D S P

Digitale Signalprozessoren

Rade Kutil

http://www.cosy.sbg.ac.at/~rkutil/teaching/dsp04.html

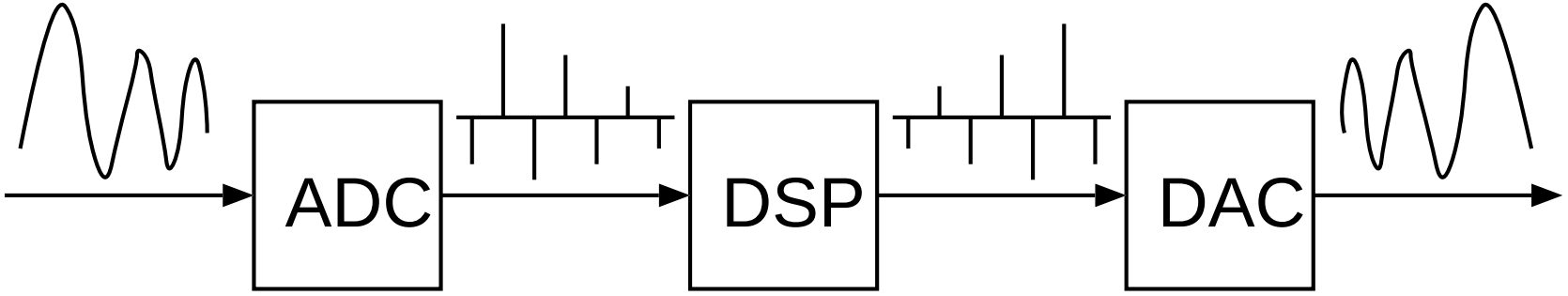
Literatur

- P. Lapsey, J. Bier, A. Shoham, E.A. Lee. **DSP Processor Fundamentals**. Wiley-IEEE Press, 1997. (45 4.7 12)
- S.W. Smith. **The Scientist and Engineer's Guide to Digital Signal Processing**. California Technical Publishing, 1997. (downloadbar unter `http://www.dspguide.com/`)
- L. Wanhammar. **DSP Integrated Circuits**. Academic Press, 1999. (45 4.7 11)
- `http://www.bdti.com/` Webseite der Berkeley Design Technology, Inc., Information zu gängigen DSPs.

Was?

DSP steht normalerweise für **Digital Signal Processing**. D.h. an sich analoge Signale werden digitalisiert und digital weiterverarbeitet. ($\Rightarrow$ **ADC** = Analog-Digital-Conversion, **DAC** = Digital-Analog-Conversion)

Techniker verwenden die Abkürzung DSP meist als **Digital Signal Processor** anstatt "DSP Prozessor". Hier soll das auch gelten.



Signale können durchaus mehrdimensional sein (z.B. Bilder). Sie werden aber zur Verarbeitung meistens serialisiert. D.h. die digitalisierten Daten werden zeilen- oder spaltenweise in den DSP "gefüttert".

Warum?

- Das Hauptanwendungsgebiet von normalen **General-Purpose-Prozessoren** (**GPPs**) ist die Verarbeitung von Text oder anderen logisch strukturierten Daten. D.h. bei der Verarbeitung müssen ständig Entscheidungen getroffen werden, die den weiteren Ablauf des Programms beeinflussen. Dagegen verläuft die Verarbeitung von Signalen regelmäßig und in wiederkehrenden Strukturen.
- Massive Bearbeitung von Fix- oder Fließkommazahlen.
- GPPs sind nicht für I/O optimiert. (Interrupts langsam)
- Dynamische Eigenschaften von GPPs (Cache, etc.) machen die Performance schwer abschätzbar. Bei DSPs ist aber immer mit dem worst case zu rechnen (wegen Echtzeit).

Wo?

Anwendungsbereiche von DSPs:

- Telekommunikation: Leitungsvermittlung, Router, Netzwerke
- Audio: Musik- und Sprachkompression, Denoising, Effektgeräte
- Image: Große Auflösungen, Digitale Kameras, Medizin
- Video: Echtzeitanforderungen
- Radar, Sonar, Seismologie

Wann?

Wann macht der Einsatz von DSPs Sinn?

- Wenn die Geschwindigkeit von normalen Prozessoren nicht ausreicht.
- Bei einfachen Algorithmen und großen Datenmengen.
- Wenn die Produktionskosten ein Hauptfaktor sind.
- Wenn die Entwicklungskosten $nicht$ ausschlaggebend sind.
- Wenn der Stromverbrauch kritisch ist.
- Wenn der Platzbedarf der Hardware kritisch ist.

Wie?

Hochoptimierte Bestandteile Z.B. Fließkommamultiplizierer, die eine Multiplikation innerhalb eines Taktzyklus schaffen.

Spezielle Instruktionen für häufig auftretende Anforderungen Z.B. Schleifensteuerung mit einer Anweisung (ohne Overhead pro Iteration).

On-Chip Memory statt Cache, Multi-Access-Memories.

Mehrfache Bussysteme Z.B. Trennung in Programm- und Daten-Memory.

Fixed Point Arithmetik wenn keine Fließkommazahlen benötigt werden. (Achtung: höherer Entwicklungsaufwand)

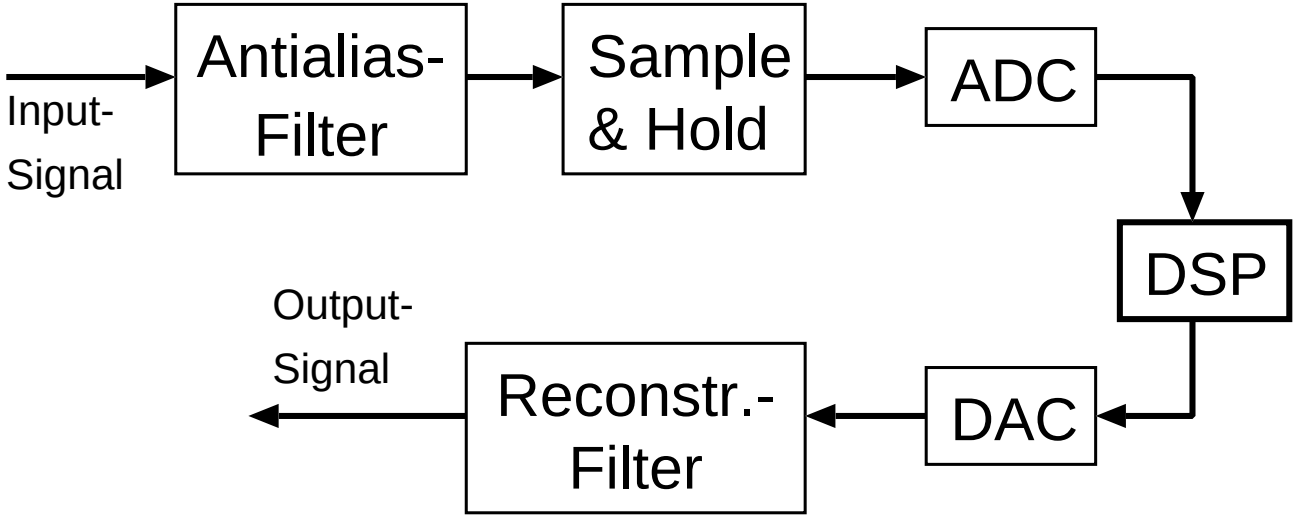
Parallelität Unabhängige Berechnungsschritte gleichzeitig ausführen. Z.B. durch doppelte arithmetische Einheiten, Pipeline-Verarbeitung oder Zusammenschalten mehrerer DSPs.

Fertige Libraries für die wichtigsten Anwendungen (Filtern, FFT, . . .)

DMA zur Kommunikation mit Peripherie.

ADC - DAC

Um Signale mit einem DSP zu verarbeiten, müssen sie zuerst digitalisiert werden.



Antialias-Filter Hohe Störfrequenzen müssen beseitigt werden.

Sample & Hold Das Signal wird für kurze Zeit konstant gehalten, damit der ADC korrekt arbeiten kann.

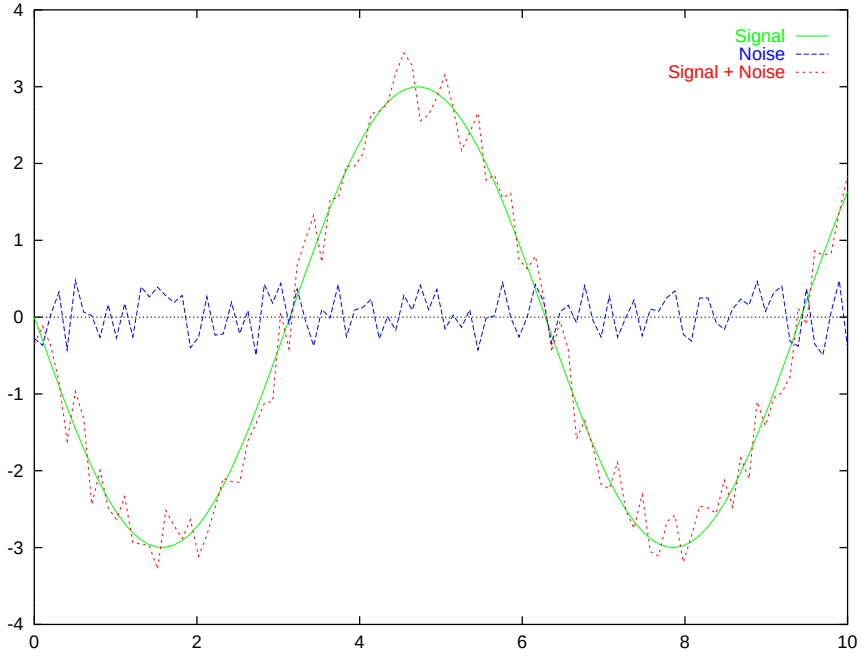
ADC Analog-Digital-Converter

DSP Hier wird das Signal digital verarbeitet.

DAC Digital-Analog-Converter

Reconstruction-Filter Das vom DAC kommende Signal muss geglättet werden.

Noise



Sei $S(t)$ ein beliebiges Signal und $\bar{S}(t)$ das gestörte Signal. Dann ist das **Rauschen (Noise)** definiert als die Standardabweichung der Differenz. Dabei wird angenommen, dass diese Differenz ein Mittel von 0 hat, d.h. das Rauschen hat keinen Gleichanteil (siehe nächste Seite).

$$N(S,\bar{S}) = N(\bar{S}-S) = \sqrt{\frac{1}{T}\int_0^T (\bar{S}(t)-S(t))^2 dt}$$

AC/DC

Einschub:

Für ein Signal $f(t) : [0, T] \longrightarrow \mathbf{R}$ kann man den Mittelwert berechnen mittels:

$$m = \frac{1}{T} \int_0^T f(t)dt \,.$$

Die konstante Funktion $m(t) : [0, T] \longrightarrow \mathbf{R}$, mit $m(t) = m$ für alle t, nennt man dann den Gleichanteil (**Bias**) der Funktion f in Anlehnung an die Elektrotechnik (Gleichstromanteil, Gleichstrom = direct current = DC). Der verbleibende Anteil $g(t) := f(t) - m(t)$ heißt u.A. Wechselanteil (Elektrotechnik: Wechselstromanteil = alternate current = AC). Im Sinne der Fourier-Transformation beinhaltet der Gleichanteil $m(t)$ die Frequenz 0 und $g(t)$ alle anderen Frequenzen.

Um auf den Noise zurückzukommen: Dieser sollte eben keinen Gleichanteil beinhalten. Man beachte, dass das bei Noise, der durch Quantisierung mit Runden (siehe später) zustande kommt, der Fall ist. Bei Abrunden hat der Noise jedoch einen Bias von $-\frac{q}{2}$, wenn q die Quantisierungsschrittweite ist.

Noise - Überlagerung

Überlagern nun zwei Störungen P und Q gleichzeitig das Signal S, dann gilt:

$$\begin{aligned} N^2(S, S+P+Q) &= \frac{1}{T}\int_0^T (P(t)+Q(t))^2 dt \\ &= \frac{1}{T}\int_0^T P^2(t)+Q^2(t)+2P(t)Q(t)dt \\ &= N^2(P)+N^2(Q)+\frac{2}{T}\int_0^T P(t)Q(t)dt \end{aligned}$$

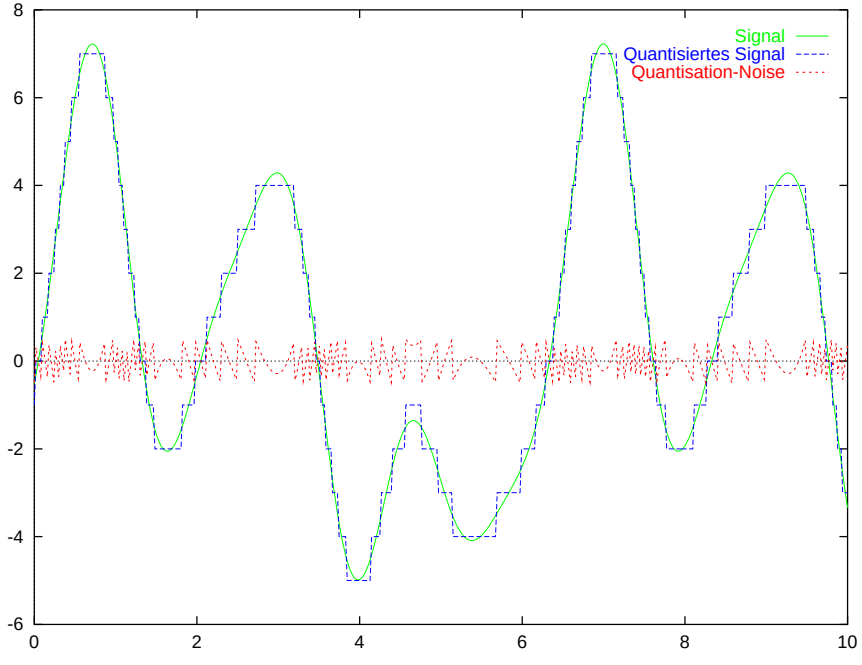
Wenn man nun weiß, dass P und Q statistisch unabhängig (nicht korreliert) sind, dann ist

$$\frac{1}{T}\int_0^T P(t)Q(t)dt \approx 0$$

und somit

$$\boxed{N^2(P+Q) \approx N^2(P)+N^2(Q)}$$

Quantisierung



Bei fixer Quantisierung wird der gemessenen Wert als binäre Fixkommazahl dargestellt. Die Quantisierungsschrittweite ist dabei gleich der Wertigkeit der letzten binäre Stelle (**LSB** = Least Significant Bit). Wenn gerundet wird, bewegt sich der Fehler zwischen $\pm\frac{\mathrm{LSB}}{2}$. Das Fehlersignal $Q(t)$ heißt **Quantisation-Noise** und ist gleichverteilt. Es hat daher eine Stärke von $N = \sqrt{\frac{1}{12}}$ LSB, denn $N^2 = \frac{1}{T}\int_0^T Q^2(t)dt = \frac{1}{LSB}\int_{-LSB/2}^{+LSB/2} x^2 dx = \frac{1}{LSB}\frac{x^3}{3}|_{-LSB/2}^{+LSB/2} = \frac{LSB^2}{3\cdot 2^2}$.

Sampling

Während Quantisierung die $abhängige$ Variable digitalisiert, geht es beim Sampling um die Digitalisierung der $unabhängigen$ Variable (meistens die Zeit).

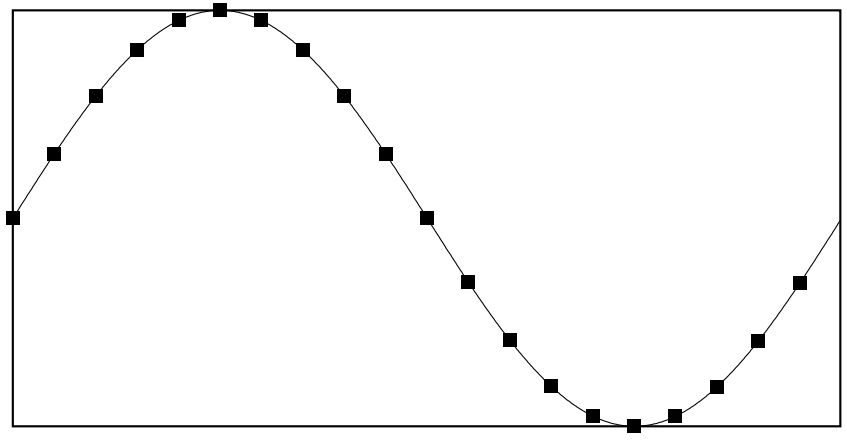
Theoretisch wird dabei das Signal mit einem sog. $Impulse\text{-}Train$ multipliziert. Das ist eine regelmäßige Folge von Einheitsimpulsen.

