

Computertechnik

Zahldarstellung & Zahlensysteme:

Benötigte Stellen m für Zahl in Basis $B: m = \lfloor \log_B Z \rfloor + 1$ $Z = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \dots + a_1 \cdot B + a_0$
 Größte Zahl, mit m Stellen Basis B darstellbar: $Z_{\max} = B^m - 1$

Zahldarstellungen sind nur Interpretationen von Bitmustern! → Jeweilige Zahldarstellung (1er/2er Komplement, Gleitkomma,...) muss bekannt sein!

| | | | | | | | | | | | | |
|------------------|---------------------------------|------|-----|-----|-----|----|----|----|---|---|---|---|
| Binär (10110001) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 2048 | 1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Hex (0xB1) | A | B | C | D | E | F | | | | | | |
| | 10 | 11 | 12 | 13 | 14 | 15 | | | | | | |
| | (Immer 4 Binärstellen zusammen) | | | | | | | | | | | |
| Oktal (261) | (Immer 3 Binärstellen zusammen) | | | | | | | | | | | |

Binärstellen nach dem Komma 0,11111111 | 0,5 | 0,25 | 0,125 | 0,0625 | 0,03125 | 0,015625 | 0,0078125 | 0,00390625

| | | |
|-------------------------------|--|---|
| Vorzeichenlos | Pos. Zahl | Einfache Umformung |
| | Neg. Zahl | Nicht darstellbar! |
| 1er Komplement | Pos. Zahl | Einfache Umformung |
| | Neg. Zahl | Pos. Zahl → Bits(+VZ) invertieren |
| | Kleinsten darstellbarer Wert: $-(2^{n-1} - 1)$ | |
| | Größter darstellbarer Wert: $+(2^{n-1} - 1)$ | |
| | Doppelte Darstellung der Null: 1111 = 0000 | |
| 2er Komplement | Pos. Zahl | Einfache Umformung |
| | Neg. Zahl | Pos. Zahl → Bits(+VZ) invertieren → +1 addieren |
| | Kleinsten darstellbarer Wert: $-(2^{n-1})$ | |
| | Größter darstellbarer Wert: $+(2^{n-1} - 1)$ | |
| | Keine Doppelte Darstellung der Null!!! | |
| | Achtung: Overflow bei Addition von zwei pos. bzw. zwei neg. Zahlen! | |
| Gleitkomma (IEEE 754) | Null: Exponent: 0...0, Mantisse: 0...0 | |
| | Nicht normalisierte Zahlen: Exponent: 0...0, Mantisse: "nicht normalisierte Zahl" | |
| 32Bit 64Bit | Unendlich: Exponent: 1...1, Mantisse: 0...0 VZ: | |
| 1 1 | Not a Number (NaN): Exponent: 1...1, Mantisse: nicht 0...0 | |
| 8 11 | Wert: $W = (-1)^s \cdot (1 + M) \cdot 2^{E-K}$ (Erst binäre Kommazahl bestimmen, dann in dezimal umwandeln!) | |
| 23 52 | | |
| K=127 K=1023 | | |
| Interpretation einer Hex-Zahl | 1er Kompl. | Stellen auf F erweitern → umformen |
| | 2er Kompl. | Stellen auf F erweitern → 1 addieren → umformen |

Zahldarstellungen sind nur Interpretationen von Bitmustern!

Floating Point (Gleitkomma) Arithmetik:

| | |
|----------------|--|
| Addition | <ul style="list-style-type: none"> ▪ Exponenten angleichen (Genauigkeit (Stellen) gehen verloren) ▪ Durchführen der Addition (Summe kann zwei Stellen vor dem Komma enthalten) ▪ Normalisierung durch Rechts-Shift des Ergebnisses (Exponent neu bestimmen) |
| Multiplikation | <ul style="list-style-type: none"> ▪ Berechnung des Produktexponenten durch Addition der Exponenten der Faktoren ▪ Multiplikation durchführen (Mantisse!) + Normalisierung 1,... ▪ Vorzeichen des Produktes aus den Vorzeichen der Faktoren ermitteln |
| Division | <ul style="list-style-type: none"> ▪ Berechnung des Produktexponenten durch Subtraktion der Exponenten der Faktoren ▪ Division durchführen (Mantisse!) + Normalisierung 1,... ▪ Vorzeichen des Produktes aus den Vorzeichen der Faktoren ermitteln |

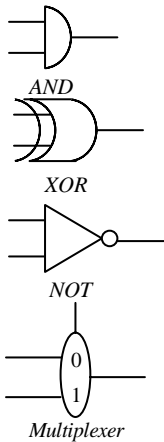
Achtung: Floating Point Arithmetik ist nicht assoziativ! (Genauigkeit!)

