basiszahl ist
 - Die notierten Reste von „unten nach oben“, d.h. den letzten Rest zuerst notieren, dann den vorletzten als zweiter Teil usw., dies ergibt den neuen Ganzzahlanteil
 - Nachkommaanteil mit Basiszahl multiplizieren, Ganzzahlanteil notieren
 - Nachkommanteil des Ergebnisses der Multiplikation mit Basiszahl multiplizieren, Ganzzahlanteil notieren usw. bis eine „genügend hohe Genauigkeit“ erreicht wurde
 - Die Ganzzahlanteile „von oben nach unten“ notieren und als Nachkommaanteil des Ergebnisses verwenden
 - ggf. Periode notieren und abbrechen oder bei 0 als Nachkommaanteil abbrechen
 - Anmerkung
 - Bei Basen größer 10 haben wir natürlich nicht genügend Ziffern, hier verwenden wir Großbuchstaben gemäß des Alphabets als Zifferersatz (A, B, C...)
- Umwandlung einer Zahl zur Basis 8 in eine Zahl zur Basis 16
 - Umwandlung der Zahl zur Basis 8 in eine zur Basis 2
 - Vom Komma links aus
 - erste Ziffer binär umwandeln
 - von links erste Stelle 2^0 , zweite 2^1 usw.
 - zweite Ziffer von rechts umwandeln usw.

- Vom Komma rechts aus
 - erste Ziffer von links umwandeln usw.
- Umwandlung der Binärzahl in eine Zahl zur Basis 16
 - Vom Komma links aus
 - Immer 4 Ziffern werden zu einer Ziffer der neuen Zahl in der Zielbasis (16)
 - Wenn in der letzten Gruppe nicht genügend Ziffern zur Verfügung stehen (Anzahl < 4), dann wird mit Nullen aufgefüllt (voranstellen)
 - Vom Komma rechts aus ebenso (Nullen hintenanstellen)
- Umwandlung einer negativen Dezimalzahl ins Zweierkomplement
 - positive Dezimalzahl binär umwandeln
 - Alle Ziffern bis auf das MSB (1. von links) „toggeln“, d.h. eine 1 wird zu 0, eine 0 zu 1
 - 1 addieren
 - von rechts für eine 1 eine 0 schreiben, solange bis die nächste Ziffer eine 0 ist, dort eine 1 schreiben (Übertrag)
 - MSB ist 1, wegen negativer Zahl
- Umwandlung einer ZK-Zahl in eine Dezimalzahl
 - 1 abziehen, MSB bleibt
 - toggeln und ggf. (MSB=1) ein „-“ vor die Zahl setzen
- Umwandlung einer IEEE-Gleitkommazahl in eine Dezimalzahl
 - Vorzeichenbit (erstes Bit von links) ablesen
 - Charakteristik (8 Bit rechts vom VZ) ablesen
 - Char. in Dezimalzahl umwandeln
 - Exponent = Char – 127 ausrechnen
 - Mantisse rechts von der Char. ablesen
 - $Zahlenwert = (-VZ) * (1, Mantisse) * 2^{Exponent}$ ausrechnen und in Dezimalzahl umwandeln
- Gegeben sind Eingangs- und Ausgangsvariablen, „generate carry“-Funktion, „propagate carry“-Funktion, Summenfunktion und Übertragungsfunktion (und \ddot{u}_0)
 - Geben Sie \ddot{u}_x (x. Übertragungsfunktion) eines 4-Bit-Carry Look-ahead-Addierers an
 - Die Funktionen rekursiv verschachteln bis man auf \ddot{u}_x gekommen ist
 - Beachte: bei $a_i \cdot b_i$ liest man aus A von rechts die a_i ab
 - ist $a_i=0$ dann ist $a_i \cdot b_i=0$, ansonsten hängt es von b_i ab
 - Beachte: Antivalenz ergibt 1, wenn beide Operanden

unterschiedlich sind

- Wieviele Prüfbits sind für eine Einzelbit-Fehlerkorrektur in einem m-Bit Datenwörtern erforderlich?
 - $2^k \geq m+k+1$
- Welche Länge haben Zeichencodierungen im Unicode? Wie ändert sich die Größe bei der Konvertierung einer 8-Bit-Datei in Unicode?
 - 16 Bit, verdoppelt sich
- Was ist der Unterschied zwischen einem *Carry Ripple*-Addierer und einem *Carry Look-ahead*-Addierer? Wovon hängt die Additionszeit beim *Carry Ripple*-Addierer ab?


<i>Carry Ripple</i> -Addierer	<i>Carry Look-ahead</i> -Addierer
Bei einer Addition einer Stelle muss auf den Übertrag der vorhergehenden Stelle gewartet werden. Die Additionszeit ist proportional zur Anzahl der Stellen.	Alle Überträge werden direkt aus den Eingangsvariablen berechnet.

