**Wiederholung - Register, Speicher, ARM, Addierwerke, Automaten**

**Rechnerarchitektur – Zusammenfassung**

**Kapitel 1: Einführung**

Binärzahlen: letzte Bit, also das niederwertigste Bit gibt an ob die Zahl gerade(0) oder ungerade (1) ist.

Hexadezimalzahlen: {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} 🡨 8 Bit = 1Byte, 24

Konventionen: Hexadezimalzahlen können auch mit dem Präfix 0x oder Suffix h geschrieben werden. Bsp: 1024 = 400 = 400h = 0x0400

#### **Kapitel 2: Kombinatorische Logik**

**Video 1: Binäre Darstellung**

In der Digitaltechnik verfügen Systeme über 2 Zustände, Strom fließt, und Strom fließt nicht. Physikalisch werden sie mit L (low, 0 - 0.4V) und H (high 2.4 - 5V) dargestellt. In der Algebra werden die Zustände 0 und 1 bezeichnet. In der Aussagenlogik werden diese Zustände mit TRUE und FALSE bezeichnet.

**Video 2: Boolesche Algebra**

In der Digitaltechnik wird die Zustandsmenge {0, 1} bevorzugt. Die verschiedenen logischen Operatoren wie OR und AND werden in der Form von sogenannten “Gattern” realisiert. Komplexere Funktionen werden durch das Verschalten von Gattern realisiert. Elementare Operatoren die Verwendung finden: **OR Operator, +, ∨. AND Operator, \*, ∧. NOT Operator, -x, ¬. Es gilt Punkt vor Strich.**

**Video 3: Boolesche Axiome und Sätze**

Damit die Boolesche Algebra nützlich wird, werden Regeln benötigt, um logische Gleichungen umformen zu können.

Kommutativgesetz: x1 + x2 = x2 + x1 x1 \* x2 = x2 \* x1

Distributivgesetz: a \*(b + c) = (a \* b) + (a \* c) a + (b \* c) = (a + b) \* (a + c)

Neutrale Elemente: 0 + x = x 1 \* x = x

Komplementäres Element: x + ¬x = 1 x \* ¬x = 0

Idempotenz: x + x = x x \* x = x

Assoziativgesetz: a + (b + c) = (a + b) + c a \* (b \* c) = (a \* b) \* c

Absorptionsgesetz: a + (a \* b) = a a \* (a + b) = a

Substitutionsgesetz: x + 1 = 1 x \* 0 = 0

Doppelnegation: ¬(¬x) = x

Komplementäre Werte: ¬0 = 1 ¬1 = 0

Abgeschlossenheit: Boolesche Operationen liefern nur boolesche Werte als Ergebnis.

Dualität: Für jede aus Axiomen ableitbare Aussage existiert eine duale

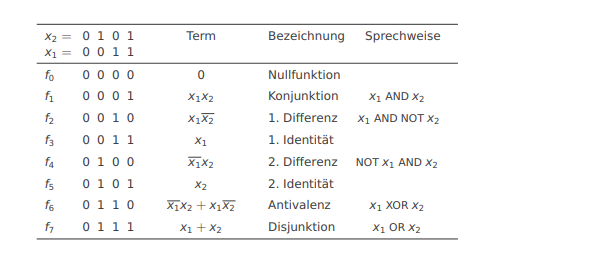
Aussage.

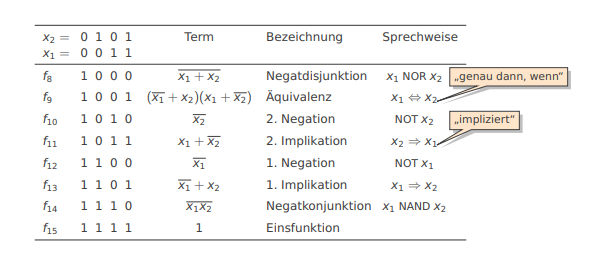
1. De Morgansche Gesetz: ¬(x1 \* x2) = ¬x1 + ¬x2 🡪 gleich NAND

2. De Morgansche Gesetz: ¬(x1 + x2) = ¬x1 \* ¬x2 🡪 gleich NOR

**Video 4: Schaltfunktionen (treffender: Boolesche Funktionen)**

Boolesche Funktionen sind eindeutig mit einer Wahrheitstabelle darstellbar, und kompakter aber nicht eindeutig mit einem Booleschen Ausdruck. Die Kontradiktion: f(x) = 0 gibt immer FALSE zurück. Die Identität: f(x) = x gibt immer x unverändert zurück. Die Negation: f(x) = -x gibt immer die Negation von x zurück. Die Tautologie: f(x) = 1 gibt immer TRUE zurück.





**Video 5: Schaltungsaufbau (treffender: Vom Booleschen Ausdruck zur Schaltung)**

Die technische Realisierung digitaler Systeme erfolgt durch die Verknüpfung binärer Werte mit elektronischen Schaltern. Diese sind aus einfachen elektronischen Bauteilen, wie Transistoren, Dioden und Widerständen zusammengesetzt. Boolesche Funktionen werde mit x1, …, xn Eingängen und einem Ausgang y dargestellt.



Darstellung der elementaren Operatoren:



**Video 6: Vergleich von Schaltungsvarianten**

Für den Entwurf von Schaltungen gibt es mehrere Entwurfskriterien. Eines dieser Kriterien ist der Flächenbedarf, da dies eine knappe Ressource auf einem DIE ist. Den Flächenbedarf kann man grob abschätzen, in dem man die Anzahl der Gatter zählt. Der Energieverbrauch ist ein weiteres Kriterium das eine Rolle spielt. Energieverbrauch steigt mit der Anzahl der Gatter, weshalb die Minimierung der Gatter essentiell ist. Ein anderes Kriterium ist die Schaltgeschwindigkeit, welche von der Anzahl der Schaltebenen abhängig ist.

**Video 7: Kanonische Normalformen**

Der Produktterm gibt die Konjunktion einfacher Variablen wieder, welche gegebenenfalls auch negiert sein können. Desweiteren gibt es noch den Spezialfall Minterm, welcher ein Produktterm ist, in welchem jede Variable einer Booleschen Funktion genau einmal vorkommt. Der **Minterm** wird aus den richtigen Ausgängen (TRUE, 1) gebildet. Die Variablen werden mit ANDs verbunden, die daraus entstandenen Terme mit ORs.

Der Summenterm wiederum gibt die Disjunktion einfacher Variablen wieder, welche gegebenenfalls auch negiert sein können. Desweiteren gibt es noch den Spezialfall Maxterm, welcher ein Summenterm ist, in welchem jede Variable einer Booleschen Funktion genau einmal vorkommt. Der **Maxterm** wird aus den falschen Ausgängen (FALSE, 0) gebildet. 0en werden bei Maxtermen zu 1en und 1en werden negiert, diese werden mit ORs verbunden. Die entstandenen Terme werden mit ANDs verbunden.

Die Disjunktive Normalform (DNF) ist eine Disjunktion von Produkttermen.

Die Kanonische Disjunktive Normalform (KDNF) ist eine Disjunktion von Mintermen.

Heißt die Zeilen wo eine 1 als Ausgabe hat und diese mit einer Disjunktion verknüpfen.

Bsp: (x1 \* x2 \*x3) + (x1\*x2\*x3)

Die Konjunktive Normalform (KNF) ist eine Konjunktion von Summentermen.

Die Kanonische Konjunktive Normalform (KKNF) ist eine Konjunktion von Maxtermen.

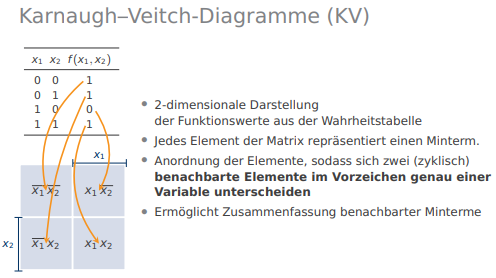
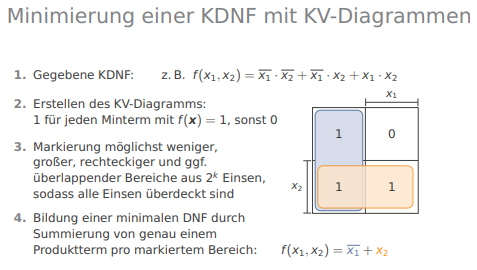
Heißt die Zeilen wo eine 0 als Ausgabe hat und diese mit einer Konjunktion verknüpfen.

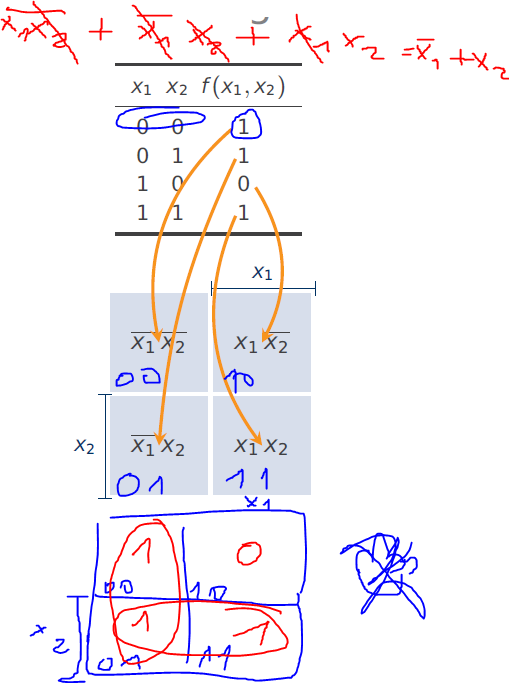
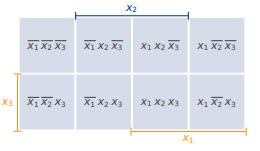
Bsp: (x1 + x2 +x3) \* (x1+x2+x3)

**Video 8: Resolutionsregel // KV Diagramm**

Der Verwendungszweck von KV Diagrammen liegt in der Minimierung der Anzahl der Gatter bei der Realisierung von einer gegebenen Booleschen Funktion. Die Resolutionsregel sagt aus: Wenn sich zwei Summanden nur in genau einer komplementären Variable unterscheiden, dann können beide Terme durch ihren gemeinsamen Teil ersetzt werden.

z.B. für (x1 \* ¬x2 \* x3 \* x4) + (x1 \* ¬x2 \* x3 \* ¬x4) = x1 \* ¬x2 \* x3

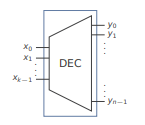




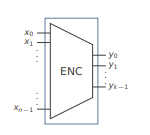
**Video 9: Dekodierer und Kodierer**

Schaltfunktionen werden in sogenannten Schaltnetzen dargestellt. Sie bilden das Wesen der kombinatorischen Logik. Ein Schaltnetz mit n Eingängen und m Eingängen, definiert damit eine m-dimensionale Schaltfunktion mit n Variablen.

Ein typisches Schaltnetz ist der Dekodierer. **Der Dekodierer** dekodiert eine Binärzahl am Eingang und gibt eine 1 an genau dem Ausgang aus, der der Zahl zugeordnet ist. Alle anderen Ausgangsleitungen sind auf 0. **Mit k Eingängen können 2k verschiedene Zahlen dargestellt werden**. Ein Dekodierer ist typischerweise so dimensioniert, das k Eingängen n = 2k Ausgänge gegenüberstehen, die von 0 aufsteigend nummeriert sind. Dekodierer treten meist als Bestandteil größerer Schaltungen auf.



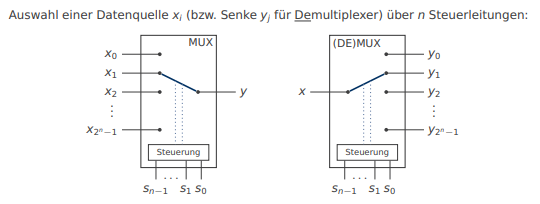
Der Kodierer arbeitet in die entgegengesetzte Richtung zum Dekodierer und reduziert die Information von n Eingängen auf k Ausgänge. Dies nennt man n-zu-k Kodierer.



**Video 10: Multiplexer und Demultiplexer**

Multiplexer und Demultiplexer sind wichtige Module in größeren Digitalschaltungen. Der MUX ist eine Erweiterung des Dekodierers. Der MUX wählt wie der Dekodierer eine Leitung aus, jedoch statt einen von vielen Ausgängen auf 1 zu setzen, leitet er den Wert der ausgewählten Leitung an einen Ausgang weiter. Die Auswahl des Eingabewerts x erfolgt mit den Steuerleitungen s.

(DE)MUX sind Datenweichen, sie haben nur einen Eingang und wählen über die Steuerleitungen aus, auf welchen ihrer Ausgänge y der Eingangswert x weitergeschaltet wird.



**Video 11: OTP-Logikanordnungen (One Trick Pony!)**

OTP-Logikanordnungen (One Time Programmable) sind einmalig programmierbare Schaltungen, welche an den Kreuzungen über Fuses verbunden sind, welche als Sicherungen fungieren, welche durch hohe Spannungen durchbrennen kann, um eine konkrete Verschaltung der Gatter zu ermöglichen. Allen OTP-Logikanordungen ist gemein, das alle Eingänge zunächst invertiert werden, damit die Signale positiv und negativ in die erste Gatterebene geführt werden können.