Praktisch hält man in regelmäßigen Abständen das Signal beim aktuellen Wert fest (**Sample & Hold**). Da der Signalverlauf zwischen den Samples verlorengeht, stellt sich die Frage, inwiefern die Samples das Signal noch repräsentieren.

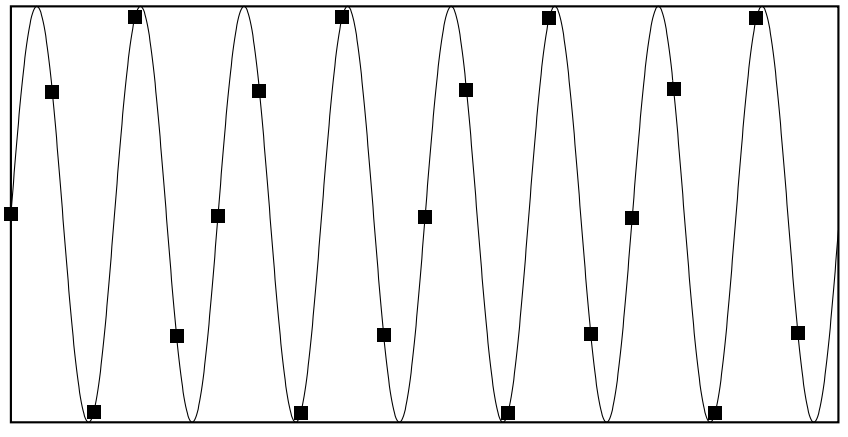
Da jedes Signal in seine Frequenzanteile (Sinus-Schwingungen) zerlegt werden kann ($Fourier\text{-}Transformation$) betrachten wir in der Folge einzelne Schwingungen.

Aliasing

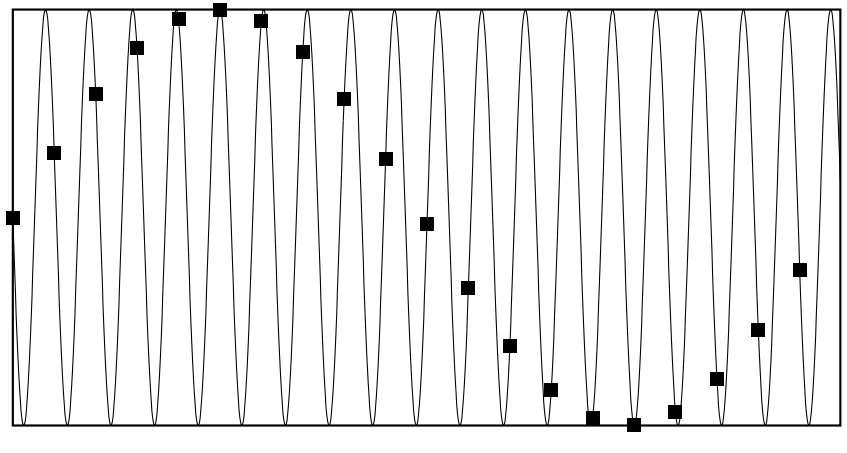
Frequenz = 0.05 Sampling-Rate



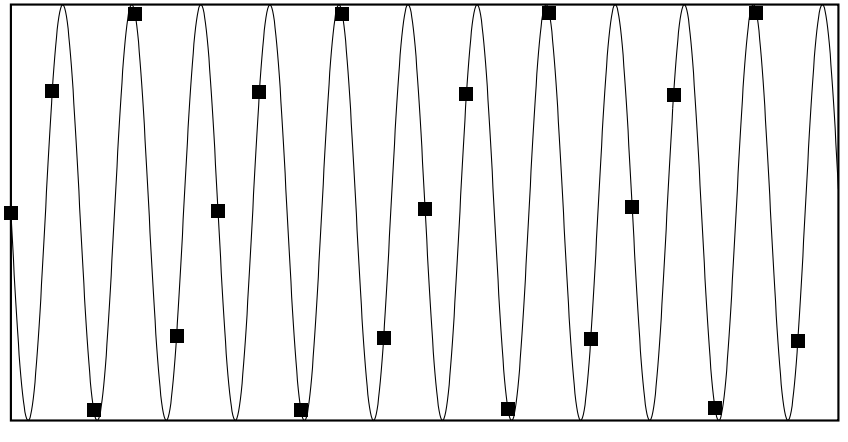
Frequenz = 0.4 Sampling-Rate



Frequenz = 0.95 Sampling-Rate



Frequenz = 0.6 Sampling-Rate



Sampling-Theorem

Eine Funktion gilt als durch eine Menge von Samples repräsentiert, wenn sie aufgrund der Samples eindeutig rekonstruiert werden kann. Dazu muss man die Menge der möglichen Funktionen einschränken. **Nyquist** und **Shannon** haben 1940 das **Sampling-Theorem** gezeigt:

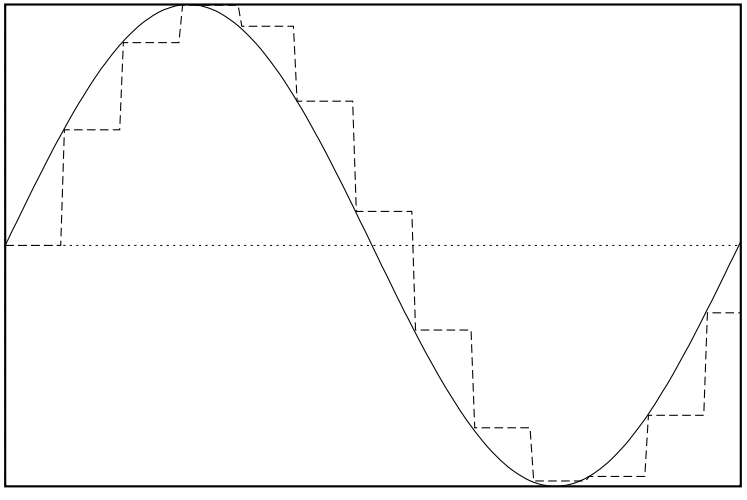
> Ein Signal kann durch Samples exakt repräsentiert werden, wenn es keine Frequenzen oberhalb der halben Sampling-Rate enthält.

Die halbe Sampling-Rate wird daher oft als $Nyquist\text{-}Frequenz$ bezeichnet, bzw. das Doppelte der maximalen Frequenz eines Signals als $Nyquist\text{-}Rate$.

Soll also ein Signal digitalisiert werden (ADC), dann ist darauf zu achten, dass es keine Frequenzen oberhalb der Nyquist-Frequenz enthält. Notfalls ist das Signal vorher analog zu filtern (*Antialias-Filter*).

DAC - Digital-Analog-Conversion

Da das gesamplete Signal nur an bestimmten Punkten genau bekannt ist, stellt sich die Frage, welchen Wert das Signal zwischen diesen Punkten haben soll. Der erste Ansatz heißt **0-th order hold**, d.h. der letzte bekannte Wert wird bis zum nächsten Sample-Punkt beibehalten.



Durch die Ecken werden Frequenzen oberhalb der Nyquist-Frequenz in das Signal eingebracht. Wird dieses Signal jedoch **gefiltert**, so dass alle Frequenzen oberhalb der Nyquist-Frequenz abgeschnitten werden, dann sollte ziemlich genau das originale Signal wieder herauskommen.

Bemerkung: Das stimmt so nicht ganz, da bei diesem Verfahren Frequenzen, die nahe an der Nyquist-Frequenz liegen, um bis zu 50% abgeschwächt werden.

Lineare Systeme

Ein System ist hier eine Funktion, die einer Folge von Eingangswerten ($x = (x[0], x[1], x[2], \ldots)$) eine Folge von Ausgangswerten ($y = (y[0], y[1], y[2], \ldots)$) zuordnet:

$$y = f(x)$$

oder genauer

$$y[k] = f(x[0], x[1], x[2], \ldots)[k] \,.$$

Ein System f gilt als **linear**, wenn:

$$f(kx) = kf(x) \quad \text{und} \quad f(x_1 + x_2) = f(x_1) + f(x_2)$$

Zusätzlich gilt ein System f als **shift-invariant**, wenn:

$$f(x[0], x[1], \ldots)[k - l] = f(x[0 - l], x[1 - l], \ldots)[k]$$

oder abstrakter:

$$\lambda_l f(x) = f(\lambda_l x)$$

wobei der Operator λ_l eine Folge um l Schritte nach rechts verschiebt.

Impulsantwort (Impulse Response)

Bei kontinuierlichen Funktionen ist der Fall etwas komplizierter, aber bei diskreten können wir den **Impuls** definieren als die **Delta-Funktion**:

$$\delta(k) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \end{cases}$$

Als **Impulsantwort** wird die Antwort des Systems auf den Impuls (also die Delta-Funktion) bezeichnet:

$$h = f(\delta) \quad \text{oder} \quad h[k] = f(1, 0, 0, \ldots)[k]$$

Das Geniale daran ist jetzt: Da man jede diskrete Folge aus Delta-Funktionen zusammensetzen kann, kann man shift-invariante lineare Systeme ganz einfach durch ihre Impulsantwort definieren.

$$\begin{aligned} f(x) &= f(x[0]\delta + x[1]\lambda_1\delta + x[2]\lambda_2\delta + \ldots) \\ &= x[0]f(\delta) + x[1]f(\lambda_1\delta) + x[2]f(\lambda_2\delta) + \ldots \\ &= x[0]h + x[1]\lambda_1 h + x[2]\lambda_2 h + \ldots \end{aligned}$$

Faltung (Convolution)

Wir formulieren jetzt die letzte Formel aus:

$$\begin{aligned} y = f(x) &= x[0]h + x[1]\lambda_1 h + x[2]\lambda_2 h + \dots \\ y[k] &= x[0]h[k] + x[1](\lambda_1 h)[k] + x[2](\lambda_2 h)[k] + \dots \\ &= x[0]h[k] + x[1]h[k-1] + x[2]h[k-2] + \dots + x[k]h[0] \\ &= \sum_m h[m]x[k-m] \end{aligned}$$

Was da steht, wird üblicherweise **Faltung** genannt. (Faltung gibt es übrigens auch bei kontinuierlichen Funktionen.) Geschrieben wird die Faltung meistens:

$$y = x * h$$

Der Einfachheit sind wir davon ausgegangen, dass es sowohl für x als auch für die Impulsantwort h keine negativen Zeitpunkte gibt. Im Allgemeinen muss man aber damit rechnen. Für h heißt das, dass das System schon eine Antwort gibt, bevor noch der Einheitsimpuls δ in das System geschickt wird (sofern man die freie Variable (k) als Zeit interpretiert).

FIR Filter (Finite Impulse Response)

Ist nun die Impulsantwort beschränkt, d.h. $h[k]$ ist ab einem bestimmten k 0, dann spricht man von einer **finite impulse response (FIR)**. Das hat den Vorteil, dass man das System mittels Faltung und endlichem Rechenaufwand implementieren kann. Der Vorgang heißt dann **Filterung**. h heißt dann auch **Filter** oder **filter kernel**.

Ein Programm könnte wie folgt aussehen:

```
for k = 0 ... 100000
    input x[k]
    Sum = 0
    for m = 0 ... 7
        Sum = Sum + x[k-m] * h[m]
    y[k] = Sum
    output y[k]
```

Die **Filterlänge** beträgt hier 8. D.h. $h[m] = 0$ für $m \geq 8$.

Eigenschaften von Filtern

- Reaktionszeit

  Füttert man ein System mit einer **Step**-Funktion (Funktion, die von Null plötzlich auf 1 wechselt und dort konstant bleibt), dann reagieren die meisten Systeme mehr oder weniger heftig, um sich dann nach einer Weile wieder zu beruhigen. Tiefpass-Filter pendeln sich irgendwann auf einen positiven Wert ein, Hochpassfilter wieder auf Null (nach einigen Schwankungen). Die Zeit, bis sich das System bis auf eine Toleranzgrenze an den neuen Wert angenähert hat, nennt man **Reaktionszeit**.

- Frequenzverhalten

  Bei linearen Systemen gilt, dass eine Sinusfunktion als Eingabe immer eine Sinusfunktion mit gleicher Frequenz als Ausgabe bewirkt. Der Ausgangs-Sinus hat aber möglicherweise eine andere Amplitude. Diese variiert meistens auch mit der Frequenz. Trägt man in einem Diagramm die Amplitude über der Frequenz auf, so sieht man den **Frequenzgang** des Systems (**power-spectrum**).

  Tiefpass-Filter drücken die Amplitude ab einer gewissen Frequenz (**Grenzfrequenz**) gegen Null. **Hochpass-Filter** drücken die Amplitude unterhalb dieser Frequenz gegen Null. **Bandpass-Filter** lassen nur einen gewissen Frequenzbereich (**pass-band**) durch, andere werden blockiert (**stop-band**).

Eigenschaften von Filtern

- Phasenverhalten

  Auch die Phase (Verschiebung des Ausgangs-Sinus gegenüber dem Eingangssinus in Grad) bleibt bei linearen Systemen nicht unbedingt gleich. Auch die Phase verändert sich mit der Frequenz ($\Rightarrow$ **Phasengang**).

  Lineare Systeme produzieren genau dann keine Phasenverschiebung, wenn der zugehörige Filter (Impulsantwort) symmetrisch um 0 ist. Ist der Filter zwar symmetrisch, aber nicht um 0, so ist der Phasengang linear. Deshalb werden diese Filter als **linear phase** klassifiziert.

- Länge des Filters

  Die Länge des Filters hat unmittelbaren Einfluss auf die notwendige Rechenzeit. *Gute* Filter (im Sinne von scharfer Abgrenzung zw. pass- und stop-band) sind üblicherweise lang.

Berechnung des Frequenzganges

Um den Frequenzgang eines Filters h zu berechnen, schicken wir eine Schwingung u mit bestimmter Frequenz und Amplitude 1 hinein und schauen, was herauskommt. Mathematisch eignet sich dazu am besten die komplexe Form $u[k] = e^{i\omega k} = \cos \omega k + i \sin \omega k$. ω ist die Frequenz (in Radianten). Der Einfachheit halber nehmen wir eine Sampling-Rate von 1 an (Einheit ist egal). Dann ergibt sich:

$$f(u) = u * h = \mathcal{F}^{-1}(\mathcal{F}u \odot \mathcal{F}h)$$

Hier ist $\mathcal{F}$ die Fourier-Transformation und $\odot$ die punktweise Multiplikation. Obige Gleichung ergibt sich aus der schönen Eigenschaft der Faltung, dass sie bei Fouriertransformation in die punktweise Multiplikation übergeht (Faltungssatz). Nun gilt aber:

$$\mathcal{F}u(\xi) = \begin{cases} 1 & \xi = \omega \\ 0 & \xi \neq \omega \end{cases}$$

Daher ist

$$\mathcal{F}u(\xi) \cdot \mathcal{F}h(\xi) = \begin{cases} \mathcal{F}h(\omega) & \xi = \omega \\ 0 & \xi \neq \omega \end{cases}$$

Alles, was also als Output aus dem System bleibt, ist die Frequenzkomponente $\mathcal{F}h(\omega)$.

Berechnung des Frequenzganges

Der Frequenzgang reduziert sich also auf:

$$\text{Frequenzgang} = \mathcal{F}h(\omega) = \sum_k h[k]e^{-i\omega k}$$

Dieser Ausdruck (als Funktion von ω) ist jetzt noch komplex. Uns interessiert aber nur der Betrag. Daher sollte man noch konjugiert komplex quadrieren.

Bei der angenommenen Sampling-Rate von 1 beträgt die Nyquist-Frequenz π (in Radianten: $\omega = 2\pi f$ und $f = \frac{1}{2}$). In diesem Bereich $(0 \ldots \pi)$ kann man den Frequenzgang des Filters betrachten.

IIR Filter (Infinite Impulse Response)

Das Problem bei FIR-Filtern ist (wer hätte das gedacht?) die endliche Impulsantwort. Zur Erinnerung (wenn l die Filterlänge ist):

$$y[k] = h[0]x[k] + h[1]x[k-1] + \ldots + h[l-1]x[k-l+1]$$

Irgendwann ist jemand auf die geniale Idee gekommen, auch die vorher berechneten Werte von y, also $y[k-1], y[k-2], \ldots$, in diese Gleichung mit einzubeziehen. Das sieht dann so aus:

$$\begin{aligned} y[k] = a[0]x[k] \quad &+ \quad a[1]x[k-1] + \ldots + a[l-1]x[k-l+1] \\ &+ \quad b[1]y[k-1] + \ldots + b[l-1]y[k-l+1] \end{aligned}$$

Dadurch kann die Impulsantwort des Systems unendlich werden (bei endlich vielen Koeffizienten). Beweis: Man setze z.B. $a[0] = b[1] = 1$ und alle anderen Koeffizienten 0. Als Antwort auf die Delta-Funktion erhält man dann $y[0] = x[0] = 1, y[1] = y[0] + x[1] = 1 + 0 = 1, y[2] = 1, y[3] = 1, \ldots$.