- Aus welchen Komponenten besteht ein von Neumann-Rechner?
 - Steuerwerk, Rechenwerk, Speicher, Bus und Eingabe-/Ausgabe-Einheiten
- Welches Register enthält den aktuell ausgeführten Befehl?
 - Befehlsregister
- Aus welchem Register entnimmt das Steuerwerk die Information über das Ergebnis einer arithmetisch logischen Operation im Prozessor?
 - Statusregister
- Wo steht die Adresse des nächsten auszuführenden Befehls?
 - Programmzähler
- Welche Einheit des Mikroprozessors berechnet die effektive Adresse?
 - Adresswerk
- Gegeben ist ein RISC-Prozessor
 - Was macht die Dekodierschaltung in einem RISC-Prozessor einfach?
 - Einheitliche Befehlslänge (und einheitliches Befehlsformat)
 - Was bedeutet *Load/Store*-Architektur?
 - Der Zugriff auf den Speicher erfolgt nur über *Load-Store*-Befehle.
 - Wie ist das Steuerwerk implementiert?
 - festverdrahtet
 - Was ist eine *Harvard*-Architektur?
 - Getrennte Speicher und Busse für Befehle und Daten


- Gegeben ist eine MIMA
 - Mikroprogramme angeben
 - Bei LDC: IR --> Akku
 - Bei STV: Akkuinhalt in den Speicher schreiben
 - Akkuinhalt ins SDR legen: Akku --> SDR
 - Adresse aus dem IR ziehen: IR --> SAR
 - In 3 Schreibtakten wird der Inhalt des SDR in die im SAR liegende Adresse geschrieben
 - Bei ADD usw
 - Adresse aus dem IR ins Speicheradressregister laden: IR --> SAR
 - 3 Lesetakte, gleichzeitig kann man schon den zweiten Summanden aus dem Akku in den Aluoperator X laden: Akku --> X
 - Nach 3 Lesetakten liegen die Daten im Speicherdatenregister, also der zweite Summand. Wir kopieren den Summanden aus dem SDR in den zweiten Summandoperator der ALU: SDR --> Y
 - Alu auf addieren
 - Ergebnis der Alu in den Akku schreiben: Z --> Akku
 - Anzahl der ALU-Operationen: $8 = 2^{\text{Anzahl von } c}$
 - Maximale Anzahl der MIMA Befehle: 31
 - 15 mit Parameter (0-E), 16 parameterlose (F0-FF) Befehle
 - Adressen extrahieren aus 24-Bit breiten Datenbus
 - Die höchsten (von links) vier Bits werden abgeschnitten

- MIPS-Assembler: Gegeben: Variablen, Register, Feld aus 32-Bit Integerzahlen und temporäre Register
 - Übersetzen Sie C-Kontrollstrukturen in MIPS-Assembler
 - $i = i + j \Rightarrow \text{add } \$s3, \$s3, \$s4$
 - i liegt in $\$s3$, j in $\$s4$
 - Multiplikation müssen je nach gegebenen Befehlen mit add abgebildet werden
 - Zugriff auf ein einzelnes Element ($A[i]$) eines 32-Bit-Feldes
 - i muss vervierfacht werden, weil jedes Element 32 Bit bzw. 4 Byte lang ist
 - mit Hilfsvariable ($\$t1$)
 - $\text{add } \$t1, \$s3, \$s3 \# \text{ verdoppelt } i$
 - $\text{add } \$t1, \$t1, \$t1 \# \text{ verdoppelt nochmal}$
 - Addition von Startadresse ($\$s6$) des Feldes