Es gibt verschiedene Varianten, wie z.B. PROM (Programmable Read-Only Memory), die PAL (Programmable Array Logic) und die PLA (Programmable Logic Array). Die PROM verfügt über eine AND- und eine OR-Matrix. Die AND-Matrix ist dabei festgelegt, die OR-Matrix ist jedoch programmierbar. Über die **PROM** kann man Wahrheitstabellen direkt realisieren. **PROM mit 2n m-Bit-Worten implementiert jede beliebige Schaltfunktion**. Die **PAL** verfügt über eine AND-Matrix die programmierbar ist, wobei nun die OR-Matrix festgelegt ist. Diese kann DNFs bis zu einer Obergrenze von Produkttermen pro Summand realisieren. Die **PLA** verfügt über eine programmierbare AND- und OR-Matrix. Sie kann DNFs bis zu einer Obergrenze an Produkttermen realisieren.

#### **Kapitel 3: Sequenzielle Logik**

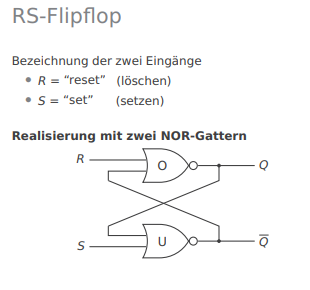
Kontrolierte Rückkopplung, Gatterlaufzeit, Zustandstabellen, Ansteuerungs- und Ausgabegleichungen

**Video 12: Rückkopplung**

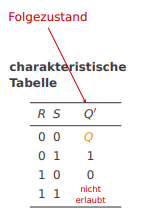
Bei einer Rückkopplung wird der Wert eines Ausgangs wieder in den Eingang des gleichen Gatters eingespeist. Rückkopplungen können schnell zu Schwingungen führen, wie z.B. bei Mikrofonen die zu nahe an einem Lautsprecher stehen. Bei der Rückkopplung bei einem NOR-Gatter würde dies z.B. dazu führen, das ein rückgekoppelter Wert für x = 1 im Output zu 0 wird, anschließend über die Rückkopplung wieder im NOR-Gatter eingespeist wird, und dadurch wieder invertiert wird, wodurch dieser zu 1 werden würde, obwohl der vorherige Output 0 war. Bei einer Eingabe x = 1 wird das OR-Gatter immer negiert und es kommt nie zu zu einem gültigen Zustand.

**Video 13: RS-Flipflop**

Flipflops sind Elemente die ein Bit speichern können und somit Digitalschaltungen ein Gedächtnis geben. Die Schaltungen mit den zwei kreuzweise rückgekoppelten NOR-Gattern ist ein Beispiel für ein RS-Flipflop. Ein RS-Flipflop hat 2 Eingänge, R für Reset zum Löschen, bzw. dem Zugang auf 0 zu stellen, und S für Set um den Zustand auf 1 zu setzen.

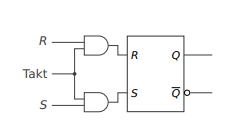


Die Belegung der Eingänge von Flipflops kann man über eine Tabelle darstellen. Diese erweitert eine Wahrheitstabelle um Zustände und Folgezustände und führt so einen zeitlichen Bezug ein.



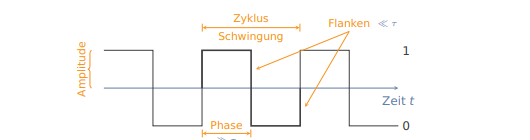
**Video 14: Taktgesteuerte Flipflops**

Der Vorteil von Flipflops, flüchtige Zustände speichern zu können, kann zu einem Nachteil werden, wenn das Schaltnetz das ein Flipflop ansteuert Laufzeiteffekte aufweist, bei denen die Gatter kurzfristig andere Werte annehmen als in der Wahrheitstabelle vorgesehen. In so einem Fall merkt sich das Flipflop diesen Störeffekt sodass er die weitere Verarbeitung beeinflussen kann. Dies ist hochproblematisch, da ungültige und nicht beabsichtigte Zustände herbeigeführt werden können.

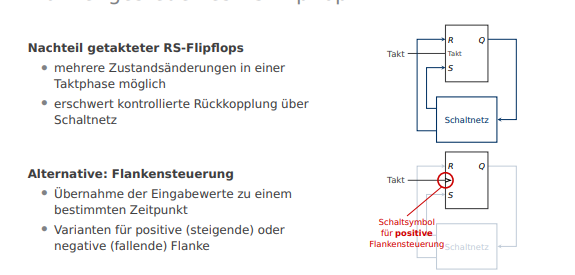


Um solche Fälle zu vermeiden, werden taktgesteuerte Flipflops verwendet. Das erreicht man, in dem man alle Flipflops in einer Schaltung synchronisiert, sodass diese Zustände nur zu vorgegebenen Zeitpunkten übernehmen. z.B. bei einem RS-Flipflop, welcher einfach mit 2 AND-Gattern und einem Taktsignal erweitert wird.

Der Takt ist ein symmetrisches Rechtecksignal mit konstanter Taktfrequenz (Schwingungen pro Zeiteinheit), gemessen in Hertz [1 Hz = 1/s]. Taktgenerierung erfolgt durch industriell gefertigte Quarze, welche sehr präzise schwingen (> 4 MHz). Niedrigere Frequenzen erreicht man durch (mehrfache) Teilung.

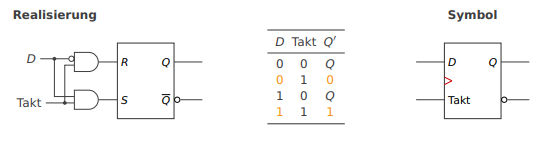


Der Nachteil von einem getakteten RS-Flipflop ist, das dieser mehrere Zustandsänderungen in einer Taktphase ermöglicht. Ebenso erschwert er kontrollierte Rückkopplung über das Schaltnetz. Eine Alternative zum getakteten RS-Flipflop ist die Flankensteuerung, bei welcher die Übernahme der Eingabewerte zu einem bestimmten Zeitpunkt passiert und nicht zu einer gegebenen Zeitphase.

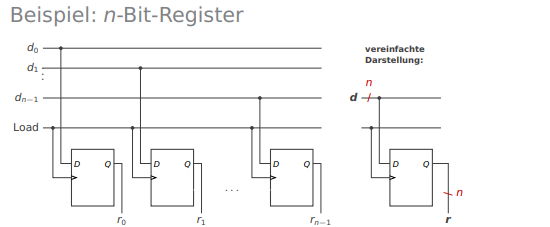


**Video 15: Register aus D-Flipflops**

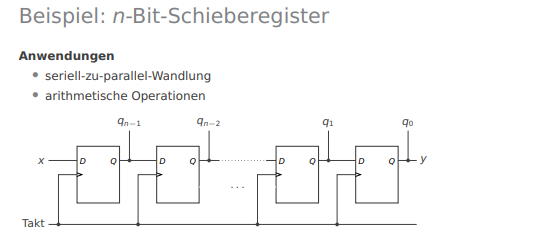
Das D-Flipflop ist ein weiterer Flipflop-Typ, wobei das D für Daten oder Delay (wegen der Verzögerung von Signalen um 1 Takt) steht. D-Flipflops setzen den Wert des Eingangs direkt als Folgezustand. Die technische Realisierung verwendet einen Inverter am AND-Gatter vor dem R Eingang eines bekannten RS-Flipflops. D-Flipflops werden häufig mit Flankensteuerung eingesetzt.



Die Parallelschaltung mehrerer D-Flipflops nennt man Register. Ein Register kann einen n Bitwert speichern. Dazu verbindet man die Eingänge mit einem n-breiten Datenbus d. Immer wenn ein Wert am Datenbus anliegt, der in das Register übernommen werden soll, wird eine positive Flanke über eine dezidierte Load-Leitung geschickt. Den Wert des Registers r kann man an den Ausgängen ablesen, bis ein neuer Wert geladen wird. Dabei muss man beachten, das der Eingang nicht direkt mit dem Takt verbunden wird, sondern die Load-Leitung nur ausgewählte Taktphasen weitergibt (die in denen das Register geschrieben werden soll). So kann die Schaltung kontrollieren, wann Werte vom Datenbus in das Register übernommen werden sollen. Das Register wird meist in der Form eines Stapels von D-Flipflops dargestellt.

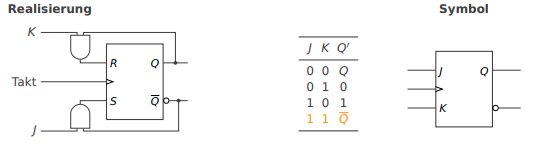


Schaltet man D-Flipflops parallel in Reihe, erhält man ein n-Bit-Schieberegister. Dabei wird der Ausgang von jedem D-Flipflop mit dem Eingang des nachfolgenden D-Flipflops verbunden. Alle werden synchron über die Taktflanke gesteuert. In dem Schieberegister wandert ein Bit vom Eingang x des ersten Flipflops zum jeweils nächsten Flipflop, bis es den Ausgang y erreicht.

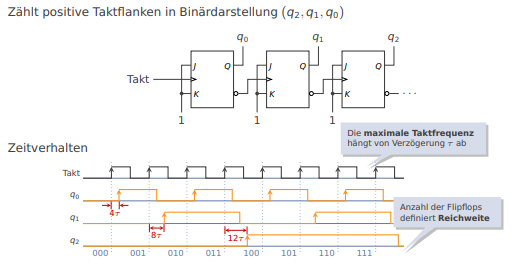


**Video 16: Zähler aus JK-Flipflops (John F. Kennedy!)**

Das JK-Flipflop (Abkürzung für Jump/Kill) ist ein abgeändertes taktgesteuertes RS-Flipflop. Bei dem JK-Flipflop wird der Zustand direkt als auch invertiert über zwei AND-Gatter an die R und S Eingänge rückgekoppelt. Wenn J und K auf 1 sind, wird der aktuelle Zustand im Folgezustand invertiert. Dies ist praktisch, wenn die Ansteuerlogik des Flipflops sonst keinen Zugriff auf den aktuellen Zustand hat.



JK-Flipflops finden z.B. bei Binärzählern Verwendung. Dabei werden der J und K Eingang aller JK-Flipflops verbunden, und auf 1 fixiert. Das bedeutet, dass der Flipflop bei jeder positiven Flanke am Takteingang seinen Zustand invertiert. Der Takt wird jedoch nur am ersten Flipflop angelegt. Der Takteingang jedes darauffolgenden Flipflops wird mit dem Ausgang des negierten Zustands des vorherigen Flipflops verbunden. Dieser Ausgang erzeugt eine positive Flanke immer dann, wenn das Flipflop von 1 auf 0 zurück schaltet. Die BInärzahl kann an den q-Ausgängen abgelesen werden, wobei das niederwertigste Bit in diesem Fall bei dem ersten Flipflop steht.



Es gibt in der sequenziellen zwei Typen von Schaltwerken, asynchrone Schaltwerke und synchrone Schaltwerke. Bei asynchronen Schaltwerken ist der Zeitpunkt stabiler Ausgangssignale abhängig von der Gatterlaufzeit. Sie können jedoch sehr schnell arbeiten.

Da sie jedoch von einer Gatterlaufzeit abhängig sind, erschwert das den Entwurf, wodurch sie weniger gebräuchlich sind.

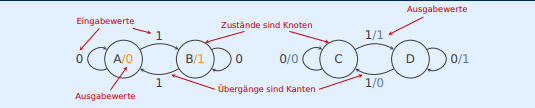
Synchrone Schaltungen führen hingegen den gleichen Takt an alle taktgesteuerten Flipflops, wodurch alle Ein- und Ausgangssignale zu festen Zeitpunkten gültig. Da es weniger fließende Übergänge gibt, ist ein systematischer Entwurf leichter zu realisieren.

**Video 17: Zustandsautomat**

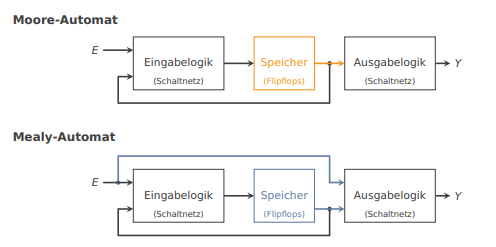
Der Zustandsautomat ist ein Entwurfswerkzeug für synchrone Schaltwerke. Zustandsautomaten befinden sich zu jedem diskreten Zeitpunkt t in genau einem Zustand St aus der endlichen Zustandsmenge S. Der Wechsel zwischen den Zuständen wird durch eine Zustandsübergangsfunktion definiert, welche vom aktuellen Zustand S und dem Eingabevektor x abhängt. Man unterscheidet zwischen zwei Varianten von Zustandsautomaten; Moore und Mealy Automat.

Beim Moore Automat hängt die Ausgabe y nur vom Zustand ab, verwendet man jedoch einen Mealy Automat, hängt die Ausgabe von Zustand und Eingabe ab.