Da bei solchen Filtern die Impulsantwort unendlich ist, nennt man sie **infinite impulse response filter (IIR)**. Da die Ausgabewerte wieder als Eingabewerte verwendet werden, nennt man sie aber oft auch **rekursive Filter**.

DFT – Diskrete Fourier Transformation

Wenn $x_0 \dots x_{N-1}$ eine Folge von reellen Werten ist, dann sind mit

$$\hat{x}[m] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}mn} \quad m \in 0 \dots N-1$$

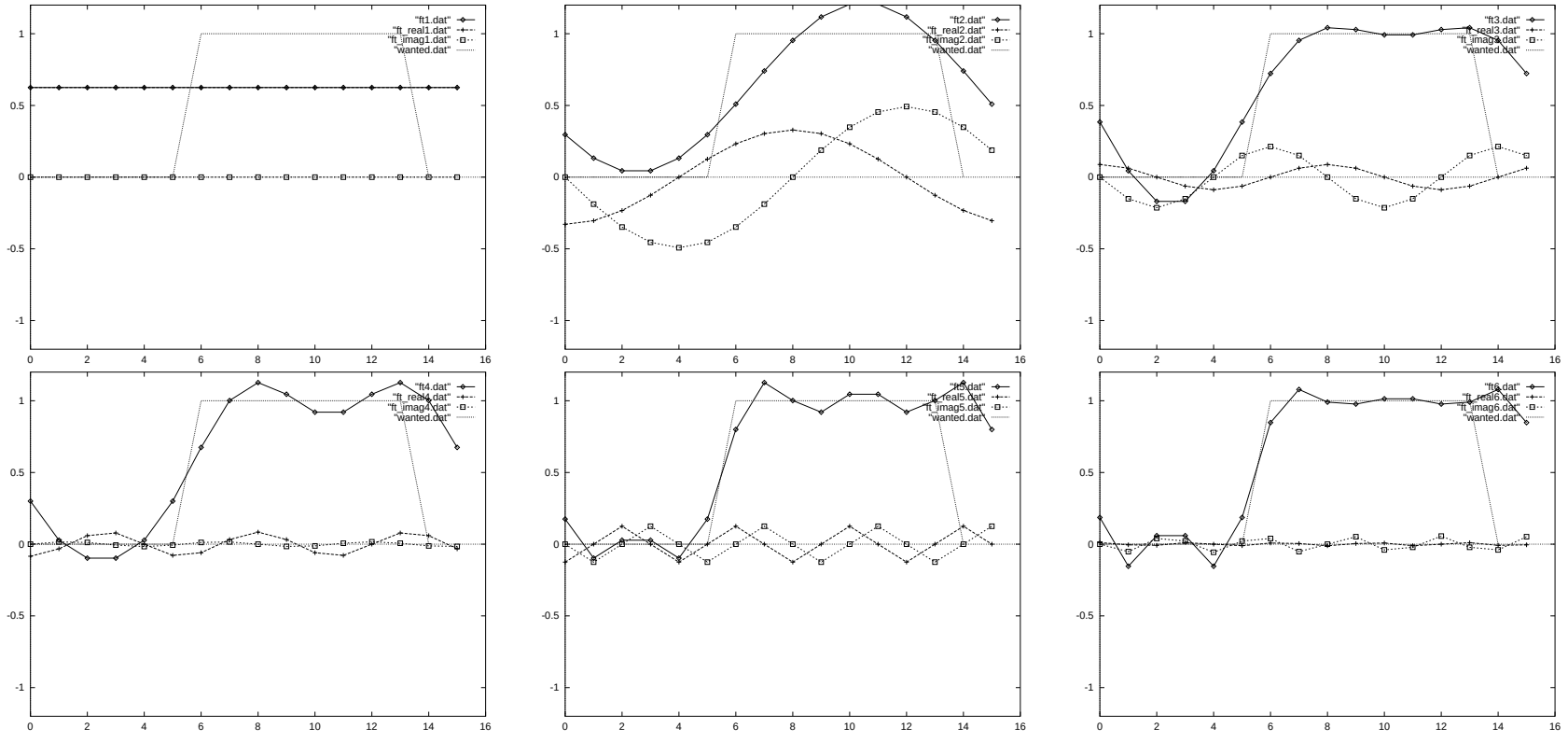
$\hat{x}[0] \dots \hat{x}[N-1]$ die transformierten Daten. Hier ist zu beachten, dass die $\hat{x}[m]$ im Allgemeinen komplex sind. Der reelle und der imaginäre Anteil gehört jeweils zur Cosinus- bzw. Sinus-Funktion.

Komisch: Bei einer Samplingrate von 1 wäre die Nyquist-Frequenz π. Die Koeffizienten von $\hat{x}$ stehen nach der Reihe für die Frequenzen $0, \frac{2\pi}{N}, 2\frac{2\pi}{N}, 3\frac{2\pi}{N}, \dots$. Bis zur Nyquistfrequenz braucht man daher nur $\frac{N}{2}$ Koeffizienten. Wofür stehen dann die Koeffizienten $\hat{x}[\frac{N}{2}] \dots \hat{x}[N-1]$? *Antwort:* für die negativen Frequenzen: $\hat{x}[-m] = \hat{x}[N-m]$.

Beachte: Bei reellen Signalen (x nicht komplex) sind die negativen Frequenzen redundant: $\hat{x}[-m] = \overline{\hat{x}[m]}$ (konjugiert komplex). Daher ist $\hat{x}$ (fast) symmetrisch.

Die Komplexität der Fourier-Transformation ist N^2. Aber es existiert ein schnellerer Algorithmus um diese Koeffizienten zu berechnen: die Fast-Fourier-Transformation (**FFT**). Diese hat eine Komplexität von $N \log(N)$.

DFT - Veranschaulichung



Die ersten sechs Koeffizienten der FT und ihr Einfluss auf die rekonstruierte Funktion. Man beachte, dass immer zwei Basisfunktionen dazuaddiert werden ($\cos$ und $\sin$). Die Koeffizienten (zu sehen als Amplitude der Basisfunktionen) werden immer kleiner.

FFT

Wir definieren:

$$w_N^\alpha := e^{-i\frac{2\pi}{N}\alpha}$$

Dann kann man die DFT schreiben als:

$$\hat{x}[m] = \sum_{n=0}^{N-1} x[n] w_N^{mn}$$

Jetzt gelten folgende Beziehungen:

$$\begin{aligned} w_N^{\alpha+\beta} &= w_N^\alpha w_N^\beta \\ w_N^{2\alpha} &= w_{\frac{N}{2}}^\alpha \\ w_N^N &= 1 \\ w_N^{\frac{N}{2}} &= -1 \end{aligned}$$

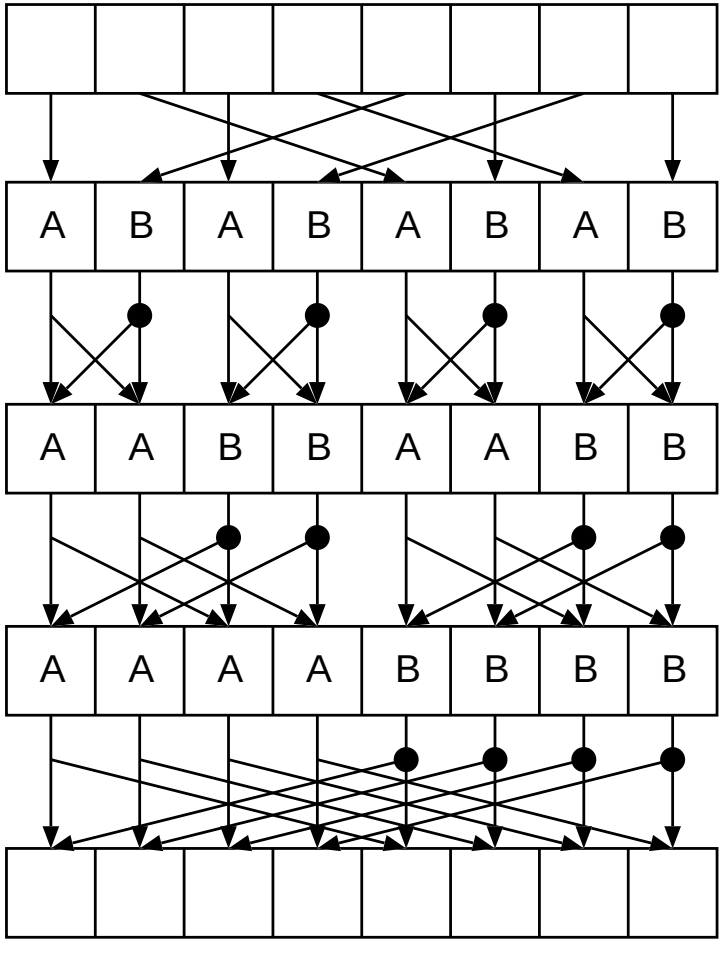
FFT

Jetzt trennen wir x in gerade $x[2n]$ und ungerade $x[2n+1]$ und $\hat{x}$ in zwei Hälften: $\hat{x}[0 \ldots \frac{N}{2}-1]$ und $\hat{x}[\frac{N}{2} \ldots N-1]$. Dann ergibt sich:

$$\begin{aligned}
\hat{x}[m] &= \sum_{n=0}^{\frac{N}{2}-1} x[2n] w_N^{m2n} + \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] w_N^{m(2n+1)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x[2n] w_{\frac{N}{2}}^{mn} + w_N^m \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] w_{\frac{N}{2}}^{mn} \\
&= A_m + w_N^m B_m \\
\hat{x}[m+\frac{N}{2}] &= \sum_{n=0}^{\frac{N}{2}-1} x[2n] w_N^{(m+\frac{N}{2})2n} + \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] w_N^{(m+\frac{N}{2})(2n+1)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x[2n] w_{\frac{N}{2}}^{mn} - w_N^m \sum_{n=0}^{\frac{N}{2}-1} x[2n+1] w_{\frac{N}{2}}^{mn} \\
&= A_m - w_N^m B_m
\end{aligned}$$

FFT

Man sieht, dass sowohl A_m als auch B_m wieder DFTs von halber Länge sind. Wenn man diese nach dem selben Schema auflöst, dann ergibt sich folgendes Schema:



- Der erste Schritt ist eine Umordnung der Input-Werte x. Dieser Schritt heißt auch **bit reversal sorting**. Denn es wird z.B. der Wert an Stelle 6 (110 binär) auf die Stelle 3 (011 binär) verschoben.
- Die darauf folgenden Schritte stellen die rekursive Anwendung der Formeln dar.
- Der dicke Punkt muss dabei so interpretiert werden: *Multipliziere den Wert mit* w_N^m, *wobei* $N = 2, 4, 8$ *und* $0 \leq m < \frac{N}{2}$, *und gib das Ergebnis positiv nach links und negativ nach unten weiter.*
- Wo sich zwei Pfeile treffen, werden die Werte addiert.
- Wie man sieht, ergibt das $\log_2 N$ Stufen (z.B. 3 Stufen bei $N = 8$ in diesem Fall) mit einer Komplexität von N. Daraus folgt: Die FFT hat Komplexität $N \log N$.

Faltung durch FFT

Bei sehr großen Filtern ist der rechnerische Aufwand sehr groß. Man kann die Faltung auch einsetzen, um Muster zu finden: Z.B. kann man ein Suchbild mit einem beliebigen Bild falten (zweidimensional). Die Position des größten Samples (nach der Faltung) gibt dann die Position der größten Übereinstimmung mit dem Originalbild an.

Setzen wir der Einfachheit halber die Filtergröße gleich der Datengröße N. Dann ist die Komplexität der Faltung N^2.

Wir wissen aber, dass die Faltung im Zeitbereich einer punktweisen Multiplikation im Frequenzbereich entspricht. D.h. dass man die Faltung durch

$$\mathrm{IFFT}(\mathrm{FFT}(x) \odot \mathrm{FFT}(h))$$

implementieren. Die Komplexität dieser Operation beträgt $N \log N + N + N \log N$ für FFT, $\odot$ (punktweise Multiplikation) und IFFT respektive. Das ergibt insgesamt eine Komplexität von $N \log N$. Das ist erheblich schneller als die direkte Methode (zumindest für große Filter).

DCT - Diskrete Cosinus-Transformation

ist gegeben durch

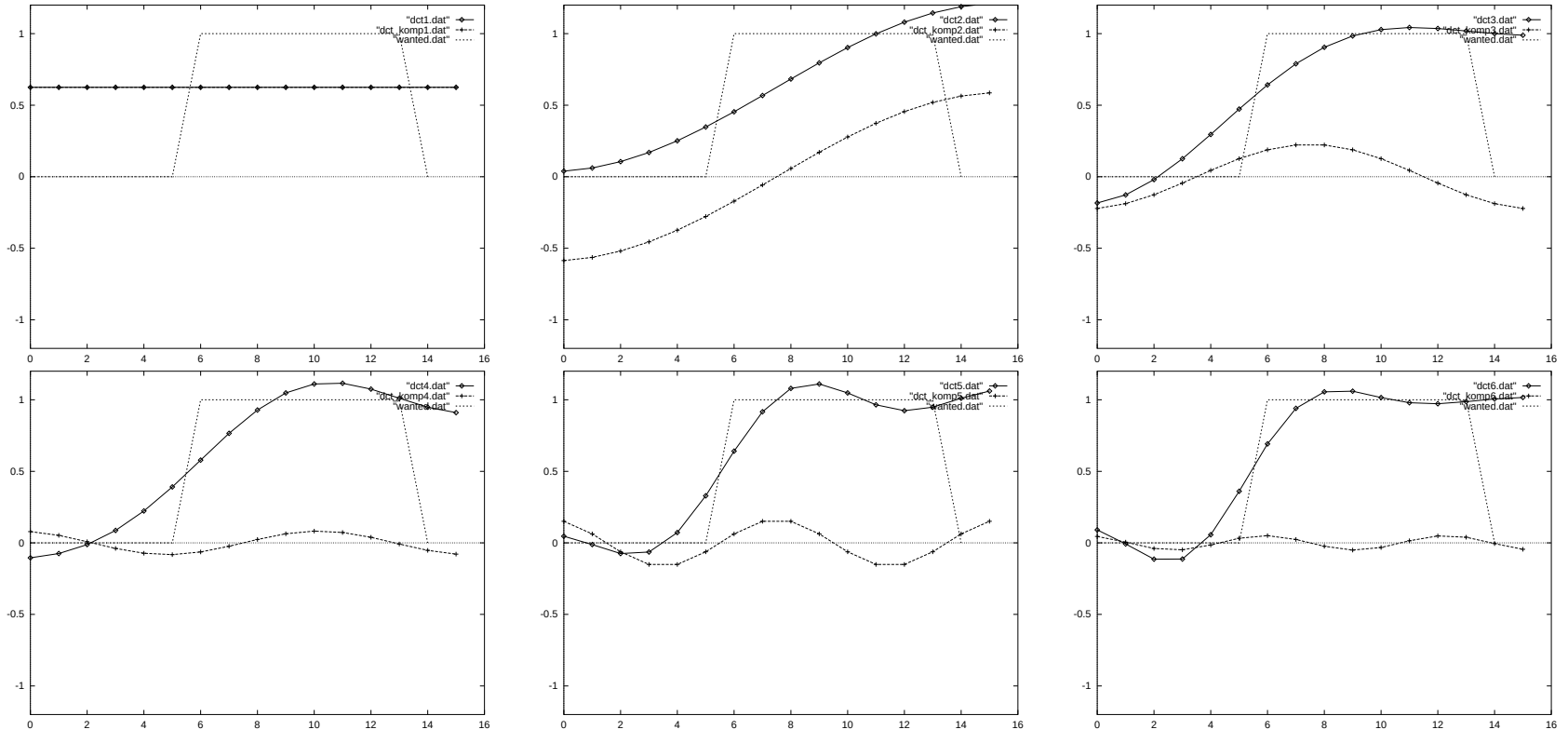
$$\hat{x}[m] = c[m] \sum_{n=0}^{N-1} x[n] \cos \frac{\pi(2n+1)m}{2N}$$

wobei

$$c[0] = \sqrt{\frac{1}{N}}, \quad c[m] = \sqrt{\frac{2}{N}} \quad m \geq 1 \, .$$

Hier sind die Koeffizienten alle reell. Die Komplexität ist die gleiche wie bei der Fourier-Transformation. Auch hier gibt es eine Fast-Version mit der Komplexität $N \log(N)$.