Boolsche Algebra:

UND: $\bullet \wedge$ (konj.)

ODER: $+ \vee$ (disj.)

NEG vor UND vor ODER!!!



Kommutativ: $a \bullet b = b \bullet a; a + b = b + a$

Assoziativ: $(a + b) + c = a + (b + c)$

$(a \bullet b) \bullet c = a \bullet (b \bullet c)$

Absorbtion: $a \bullet (a + b) = a; a + (a \bullet b) = a$

Distributiv: $(b + c) \bullet a = (b \bullet a) + (c \bullet a)$

$(b \bullet c) + a = (b + a) \bullet (c + a)$

DeMorgan: $(\overline{a + b}) = \overline{a} \bullet \overline{b}; (\overline{a \bullet b}) = \overline{a} + \overline{b}$

Neutrales Element: $a \bullet 1 = a; a + 0 = a$

Komplement Element: $a \bullet \overline{a} = 0; a + \overline{a} = 1$

Involution: $\overline{\overline{a}} = a$

Idempotenz: $a \bullet a = a; a + a = a$

Domiananz: $a \bullet 0 = 0; a + 1 = 1$

UMWANDLUNG: Zuerst Distributivgesetz!!!

$f_1(x, y) = 0$ Nullfunktion

$f_2(x, y) = x \cdot y$ UND

$f_3(x, y) = x \cdot \overline{y}$ Inhibition

$f_4(x, y) = x$ Identität in x

$f_5(x, y) = \overline{x} \cdot y$ Inhibition

$f_6(x, y) = y$ Identität in y

$f_7(x, y) = (x + y) \cdot \overline{(x \cdot y)}$ xOR Antiv

$f_8(x, y) = x + y$ ODER

$f_9(x, y) = \overline{x + y}$ NOR

$f_{10}(x, y) = (x \cdot y) + \overline{(x + y)}$ Äquivalenz

$f_{11}(x, y) = \overline{y}$ Negation von y

$f_{12}(x, y) = x + \overline{y}$ Implikation ($y \rightarrow x$)

$f_{13}(x, y) = \overline{x}$ Negation von x

$f_{14}(x, y) = \overline{x} + y$ Implikation ($x \rightarrow y$)

$f_{15}(x, y) = \overline{(x \cdot y)}$ NAND

$f_{16}(x, y) = 1$ Einsfunktion

MMIX Assembler Programmierung:

MMIX Befehle:

Marke *LZ Anweisungen* *LZ Kommentar*
optional nicht optional optional

Befehlsarten:

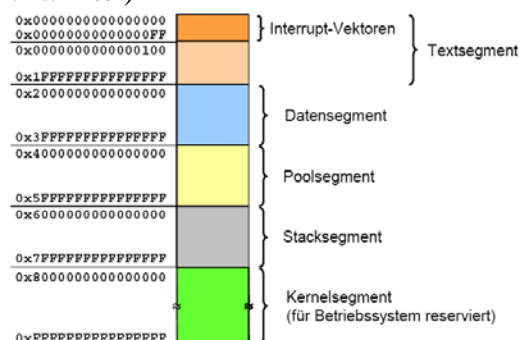
- Befehle aus MMIX-Befehlssatz (werden vom Assembler in 8Bit Befehlscode + 24Bit Operanden übersetzt)
- Loader-Befehle (vom Loader beim Laden des Programmes in Arbeitspeicher ausgeführt z.B.: LOC #100)
- Assembler Befehle (Befehle, die während der Übersetzung vom Assembler ausgewertet werden, Lesehilfe für Programmierer, zB: ADD x,x,y statt ADD \$4,\$4,\$5)

Speicher:

Der Speicher ist 2^{64} Byte groß. Die kleinste adressierbare Einheit ist ein Byte. Darüberhinaus kann Wyde (2 Byte), Tetra (4 Byte) und Octa (8 Byte) adressiert werden. Jedoch können diese immer nur an ganzzahligen Vielfachen von sich selbst adressiert werden!