(Moore links, Mealy rechts)

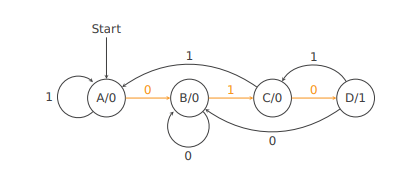


In der schaltungstechnischen Realisierung eines Zustandsautomaten werden soviele Flipflops verwendet, das man mit den binären Zuständen Q der Flipflops ausreichend viele Kombinationen bilden kann, um jeden Zustand des Automaten, also der Zustandsmenge S identifizieren zu können. z.B. reichen 2 Flipflops für 4 Zustände, 4 Flipflops für 16 Zustände und 16 Flipflops für 65536 Zustände. Diese Flipflops sind als Speicher eingezeichnet.



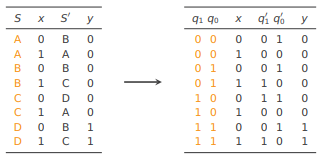
**Video 18 und 19: Entwurf synchroner Schaltwerke**

Beim Entwurf eines synchronen Schaltwerk gibt es eine genaue Vorgehensweise. Man beginnt mit der Erstellung eines Zustandsdiagramms. Dafür zeichnet man eine bestimmte Anzahl an Knotenpunkten auf, welche man anschließend alphabetisch beschriftet. Hätte man z.B. 3 Zustände die zu erfüllen wären, benötigt ein Moore-Automat einen zusätzlichen Zustand für die Lösung, wobei man bei einem Mealy-Automat darauf verzichten kann. Anschließend markiert man den Start-Knotenpunkt und gibt die nötigen Eingaben an, um von einem Knotenpunkt zum nächsten zu gelangen. Daraufhin muss man noch alle anderen Fälle abdecken, sollten die Eingaben nicht die Bedingungen erfüllen, die von den Knotenpunkten vorausgesetzt wurden. Anschließend müssen noch die Knotenpunkt so beschriftet werden, das der Knotenpunkt der über das Ergebnis verfügt, eine 1 zurückgibt, und der Rest eine 0.



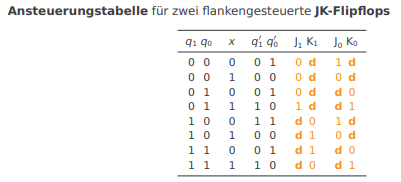
Im **zweiten Schritt** erstellt man eine **Zustandstabelle**, welche für jeden Zustand und jede mögliche Eingabe eine Zeile in der der Folgezustand eingetragen ist. Jede Kante der grafischen Darstellung wird also eine Zeile. Ebenso verfügt die Zustandstabelle für die Ausgabewerte, welche ganz rechts dargestellt werden.

(Zustandstabelle links, binär kodierte Zustandstabelle rechts)

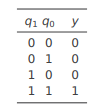


Im **dritten Schritt** erstellt man eine **binär kodierte Zustandstabelle**, welche die Spalten für S und S’ durch mehrere binäre Zustände q und q’ (für die Folgezustände) ersetzen. Wie viele q-Spalten und damit Flipflops man braucht, hängt von der Anzahl der Zustände ab. Mit k Bits kann man 2k Zustände unterscheiden, also ist k der Log2 der Kardinalität von S. Wie q und q’ belegt werden ist dem Ersteller überlassen, wichtig ist dabei nur, das die Zustände immer den Zuständen der normalen Zustandstabelle entsprechen. Zum Beispiel für Zustand A 0 1, und für Zustand B 0 0.

Ausgangspunkt für den **vierten Schritt** des Entwurfs eines synchronen Schaltwerks ist die binäre Zustandstabelle von dem dritten Schritt, welche in die **Ansteuerungstabelle** überführt wird. Anstelle von dem Ausgangswert y, wird dieser durch die Flipflops ersetzt, bei 2 JK-Flipflops z.B. J1 + K1 und J0 + K0. Anschließend überprüft man jeden möglichen Übergang von jedem möglichen Zustandsbit.

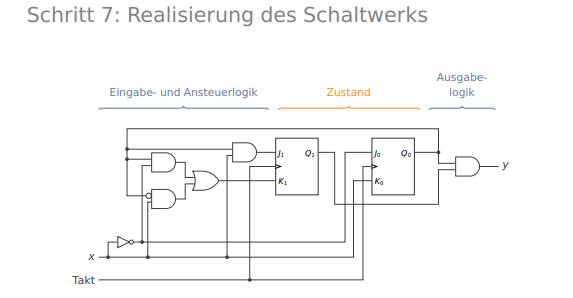


Im **fünften Schritt** wird die **Ausgabegleichung** des synchronen Schaltwerks ermittelt. Beim Moore-Automat kommt es bei der Ausgabe nur auf den aktuellen Zustand an. Man erhält also eine kurze Tabelle mit jeder möglichen Darstellungsvariante für q. Die Werte für y erhält man aus der binär kodierten Zustandstabelle. Daraus kann man eine Boolesche Funktion für die Ausgabe y bestimmen. Sollte die Boolesche Funktion nicht minimal sein, müsste man diese minimieren.



Im **sechsten Schritt** werden die **Ansteuergleichungen minimiert**, welche man über die von der Ansteuertabelle gebildeten Booleschen Funktionen erhält. Dies passiert für die niederwertigsten und hochwertigsten Bits der Zustandskodierung.

Im **siebten und letzten Schritt** **realisiert man** schlussendlich **das Schaltwerk** mit den Flipflops und Schaltgattern, welche einem zur Verfügung stehen.



#### **Kapitel 4: Arithmetik**

**Video 20: Addieren von Binärzahlen**

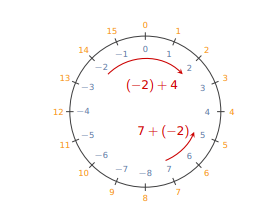
Bei der **Addition von Binärzahlen** kann ein sogenannter Übertrag (Carry) auftreten.

Bei der Addition von Binärzahlen muss man einige Gesetzmäßigkeiten beachten.

**0 + 1 = 1, kein Übertrag, 1 + 1 = 0, 1 Übertrag, 0 + 0 = 0, kein Übertrag**. Ebenfalls kann ein Overflow auftreten falls die Summe y = a + b >= 2n. Dann reichen n Bit nicht mehr für die Darstellung des Ergebnisses aus. Das (n + 1)-te Summenbit wird als Überlauf (Overflow) bezeichnet.

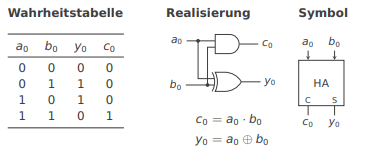
**Video 21: Negative Ganzzahlen im Zweierkomplement**

Wenn man nach einem Überlauf bei der Addition wieder zu zählen beginnt, erhält man einen Zahlenring. Eine typische Interpretation ist die Darstellung der Ganzzahlen 0 - 15. Addition bedeutet im Zahlenring Bewegung im Uhrzeigersinn. Wenn man in der 0 - 15 Darstellung z.B. 13 + 5 rechnet, kommt es zu einem Überlauf, wodurch das Ergebnis zu 2 wird. Wenn man im Zahlenring 0 - 15 z.B. eine Subtraktion durchführt wie 7 - 4, ist das Ergebnis = 3. Da es aber den Overflow gibt, kann man auch über die Addition subtrahieren, wie z.B. bei 7 + 12, wo das Ergebnis 3 ergeben würde. Daran kann man erkennen, das die 12 die Rolle von -4 einnimmt. Im Zweierkomplement entscheidet die vorderste Zahl darüber ob eine Zahl positiv oder negativ ist, z.B. 0 = negativ und 1 = positiv. **Negation von Zahlen im Zweierkomplement erfolgt folgendermaßen; Invertierung aller Bits und Addition von 1**.

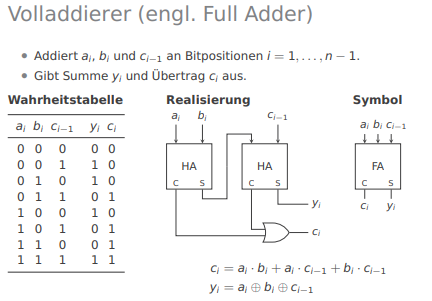


**Video 22: Logik für die Addition**

Der elementare Schritt bei der schriftlichen Addition ist das Addieren der Werte zweier Ziffern. Dieser Schritt lässt sich schaltungstechnisch mit sogenannten **Halbaddierern** realisieren. Deren Aufgabe ist es, zwei Eingangsbits a0 und b0 zu einer Summe y0 und einem Übertrag c0 zu verrechnen.

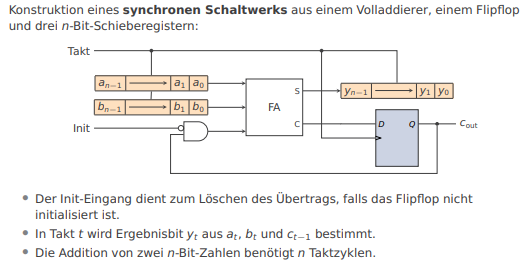


**Der Volladdierer** hat drei Eingänge und führt 2-Bit Additionen durch, um ai, bi und ci-1 an Bitpositionen i = 1, …, n - 1 zu berechnen. Schaltungstechnisch kann man sich einen Volladdierer als Kombination von zwei Halbaddierern vorstellen. Dabei werden die Übertragsbits mit OR verknüpft, denn ein Übertrag kann dann entstehen wenn beide Summanden 1 sind, als auch wenn ein Summand 1 ist und ein Übertrag aus der nächstniedrigeren Stelle verrechnet werden muss.



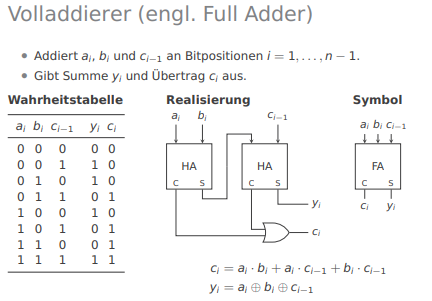
**Video 23: Addierwerke**

**Das serielle Addierwerk** besteht aus einem Volladdierer, einem Flipflop und drei n-Bit-Schieberegistern. Die beiden Summanden für den Volladdierer sind in n-Bit-Schieberegistern gespeichert, und auch die Summe wird in ein Schieberegister geschrieben. Schieberegister sind Schaltwerke, weshalb sie eine Taktleitung benötigen. In jedem Takt wird die niederwertigste Stelle beider Summanden ausgelesen, im Volladdierer verrechnet und das Ergebnisbit zunächst in das höchstwertige Bit des Schieberegisters y geschrieben, wo es dann weitergeleitet wird, bis es y0 erreicht. Überträge müssen auch verrechnet werden. Dieser wird in einem D-Flipflop gespeichert, welcher im nächsten Takt im Carry-Eingang des Volladdierers gespeichert wird.



Der **Init-Eingang** **dient** zum **Löschen des Übertrags**, **falls** das **Flipflop nicht initialisiert** ist. kommt es zum Überlauf, kann man dies am Zustand des D-Flipflops im n-ten Takt ablesen. Insgesamt benötigt die **Addition** **von zwei n-Bit-Zahlen n Taktzyklen**.

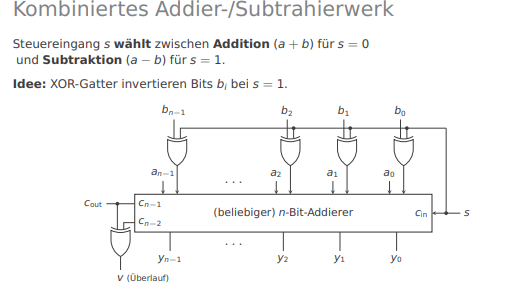
**Das Parallele Addierwerk** ist eine Konstruktion aus n - 1 Voll - und einem Halbaddierer. Der Übertrag an Position i = 0 kann alle Bitstellen 1 bis n - 1 durchlaufen, daher: **“Ripple Carry”-Addierer (RCA)**. Im Prinzip kann dieses Schaltwerk pro Taktzyklus 2n - Bitwerte addieren, allerdings ist die maximale Taktfrequenz durch das Warten auf die Überträge begrenzt, wenn n realistisch groß wird. Die Überträge führen zu einem sehr langen Pfad.



**Der CLA (Carry Look-Ahead)** versucht den Wert der Carrybits vorherzusagen um damit die höherwertigen Stellen bereits früher ausrechnen zu können.

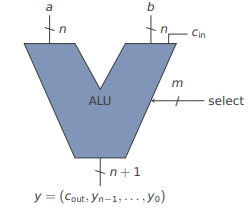
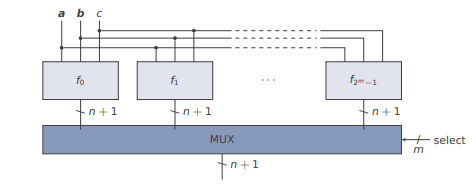
**Video 24: Kombiniertes Addier-/Subtrahierwerk**

Ein **kombiniertes Addier- Subtrahierwerk** ist ein Rechenwerk welches zwei n-Bit Wörter a und b abhängig von einer Steuerleitung s entweder addiert, für s = 0, oder subtrahiert, für s = 1. Für S = 1 invertieren die XOR-Gatter alle Bits von b0 bis bn-1, das ist der erste Schritt der arithmetischen Negation. Im zweiten Schritt wird die 1 am Übertragseingang mit der niederwertigsten Stelle bei a0 und b0 verrechnet, wodurch die Invertierung komplett ist. Die Schaltung verfügt über 2 Ausgänge, cout, welcher der Übertrag der höchstwertigen Stelle ist. Der Ausgang V zeigt einen arithmetischen Überlauf an.