DCT Veranschaulichung



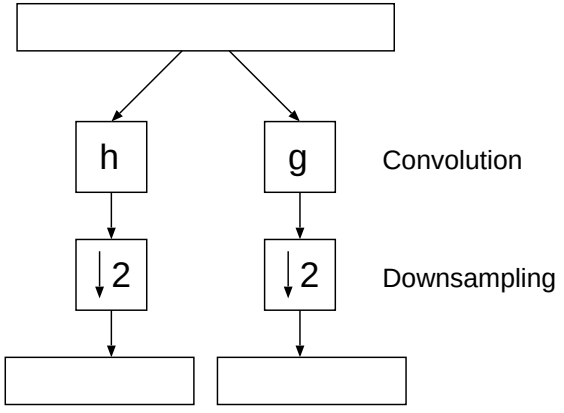
Die ersten sechs Koeffizienten der DCT und ihr Einfluss auf die rekonstruierte Funktion.

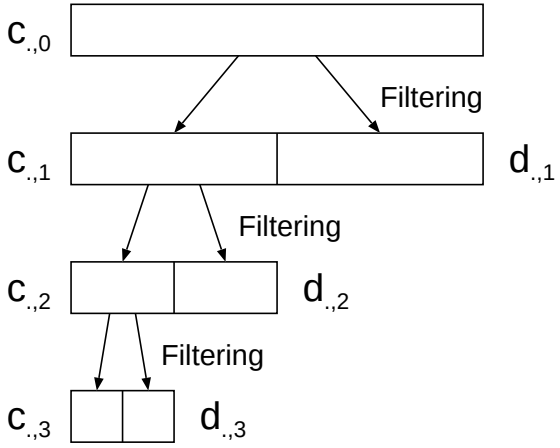
Wavelet-Transformation (1D)

Die diskrete Wavelettransformation (**DWT**) basiert auf einer schrittweisen Filterung des Signals x.

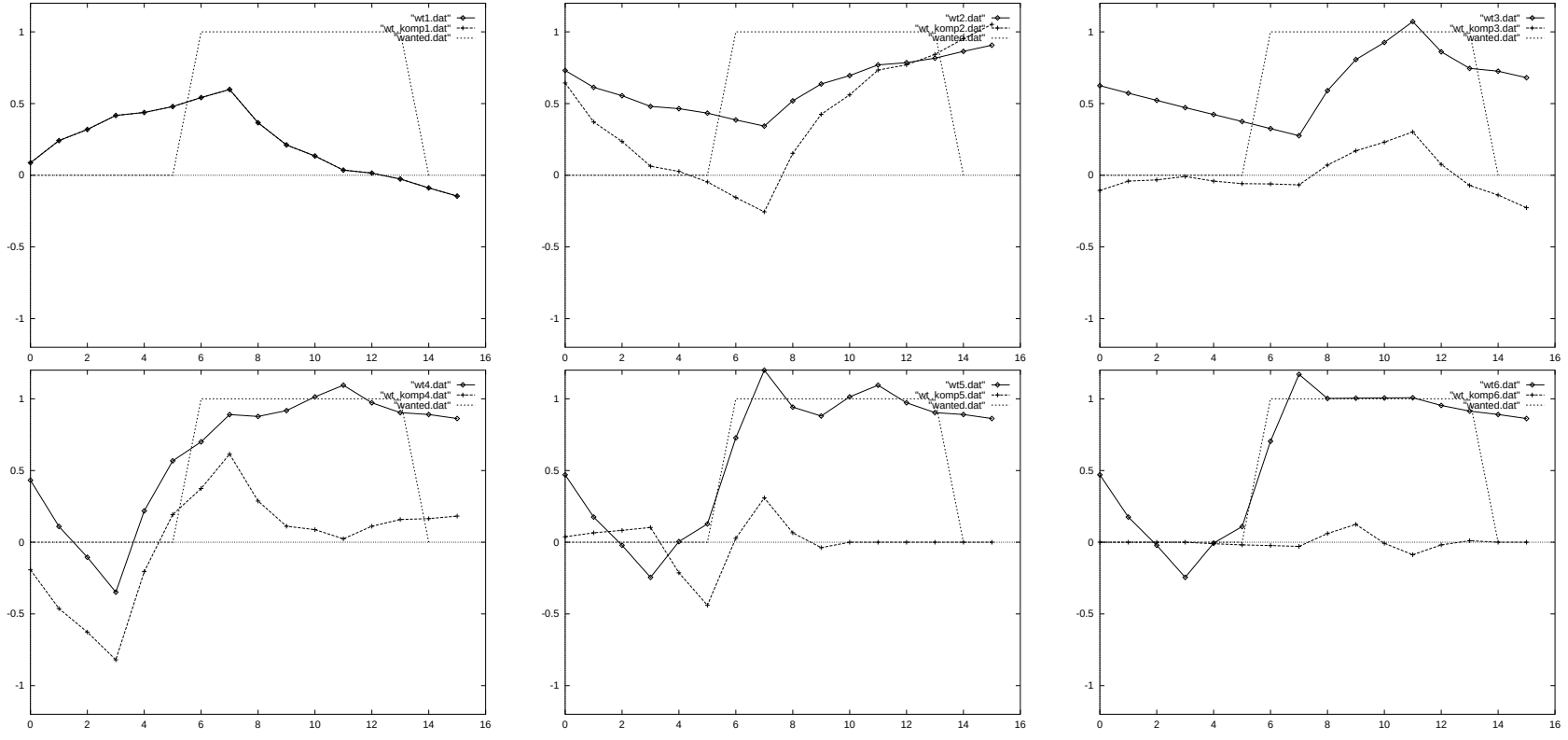
In der Wavelettransformation finden immer zwei Filter (Filterpaar) Anwendung. Ein Hochpassfilter g und ein Tiefpassfilter h. Die Frequenzgänge dieser Filter sollen sich ausschließen, sodass die Daten nach der Filterung in einen tief- und einen hoch-frequenten Teil getrennt ist.

Beim Ergebnis lässt man jeden zweiten Wert aus: **Downsampling**. Dadurch bleibt die Datenmenge gleich. Die gefilterten Datenreihen nennt man **Subbands**. Danach fährt man mit dem tiefpassgefilterten Subband (**approximation subband**, i.Gs. zu **detail subband**) rekursiv fort.





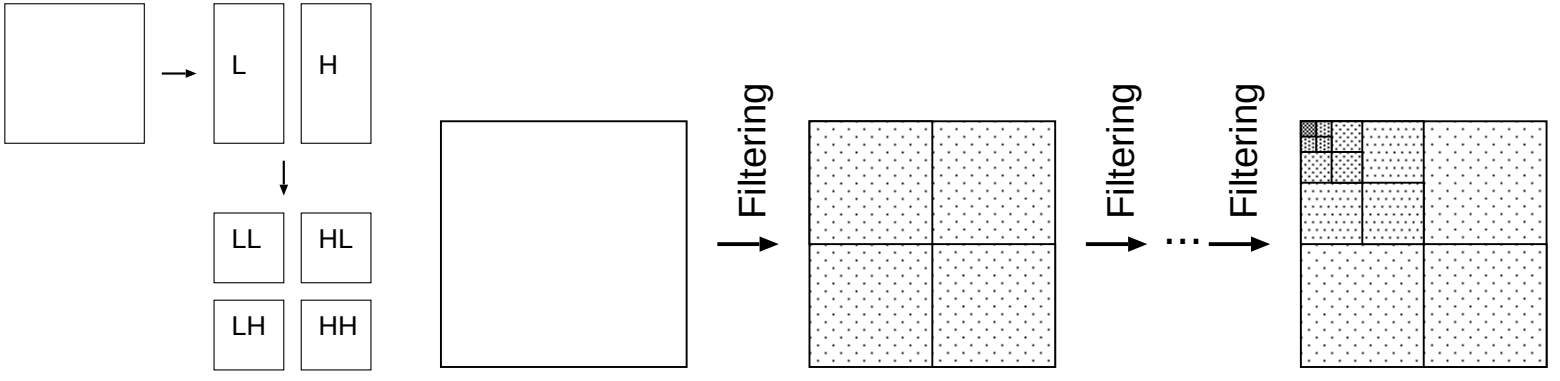
WT Veranschaulichung



Die ersten sechs Koeffizienten der DWT und ihr Einfluss auf die rekonstruierte Funktion. Man beachte, dass vor allem die letzteren (höherfrequenten) Basisfunktionen (*Wavelets*) sich nicht über den ganzen Bereich erstrecken, sondern auch horizontal kleiner werden.

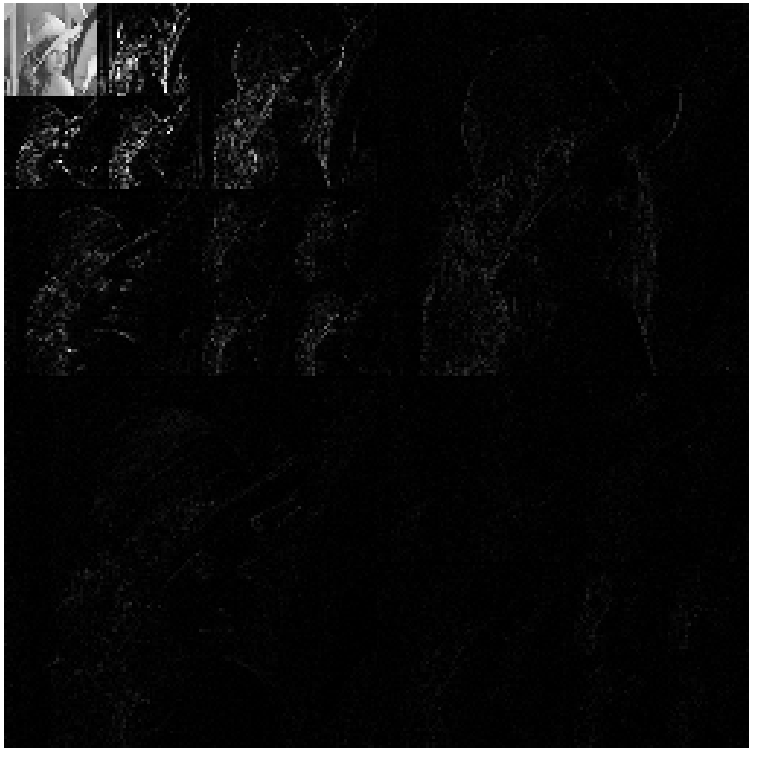
Wavelettransformation (2D)

Bei der zweidimensionalen DWT führt man eine horizontale und eine vertikale Filterung (auf Zeilen und Spalten) hintereinander aus. Dadurch ergeben sich 4 neue Subbands. Davon wird nur eines (LL, das in beide Richtungen tiefpassgefiltert worden ist) weiterzerlegt.



Diese Transformation hat **lineare** Komplexität (N). Die "Energie" des Bildes ist dann in den tieferfrequenten Subbands (links oben) konzentriert.

DWT (2D) Veranschaulichung



Entropiekodierung

Eine wichtige Anwendung für DSPs ist das Komprimieren von Daten, da das bei Multimediaanwendungen aufgrund der Fülle der Daten notwendig ist. Der Kern jeder Datenkomprimierung ist die Entropiekodierung. Die zu kodierende Information wird unter Verwendung einer Menge von Symbolen dargestellt. Jedes Symbol hat dabei eine gewisse Wahrscheinlichkeit. Ziel ist es nun, Symbole mit hoher Wahrscheinlichkeit mit kürzeren Codes zu versehen. Damit erreicht man die Entropie (Shannon) ziemlich gut. Entropie $\approx$ minimale Datenlänge (Informationsgehalt) für bestimmte Anzahl von Symbolen.

Die zwei Hauptvertreter dieser Methode sind:

Huffman-Kodierung Jedem Symbol wird eine eindeutige Bitfolge zugeordnet. Häufigere Symbole bekommen eine kürzere Folge.

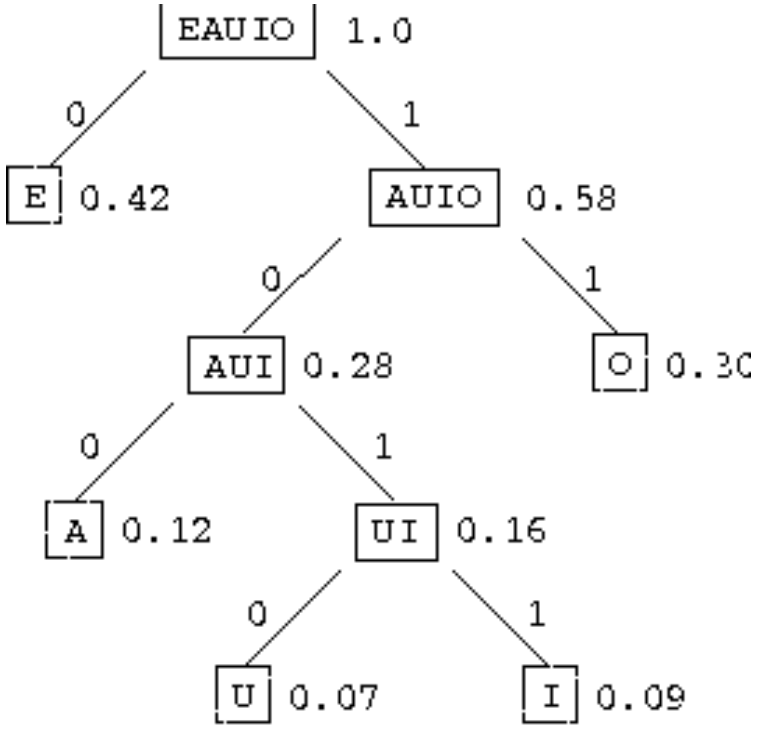
Arithmetisches Kodieren Hier kann man einem Symbol eine nicht-ganzzahlige Anzahl von Bits zuordnen (Wie das funktioniert, ist hier wurscht). Dadurch ist man nicht gezwungen, zwei Symbole einfach als 1 und 0 zu kodieren.

Huffman-Kodierung

Jedem Symbol wird ein Code (Folge aus bits) zugeordnet, so dass kein Code ein Präfix eines anderen ist. Dies wird erreicht durch einen binären Baum mit den Symbolen an den Blättern. Verzweigungen nach links mit 0, nach rechts mit 1 kodieren. Unwahrscheinliche Symbole werden weiter vom Ursprung entfernt angeordnet.

Die (bitweise) in einen **Bitstream** hintereinandergeschriebenen Codes können dann eindeutig aus diesem wieder rekonstruiert werden.

Nachteil: Bei z.B. nur zwei Symbolen muss man von einer 50:50 Wahrscheinlichkeitsverteilung ausgehen.



Herkömmliche Audio-Kompression

Normale Kompressionsverfahren, wie z.B. Huffman, LZW, . . . basieren auf wiederkehrenden Datenteilen mit bestimmter Wahrscheinlichkeitsverteilung. Audio-Daten entsprechen dieser Voraussetzung nicht. Deshalb muss man für Audio andere Verfahren entwickeln. Da sich diese Aufgabe als schwierig herausstellt, sind die meisten Verfahren **lossy**. D.h. die Qualität des Audio-Signals leidet unter der Kompression.

Herkömmliche (tlw. veraltete) Verfahren sind:

Silence Compression Stille (oder einfach leise) Sequenzen werden entdeckt und einfach unter Angabe der Länge mit ein paar Bytes kodiert.

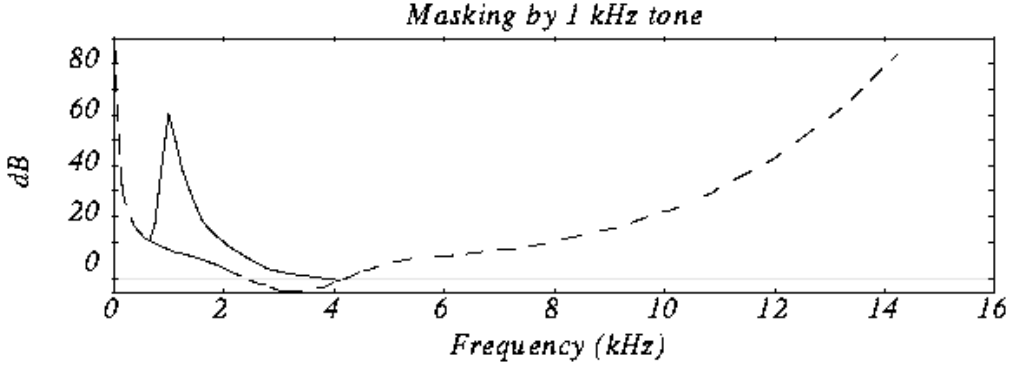
Adaptive Differential Pulse Code Modulation (ADPCM) Kodiert nicht die Samples selbst, sondern die Differenz zwischen dem Sample und einer Voraussage (waveform prediction) aufgrund vorangegangener Samples. Adaptiert die Quantisierung/Kodierung, so dass für kleinere Differenzen weniger Bits verwendet werden.