Beim Abspeichern wird jedem Byte eine 8Bit Speicherzelle zugewiesen und Elemente eines Byte Arrays werden hintereinander im Speicher abgelegt. Beim Abspeichern eines aus mehreren Byte zusammengesetzten Wort (z.B.: Integer mit vier Byte) gibt es zwei Formate für die Adressierung (→ Probleme beim Datenaustausch?!):

- Big-Endian: höchstwertiges Byte wird adressiert (z.B.: MMIX)
- Little-Endian: niederwertigstes Byte wird adressiert (z.B.: Intel)

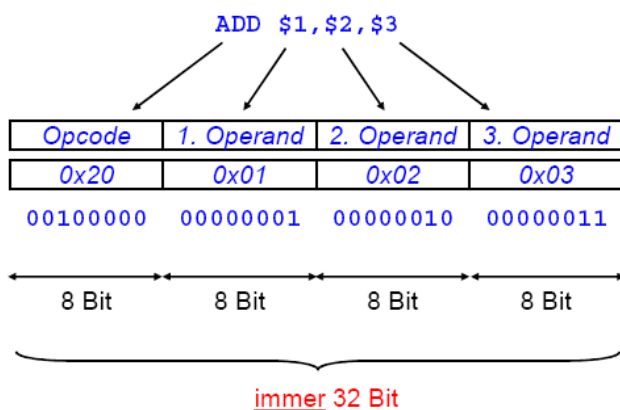


Übersicht MMIX Befehle:

| Art | Befehl | Operanden | Bedeutung |
|---------------------------|--------------------|--------------------|--|
| | LOC | #100 | Programm beginnt an dieser |
| | LOC | Data_Segment | Arbeitsspeicher-Adresse (0x2000000000000000) |
| | GREG | @ | Momentane Speicheradresse wird in nächstes (1. 254, 2. 253,...) globales Register eingetragen. |
| | a IS | \$1 | Reg. 1 kann jetzt mit a bezeichnet werden |
| | SET | \$1,xxx | Für \$1 Wert setzen (direkt max. 65535) |
| Stack | BOS | GREG | #4000000000000000 |
| | SP | GREG | #4000000000000000 |
| | | | Bottom of stack |
| | | | Stackpointer |
| Funktionen | PREFIX | Main: <i>start</i> | allen Variablen wird automatisch „Main:“ vorangestellt („:“ vor Variable → „Main:“ wird nicht vorangestellt, auch bei Funktionen verwenden!!!) |
| | PREFIX | : <i>ende</i> | |
| Arithmetik | MUL | \$X,\$Y,\$Z | $\$X = \$Y * \$Z$ |
| | SUB | \$X,\$Y,\$Z | $\$X = \$Y - \$Z$ |
| | ADD | \$X,\$Y,\$Z | $\$X = \$Y + \$Z$ |
| | DIV | \$X,\$Y,\$Z | $\$X = \$Y / \$Z$ |
| Gleitkommaarithmetik | FLOT | \$X,\$Y | $\$X = \Y in GKZ transformiert |
| | FSQRT | \$X,\$Y | $\$X = \text{SQRT}(\$Y)$ |
| | FADD | \$X,\$Y,\$Z | $\$X = \$Y + \$Z$ |
| | FSUB | \$X,\$Y,\$Z | $\$X = \$Y - \$Z$ |
| | FDIV | \$X,\$Y,\$Z | $\$X = \$Y / \$Z$ |
| Speicher Reservieren | <i>Marke</i> BYTE | <i>Wert</i> | Reserviert 1 Byte im Speicher |
| | <i>Marke</i> WYDE | <i>Wert</i> | Reserviert 2 Byte im Speicher |
| | <i>Marke</i> TETRA | <i>Wert</i> | Reserviert 4 Byte im Speicher |
| | <i>Marke</i> OCTA | <i>Wert</i> | Reserviert 8 Byte im Speicher (a OCTA 4) |
| Speicher Laden | LDB | \$4,\$5,\$6 | In Register 4 den Inhalt des Speichers eintragen! [\$5+\$6 → \$4] (z.B.: LDO \$2,x2 Wert von x2 an Register 2 speichern) |
| | LDW | \$4,\$5,\$6 | |
| | LDT | \$4,\$5,\$6 | |
| | LDO | \$4,\$5,\$6 | |
| | LDBU,LDWU,.. | \$4,\$5,\$6 | Vorzeichenloses (unsigned) laden! |
| Speicher Speichern | STB | \$4,\$5,\$6 | \$4 in \$5+\$6 speichern |
| | STW | \$4,\$5,\$6 | |
| | STT | \$4,\$5,\$6 | |
| | STO | \$4,\$5,\$6 | |
| | STBU,STWU,.. | \$4,\$5,\$6 | Vorzeichenloses (unsigned) speichern! |
| Sprungbefehle (unbedingt) | JMP | xyz | xyz (24Bit) gibt Sprungweite in Byte an |
| | JMP | Main | Direkter Sprung zu Marke |
| | GO | \$4,\$5,\$6/6 | Sprung zu \$5+\$6 oder \$5+6 |
| | GO | \$4,Main | Direkter Sprung (\$4 Rücksprungadresse) |
| | | | |
| Sprungbefehle (bedingt) | BZ | \$X,Ende | Falls $\$X == 0$, dann Verzweigung zu Ende |
| | BN | \$X,Ende | Falls $\$X < 0$, dann Verzweigung zu Ende |
| | | | |
| | | | |
| | | | |