**Video 25: Arithmetisch-logische Einheit**

**Die arithmetisch-logische Einheit (ALU)** ist ein Modul mit mindestens 2 n-Bit-Eingängen, das arithmetische Operationen mit logischen Operationen verknüpfen kann. Jeder Prozessor verfügt über mindestens eine ALU. Die ALU wird in der Form von einem V dargestellt, wobei die beiden Eingänge an den jeweiligen Spitzen von V sind. Das Tal bildet den Ausgang, an welchem die Ergebnisse der Operationen vereinigt werden.



**Video 26: Multiplikation positiver Binärzahlen**

Das Produkt y = a \* b aus zwei n-Bit-Faktoren hat 2n Stellen.

Der Algorithmus wie verläuft wie bei schriftlicher Dezimal-Multiplikation.

**0 \* 0 = 0, 1 \* 0 = 0, 0 \* 1 = 0, 1 \* 1 = 1.**

Multiplikation von 100 und 1001:

1100 \* 1001

0011

00000

000000

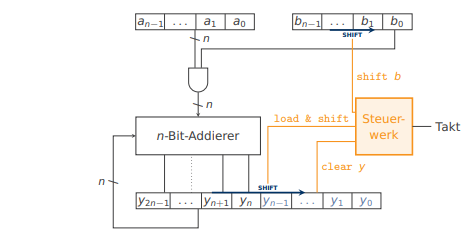
+ 0000011

0011011

**1101100**

<https://www.youtube.com/watch?v=oFseqcdcZFU> // Multiplikation von Binärzahlen

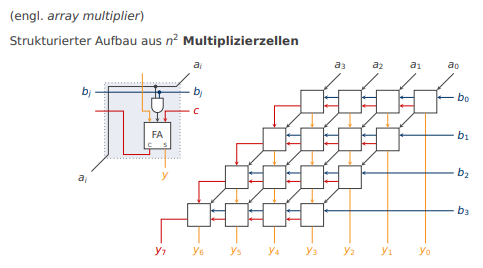
**Das serielle Multiplizierwerk** realisiert die Multiplikation von Binärzahlen. Dabei beginnt man mit einem n-Bit-Addierer, der mit 2n-Bit-Schieberegister so verbunden ist, dass das Ergebnis der Addition im Folgezustand in die höherwertigen Stellen des Registers übernommen wird. Gleichzeitig geht der aktuelle Zustand dieser Bits als Summand in den Addierer. Der zweite Summand ist der erste Faktor a. Dieser geht allerdings nicht direkt rein, denn dieser muss zuvor noch eine Bedingung erfüllen. Dazu wird jedes Bit von a mit dem niederwertigsten Bit des zweiten Faktors von b verknüpft. Auch dieser wird nur in einem n-Bit-Schieberegister gespeichert.



Abschließend wird noch ein Steuerwerk benötigt, das die vorhandenen Komponenten so betreibt, wie es der Algorithmus vorsieht. Der clear y Ausgang dient dazu, das Schieberegister zu löschen und wird nur einmal im Startzustand 1 benötigt, der load & shift Ausgang übernimmt das Ergebnis des Addierers in die obere Hälfte, und schiebt dann das Register um ein Bit nach rechts. Schließlich steuert shift b den Takteingang des kleinen Schieberegisters. Der Takteingang dieses Schieberegisters geht in das Steuerwerk.

**Video 27: Feldmultiplizierer**

Schneller als **ein serielles Multiplizierwerk ist der Feldmultiplizierer**. Dieser besteht aus einem Schaltnetz, welches die schriftliche Multiplikation mit je einem Volladdierer pro Zelle realisiert. Volladdierer werden in dieser Realisierung nicht wieder verwendet. Deshalb ist der Preis für sehr schnelle Ergebnisse ein sehr hoher Flächenbedarf. Die Bits des ersten Faktors a werden zeilenweise um 1 verschoben aufsummiert. Ob eine Zeile in die Summe einfließt entscheidet das entsprechende Bit im zweiten Faktor b. Es sind insgesamt n2 Bitprodukte zu berechnen und deren Werte zu addieren.



Die technische Realisierung geschieht über Multiplizierzellen, von denen es pro Bitprodukt 1 Zelle gibt. Jede Zelle benötigt einen Volladdierer, sowie ein AND-Gatter, um den Summand ai \* bj zu berechnen. Die Bits des ersten Faktors ai verschieben sich in jeder Zeile um eine Stelle und werden diagonal weitergeleitet. Die bits des zweiten Faktors bj entscheiden über die Addition einer ganzen Zeile und werden horizontal weitergeleitet. Der Wert der Summe geht als zweiter Summand in den Volladdierer der darunter liegenden Zeile ein. Wenn es keine Multiplizierzellen in der Zeile mehr gibt, liegt direkt das Ergebnisbit vor. Abschließend müssen nur noch die Überträge weitergeleitet werden. Diese folgen dem bekannten Prinzip des Ripple Carry Addierers.

Video 28: Division mit “Restoring”

Binäre Subtraktion

**1 - 1 = 0, 1 - 0 = 1, 0 - 0 = 0, 0 - 1 = 1, 1 Rest**

Schriftliche Division von Binärzahlen:

10100 : 100 = 101

100

001

0010

00100

00100

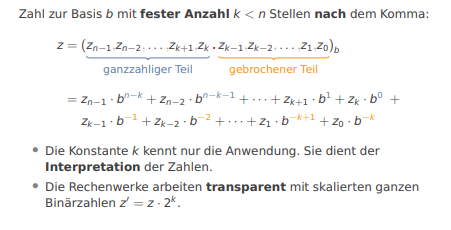
00000

**Video 29: Rationale und reelle Zahlen**

**Festkomma:** Jede darstellbare Kommazahl z wird durch lineare Skalierung auf eine Zahl z’ abgebildet. Diese kann man dann in den bekannten Rechenwerken ohne Probleme verarbeiten. Gleitkomma: Die Darstellung von Kommazahlen durch ein Argument (Mantisse) a und Charakteristik (Exponent) c zur Basis r.

z = a x rc

Dafür sind spezielle Rechenwerke erforderlich. Ein solches Rechenwerk ist z.B. das Gleitkomma-Addierwerk. Die exakte Darstellung rationaler Zahlen erfolgt durch separate Speicherung und Verarbeitung von Zähler und Nenner als Ganzzahlen. Dies wird in Software realisiert, um exaktes wissenschaftliches Rechnen zu garantieren.



Festkommazahlen werden in einen ganzzahligen Teil, und einen gebrochenen Teil aufgeteilt.

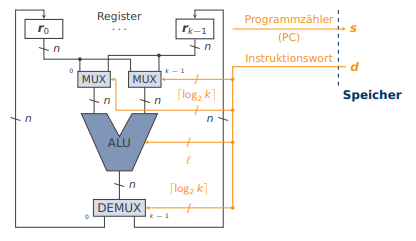
**Video 30: Gleitkommazahlen**

Mantisse aus Vorzeichen s und normalisiertem Betrag a = 1.f, im Bereich 1.00 … 0 bis 1.11 … 11 ohne die führende Eins.

#### **Kapitel 5: Befehlssatzarchitektur (ARM 32)**

**Video 31: Vom Dividierwerk zum Mikroprozessor**

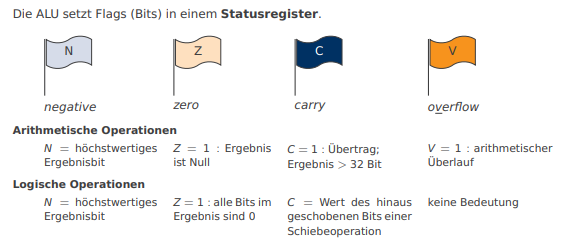
Für den Entwurf eines Mikroprozessors benötigt man einige Mittel. Der erste Baustein ist eine ALU, welche man mit k Registern füllt. Diese Register verbindet man dann mit Multiplexern, welche auswählen, aus welchem Register ein Operand übernommen wird. Auch das Ergebnis der ALU-Operation wird wieder in ein Register zurückgeschrieben, welches der Demultiplexer, welcher sich am Ausgang der ALU befindet, auswählt. Damit sind die wesentlichen Datenpfade gegeben. Es fehlen nur noch die Steuerleitungen, wie der Programmzähler, das Instruktionswort und der Speicher. Bei dem Instruktionswort wird an die MUX oder (DE)MUX weitergegeben, welche Operanden aus dem Register genommen werden sollen. Der Programmzähler wird in jedem Takt um 1 Wort erhöht. Er ist dafür zuständig, zu verständigen, auf welche Register zugegriffen werden soll.



Der Prozessor (Central Processing Unit, CPU) ist die Hauptkomponente jedes Rechners.

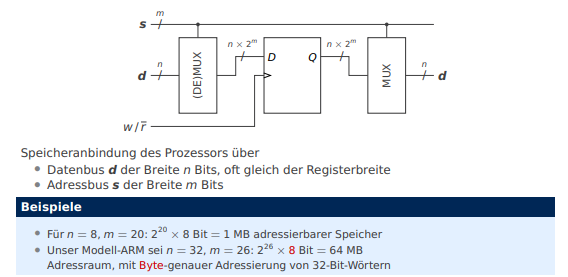
**Video 32: ARM-Registersatz und Flags**

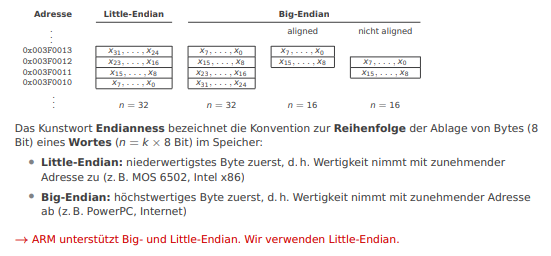
CPUs sind Zustandsautomaten. Ihr Zustand wird in wenigen, direkt mit der Logik verbundenen Registern gespeichert. Bei ARM stehen im User-Modus 16 Register mit je 32 Bit zur Verfügung. Dabei sind R13, R14 und R15 reserviert, und können nicht vom User genutzt werden. Ebenfalls gibt es sogenannte Flags, die die ALU in einem Statusregister setzt. Es gibt vier Flags die von Bedeutung sind, N (negative), Z (zero), C (carry) und V (overflow).



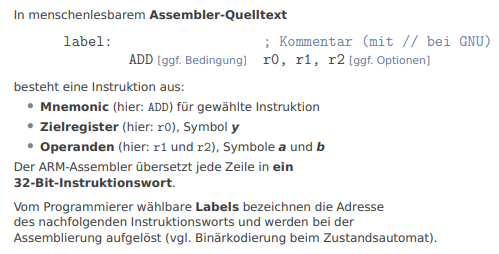
**Video 33: Speichermodell, Endianness und Alignment**

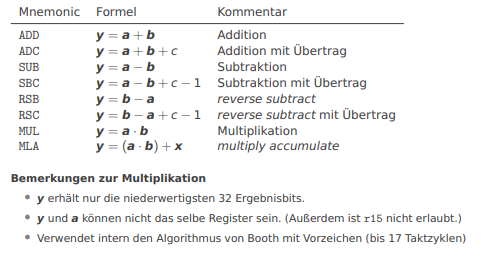
Der Prozessor ist über zwei Busse, also Leitungsbündel mit dem Speicher verbunden.

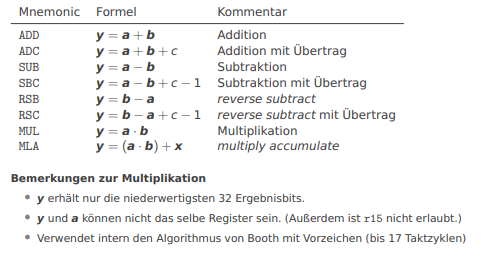


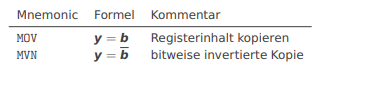


**Video 34: Allgemeines ARM-Instruktionsformat**





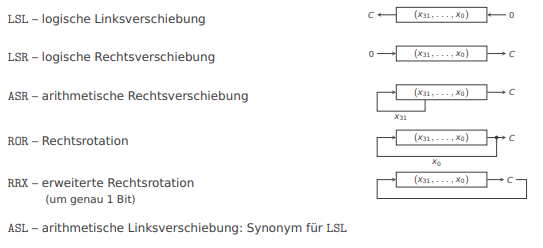




**Video 35: Barrel-Shifter und Immediate-Werte**

Bei der ARM-Architektur kann der zweite Operand von einem Barrel-Shifter bearbeitet werden, bevor in die ALU weiter gelangt. Dieser Barrel-Shifter kann die Bits im Register

um 0 bis 31 Stellen verschieben oder rotieren, und zwar in der gleichen Instruktion. Dies ermöglicht viele Kombinationsmöglichkeiten und spart gegenüber anderen Architekturen eine Menge Instruktionen ein. Wie der Barrel-Shifter selbst angesteuert werden kann, hängt davon ab woher der Wert von b bezogen wird.