Linear Predictive Coding (LPC) Nach einem Modell des Sprachorgans wird aus vergangenen Samples das jeweils nächste vorausgesagt. Kodiert werden die Parameter des Modells. Eignet sich sehr gut vor hohe Kompressionsraten, aber nur für Sprache. Klingt wie computer-generierte Sprache.

Code Excited Linear Predictor (CELP) Wie LPC, kodiert aber auch den Voraussagefehler (prediction error).

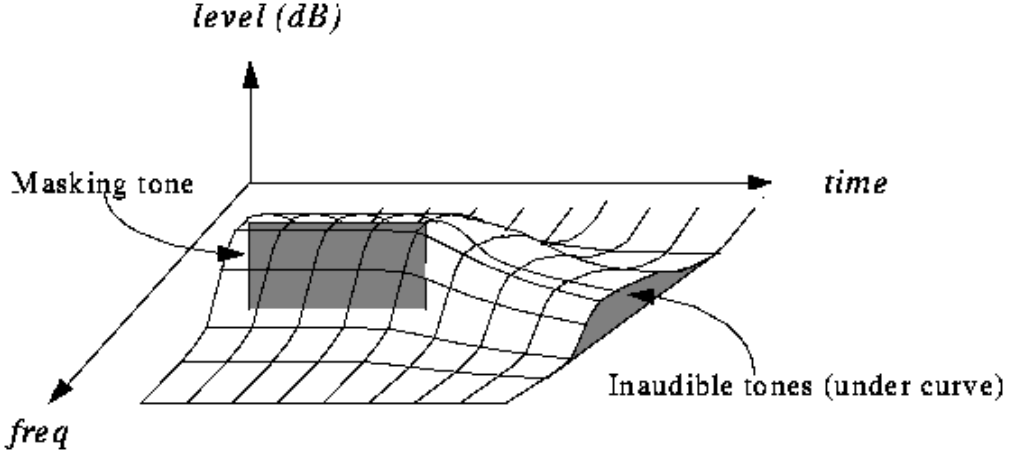
Psychoakustik

Gewisse Töne sind für den Menschen nicht hörbar. Das hat mehrere Gründe: Der Ton ist (nona) zu leise, der Ton ist zu hoch/tief, der Ton wird von einem ähnlichen Ton überdeckt (Masking). Folgendes Diagramm stellt die Schwelle dar, ab der der Mensch einen Ton hört:



Die frequenzabhängige Hörschwelle wird durch das **Frequenz-Masking** auch in der Frequenz-Umgebung des maskierenden Tons erhöht.

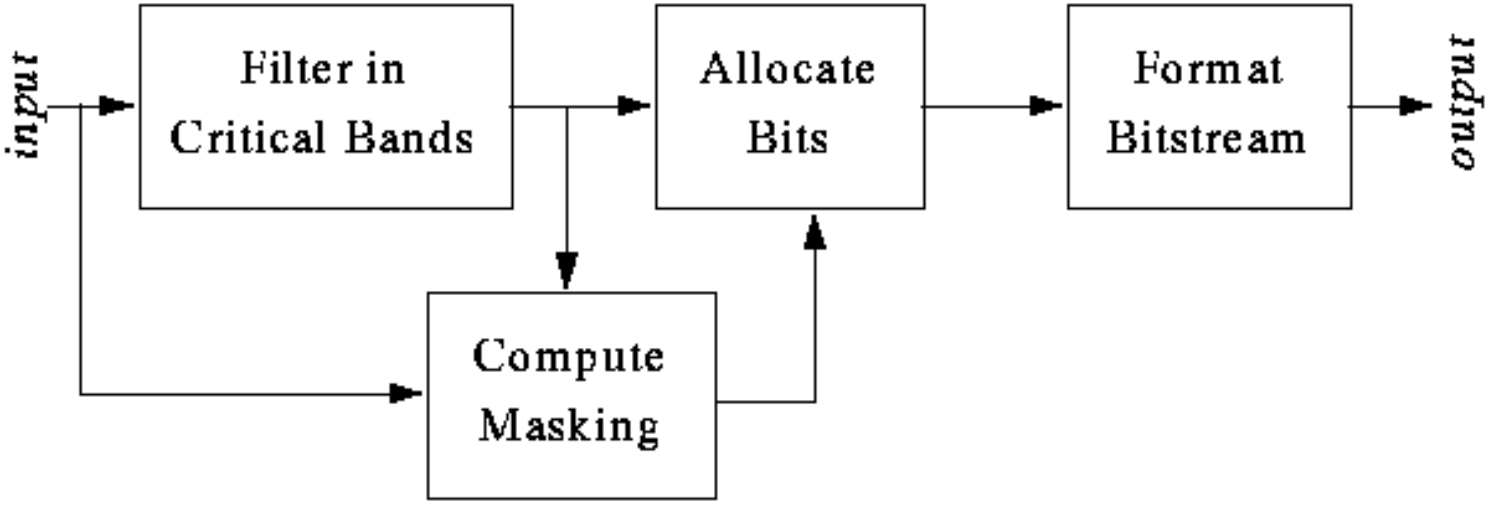
temporal masking: Unhörbare Töne sind auch eine Weile nach Abschalten des überdeckenden Tons unhörbar.



MPEG – Audio Layers

Im MPEG-Videokompressions-Standard sind drei Layers von Audio-Kodierung vorgesehen. Die Layers basieren alle auf dem selben Prinzip, sind aber in Bezug auf Kompressionsleistung/Qualität unterschiedlich. Der beste Layer 3 (auch der bekannteste . . .) ist aber auch der rechenintensivste und deshalb nicht für jedes Gerät geeignet.

Funktionieren tut das so:



MPEG – Audio Layers

Das Signal wird mit einer modifizierten DCT-Transformation blockweise (Frames) in mehrere Subbands transformiert. Durch **Masking** zwischen den Subbands werden gewisse Koeffizienten ausgeschaltet und gewisse Koeffizienten genauer/ungenauer quantisiert (Quantisierungs-Noise sollte dabei unter der Masking-Schwelle bleiben). Anschließend wird das ganze in einen Bitstream kodiert.

Der Hauptunterschied zwischen den Layers ist:

Layer 2 verwendet mehrere Frames für Masking ($\Rightarrow$ leichtes temporal masking)

Layer 3 verwendet verschieden breite Subbands (aufwendiger) und volles temporal masking. Zusätzlich wird Stereo-Redundanz und Huffman-Kodierung genutzt.

Ein weiterer Unterschied zwischen den Layers ist der Coding-Delay (Zeit, die verlorengeht, bis ein Ton beim Empfänger ankommt). Die theoretischen Delays liegen bei 19 ms, 35 ms und 59 ms für Layer 1, 2, 3 respektive. Die echten Delays sind meistens 3 mal so hoch.

Image- und Video-Compression

Bild- und Videodaten werden meist verlustbehaftet komprimiert. D.h. nach der Wiederherstellung ist das Bild leicht verändert. “Unwichtige” Teile werden einfach unterschlagen. Was unwichtig ist kann über das “human vision system” (**HVS**) modelliert werden.

Um in einem Bild wichtige und unwichtige Information voneinander zu trennen, wird meistens eine frequenz-basierte Transformation angewandt (**transform coding**). Dadurch konzentriert sich die Bildinformation auf tieffrequente Koeffizienten.

Bildkompression besteht daher aus folgenden Schritten:

- Transformation (DCT oder WT)
- Kodieren der Information, welche Koeffizienten vernachlässigt werden
- Kodieren der nicht vernachlässigten Koeffizienten (meist gerundet)
  Dabei wird meist zusätzlich eine verlustfreie Kompressionsmethode angewandt.

Bild- und Videoformate

JPEG Ein Bildformat, das wohl jedem bekannt ist. Das Bild (jede Farbkomponente) wird dabei in 8×8-Blöcke unterteilt. Diese werden DCT-transformiert (2D). Quantisierung mittels einer 8×8-Quantisierungsmatrix (ein Quantisierungsfaktor für jeden Koeffizienten). Die quantisierten Koeffizienten jedes Blockes werden lauflängen- und Huffman-kodiert.

JPEG2000 Neue Version von JPEG. Das Bild wird wavelet-transformiert (2D). Die Subbands werden in Blöcke unterteilt. Die Koeffizienten werden Bit für Bit (beginnend mit dem high-order-bit) mittels arithmetischem Coder kodiert. Wobei die *Bit-Planes* der Blöcke so ausgesucht werden, dass die Bildqualität maximiert wird.

SPIHT Akademisch. Ebenfalls Wavelet-basiert. Einfach – schnell – gut.

MPEG Das bekannte Videoformat. Fast alle anderen Videoformate basieren darauf. Die Einzelbilder werden per *block-matching* vorausgesagt. D.h. es werden *Motion-Vektoren* für Blöcke kodiert und die Differenz zwischen dem (durch Block-Verschiebung) angenäherten Block und dem wirklichen Block wird wie in JPEG kodiert.

xDSL

xDSL (also ADSL, HDSL, VDSL, . . .) ist eine Technik zur bidirektionalen Übertragung von Daten auf herkömmlichen Kupferleitungen. Der Unterschied zu herkömmlichen Verfahren ist, dass die Modulation der Signale (Information in Frequenzbänder packen) digital erfolgt, wodurch einige Techniken erst möglich werden. Dazu sind schnelle Prozessoren notwendig: DSPs.

- Auftrennen der Daten in mehrere (z.B. 200) Teilströme.
- Die Teilströme werden mit einem Fehlerkorrektur-Code versehen (FEC – forward error correction, Reed-Solomon-Codes)
- Eine gewisse Anzahl von Bits werden zu Symbolen zusammengefasst. Ein Symbol ist eine komplexe Zahl. Die Information wird in Amplitude und Phase der Zahl gepackt (PAM, QAM).
- Die Symbole (von jedem Teilstrom eines) werden zu einer Folge zusammengesetzt und IFFT-transformiert (inverse Fast-Fourier-T.). Auf diese Weise bekommt jeder Teilstrom ein Frequenzband zugewiesen.
- Tiefe Frequenzbänder werden oft nicht benutzt, um nicht mit POTS (plain old telephone service) in Konflikt zu kommen.
- Die sich ergebende Wellenform wird (mit ein paar Modifikationen) auf eine Leitung geschickt. Diese Leitung wird aber auch für die Übertragung in die Gegenrichtung benutzt. Um nicht die eigenen Signale zu empfangen, werden diese digital ausgefiltert (EC – echo cancellation). So können die Frequenzbänder gleichzeitig in beide Richtungen verwendet werden.

Hardware

Der Aufbau von Digitalen Signalprozessoren orientiert sich stark an den häufigsten Anwendungen. Im Gegensatz zu "normalen" Prozessoren sind diese Anwendungen nicht so breit gestreut, bzw. basieren auf ähnlichen Algorithmen und Abläufen.

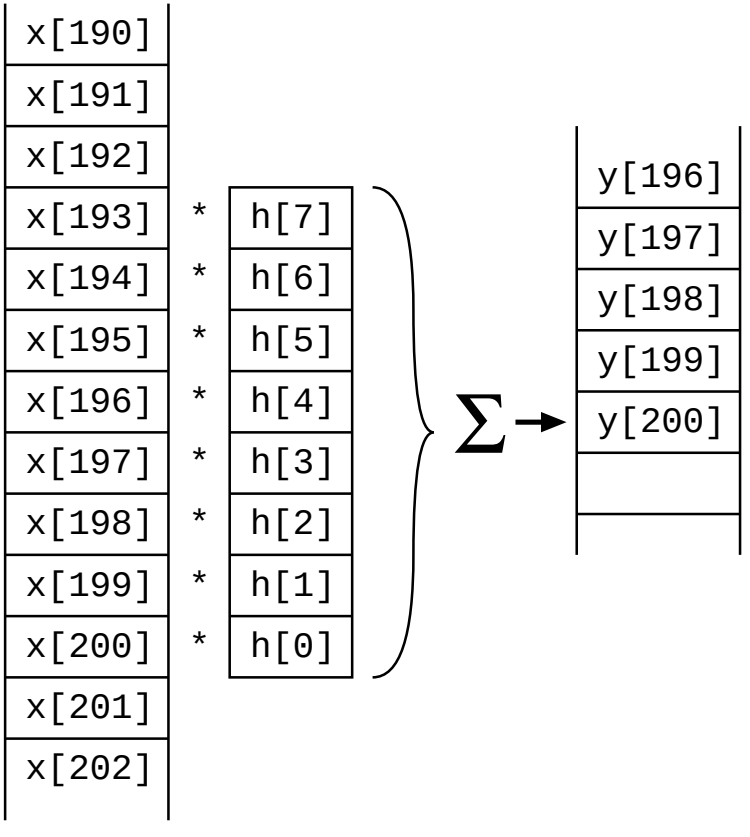
	"Normale" Prozessoren	DSPs
Typische Anwendungen	Textverarbeitung Datenbanken Betriebssysteme	Filterung Bild-/Videokompression
Typische Operationen	Daten verschieben/kopieren Testen / Verzweigen	Addition Multiplikation

Daraus erkennt man, dass für die Performance von DSPs hauptsächlich von der Geschwindigkeit der arithmetischen Operationen abhängt. Diese werden derart hochgezüchtet, dass sie meist nur einen Taktzyklus zur Ausführung benötigen. (Gewisse DSPs verzichten daher sogar auf eigene Integer-Operationen.)

Filtern – genau betrachtet

Die wichtigste Anwendung für DSPs ist wohl das **Filtern**. Hier zur Erinnerung noch einmal die Formel:

$$y[t] = \sum_{i=0}^{n} x[t-i]h[i]$$




```
for k = 0 ... 100000
    Sum = 0
    for m = 0 ... 7
        Sum = Sum + x[k-m] * h[m]
    y[k] = Sum
```

Filtern – Schleifenoptimierung

Ziel ist es, die innere Schleife `Sum = Sum + x[k-m] * h[m]` in einem Taktzyklus pro Durchlauf ausführen zu lassen. In dieser einen Zeile passieren aber folgende Sachen:

- Mindestens ein Maschinenbefehl muss gelesen werden
- Ein Sample und ein Koeffizient muss gelesen werden
- Eine Addition *und* eine Multiplikation wird durchgeführt

Zusätzlich muss ein Buffer für die `x[*]` verwaltet werden. Es müssen ja nur so viele Samples gespeichert werden, wie der Filter Koeffizienten hat. Eine Möglichkeit ist, in der Schleife die `x[*]` um einen Wert zu verschieben. Das bedeutet aber noch einen Speicherzugriff.

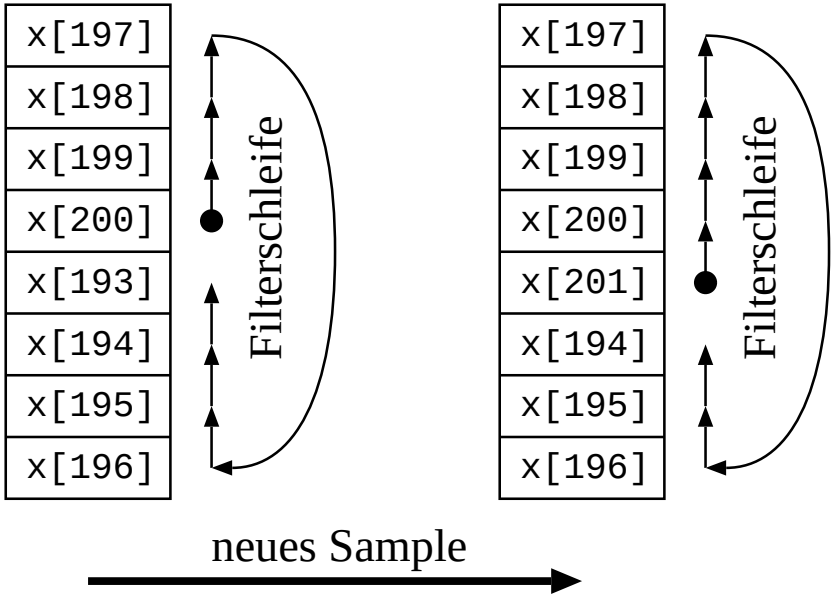
Insgesamt müssen also 3 (bzw. 4) Speicherzugriffe, eine Multiplikation und eine Addition innerhalb eines Taktzyklus erledigt werden.