| | | | |
|------------------------|------|-------------|---|
| Bitoperatoren | SL | \$X,\$Y,a | \$X = \$Y um a Bit nach links verschoben (a=2 → Multiplikation mit 4) |
| | SR | \$X,\$Y,a | \$X = \$Y um a Bit nach rechts verschoben |
| | | | |
| | SRU | | Vorzeichenfreies schieben (sonst wird MSB nachgezogen) |
| logische Bitoperatoren | OR | \$X,\$Y,\$Z | Oder Verknüpfung von \$Y mit \$Z nach \$X |
| | XOR | \$X,\$Y,\$Z | XOR Verknüpfung von \$Y mit \$Z nach \$X |
| | | | |
| | | | |
| Vergleiche | CMP | \$X,\$Y,\$Z | \$X=(-1 \$Y<\$Z, 1 \$Y>\$Z, 0 \$Y==\$Z) |
| | | | |
| | | | |
| | CMPU | \$X,\$Y,\$Z | Vorzeichenloser Vergleich |
| Ende | TRAP | 0,HALT,0 | Ende des Programms |

Assemblierung der MMIX Befehle:



Befehlsdauer $\left\{ \begin{array}{l} \mu \quad \text{Dauer eines Speicherzyklus} \\ \nu \quad \text{Dauer eines Taktzyklus} \\ \pi \quad \text{Zeitaufwand für Programmsprung} \end{array} \right.$

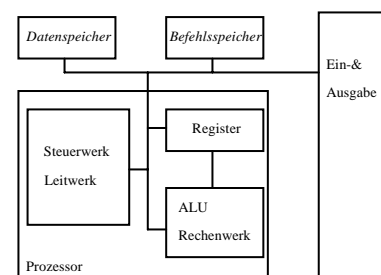
$$f = \frac{1}{T}$$

Datenpfad:

Harvard Architektur:

Befehls- & Datenspeicher sind getrennt.

→ Gleichzeitiger Zugriff auf Befehls- und Datenspeicher möglich (Vorteil für Pipelining)



Neumann Architektur:

Befehle und Daten befinden sich im selben Speicher.

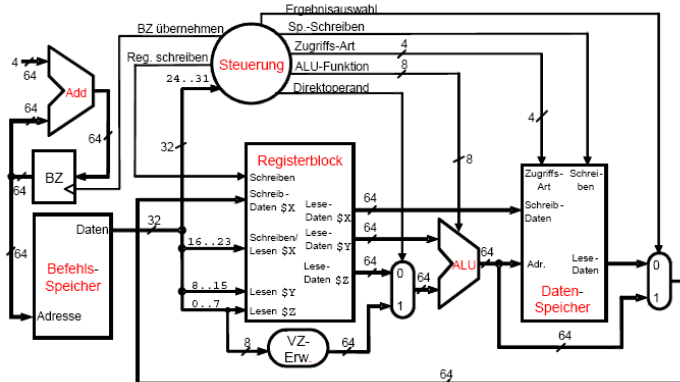
→ Programmstruktur einfacher, da es nicht zuerst vom Datenspeicher in den Befehlsspeicher geladen werden muss.