Mit dem Barrel-Shifter kann man Multiplikationen mit 2k +- 1 in einem Taktzyklus berechnen.

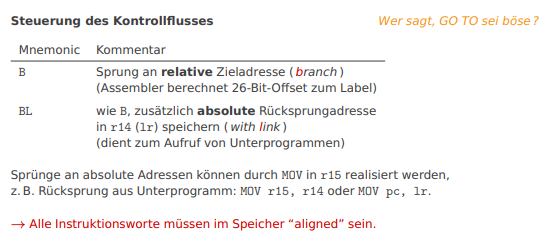
MOV r2, r0, LSL #2 ; r2 = r0 \* 4, da LSL #2 22, also 4 gleicht.

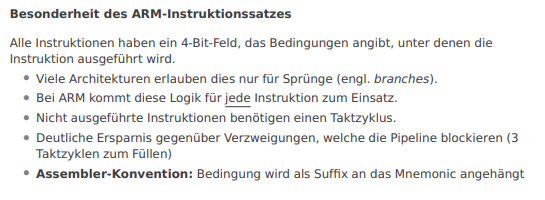
ADD r9, r5, r5, LSL #3 ; r9 = r5 \* 9, da LSL #3 23, also 8 gleicht.

RSB r9, r5, r5 LSL #3 ; r9 = r5 \* 7

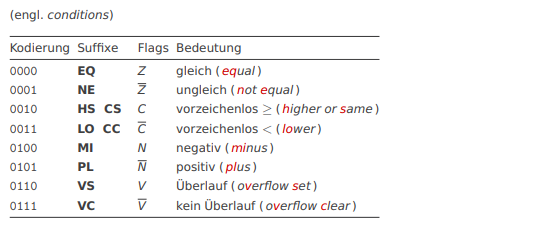
Immediate-Werte werden in der Assembler-Notation mit vorangestellter Raute # dargestellt. Jedes Instruktionswort ist 32 Bit lang. Damit stehen nur 12 Bit für den zweiten Operanden b zur Verfügung. 8 Bit davon werden für Konstanten verwendet, die anderen 4 Bit für ROR-Verschiebung in Vielfachen von 2. Wenn möglich, kümmert sich der Assembler um die Kodierung.

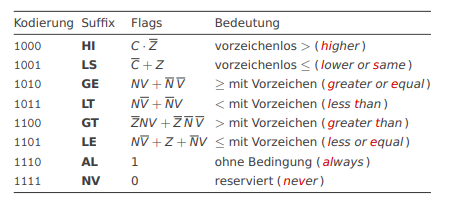
**Video 36: Sprünge und Bedingungen**





Bedingungen für Mnemonics





**Video 37: Speicherzugriff**

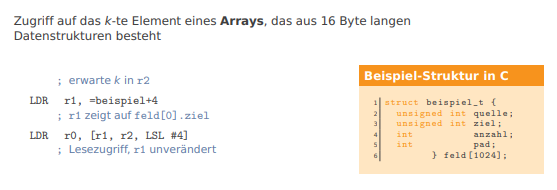
Der Speicherzugriff wird bei ARM über spezielle Instruktionen ausgelöst, wie LDR (lese 32-Bit-Wort), LDRB (lese Byte), LDRH (lese Halbwort (16 Bit)), LDRSB (lese Byte mit Vorzeichenerweiterung), LDRSH (lese Halbwort mit Vorzeichenerweiterung). Die Adresse wird über ein Basisregister plus Offset angegeben:

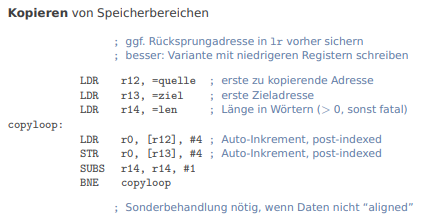
z.B. STR r0, [r1] ; Inhalt von r0 an Adresse speichern, die in r1 steht.

LDR r2, [r1, #-12] ; Speicherinhalt an der Adresse (r1-12) nach r2 laden.

Bemerkungen sind möglich und werden zwischen Stamm-Mnemonic und Größensuffix eingeschoben, z.B. LDREQB.

**Video 38: ARM-Adressierungsarten**





**Video 39: ARM-Instruktionskodierung**

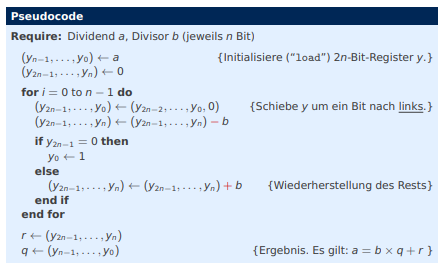


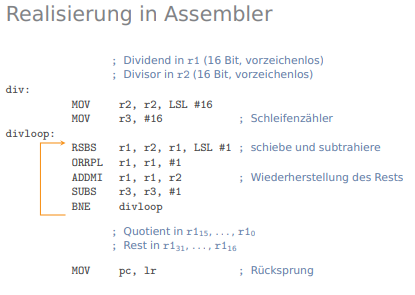
Auf dieser Abbildung sieht man die Zuordnung der 32-Bit im Instruktionswort welche nach Befehlstypen aufgeschlüsselt sind. Die Bits 28 bis 31 kodieren unabhängig vom Befehlstyp die Bedingung. Alle ALU-Operationen sind im Befehlstyp Data Processing zusammengefasst. S sagt aus, ob Flags gesetzt werden oder nicht. L/S bestimmen ob Lese- oder Schreibzugriff erfolgen darf. Bei dem Branch sind 26 Bits für den Abstand zur Zieladresse vorgesehen. Sowie 1 Bit um B von BL (Link) zu unterscheiden.

Rn und Rd wählen das Register des ersten Operanden und das Zielregister. Für den zweiten Operanden bleiben die niederwertigsten 12 Bit. w ist das Auto-Inkrement. B steht für ein Byte, wenn es nicht für ein Byte steht, dann steht es für ein 32 Bit Wort.

U ist das Vorzeichen für Offset. P steht für Post/Pre-inkrement.

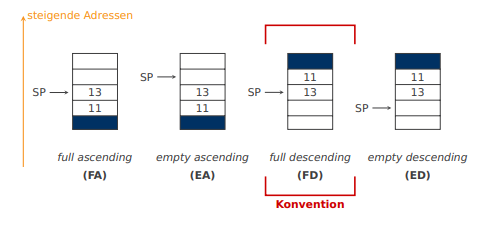
**Video 40: Division in Assembler**



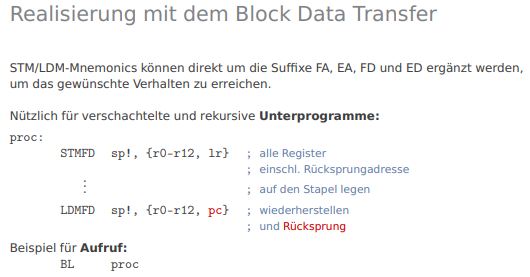


**Video 41: Stapelorganisation**

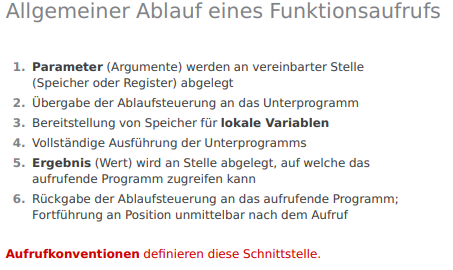
Als Stapel bezeichnet man eine besondere Struktur im Arbeitsspeicher. Und zwar ist ein Stapel eine Datenstruktur die wächst, wenn man neue Daten darauf ablegt (push) und schrumpft, wenn man Daten von oben wegnimmt (pop). Bei der Realisierung im Speicher definieren zwei Zeiger die aktuellen Grenzen des Stapels. Dabei zeigt der Base Pointer (BP) auf den Boden des Stapels. Der Stack Pointer zeigt wiederum auf die Spitze des Stapels, wo sich das zuletzt abgelegte Element befindet. Elemente die in den Stapel gelegt werden, werden von unten nach oben geschoben. Sollte es zu einem “Pop”-Zugriff kommen, wird das oberste Element verworfen, und der Stack Pointer zeigt auf das neue höchstgelegene Element.

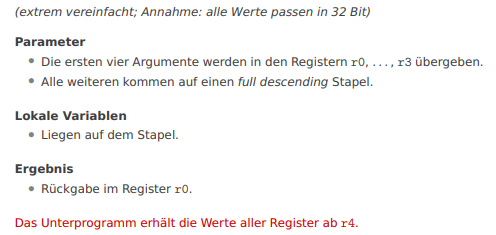


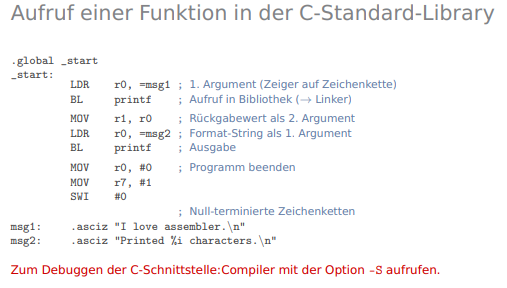
Es gibt verschiedene Varianten der Stapelorganisation; welche Variante verwendet wird, ist abhängig davon, wohin der Stack Pointer zeigt. Zeigt er auf das letzte Element im Stapel, spricht man von der “full”-Variante, zeigt der Stack Pointer jedoch auf die erste freie Adresse, spricht man von der “empty”-Variante. Daraufhin unterscheidet man noch zwischen einem ascending oder descending Stapel, wobei bei einem ascending Stapel der Stapel steigend wächst, und bei einem descending Stapel fallend wächst. Damit ergeben sich 4 Kombinationsmöglichkeiten aus Full, Empty, Ascending und Descending.



**Video 42: Aufrufkonventionen**







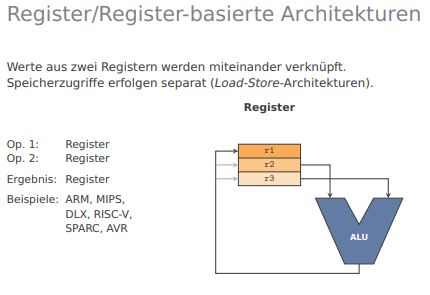
#### **Kapitel 6: Prozessorarchitekturen**

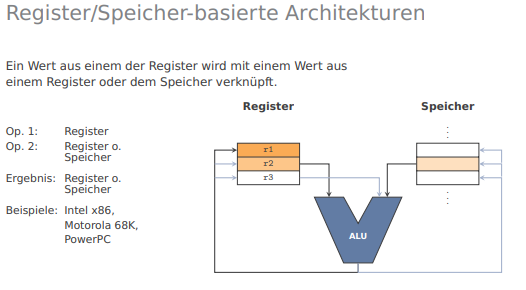
**Video 43: Klassifikation von Prozessorarchitekturen**

Die bekannteste Unterscheidung von Prozessoren liegt darin, wie der Prozessor mit dem Speicher verbunden ist. Bislang wurde in der Vorlesung immer die Von-Neumann-Architektur verwendet. Charakteristisch für die Von-Neumann-Architektur ist das sich Daten und Programme den gleichen Speicher teilen.

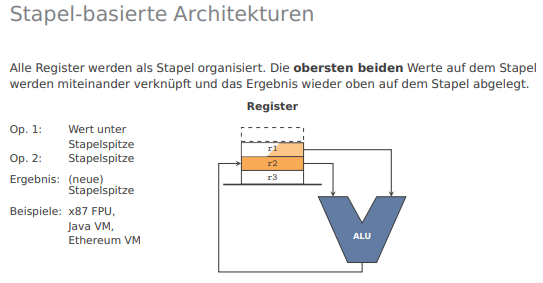
Dabei unterscheidet sie sich von der Harvard-Architektur, diese verwendet zwei Speicher, einen für Instruktionen und einen für Daten. Jeder Speicher kann für seinen jeweiligen Einsatzzweck optimiert angebunden werden, wodurch es zu weniger Stau auf dem Systembus kommt. Außerdem schützt die Trennung von Code und Daten, davor, das Instruktionen fälschlicherweise überschrieben, oder Daten als Instruktionen interpretiert werden. Der größte Nachteil der Harvard-Architektur ist, das man beim Systementwurf festlegen muss, wie man den Speicher aufteilt. Zu diesem Zeitpunkt kennt man aber in der Regel nicht die zukünftigen Anwendungen im System nicht. Von-Neumann ist weiter verbreitet.

Eine andere Art der Prozessorklassifizierung ist die Anbidnung der ALU innerhalb der CPU. Man betrachtet dabei, wie flexibel Quellen und Ziele verschaltet werden können. Hier gibt es vier Varianten, Register/Register-, Register/Speicher-, Akkumulator- und Stapelbasiert.