Außerdem bleibt für die Schleifensteuerung (Überprüfung der Endebedingung, Sprung zu Beginn) kein Taktzyklus mehr übrig.

Im folgenden sehen wir, wie das gehen kann.

Circular Buffers

Um sich den Speicherzugriff zum Verschieben der `x[*]` zu sparen, kann man sog. **circular buffers** verwenden.

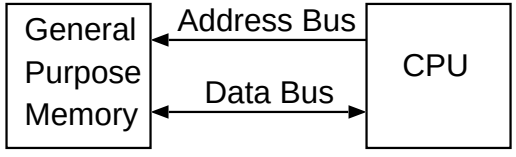


Die "Filterschleife" unseres Beispielprogramms wird dann aber komplizierter (und daher langsamer):

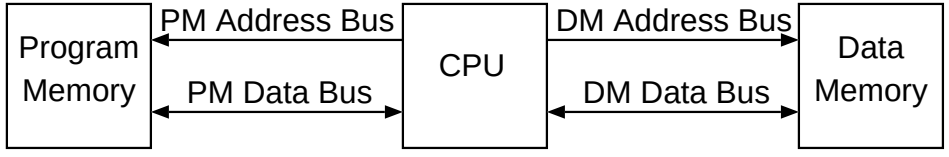
```
...
Sum = 0
for m = 0 ... 7
   Sum = Sum + Buffer[Pointer] * h[m]
   if Pointer > 0 then
      Pointer = Pointer - 1
   else
      Pointer = Bufferlength - 1
...
```

Diese `ifs` und Vergleiche und so weiter gehen sich nun aber in einem Taktzyklus wirklich nicht mehr aus. Daher bieten DSPs dafür Hardwareunterstützung.

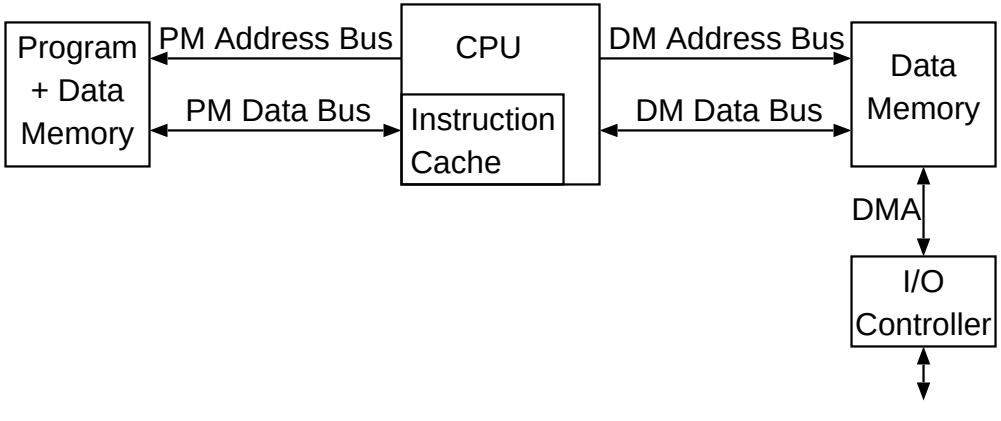
Memory: Neumann versus Harvard



Die meisten heutigen General-Purpose-Prozessoren (GPP) sind nach dem Prinzip des amerik. Mathematikers John Von Neumann (1903-1957) aufgebaut. ($\Rightarrow$ **Von Neumann Architektur**) Dabei befinden sich Programm und Daten in ein und dem selben Speicher.



Die Alternative dazu heißt **Harvard Architektur**. Hier befinden sich Programm und Daten in verschiedenen Speicher-Blöcken. Nachteil: Doppelte Busstruktur $\Rightarrow$ komplexer. Vorteil: Ein Maschinenbefehl und ein Datenwort kann gleichzeitig gelesen werden.



Die **Super Harvard Architektur** wurde von dem DSP-Hersteller *Analog Devices* entwickelt und inzwischen von den meisten anderen DSP-Herstellern übernommen. Die Programmspeicherzugriffe wurden minimiert durch einen **Instruction Cache**. Dadurch bleibt Bandbreite übrig, die für spezielle Daten genützt werden kann (z.B. den Filter `h[*]`).

Funktionseinheiten

Um die verschiedenen Typen von DSPs zu verstehen, muss man die verschiedenen Module (Funktionseinheiten, functional units) kennen, aus denen ein DSP aufgebaut ist. Es sind dies:

Control Unit Veranlasst das Laden der Instruktionen, dekodiert diese und steuert die Ausführung der Operationen durch die anderen Einheiten.

ALU Arithmetic Logic Unit.
Ist für Integer-Addition, -Subtraktion, logische Operationen, Datenformat-Konvertierung, etc. zuständig.

Multiplier, Adder, etc. Einzelne arithmetische Operationen werden oft als eigene Bausteine betrachtet/entwickelt.

MAC * Multiply-Accumulator (nur bei DSPs)
Multiplikation mit anschließender Aufsummierung.

Shifter Shift, Rotate, Bit-Selection, etc.

Address Generator Eigene Einheit, die darauf spezialisiert ist, die Adressen für den Speicherzugriff oder Programmsprünge zu berechnen.

Memory * (on-chip)

DMA * (Direct Memory Access für I/O)

MAC – Multiply-Accumulator

Bei DSP-Anwendungen tritt sehr häufig die Kombination "Multiplikation mit anschließender Aufsummierung" auf. Bestes Beispiel ist wieder einmal das Filtern. Zur Erinnerung:

```
Sum = Sum + x[?] * h[?]
```

Um diese Operation zu unterstützen, haben viele DSPs einen eigenen Baustein integriert, der diese Operation durchführt – meist in nur einem Taktzyklus.

Der Akkumulator, also das Register, das die laufende Summe enthält, bewegt sich üblicherweise in einer Größenordung vom durchschnittlichen Summierungswert (`x*h`) multipliziert mit der Anzahl der Summanden. Deshalb (vor allem bei Fixkommazahlen) sollte der Akkumulator größer dimensioniert sein (z.B. 80 bit bei SHARC ADSP-21160).

Auch bei Fließkommazahlen ist das notwendig, da die Addition einer kleinen Zahl (Summand) mit einer großen (Akkumulator) immer einen Fehler verursacht, der sich dann aufsummiert.

On-Chip Memory

On-Chip Memory, also ein im Prozessor integrierter Speicher, ist aus mehreren Gründen vorteilhaft/notwendig:

- Große Datenmengen benötigen hohe Bandbreite bei Speicherzugriffen. Wenn der Speicher am gleichen Chip ist, ist der Zugriff schneller.
- Die komplizierte Struktur (Dual-Bus – Super-Harvard) würde externe Hardware noch zusätzlich aufblähen bzw. die Produktentwicklung erschweren.
- Caches sind bei DSPs unbeliebt. Das hat damit zu tun, dass DSP-Anwendungen hauptsächlich Echtzeitanwendungen sind. D.h. die Verarbeitung von soundsovielen Samples pro Sekunde muss *immer* garantiert sein und nicht nur, wenn der Cache gerade korrekt gefüllt ist. Ein On-Chip Speicher ist bei DSPs die bessere Variante.
  Wegen dieser **Performance-Abschätzbarkeit** *sind auch viele andere Features von GPPs bei DSPs nicht anwendbar.*
- Vor allem bei mobilen Geräten (z.B. Mobiltelefonen) ist die Größe des Gesamtsystems ausschlaggebend. Wenn Speicher und Prozessor am selben Chip sind, wird das Ding natürlich kleiner.
- Es sind sogenannte Multiaccess-Memories möglich, bei denen innerhalb eines Taktzyklus mehr als ein Datenwort gelesen/geschrieben werden kann. (Wir erinnern uns an die Forderung von drei Speicherzugriffen pro Taktzyklus beim Filtern.)

DMA – Direct Memory Access

Wegen der hohen I/O Bandbreiten, die bei DSP-Anwendungen notwendig sind, ist I/O auf der Basis von Interrupts (wo sich der Prozessor um die Abholung der I/O-Daten von I/O-Interfaces kümmern muss) inakzeptabel.

Deshalb muss I/O direkt auf den Speicher zugreifen. Da der Speicher on-chip ist, muss auch der I/O-Controller on-chip sein. DSPs verfügen deshalb über eigene I/O-Schnittstellen.

Unterschiedlich ist, abgesehen von der Anzahl und Art der I/O-Schnittstellen, inwieweit der Prozessor durch DMA-Aktivitäten eingeschränkt wird. (Beim ADSP-21160 verliert der Prozessor durch DMA-Zugriffe z.B. gar keine Taktzyklen. **zero overhead DMA**)

Designkriterien

Beim Design *und auch bei der Auswahl* von DSPs gelten folgende Kriterien:

Taktrate Je höher desto schneller. Aber Vorsicht: Energieverbrauch, Wärmeentwicklung.

Energieverbrauch Hängt das zu entwickelnde Gerät an einer Steckdose, ist der Energieverbrauch nebensächlich. Ansonsten . . .

Bauteilgröße Bei mobilen Geräten oft sehr wichtig (Mobiltelefone). Beachte: Am meisten Platz brauchen meistens prozessorexterne Bauteile (wie z.B. zusätzliches Memory, falls das prozessorinterne nicht ausreicht).

Praktische Performance DSP-Hersteller geben meistens mit theoretischen Performancewerten an (wie z.B. MFLOPS berechnet aus Taktrate mal Anzahl der Taktzyklen pro Fließkomma-multiplikation). Betrachtet man die Performance bei Benchmarks (meistens FFT bei DSPs), so sacken gewisse DSPs gewaltig ab. Auch ist es ein Unterschied, ob man die maximale Performance nur per händischer Assembler-Code-Optimierung oder auch mit C-Compiler erreicht.

Time-To-Market IT-Business ist schnelllebig! Wenn der Prozessor schwierig zu handhaben ist (hardware- und/oder softwaretechnisch) und man deswegen das Produkt erst ein halbes Jahr nach der Konkurrenz auf den Markt bringen kann, geht man in Konkurs. :-(

Entwicklungstools Für die meisten DSPs gibt es *Evaluation-Boards*, Assembler, Debugger, C-Compiler, Simulatoren oder ganze grafische Entwicklungsumgebungen.

Energieverbrauch

Der Energieverbrauch kann verringert werden durch:

- Technologie mit niedriger Versorgungsspannung (z.B. 3.3V oder 3V).
- Taktfrequenzregelung. Bei manchen DSPs kann man die Taktfrequenz so herunterregeln, dass sie gerade noch zur Verarbeitung der Daten reicht. Geringere Frequenz $\Rightarrow$ weniger Energieverbrauch.
- Sleep und Idle Modes. Der DSP kann sich schlafen legen (mit minimalem Energieverbrauch), bis er durch ein spezielles Pin, externe Interrupts oder interne Interrupts aufgeweckt wird. *Achtung:* Wie der Mensch braucht auch ein DSP eine gewisse Zeit zum Aufwachen.
- Abschalten von nicht benötigter Peripherie und I/O-Hardware.

Instruktionen und Operationen

Eine **Operation** ist im Prinzip das, was eine einzelne Funktionseinheit ausführt.

Eine **Instruktion** ist ein einzelner Befehl an den Prozessor.

Die Ausführung einer Instruktion beinhaltet also mehrere Operationen. Zum Beispiel:

- Instruktion:
  `ADD R5,R2,R3(100)`
- Operationen:

Instruction Fetch	Hole Instruktion aus Program-Memory
Instruction Decode	Control Unit separiert Op-code und Registernummern, etc.
Address Generation	Adresse = R3 + 100
Operand Fetch	Hole X aus Data-Memory an obiger Adresse
Execute	Ergebnis = X + R2
Write Back	R5 = Ergebnis

Instruction Set

CISC (Complex Instruction Set Computer)

- Am weitesten verbreitet in GPPs
- Gut verständliche Anweisungen mit vielen Varianten

RISC (Reduced Instruction Set Computer)

- Weniger Befehlsvarianten, mehr Register
- Dadurch einfacher im Aufbau $\Rightarrow$ höhere Taktraten möglich
- Dadurch auch einfacher zu programmieren
- *Horizontale Befehlsstruktur*: Bestimmte Bits haben bestimmte Auswirkung auf bestimmte Einheiten.
- Daher sind aber auch längere Code-Wörter notwendig $\Rightarrow$ große Programme
- Eignen sich besser als **Compiler-Target**

Spezialisiert Hauptaugenmerk auf bestimmte Anwendungen

- Verbreitet bei konventionellen DSPs
- Wenige elementare, dafür einige komplizierte spezielle Befehle
- bestimmte Anwendungen sehr schnell
- kleinere Programme
- Willkürliche Einschränkungen beim Benutzen von Registern
- $\Rightarrow$ schwierig zu programmieren, schlechte **Compiler-Targets**

DSP-spezifische Instruktionen

Spezialisierte Instruktionssätze von DSPs umfassen meistens:

Zero-Overhead Loops Mit einer einzigen Instruktion zu Schleifenbeginn wird die ganze Schleife gesteuert. Es ist kein "Decrement – Jump if not zero" am Schleifenende notwendig. Zu Beginn gibt man die Anzahl der Durchläufe und die Anzahl der Instruktionen des Schleifenkörpers an.

Adressierung mit Pointer-Increment Spezielle Adressierung über ein Pointerregister, wobei der Registerinhalt (der Pointer) nach dem Speicherzugriff erhöht wird. So spart man sich eine eigene Instruktion für den Pointer-Update.

Circular Addressing Bei Verwendung von "Circular Buffers" verkompliziert sich natürlich das Pointer-Erhöhen. Auch das wird von vielen DSPs (ohne Overhead) unterstützt.

Multiply-Accumulate Multiplikation zweier Zahlen mit anschließender Aufsummierung. Meistens dient zur Aufsummierung ein speziell dafür vorgesehenes Register (Akkumulator).

Beachte: Mittels "Zero-Overhead Loops" plus "Multiply-Accumulate"-Instruktionen ist es möglich, die beliebte Anwendung "FIR-Filterung" so zu implementieren, dass sie nur einen Taktzyklus pro Filterkoeffizient benötigt (abgesehen von der Schleifeninitialisierung und dem Speichern des Ergebnisses).

Fixkomma versus Fließkomma

In der Anfangszeit der DSPs gab es *nur* Fixkommazahlen, da Fließkommaeinheiten einfach nicht schnell genug waren. Mittlerweile können viele DSPs aber auch Fließkommarechnen.

	Fixkomma	Fließkomma
Performance	gut	nicht immer, aber immer öfter
Entwicklungszeit	hoch	niedrig
Produktkosten	niedrig	hoch
Präzision	schlecht	gut
Dynamischer Bereich	nein	ja

Wegen der obigen Gegenüberstellung steigen viele DSP-Anwender heute auf Fließkommasysteme um.

Wie immer muss man aber abwägen, wie wichtig Performance/Produktkosten im Vergleich zum Entwicklungsaufwand sind (bei hohen Stückzahlen z.B. sehr).

Parallelität 1

Um die hohe Verarbeitungsgeschwindigkeit bei DSPs zu erreichen, müssen gewisse Operationen parallel ausgeführt werden. Generell gilt: Je mehr Parallelität, desto schneller der Prozessor.

Es sind folgende Arten von Parallelität üblich:

- Parallelität auf Operationsebene
  Pipelining Jede Instruktion durchläuft mehrere Stationen, in denen je eine Operation abgearbeitet wird.
  Spezielle Parallelität z.B. zur Schleifensteuerung (Zero-Overhead Loop)
  Multiple Funktionseinheiten Meistens doppelte Multiplizierer und/oder Addierer. Instruktionssatz wird um Befehle für parallele Operationen erweitert.
- Parallelität auf Instruktionsebene (dazu später)
- Parallelität auf Programmebene (dazu später)

Pipelining ist auch bei konventionellen DSPs üblich. Als **Enhanced Conventional DSPs** bezeichnet man üblicherweise jene mit multiplen Funktionseinheiten.