- add \$t1, \$t1, \$s6
 - Wort (Zugriff 0(REGISTER)) in Hilfsvariable laden (\$t0)
 - lw \$t0, 0(\$t1)
 - Sprungmarken werden im Format „Name:“ vor die Zeile geschrieben, Sprünge darauf verweisen ohne Doppelpunkt
 - j Ende
 - beq \$t0, \$s5, Loop
 - else-if
 - 1. Anweisung ist der Vergleich (Achtung: Vergleich negieren und ggf. zum nächsten Elself/Else-Block springen)
 - letzte Anweisung im If/ElselfBlock ist der Sprung zum Ende, wird dieser vergessen, wird der Else-Block zusätzlich ausgeführt, was nicht gewollt ist
 - (Bei Musterlösung SS2006 fehlt die „fertig“ (fertig) - Sprungmarke)
 - Adressen kann man entweder mit 0(REGISTER) (0(\$s3)) oder dem Variablennamen (i) ansprechen
 - C-Pointer erstellen (ptr = &i;)
 - la \$t0, i # Laden der Adresse als Wort in das Register \$t0
 - sw \$t0, ptr # Speichern der Adresse als Wort in die Adresse ptr
 - Konsequenz: In ptr steht die Adresse i
 - C-Pointer dereferenzieren (i=*ptr;)
 - lw \$t0, ptr # In ptr steht die referenzierte Adresse als Wort, wir laden die Adresse als Wort in \$t0
 - lw \$t0, 0(\$t0) # Nun holen wir uns den Inhalt (das Wort/Datum) der Adresse
 - sw \$t0, i # und speichern es an die Adresse i
 - C-Pointer auf C-Pointer dereferenzieren (i=**ptr;)
 - gleiches wie oben nur der 2. Schritt muss doppelt angewandt werden, da man wieder eine Adresse als Wort lädt
 - Wort einer Adresse an die referenzierte Adresse eines Pointer schreiben (*ptr = i;)
 - lw \$t0, ptr # Adresse als Wort aus ptr laden
 - lw \$t1, i # Wort der Adresse i laden
 - sw \$t1, 0(\$t0) # Adresse aus \$t0 auflösen und das Wort von \$t1 (quasi der Inhalt von i) in die aufgelöste Adresse schreiben
- MIPS-Programm gegeben, DLX-Pipeline wird verwendet
 - Bestimmen Sie die echten Datenabhängigkeiten
 - Zeilen werden mit S_x bezeichnet

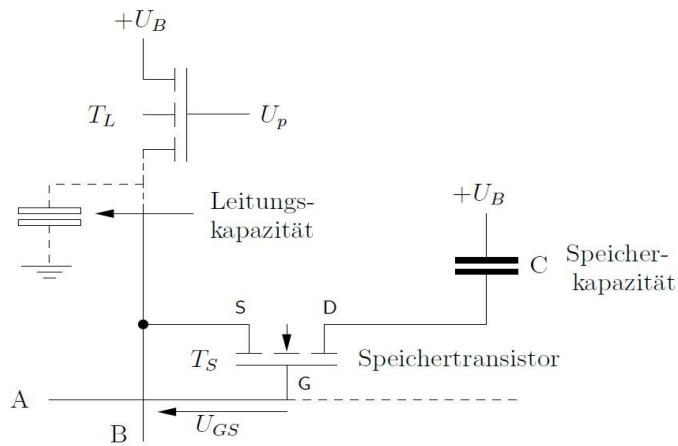
- Befehle checken, welche aus Registern lesen, in die vorher geschrieben wurde
 - Beachte
 - sw schreibt in die Adresse, die im zweiten Operanden angegeben ist
 - lw lädt das Wort in das Register (erster Operand) aus der Adresse (zweiter Operand)
 - Notiere $S_1 \rightarrow S_3$, wenn S_3 aus einem Register liest, das von S_1 geschrieben wurde, in Klammern schreibt man das „schädliche“ Register (R1)
 -  Oli sagt: „ $S_1 \rightarrow S_3 := 'S_1$ schießt einen Pfeil auf S_3 , damit kann es treffen (schaden) oder vorbeischießen (don't care).“
- Beheben Sie die Konflikte
 - NOPs einfügen
 - Schreibe NOP nach dem Befehl, der schreibt und vor den, der liest
 - Ohne Forwarding
 - Wenn während der ID und EX Phasen gelesen wird, müssen 1-2 NOPs geschrieben werden
 - Ist der lesende Befehl der direkte Nachfolger, schreibe 2 NOPS
 - Ist er der zweite, schreibe ein NOP
 - Result Forwarding
 - Keine NOPs benötigt, da der Inhalt des Ausgaberegisters der ALU in das Eingaberegister der ALU für den nächsten Befehl geschrieben wird
 - Load Forwarding
 - Bei Speicherladebefehlen wird der Wert aus Speicher gelesen und dem nächsten Befehl als Eingaberegister an die ALU gelegt
 - Ein NOP einfügen
 - Beachte
 - Bei Verzweigungen werden 3 NOPs benötigt
- Gegeben ist ein Cache-Speicher mit einer Speicherkapazität, Länge der Hauptspeicheradresse in Bit, Länge des Index-Feldes und die Länge des Byte-Offsets
 - Bestimmen Sie die Blockgröße in Bytes
 - $2^{\text{Bitzahl der Länge des Byte-Offsets}}$ ergibt Blockgröße in Byte
 - Beispiel: 4 Bit Byte Offset: $2^4 = 16 \text{ Byte}$
 - Wieviele Einträge besitzt der Cache-Speicher?