**Video 44: CISC am Beispiel der x86-Architektur**

Eine weitere Art, Prozessorarchitekturen einzuordnen, ist nach der Komplexität des Befehlssatzes. Dabei unterscheidet man zwischen RISC (Reduced Instruction Set Computer) und CISC (Complex Instruction Set Computer). Bei ARM handelt es sich um eine RISC-Architektur.

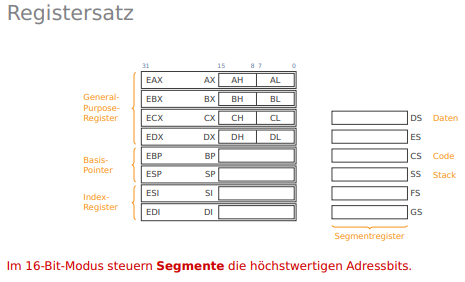
RISC verfügt über wenige elementare Maschinenbefehle, ermöglicht aber schlanke Pipelines, für welche der Richtwert ein Taktzyklus pro Stufe ist. Es verfügt ebenfalls über eine kompakte Instruktionskodierung, welche oft orthogonal ist. Sie wird meist in Load-Store Architekturen verwirklicht.

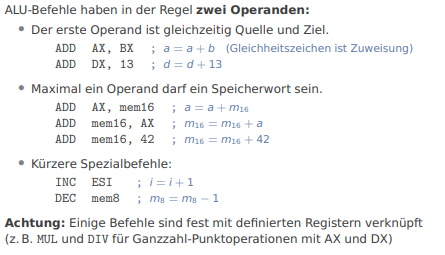
CISC verfügt über viele Spezialbefehle, welche zum Teil in Mikroprogrammen realisiert sind. Die Optimierung von CISC ist sehr schwierig, da ihre Befehlssätze oft organisch über Prozessorgenerationen gewachsen sind, wodurch Befehlssätze unübersichtliche Ausmaße angenommen haben. RISC und CISC existieren selten in Reinform.

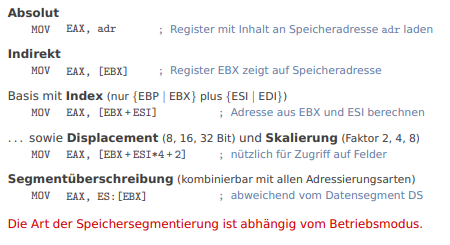
Es sollte erwähnt werden, das RISC und CISC selten in Reinform existieren, z.B. heutige CISC-Architekturen verfügen über einen RISC-Kern.

Nun wird eine x86-Architektur betrachtet.







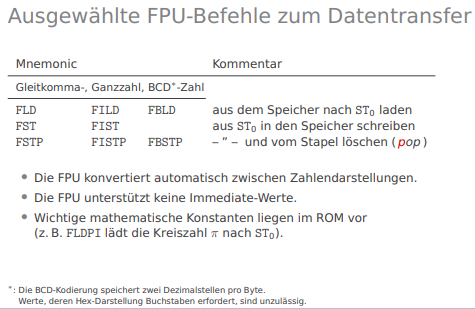


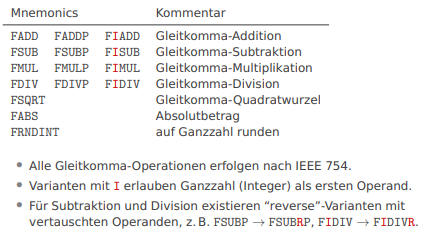
**Video 45: Gleitkommaeinheit**

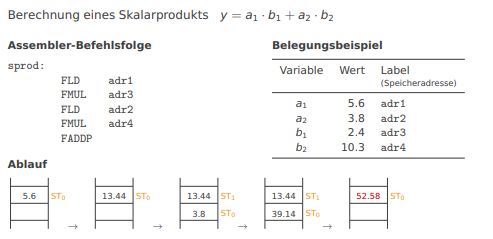
Als ein Beispiel für stapel basierte Architekturen dient der Coprozessor für Gleitkommazahl aus der x86 Familie, die x87 FPU (Floating Point Unit). Der Coprozessor verfügt über eigene Register und eine Befehlssatzarchitektur, die für die Bearbeitung von Gleitkommaberechnungen ausgelegt ist. Die FPU verfügt über 8 80-Bit-Register, die als full descending Stapel organisiert sind. Die Inhalte werden grundsätzlich als Gleitkommazahlen interpretiert und zwar mit 64-Bits für die Mantisse, 15 Bits für den Exponenten, und ein Vorzeichenbit.

Ein 16 Bit breites Statusregister enthält den Zeiger auf die Stapelspitze, ein Flag das den Stapelüberlauf oder Unterlauf anzeigt, sowie weitere Flags für die Ergebnisse von Vergleichsoperationen und Fehlerausgänge der Gleitkomma-ALU. Die Register werden über ST0, …, STn relativ zur Stapelspitze angesprochen, das heißt der Wert in ST0 befindet sich

in ST1 sobald ein weiteres Element auf dem Stapel abgelegt wird.

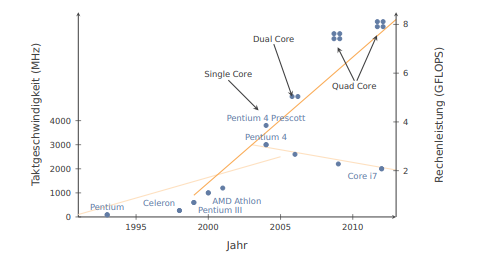




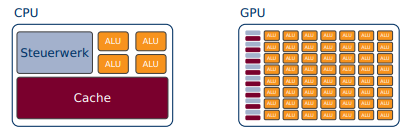


**Video 46: Datenparallele Architekturen am Beispiel einer GPU**

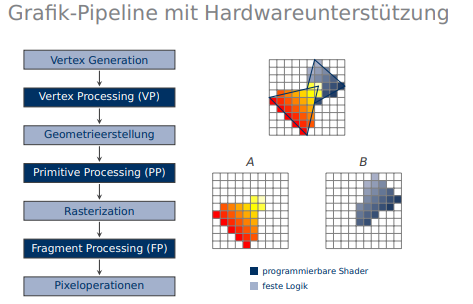
Bis 2005 wurden bei den x86-Prozessoren Leistungssteigerungen vor allem durch höhere Taktfrequenzen erreicht wurden. Jedoch geht es nicht weit über 3 - 4 GHz hinaus. Das sind die Taktgeschwindigkeiten die auch heute übliche Rechner verwenden. Grund dafür ist, dass man hier den physikalischen Grenzen sehr Nahe gekommen ist. Je höher der Takt, desto mehr Spannung braucht eine Schaltung und desto mehr erwärmt sie sich. Wenn die Wärme nicht abgeführt werden kann, treten unerwünschte Stromflüsse ein, der Prozessor verrechnet sich, das System stürzt ab und im schlimmsten Fall verkürzt sich die Lebensdauer der CPU erheblich. Weshalb die Taktgeschwindigkeiten seit 2005 tendenziell wieder gesunken sind.



Trotzdem sind moderne CPUs leistungsfähiger. Die Hersteller erreichen dies, durch die Integration von mehr CPU-Kernen in einem Chip. Diese können bei entsprechender Programmierung parallel unterschiedliche Aufgaben erledigen und so in Summe pro Zeiteinheit ein Vielfaches der Anzahl der Operationen erledigen als eine CPU mit nur einem Kern. Die SISD (Single Instruction Single Data) verfügt, wie der Name sagt über 1 Befehlsstrom, und dadurch über nur eine Instruktion. Die MIMD ist ein Mehrkern-Prozessor, welcher wie der Name sagt über mehr Prozessorkerne und dadurch über mehr Instruktionen verfügt. Die MISD garantiert zuverlässige Systeme und weniger Fehler. Computerspiele sind ein Treiber der Hardware-Entwicklung. Moderne Grafik Engines berechnen mit Hardwareunterstützung mehrere Multiplikationen und dutzende Additionen pro Pixel. Jedoch ähneln sich die Aufgaben pro Pixel enorm, weshalb man Grafikkarten massiv datenparallel auszulegen.



Eine CPU verfügt über ein kompliziertes Steuerwerk, eine oder nur wenige ALUs und sehr viel Cachespeicher zur Beschleunigung des Speicherzugriffs. Eine GPU nutzt jeden Quadratnanometer für ALUs. Diese sind in Zeilen organisiert und teilen sich pro Zeile ein Steuerwerk. Die GPU ist organisiert wie ein alter Industriebetrieb, bei welchem alle Maschinen, z.B. Webstühle parallel an einer Welle hängen, die die Kraft verteilt. GPUs erreichen maximale Leistung genau dann, wenn viele gleichartige Daten, auf die gleiche Weise zu bearbeiten sind. Dies funktioniert allerding nur dann, wenn die Zwischenergebnisse einzelner ALUs den Kontrolfluss nicht beeinflussen (geringe Datenabhängigkeit).

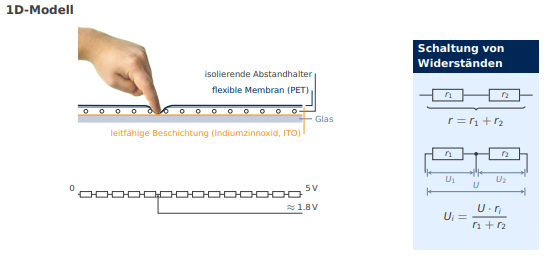


Eine typische Grafik-Pipeline kann man in 7 Stufen einteilen. In der Vertex Generation werden Punkte im dreidimensionalen Raum angeordnet, so das eine Szene entsteht. Diese werden im nächsten Schritt, dem Vertex Processing (VP), jeweils einzeln verarbeitet. Bei der Geometrieerstellung werden Punkte zu geometrischen Primitiven, oft Dreiecken, verbunden. Anschließend wird jedes Primitiv im Primitive Processing (PP) weiterverarbeitet. Zum Beispiel werden Normalvektoren für die Beleuchtung und Texturerstellung berechnet. Als Rasterization bezeichnet man den Übergang von der Vektor- zur Pixelgrafik. Diese passiert zunächst für jedes Primitiv einzeln, und wird dann auch als Fragment bezeichnet. Auch jedes Fragment kann in der folgenden Stufe auf Pixelebene bearbeitet werden. In der letzten Stufe werden die Fragmente unter Beachtung der Tiefeninformation zu einem Bild zusammengesetzt.

#### **Kapitel 7: Ein/-Ausgabe**

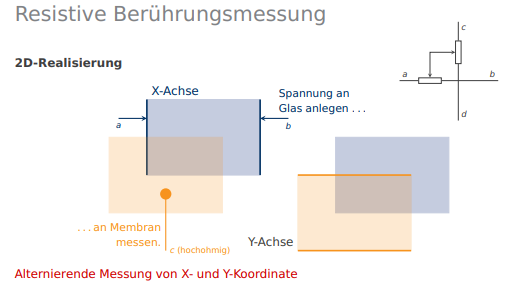
**Video 47: Resistive Berührungsmessung**

Bei einem 1D-Modell werden die berührungsempfindlichen Bauteile auf ein Display montiert und durch eine Glasscheibe von ihm getrennt. Ganz oben befindet sich eine durchsichtige Membran aus PET. Glas und Membran werden durch ein Gitter aus flexiblen Abstandshaltern getrennt. Sowohl unterhalb der Membran als auch oberhalb des Glases wird eine leitfähige Beschichtung aus ITO (Indiumzinnoxid) aufgebracht. Es ist ein durchsichtiger Halbleiter. Die auf dem Glas liegende ITO-Schicht wird nun so dotiert, dass sie pro Streckeneinheit einen festgelegten kleinen Widerstand hat.



Wenn die flexible Membran durch Druckeinwirkung mit Glas in Verbindung gebracht wird, greift die Schicht unter der Membran eine Spannung ab. Die Spannung ist proportional zur Lage des Berührpunktes auf der Strecke von 0 bis 5V. Im Beispiel 5V geteilt durch 14 Teilabschnitte im Modell mal 5 Abschnitte von links vom Berührpunkt ist ca. 1.8V. Die gemessene Spannung lässt sich digitalisieren und als Binärzahl in einer Digitalschaltung weiterverarbeiten.

In der 2D-Realisierung werden die Widerstände an den ITO-Schichten so geschaltet, das auf dem Glas die Spannung entlang der x-Achse abfällt, und auf der Membran entlang der y-Achse von oben nach unten. Die Anschlüsse a und b sind links und rechts am Glas, und die Anschlüsse c und d oben und unten an der Membran.



Die Messung der Koordinaten eines Berührpunkts geschieht in zwei Schritten; zunächst wird eine Spannung an das Glas angelegt, zwischen a und b, und an der Membran gemessen. Daran erhält man die horizontale Position, bzw. die x-Koordinate des Berührungspunktes. Dazu schaltet man zwischen c und dem Eingang einen relativ hohen Widerstand, um die Messung von der vertikalen Position weitgehend unabhängig zu machen. In einem zweiten Schritt legt man die Spannung an der Membran an, zwischen c und d, und misst am Glas. Damit erhält man die y-Koordinate. Ein Schaltwerk wechselt beide Schritte mehrmals pro Sekunde ab. Die digitalen Messergebnisse werden in Registern zur weiteren Verarbeitung gespeichert.