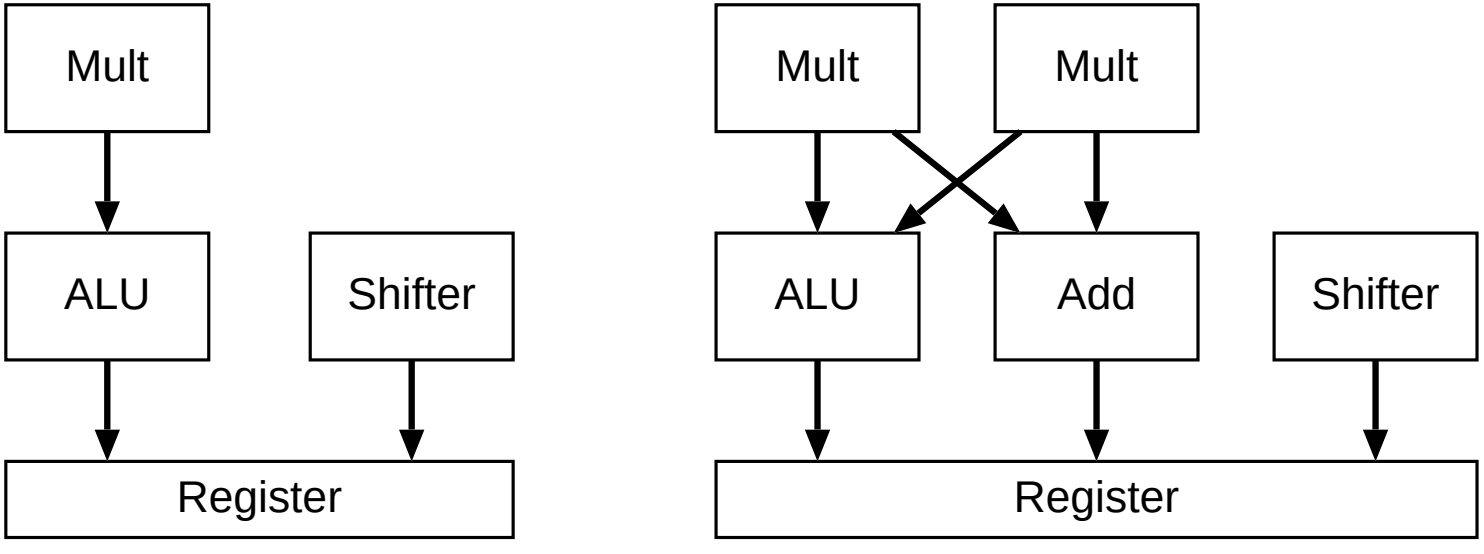
Pipelining

Pipelining ist die am weitesten verbreitete Art der Parallelität und auch bei allen DSPs state-of-the-art. Die Operationen einer Instruktion werden nacheinander abgearbeitet und passieren dabei die Funktionseinheiten. Wenn eine Funktionseinheit frei wird, kann sie schon die nächste Instruktion bearbeiten.

Dabei ist zu beachten: Wenn eine nachfolgende Instruktion vom Ergebnis einer vorhergehenden abhängt, kommt es zu **Pipeline Hazards** und in der Folge zu **Stalls**. Bei Programmverzweigungen passiert das regelmäßig.

Multiple Funktionseinheiten

Beispiel:



Welche Operationen wann und mit welchen Daten durchgeführt werden, wird durch erweiterte Instruktionen bestimmt. Folgen: Sowohl Daten- als auch Programmspeicher-Bus müssen breiter werden. Assembler-Programmierung wird schwieriger. Auch Compiler tun sich schwer.

Parallelität 2

Weitere Formen von Parallelität:

- Parallelität auf Operationsebene (hatten wir schon)
- Parallelität auf Instruktionsebene (Multi-Issue Architektur)

  Superscalar Mehrere Instruktionen werden auf einmal gelesen und dekodiert. Sofern sie nicht voneinander abhängig sind, werden sie parallel abgearbeitet.

  VLIW Wird auch oft *static superscalar* genannt. Hier muss der Compiler entscheiden, welche Instruktionen parallel abgearbeitet werden.

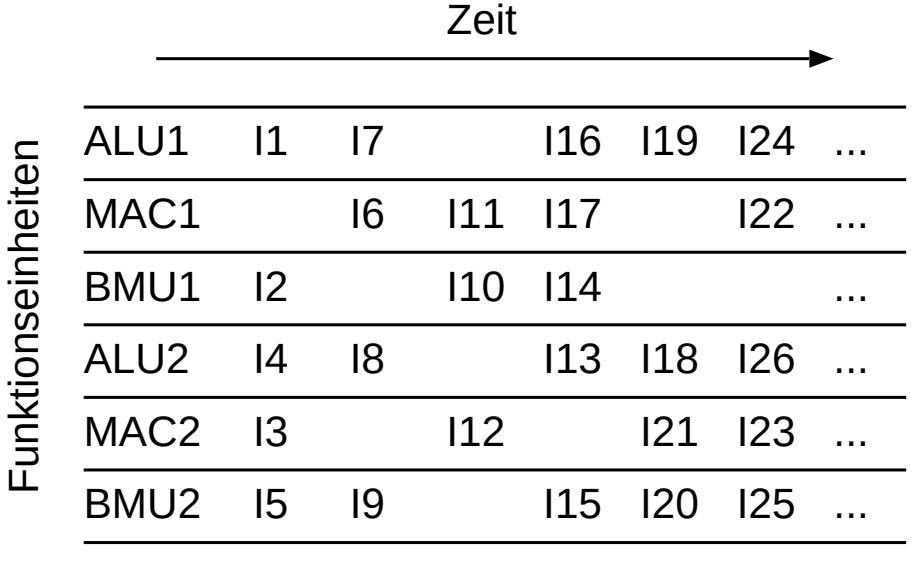
  SIMD Single Instruction Multiple Data. Eine Instruktion wird auf mehrere Datenwörter gleichzeitig angewendet.
- Parallelität auf Programmebene

  MIMD Multiple Instruction Multiple Data (Oft auch SPMD: Single Program Multiple Data). Hier arbeiten im Prinzip mehrere Prozessoren unabhängig.

Superscalar

Bei superskalaren oder **Multi-Issue**-Architekturen werden sehr einfache Instruktionen verwendet, die meist nur eine einzige Operation beinhalten. Dadurch sind auch höhere Taktraten möglich.

Es werden prinzipiell so viele Instruktionen parallel abgearbeitet, wie dazu Funktionseinheiten zur Verfügung stehen. Gewisse Konstellationen ergeben jedoch Konflikte durch Abhängigkeiten. Dadurch müssen des öfteren gewisse Instruktionen warten. Die Steuerung ist sehr kompliziert und macht die Prozessoren größer.

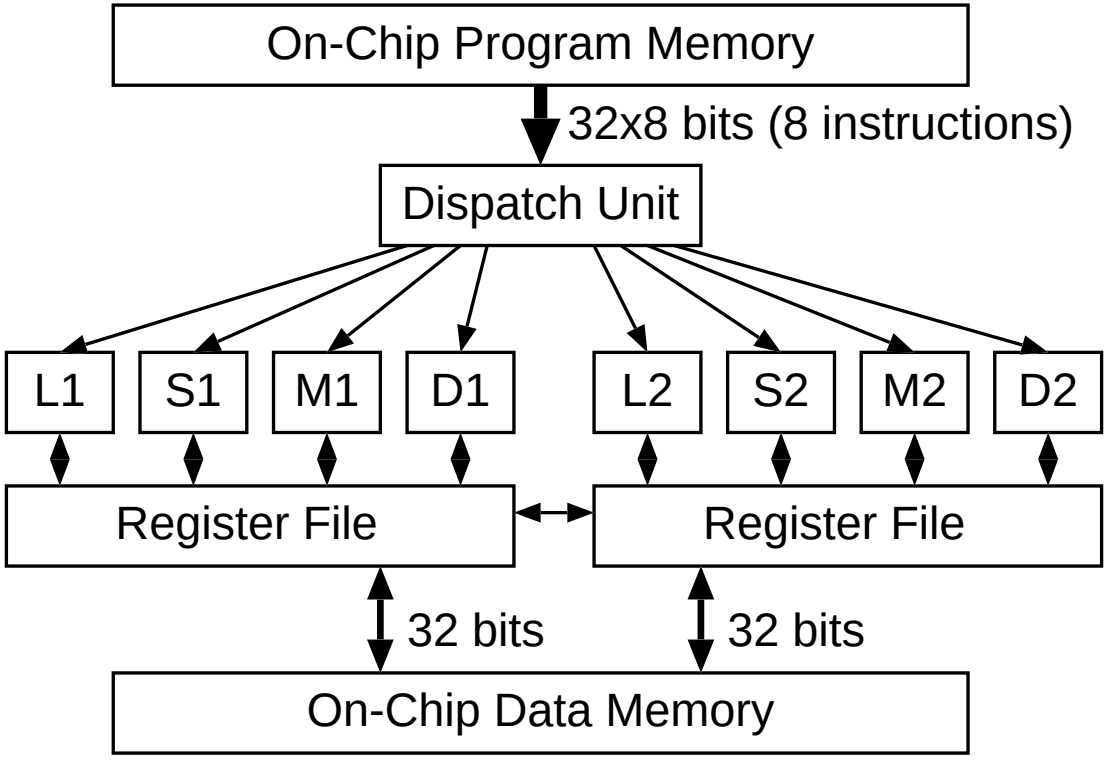


Nachteil für DSPs: Das Scheduling der Instruktionen ist dynamisch. Deshalb ist die Ausführungszeit nicht mehr einfach im voraus zu berechnen.

VLIW – Very Long Instruction Word

VLIW heißt oft auch **static superscalar**. Das soll heißen, dass die Instruktionen schon die Information beinhalten, von welcher Einheit sie verarbeitet werden ⇒ **compile time instruction scheduling**.

Beispiel TMS320C62xx:

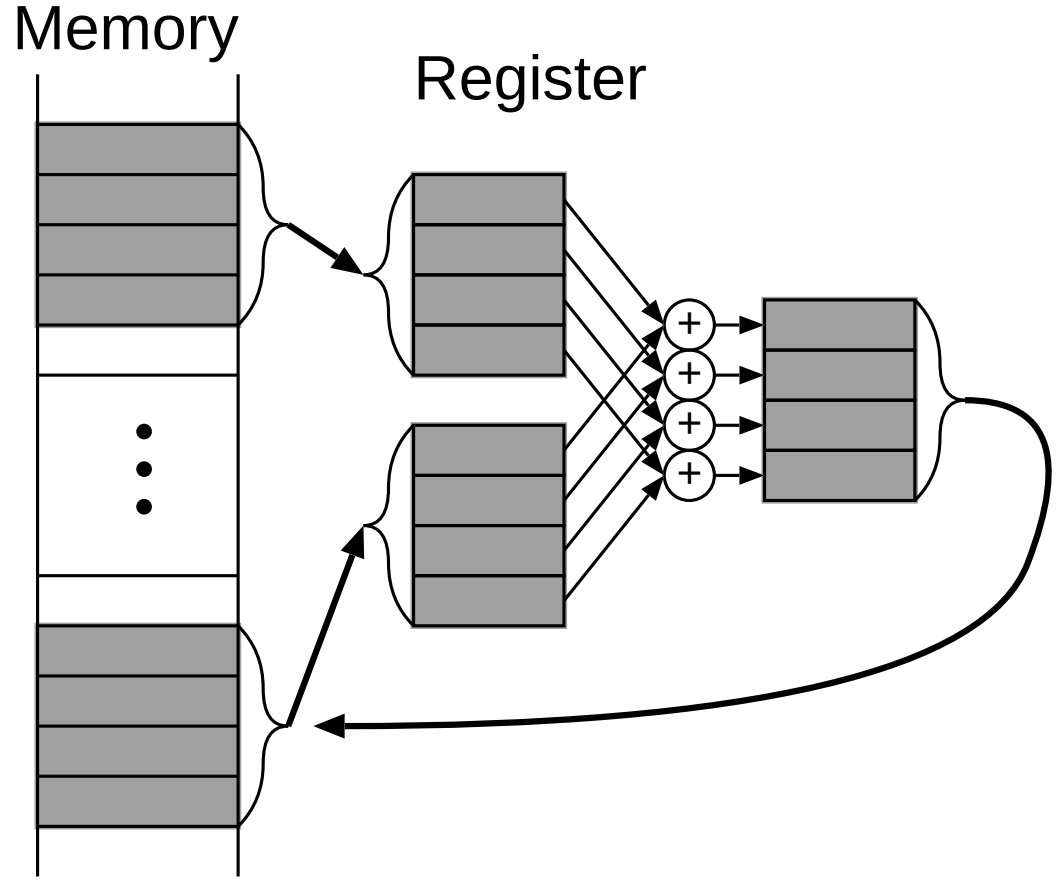


Nachteile: Schwer zu programmieren, schlechte Compiler-Targets.

SIMD – Single Instruction Multiple Data

Eine Instruktion wird gleichzeitig auf mehrere Datenwörter angewandt.

Beispiel `Add`:



- große Memory-Zugriffsbandbreite benötigt
- meist hierarchisch, z.B.: 2×64 bit oder 4×32 bit oder 8×16 bit oder 16×8 bit
- Wenn ein Register mehrere kleinere Wörter enthält (z.B. 16×8 bit), dann nennt man das **packed data**

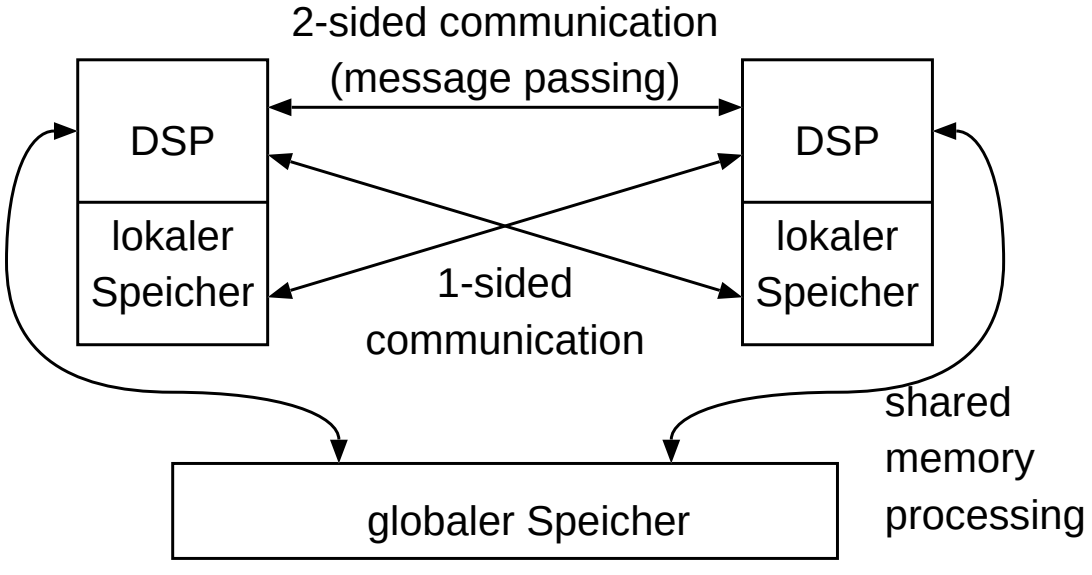
MIMD – Multiple Instruction Multiple Data

- Physischer Aufbau:
  - Mehrere DSPs miteinander verbunden, oder
  - Ein DSP mit mehreren unabhängigen Verarbeitungseinheiten
- Speicherzuordnung

  lokal einem einzelnen DSP zugeordnet (egal ob intern oder extern)
  global allen DSPs *gleichwertig* zugeordnet
  shared für mehrere DSPs erreichbar

  Bemerkung: Aus *global* folgt *shared* aber nicht umgekehrt.

MIMD – Kommunikation



- Direkte Unterstützung durch DSP (z.B. ADSP-2106x unterstützt alle Arten)
- Es ist möglich, I/O-Ports für Message-Passing zu verwenden
- Extra Hardware für shared memory

Gängige DSPs

- Texas Instruments
  TMS320C54xx Konventionell. 16-bit fixed-point. 40-bit MAC.
  TMS320C55xx 2-weg VLIW basierend auf C54xx.
  TMS320C62xx 8-weg VLIW. 16-bit fixed-point.
  TMS320C67xx wie C62 aber mit 32-bit floating-point.
  TMS320C80 4-weg MIMD + Mastercontroller. 8/16/32-bit fixed-point (SIMD).
- Analog Devices
  ADSP-2106x "SHARC". 32-bit floating-point. 80-bit Akku. Guter Support für Multi-Prozessor MIMD.
  ADSP-2116x "Hammerhead". 2-weg SIMD basierend auf 2106x.
  ADSP-TS0xx "TigerSHARC" 4-weg VLIW plus SIMD. 8/16/32/64-bit fixed, 32/40-bit floating-point.
- Motorola
  DSP563xx Konventionell. 24-bit fixed-point.
- Lucent Technologies
  DSP16xxx Konventionell. 16-bit fixed-point.
- und einige andere . . .