- $Anzahl\ der\ Einträge = \frac{Kapazität}{Blockgröße}$
- Beispiel: $\frac{512\ KByte}{16\ Byte} = 32\ K\ Einträge$
- Wie ist der Cache-Speicher organisiert?
 - Die Frage ist eigentlich: Wie hoch ist die Assoziativität?
(Assoziativität ist das n des n -way set associative Caches, gibt es ein Index-Feld ist ein ein n -way set associative cache)
 - $Assoziativität = \frac{Anzahl\ der\ Einträge}{Anzahl\ der\ Sätze}$
 - Beispiel:
 - 12 Bit Index-Feld \Rightarrow
 - Es lassen sich $2^{12} = 4\ K$ Sätze im Cache adressieren
 - $Assoziativität = \frac{32\ K}{4\ K} = 8$
 - Der Cache ist als 8-fach assoziativer Speicher (8-way set associative) organisiert.
- Gegeben n -fach-assoziativer Cache-Speicher mit einer Anzahl an Sätzen und einer Blockgröße in Byte, der Größe der Hauptspeicheradresse in Bit und ein Statusbit (Valid-Bit: V) zur Verwaltung
 - Bestimmen Sie den erforderlichen Speicherbedarf
 - Für jede Zeile werden der Tag, das Statusbit und die Daten benötigt
 - Daten pro Zeile in Bit umrechnen
 - $Blockgröße \times 8$
 - Beispiel
 - $Blockgröße = 8\ Byte$
 - in Bit: $Blockgröße = 64\ Bit$
 - Satzindex in Bit umrechnen
 - $ld(\text{Anzahl der Sätze})$
 - Beispiel
 - $Anzahl\ der\ Sätze = 256$
 - $ld(256) = 8$
 - Byte-Offset berechnen
 - $Byte\text{-Offset in Bit} = ld(\text{Blockgröße in Byte})$
 - Beispiel:
 - $Byte\text{-Offset} = ld(8) = 3\ Bit$
 - $Tag\text{-Länge} =$
Länge der Hauptspeicheradresse – Satzindex – Byte-Offset
 - Beispiel:

- Länge der HPS=32 Bit
- Satzindex=8 Bit
- Tag-Länge=32 Bit – 8 Bit – 3 Bit=21 Bit
- Speicherbedarf für eine Zeile=
 - Tag-Länge+ Statusbit+ Daten pro Zeile
- Beispiel:
 - Speicherbedarf für eine Zeile=21 Bit + 1 Bit + 64 Bit = 86 Bit
- Speicherbedarf für den gesamten Cache=
 - Speicherbedarf für eine Zeile × Anzahl der Sätze × n
- Beispiel:
 - Gesamter Speicherbedarf =
 - $86 \text{ Bit} * 256 * 4 = 88064 \text{ Bits} = 11008 \text{ Byte}$
- Gegeben ist ein direct-mapped cache mit einer Speicherkapazität und Blockgröße, als Aktualisierungsstrategie wird ein Durchschreibverfahren (write through policy) verwendet.
 - Die Tags kann man mittels der Speicherkapazität berechnen
 - $m = \text{Id}(\text{Blockgröße}) + \text{Id}(\text{Anzahl der Zeilen s.o.})$
 - $\text{Tag} = \text{Adresse} \text{ div } 2^m$
 - $\text{Index} = (\text{Adresse} - \text{Tag} * 2^m) \text{ div } \text{Blockgröße}$
 -  Oli sagt: „Immer binär!“
 - Adresse binär umwandeln
 - Tag und Index lassen sich aus der Binärzahl ablesen
 - Von rechts an ab der $\text{Id}(\text{Blockgröße}) + 1$. Stelle $\text{Id}(\text{Anzahl der Zeilen s.o.})$ Stellen ablesen, damit erhält man den Index
 - Die restlichen Stellen links davor sind der Tag
 - Man schreibt einen Hit, wenn die Kombination schon einmal vorkam, aber inzwischen der gleiche Index nicht mit einem anderen Tag aufgerufen wurde
 - In jedem anderen Fall schreiben wir ein Miss

BEISPIEL!

- Skizzieren Sie den Aufbau einer dynamischen RAM-Speicherzelle



-
- Wieviele Adressleitungen sind erforderlich bei einem Speicherbaustein mit einer Kapazität von 4096 Bits und einer 512x8-Organisation?
 - 512x8 Organisation bedeutet, dass der Speicher aus 512 Zellen und 8-Bit Wörtern besteht
 - Um 512 Zellen zu adressieren sind $9 (2^9=512)$ Adressleitungen notwendig
- Wieviele RAM-Bausteine (8kx1-Organisation) sind nötig, um einen Speicher mit der Kapazität von 8k Wörtern und einer Wortbreite von 8 Bit zu realisieren?
 - Es sind $\frac{8}{1}$ notwendig
- Wie ist ein ROM-Baustein mit der Speicherkapazität von 2048 Bits und 8 Adressleitungen organisiert?
 - Es können $\frac{2048}{8}=256$ Zellen adressiert werden
 - 256x8-Organisation