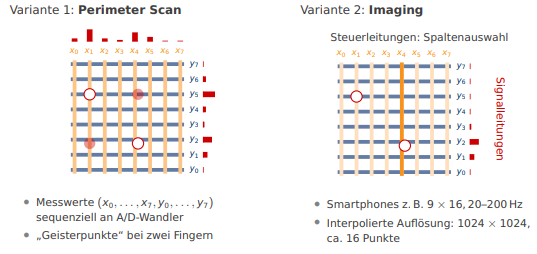
**Video 48: Projiziert-kapazitive Berührungsmessung**

Bei der kapazitiven Berührungsmessung besteht der Sensor aus einem Wechselstromkreis, der ganz klassisch aus Kondensator und Spule oder Timer und Widerstand aufgebaut ist. Menschliche Körper wirken wie Kondensatoren. Wenn Menschen einen Leiter eines Schwingkreises berühren, oder nur in die Nähe kommen, ändert sich die Gesamtkapazität aller Kondensatoren im Schwingkreis und damit deren Sequenz. In einer Schaltung wird die Frequenzabweichung von einem Controller ausgewertet und bewirkt im einfachsten Fall das Ein- und Ausschalten einer Lampe.

Da Menschen als Kondensatoren dienen, kann man mit einem Finger Ladung von einer X-Elektrode stehlen, welche sich unter einer Glasscheibe befindet. Das führt dazu, dass sich die Kapazität zwischen den Elektroden verändert. Bei Berührung werden die Feldlinien über die berührungsempfindliche Oberfläche hinaus projiziert.



Das Ausleseverfahren bei “Pro-Cap”-Touchscreens kann über 2 Varianten verlaufen. Die erste Variante, der Perimeter Scan verfügt über Messwerte (X-Elektroden und Y-Elektroden), welche sequenziell an einen A/D-Wandler angeschlossen sind. Die Berührungsposition kann am Rand anhand der Messwerte bestimmt werden. Diese Methode funktioniert nicht für mehrere Finger gleichzeitig, da der A/D-Wandler dann nicht mehr genau sagen kann, wo sich der Berührungspunkt befindet, wodurch “Geisterpunkte” bei der Nutzung von 2 Fingern entstehen.

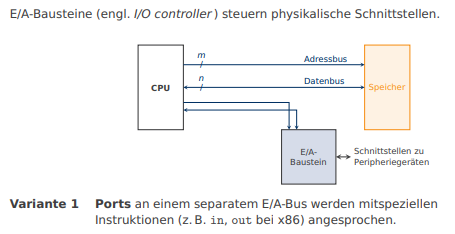


Die zweite Variante ist das sogenannte “Imaging”, bei welchem ein Schaltwerk zyklisch alle Spalten von links nach rechts auswählt, und die jeweiligen Messwerte der Signalleitungen abspeichert. Damit kann genau festgestellt werden, wo genau sich die Berührungspunkte befinden, wodurch die Nutzung von mehreren Fingern ermöglicht wird.

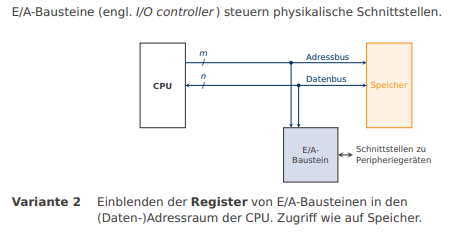
**Video 49: Ansteuerung von E/A-Bausteinen**

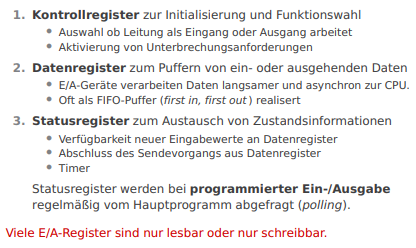
Der Systembus ist in vielen Systemen die Universalschnittstelle für die Kommunikation der CPU mit allen anderen Systemkomponenten. Ein E/A- Baustein ist eine Schnittstelle zu Peripheriegeräten. Es gibt 2 Varianten, wie man diese ansteuern kann.

Die erste Variante sind Ports, welche an einem separaten E/A-Bus mit speziellen Instruktionen angesprochen werden. Ein Nachteil dieser Variante ist, dass die Adress- und Datenwortbreite durch die Anzahl der zur Verfügung stehenden Pins bei der CPU stark begrenzt ist.



Bei der zweiten Variante werden die Register der E/A-Bausteine in den Adressraum der CPU eingeblendet. Das bedeutet technisch, das ein Schaltnetz die Schnittstelle zum Speicher für wenige Adressen im allgemeinen Adressraum auf den E/A-Baustein umleitet. Dabei lassen sich E/A-Bausteine wie Speicher ansprechen.

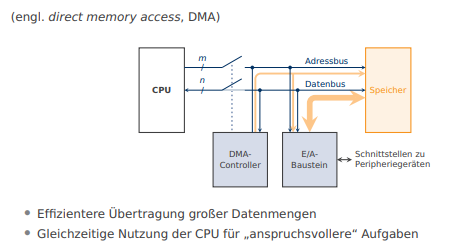




**Video 50: Speicherdirektzugriff (DMA)**

Wenn die E/A-Bausteine große Datenmengen liefern oder verarbeiten müssen, stößen die E/A-Bausteine an ihre Grenzen. Beim Laden von Dateien müssen z.B. große Mengen an Daten von einem Datenträger in den Hauptspeicher bewegt werden. Auch Grafikkarten benötigen regelmäßig sehr viele Informationen aus dem Hauptspeicher. Der Weg über E/A-Register und die CPU ist deshalb ineffizient. Erstens müssen Daten zweimal über den Systembus, und damit einmal mehr als nötig und Zweitens wird die CPU mit unnötigen Aufgaben wie Datenschaufeln blockiert.

Deshalb unterstützen sogut wie alle Systeme Speicherdirektzugriff (DMA). Dazu wird ein weiterer E/A-Baustein, der DMA-Controller in das System integriert, und zwar in privilegierter Stelle. Der Controller verfügt über die Möglichkeit auf Anweisung der CPU diese selbst vorübergehend vom restlichen System abzukoppeln. Daraufhin übernimmt der DMA-Controller die Kontrolle über den Adressbus und kann den Datenaustausch zwischen anderen E/A-Bausteinen und dem Speicher organisieren. Die CPU ist dafür in der Zwischenzeit für anspruchsvollere Aufgaben frei.

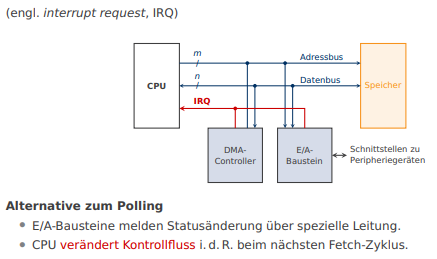


Im besten Fall arbeitet der DMA-Controller mit dem E/A-Baustein zusammen. Dann muss der DMA-Controller nicht mehr zwischen den Adressen des E/A-Datenregisters und den Quell- oder Zieladressen im Speicher vermitteln. Stattdessen legt er lediglich die Speicheradresse auf den Adressbus und der E/A-Baustein weiß welche Daten er beim Lesezugriff auf den Datenbus anlegen muss (aus Sicht des Speichers ist das dann ein Schreibzugriff). Auf diese Weise muss jedes auszutauschende Datenbyte nur 1-mal über den Systembus laufen. In der Praxis wechseln sich CPU und DMA Zugriffe auf den Systembus in schnelle Folge ab.

**Video 51: Unterbrechungsanforderung (IRQ)**

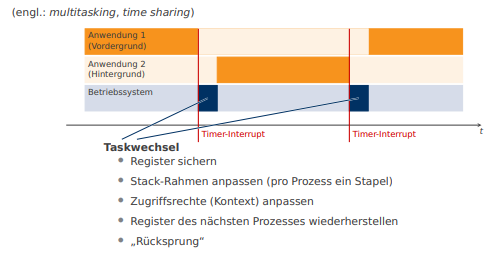
Das Polling ist die programmierte Ein/- und Ausgabe. Ein Nachteil des Pollings ist, das die CPU regelmäßig alle relevanten E/A-Statusregister auslesen muss um auf Ereignisse reagieren zu können. Das kostet Taktzyklen und schränkt die Programmierung unnötig ein.

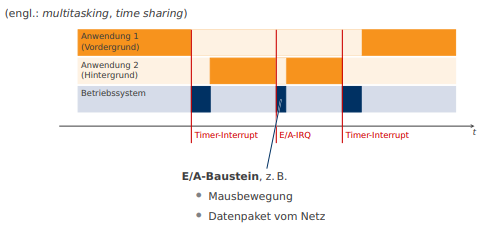
Hilfe bietet hier die Interrupt Request (IRQ).



Technisch funktioniert dies über eine zusätzliche Eingangsleitung an der CPU, über die ausgewählte E/A-Bausteine nach entsprechender Aktivierung über deren Statusregister die CPU in ihrem Kontrollfluss unterbrechen können. Das heißt, dass diese beim Fetch des nächsten Zyklus ein Instruktionswort an einer vordefinierten Stelle einliest. Dort liegt in der Regel ein vom Betriebssystem bereitgestellter Programmteil, der als Interrupt Handler bezeichnet wird. Der Interrupt Handler sichert alle Register und Flags des unterbrochenen Programms und kümmert sich dann um die Bearbeitung der anstehenden E/A-Aufgaben.

Am Ende stellt der Interrupthandler alle Register wieder her und übergibt die Kontrolle zurück an das Hauptprogramm, and der unterbrochenen Stelle.

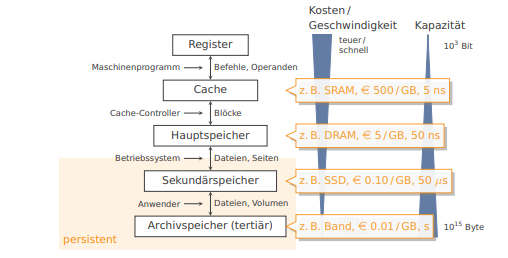




#### **Kapitel 8: Speicher**

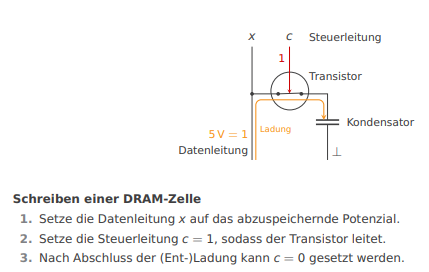
**Video 52: Speicherhierarchie**

Moderne Rechner vereinigen viele verschiedene Arten von Speicher. Damit können die Anforderungen an Speicherkapazität, Persistenz und Zugriffsgeschwindigkeit unter Berücksichtigung der Kosten so gut wie möglich erfüllt werden. Grundlegend dafür ist das Konzept der Speicherhierarchie, die das Zusammenspiel der unterschiedlichen Speicherarten veranschaulicht.

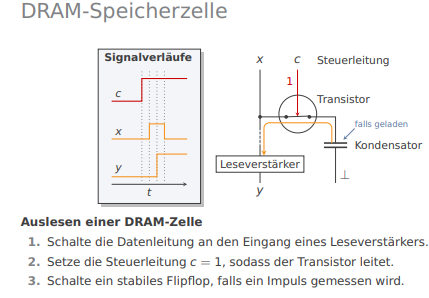


**Video 53: Dynamische RAM**

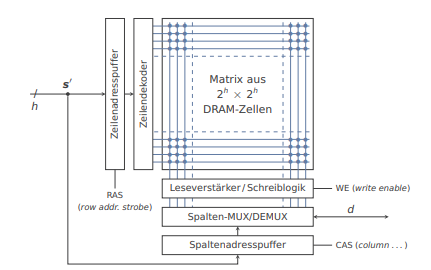
Dynamisches RAM speichert Informationen im Ladungszustand eines Kondensators. Im Gegensatz zu Flipflops kommt keine Rückkopplung zum Einsatz. Damit benötigt jedes DRAM-Bit nur einen Transistor und ist wesentlich dichter und billiger realisierbar als statisches SRAM oder Register mit Taktflankensteuerung.



Bei der DRAM-Speicherzelle hängen an der Datenleitung viele DRAM-Zellen, jede von ihnen hat eine eigene Steuerleitung die mit den Ausgang eines Dekodierers verbunden ist, der die Adresse als Eingabe erhält. Beim Schreibzugriff wird an die Datenleitung x das abzusichernde Potenzial angelegt. Zum Beispiel +5V für eine 1 bei den Pegeln. Dadurch wird die Steuerleitung c auf 1 gesetzt. Dies schließt den elektrischen Schalter im Transistor, so das sich der Kondensator gegenüber Erde auflädt. Würde man 0V laden, dann würde sich der Kondensator entladen, wenn er davor geladen war. Der Ladezustand bleibt für einige Zeit erhalten wenn die Steuerleitung c = 0 gesetzt wird. Dadurch kann die Datenleitung für Schreib- oder Lesezugriff für andere DRAM-Zellen benutzt werden.