Mehr Informationen gibt es auf: www.bdti.com

FPGAs – Field Programmable Gate Arrays

- Geschichte

  PLD Arrays von AND-Gattern werden mit Arrays von OR-Gattern verbunden. Damit kann man eine große Menge von logischen Schaltkreisen abdecken.

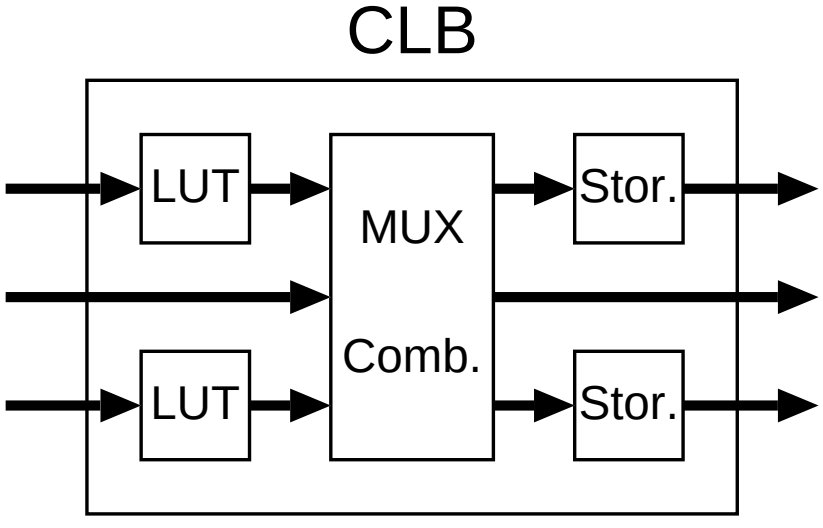
  MPGA Mask Programmable Gate Arrays. Arrays aus Logikeinheiten werden nach Benutzervorgaben beim Hersteller konfiguriert.

  FPGA Wie MPGA. Nur kann hier der Benutzer das FPGA nach Belieben selbst programmieren.

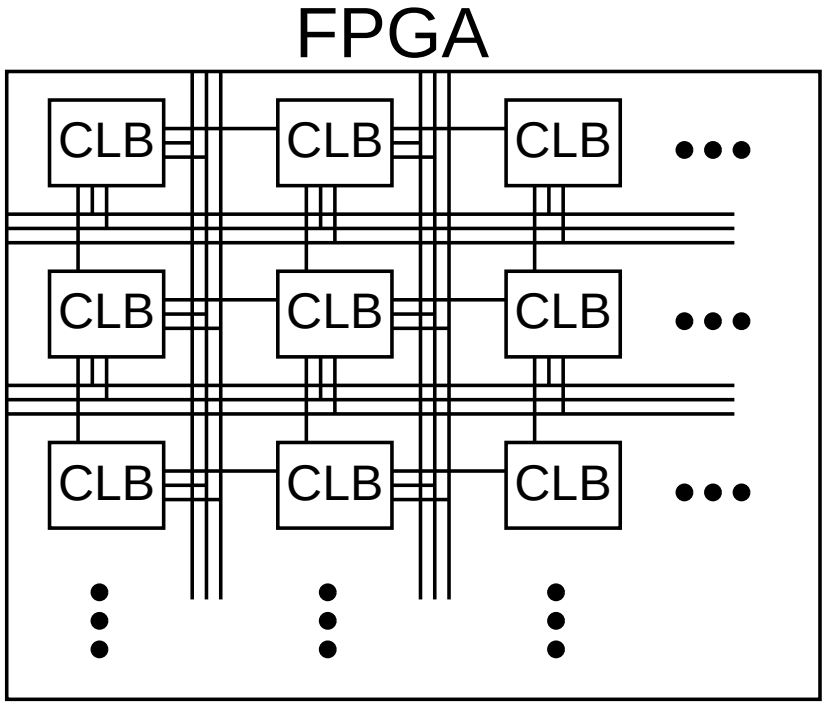
- Aufbau
  - Ansammlung von **CLBs** (Configurable Logic Blocks). Das sind primitive Logikeinheiten, die programmierbar sind mittels:
    * Lookup Tables (LUTs): kleine Speicher (8×8 bits)
    * MUX (Multiplexer, konfigurierbare N:N-Verbindungen zwischen Bauteilen)
  - Verbindung der CLBs über kurze (zu benachbarten Einheiten) und längere (aufwändigere) Verbindungen

FPGAs besitzen die Fähigkeit, bei richtiger Programmierung einen ganzen Prozessor zu ersetzen. Durch die Konfigurierbarkeit können sie auf spezielle Anwendungen “zurechtgeschneidert“ werden und so Parallelität optimal nutzen.

FPGA – Aufbau



Ein FPGA ist aus einer Menge von CLBs zusammengesetzt. Obiges ist nur ein Beispiel für einen CLB. CLBs können sehr verschiedenartig aussehen.
Konfiguriert/Programmiert werden kann ein CLB über die LUTs und die Multiplexer (MUX).



Zusätzlich können die Verbindungen zwischen den CLBs konfiguriert werden.

Programmiert wird ein FPGA über eine **HDL** (Hardware Definition Language). Weit verbreitet: VHDL und Verilog HDL.

Media Processors

Media Processors sind Prozessoren, die speziell für den Einsatz in Audio- und Videokodierung bestimmt sind. Sie werden daher oft auch **VSPs** (Video Signal Processors) genannt. Ihr Einsatzgebiet ist noch enger als das der DSPs.

Audio- und Videokodierung beinhaltet:

- Motion-Estimation/-Compensation (Verschieben und Subtrahieren von Blöcken von Pixeln)
- Frequenztransformation von Blöcken (DCT oder Wavelet)
- Entropiekodierung (Huffman oder arithmetisch)

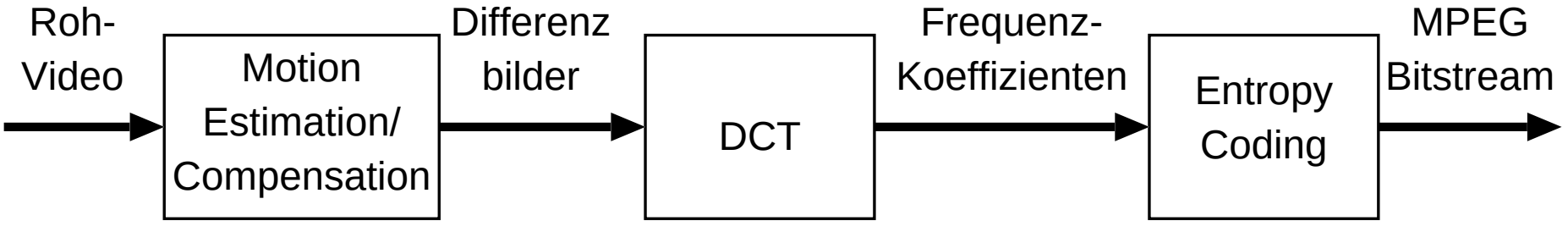
Dabei werden verschiedene Operationen benötigt (gleiche Reihenfolge):

- Arithmetische Operationen auf Bytes (Pixel)
- Fließkommaoperationen (DCT-Koeffizienten)
- Bitmanipulation

Media Processors 2

Es drängt sich auf, diese grundverschiedenen Aufgaben als verschiedene Bauteile in der Hardware zu implementieren. Diese können als Pipeline zusammengeschaltet werden. Diese Art der Parallelität wird übrigens "funktionale Parallelität" genannt.

Konkretes Beispiel:



Darüber hinaus muss wegen der hohen Verarbeitungsrate zusätzliche Parallelität eingeführt werden. Am besten eignet sich dazu die Blockaufteilung bei der Motion-Compensation und der DCT.

Philips Trimedia TM1000

- Seit 1996 erhältlich
- VLIW Kern mit
  - über 20 Functional Units
  - Bis zu 5 parallele Instruktionen
  - 256Kbit instruction cache (mit 220bit breitem Zugriff), 128Kbit data cache
- Image Koprozessor
  - ist eine Pipeline aus spezialisierten Teilprozessoren
  - 1. ein 5-tap FIR Filter
  - 2. ein YUV/RGB Converter
  - 3. Alpha-Blending Einheit
  - 4. Output Formatting
  - All das ist mikroprogrammierbar
- Variable Length Decoder (für Huffman wie in MPEG-1 und -2)
- Verbindung aller Proz. und Koproz. durch einen 32bit Bus
- Kommunikation über Interrupts gesteuert

MicroUnity MediaProcessor

- Einzelner Prozessor, der für alles geeignet ist (großer Instruction-Set, keine Koprozessoren)
- Massives SIMD
- Komplexe Control Operationen (Masking) um Conditional Jumps zu vermeiden
- Sehr guter Shifter (alle möglichen Befehle zum beliebigen Vertauschen von Bits und Bytes)
- Sehr hohe I/O Bandbreite
- Unterstützung von Hyperthreading: D.h. der Prozessor unterstützt die Koordination von Threads (Multitasking wie bei Betriebssystemen), wobei durch Verwendung von disjunkten Teilen des Register-Sets das aufwändige Umschalten zwischen den Threads durch Interrupts wegfällt. Durch das Verwenden von gemeinsamen oder getrennten Pipelines ergeben sich Parallelisierungseffekte. (Unter den GPPs ist das jetzt auch zu finden, z.B. bei Xeon-Prozessoren.)
- Erweiterte Mathematikfunktionen
  - Logarithmus des Most-Significant-Bit
  - Summe der Bits
  - Unterstützung von Galois-Fields für Reed-Solomon Error Correcting Codes
- Gute Möglichkeiten, mehrere Prozessoren zusammenzuschalten (MIMD)

Chromatics MPact2

- VLIW Kern mit 1 oder 2 Instruktionen parallel
- Statt Cache+Register: Ein 73 kbit RAM
- Eine spezialisierte FU für 3D-Grafik
- Eine spezialisierte FU für Motion-Estimation (Eine Differenz zwischen zwei Vektoren mit je 128 8bit-Elementen pro Taktzyklus!)

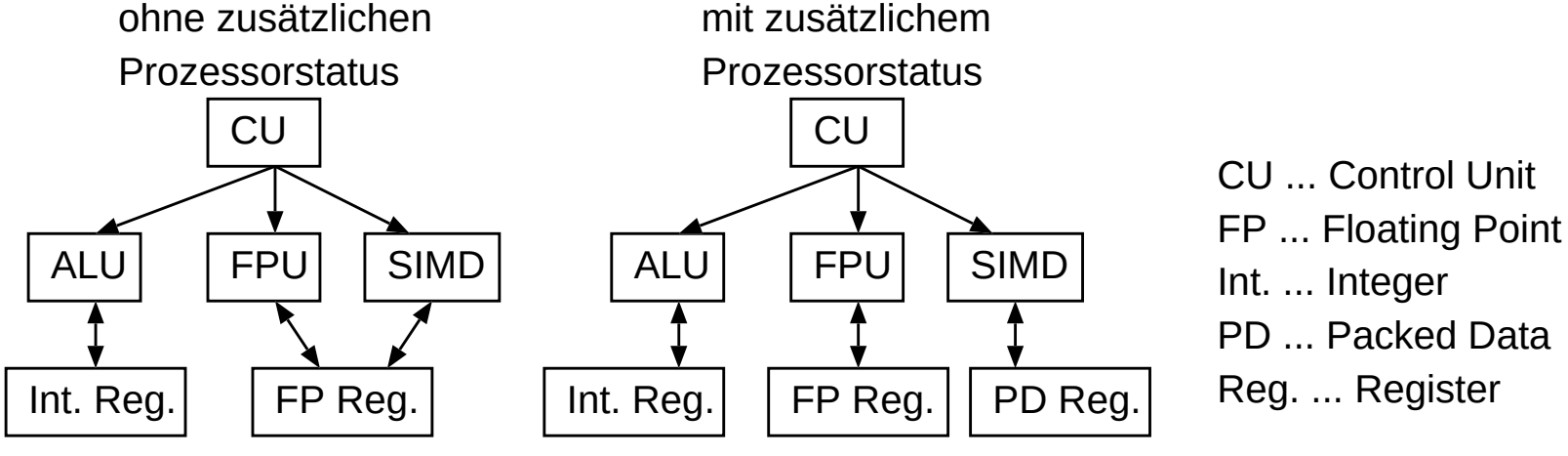
Samsung Media Signal Processor

- Konventioneller 32b RISC Kern (ARM7)
- Der ARM7 ist als Steuer-Modul für verschiedenste Anwendungen designed (Auch/hauptsächlich für *embedded architectures* also Gerätesteuerungen)
- Die Hauptaufgabe des ARM7 ist die Steuerung von bis zu 16 Koprozessoren
- Als Koprozessor wird hier hauptsächlich ein Vektorprozessor eingesetzt
- Dieser ist im Prinzip ein SIMD Prozessor mit 16 32bit Units
- Zusätzlich ist ein MPEG2-bitstream-Koprozessor untergebracht

Multimedia Extensions - Grundidee

Die häufigen Operationen auf Bytes, die in Multimediaanwendungen auf General-Purpose-Computern auftreten verschwenden eindeutig Memory-Bandbreite und Prozessor-Kapazität. Deshalb hat man versucht, SIMD-Elemente in Prozessoren einzubauen.

- Eine zusätzliche Einheit am Prozessor, die SIMD-Operationen durchführt
- Erweiterung des Befehlssatzes für die Einheit
- Entweder werden dafür neue Register vorgesehen oder bestehende verwendet (meist Fießkomma-Register)



Multimedia Extensions - Grundidee 2

- Der Vorteil beim Verwenden von bestehenden Registern ist, dass der Prozessorstatus nicht erweitert wird. Es muss dann das Betriebssystem nicht umgeschrieben werden. Das Problem ist, dass bei Interrupts der ganze Prozessorstatus gesichert werden muss. Dieses Problem besteht bei DSPs nicht.
- Der Nachteil dabei ist, dass Fließkommaoperationen und SIMD Operationen nicht gleichzeitig verwendet werden können.
- Zusätzlich muss es Instruktionen geben, um Packed-Data zu und von normalen Integer-Registern zu kopieren (`pack` und `unpack`).
- Es gibt auch MM-Extensions mit gepackten Fließkommadaten. Dazu sind große Wortbreiten notwendig (≥ 64 bit).

Intel MMX, SSE, SSE2

- MMX (MultiMedia eXtension)
  - verwendet die 8 Fließkommaregister des Intelx86 als 8 64bit-Register für *packed data* $\Rightarrow$ kein zusätzlicher Prozessorstatus
  - Mögliche Datentypen: 8×8 bits, 4×16 bits, 2×32 bits und 1×64 bits (alles Integer)
  - Saturated Arithmetics
- SSE (Streaming SIMD Extension)
  - erweitert MMX
  - 8 **neue** 128bit Register für SIMD
  - Mögliche Datentypen (außer MMX): 4×32 bits Fließkomma
  - Zusätzliche Instruktionen für die neuen Fließkommaregister
  - Zusätzliche MMX-Instruktionen für spezielle Anwendungen (Bildverarbeitung, MPEG, 3D Grafik)
- SSE2 (für Pentium4)
  - Mögliche Datentypen: 4×32 bits und 2×64 bits Fließkomma
  - Integer-Teil (MMX) auf 128bit-Register erweitert (also 16×8 bits usw.)

Motorola's AltiVec für PowerPC

- Zusätzliche 32 Register mit 128bit für SIMD-Operationen
- Mögliche Datentypen: 16×8 bits, 8×16 bits, 4×32 bits Integer und 4×32 bits Fließkomma
- Saturation und Modulo Arithmetik
- Beliebiges Permutieren der Teilwörter
- Gute Unterstützung für: Vektorprodukt, Vektorsumme, Multiply-Accumulate

Andere Multimedia-Extensions

- UltraSPARC – VIS – Visual Instruction Set
  - Hier werden die Fließkommaregister für SIMD-Operationen verwendet
  - Mögliche Datentypen: 4×8 bits, 4×16 bits und 2×32 bits (Integer)
  - Diese Datenformate zielen darauf ab, RGB bzw. YUV plus Alpha-Daten zu beinhalten
  - Befehlssatz sehr spezialisiert auf 2D-, 3D-Grafik, JPEG und MPEG-Verarbeitung
- HP: MAX für PA-RISC
- Silicon Graphics: MDMX für MIPS
- Compaq(Digital): MVI für Alpha