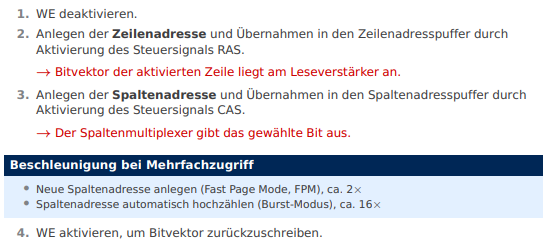


Der DRAM vergisst seinen Inhalt in wenigen Sekunden. Alle 60 Millisekunden werden alle DRAM Zellen ausgelesen und mit ihren jetzigen Werten neu beschrieben. Ein Steuerwerk im DRAM-Baustein übernimmt diese Aufgabe (nicht die CPU).



Ein DRAM-Baustein besteht aus einer Matrix bestehend aus 2h x 2h quadratisch angeordneten DRAM-Zellen. Die Steuerleitungen jeder Zelle werden zeilenweise mit den Ausgängen eines Zeilendekoders verbunden. Die Datenleitungen verlaufen spaltenweise und sind mit einem kombinierten MUX/(DE)MUX verbunden, der Bits auswählt, welche mit den Datenbus d verbunden werden sollen. Da der Multiplexer nur mit stabilen Pegeln arbeitet, ist der Leseverstärker dazwischen geschaltet. Dieser enthält zugleich auch die Schreiblogik, und wird durch eine zusätzliche Steuerlogik, WE (Write Enable) kontrolliert.

Die Dekodierer und MUX benötigen stabile Adresseingaben, weshalb h-Bit-Register als Puffer vorgeschaltet sind. Der Adressbus innerhalb des Bausteins ist mit h-Bit halb so breit wie der Adressbus des Systems. Deshalb wird er in der Skizze mit s’ gekennzeichnet, und nicht s.

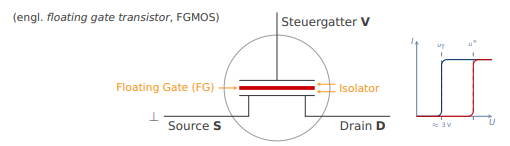


Nun wird der Lesezugriff betrachtet. An erster Stelle wird WE (Write Enable) deaktiviert, daraufhin wird die Zeilenadresse an s’ angelegt, und die Steuerleitung RAS auf 1 gesetzt. Damit wird die Zeilenadresse in den Zeilenadresspuffer übernommen. Alle DRAM-Zellen in der Zeile der dekodierten Steuerleitung machen ihre Transistoren leitend. Wenn ihre Kondensatoren geladen waren, übergeben sie einen kurzen Impuls spaltenweise nach unten. Damit liegt am Leseverstärker der Bitvektor der kompletten ausgewählten Zeile an.

Nun legt man die Spaltenadresse bei s’ an und aktiviert die Steuerleitung CAS. Damit gibt der MUX das ausgewählte Bit aus. Zum Abschluss des Lesezugriffs muss WE (Write Enable) aktiviert werden, um den Bitvektor wieder zurück zu schreiben. Sonst wäre der Inhalt aller nicht ausgewählten Spalten verloren.

**Video 54: Flash-Speicher**

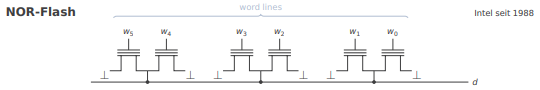
Flash-Speicher verdrängt Festplatten als persistenter Massenspeicher, da er ohne mechanische Komponenten auskommt. Das wird durch spezielle Feldeffekttransistoren mit isoliertem Gatter (FGMOS, Floating Gate Transistor) ermöglicht. Wie bei jedem Transistor überprüft ein Steuereingang ob die Verbindung zwischen den Pins leitend ist oder nicht. Das Floating Gate wird nun isoliert zwischen das Steuergatter und das gesteuerte Gatter eingebaut. Unter der kontrollierter Ausnutzung von quantenmechanischen Tunneleffekten ist es trotzdem möglich das isolierte Gatter aufzuladen. Dazu benötigt man Spannungen in der Größenordnung von 10V. Sind Ladungsträger im Floating Gate “eingesperrt”, verharren sie dort unabhängig von der Versorgungsspannung und können dadurch Informationen persistent behalten.



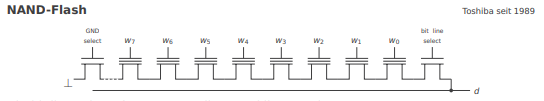
Wenn das FG nicht geladen ist, reagiert der FGMOS wie ein Transistor: er leitet, wenn die Spannung am Steuergatter größer als uT ist.

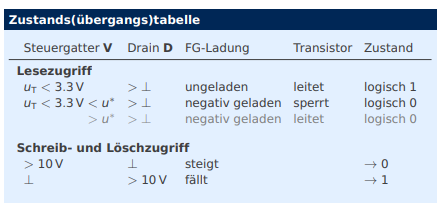
Das NOR-Flash besteht aus vielen Feld-Transistoren, welche sich eine Datenleitung teilen, die mit Drains verbunden ist. Jeder Transistor verfügt über eine eigene Steuerleitung wi. Die

Datenleitung d geht auf 0, wenn an mindestens einer word line wi eine Spannung über dem Arbeitspunkt des Feldeffekttransistoren anliegt. Bei einem NOR-Flash kann man Bits wahlfrei und direkt auslesen in dem man alle wi auf 0 setzt und nur die Steuerleitung der adressierten Zellen auf einen Wert zwischen uT und u\* bringt. Es ist daher gut geeignet um direkt mit dem Systembus verbunden zu werden.

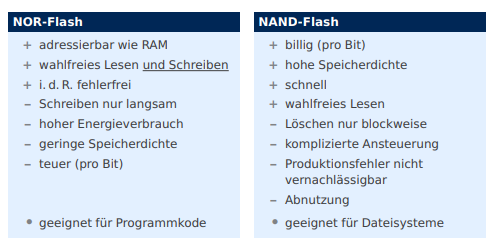


Das NAND-Flash besteht aus Ketten von Transistoren, deren Source-Leitungen mit den Drains der benachbarten Transistoren verbunden werden. Jede Kette ist mit einer Selectleitung auf der Datenleitung schaltbar und hat eine weitere Steuerleitung (GND select), um das andere Ende mit Erde zu verbinden. Die Bitline geht auf 0, wenn an allen WL (Word Lines) wi eine Spannung über den jeweiligen Arbeitspunkten der Feldeffekttransistoren anliegt. Beim Lesen muss man an alle wi eine Spannung größer als u\* anlegen, außer an derjenigen Zelle, deren Wert man auslesen möchte. Dort legt man eine Spannung zwischen uT und u\* an, um den Arbeitspunkt zu ermitteln.



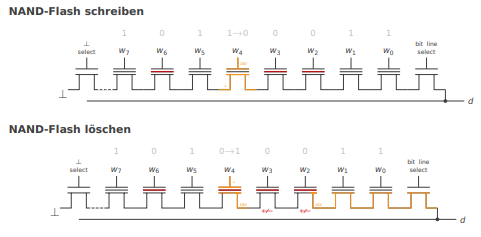


**Video 55: Besonderheiten bei NAND-Flash**



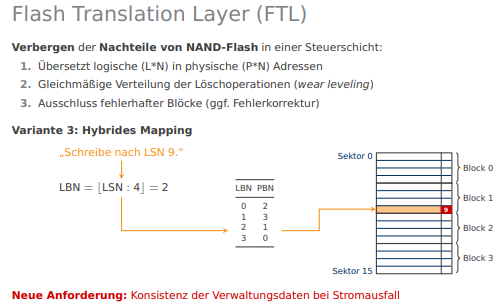
Ein Beispiel für NAND-Flash; sind die roten Floating Gates geladen, die dunklen nicht. Damit repräsentiert diese Bitline einen logischen 8-Bit Zustandsvektor, dessen Werte in grau angeschrieben sind. Möchte man die 1 bei w4 auf 0 ändern, dann benötigt man einen Schreibzugriff.

Um das Floating Gate zu laden, müssen Source und Drain auf Erde, und die Steuerleitung w4 auf +10V geschaltet werden. Durch die Verkettung im NAND-Flash kann man Source und Drain nicht direkt ansteuern, weshalb alle anderen Steuerleitungen ein Potenzial größer als u\*, aber deutlich kleiner als 10V annehmen müssen. Außerdem wird die Datenleitung d auf Erde geschaltet. Dadurch lädt sich das Floating Gate, und das Bit ändert wie gewünscht seinen logischen Zustand von 1 auf 0. Auf diese Art kann man jedes Bit beliebig von 1 auf 0 schalten.



Um das NAND-Flash zu löschen (Bit im Floating Gate von 0 auf 1 setzen), muss man die Datenleitung d auf 10V setzen. Allerdings muss man nun den Weg aller Feldtransistoren rechts von der Zelle nehmen, welche man verändern möchte. Selbst wenn man an deren Steuergatter Spannungen oberhalb der jeweiligen Arbeitspunkte anliegt, um sie leitend zu machen, liegt an ihnen nun ebenfalls 10V an Drain an. Dadurch wird man der Gefahr ausgesetzt, das man die geladenen Floating Gates entlädt, wodurch die anderen logischen 0en ebenfalls zu 1en kippen. Deshalb ist die Vorgehensweise bei NAND-Flashs immer ganze Bitlines von rechts nach links zu löschen, bevor man sie wieder beschreiben kann. Damit erklärt sich die Einschränkung, dass NAND-Flash nur blockweise gelöscht wird, wodurch man keine wahlfreien Schreibzugriffe ermöglichen kann.

Der Flash Translation Layer (FTL) ist eine Steuerschicht, die den Zweck erfüllt, die Nachteile von NAND-Flash vor dem Rest des Systems zu verbergen. Der FTL bildet die logischen Adressen auf dem Systembus in neue physische Adressen in dem Flash-Baustein ab. Diese Abbildung ist bewusst flexibel, so dass Löschoperationen bewusst über alle Zellen verteilt werden können (Wear Leveling). Sollten dennoch Zellen versagen, so erkennt der FTL das, und verwendet diese Zellen zukünftig nicht mehr. Dadurch wird die Lebenszeit von Flash-Speichern erhöht, sie verlieren jedoch an Kapazität.

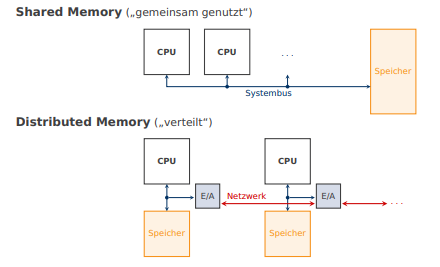


Jede Implementierung eines FTLs benötigt eine Tabelle, die die dynamische Abbildung von logischen auf physische Adressen verwaltet. Die einfachste Variante ist das Sektor-Mapping.

Jeder physische Block (Elemente die gelöscht werden müssen), wird in mehrere gleichgroße Sektoren eingeteilt und durchnummeriert. Sektoren sind die Einheiten, die das System als Ganzes schreibt oder aktualisiert. Zum Beispiel: Bei der Anforderung vom System “Schreibe nach LSN 9.”, wird mithilfe der Tabelle ein physischer Sektor ermittelt, wo die Information abgelegt wird. Durch binäre Rechtsverschiebung der physischen oder logischen Nummer erhält man die entsprechende Blocknummer. Bei einem weiteren Schreibzugriff auf den logischen Sektor Nummer 9 könnte man nicht den ganzen Block löschen. Stattdessen würde ein anderer Sektor gewählt werden, wo die Daten hingeschrieben werden. Daraufhin wird der Eintrag von dem FTL in der Tabelle angepasst, und der alte physische Sektor 3 wird als ungültig markiert. Erst sobald alle physischen Sektoren in einem Block als ungültig markiert wurden, wird der gesamte Sektor gelöscht und wieder als frei markiert.

**Video 56: Speicherarchitekturen für Mehrprozessorsysteme**

Die letzte Art, wie man zwischen der Klassifikation von Speichersystemen unterscheiden kann, ist über die Anbindung des Speichers. Wenn alle Prozessoren auf denselben Systembus zugreifen, und damit Daten und Programme auf einem gemeinsam genutzten Speicher ablegen, dann spricht man von einer “Shared Memory” Architektur. Diese Architektur skaliert bis zu einem gewissen Grad, doch irgendwann wird der Systembus zum Flaschenhals. Außerdem wird die Vermeidung von Fehlern durch konkurrierende Zugriffe mehrerer CPUs auf die gleiche Speicheradresse zunehmend komplex und manchmal selbst fehleranfällig.



In der Wissenschaft setzt man deshalb auf eine andere Variante, die “Distributed Memory” Architektur. Hier hat jede CPU ihren eigenen Speicher. Dieser ist vergleichsweise kleiner, steht aber exklusiv zur Verfügung, sodass sich CPUs nicht den Systembus teilen müssen. Jede Art von Kommunikation in “Distributed Memory” Systemen erfolgt aus Sicht jeder einzelnen CPU über ein E/A - Operationen. Dazu werden die Knoten eines solchen “Rechenclusters” über ein speziell dafür ausgelegtes Hochgeschwindigkeitsnetzwerk verbudnen. Ein Nachteil dieser Architektur ist, das Daten und Programme, welche von mehreren CPUs benötigt werden, mehrfach in verschiedenen Speichern des Gesamtsystems zu behalten sind.