Grundlegende Schritte der Befehlsausführung:

- Speicher mit Wert des Befehlszählers adressieren und nächste Instruktion laden
- Ein oder zwei Register lesen um die Operanden zur Verfügung zu stellen
- Operationsführung oder Adressberechnung mit ALU
- Abschließen der Instruktion (Speicherzugriff, ALU Ausgang in Register schreiben, Neuen Wert in Befehlszähler schreiben)

Single-Cycle-Datenpfad:

$$\text{Programmdauer} = \text{"Dauer längster Befehl"} \cdot \sum (\text{alle Befehle})$$

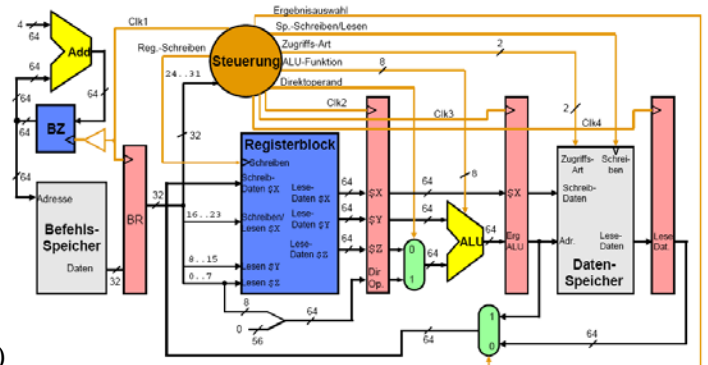


- Jeder Befehl benötigt eigenen Taktzyklus
- → Jeder Befehl dauert so lange wie der längste unterstützte Befehl
- 8ns pro Instruktion
- → Takt orientiert sich an der langsamsten Instruktion
- → Mangelnde Effizienz, da Leerlauf bei kürzeren Instruktionen
- Jedes Hardwaremodul kann nur einmal pro Taktzyklus verwendet werden

MultiCycle Datenpfad:

$$\text{Programmdauer} = \sum_i (\text{HäufigkeitBefehl} \cdot \text{DauerBefehl})$$

- Leerezeiten im Datenpfad werden minimiert
- Befehle werden in mehreren Taktzyklen abgearbeitet → Verkürzung kritischer Pfad
- Ausführungszeiten der Maschinenbefehle unabhängig voneinander (Ausführungszeit ‚langer‘ hat keine Auswirkung auf ‚kürzere‘)
- Zusätzliche Hardware (Register für Zwischenspeicherung)
- Einsparung von ALUs (Nur ein Speicher, ALU)

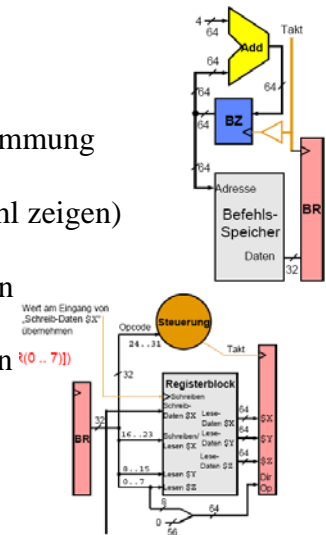


5 Zyklen einer Instruktion:

- BH: Befehl holen (IF = Instruction Fetch)
- BD: Befehl dekodieren (ID = Instruction Decode)
- AF: Befehl ausführen (EX = Execute)
- SP: Speicherzugriff (MEM = Memory Access)
- ES: Ergebnis schreiben (WB = Write Back)

BH: Nächster auszuführender Befehl in Befehlsregister laden (Bestimmung durch Befehlszähler, enthält Adresse des nächsten Befehls);
 Befehlszähler für nächsten Befehl anpassen (auf nächsten Befehl zeigen)

BD: Im Befehlsregister stehenden Befehl analysieren und decodieren
 → Steuerung bestimmt Steuersignale
 Operanden für Befehl werden werden aus Registerblock gelesen (liest Quellenregister aus dem Registerblock)



AF: Ausgelesene Operanden zu ALU leiten (Achtung: Unterscheidung Z / \$Z)
 ALU berechnet Ergebnis mit Operation (Steuerleitung bestimmt Operation)
 Für Abspeichern Inhalt von \$X an die nächste Stufe weiterleiten (berechnet effektive Speicheradresse)

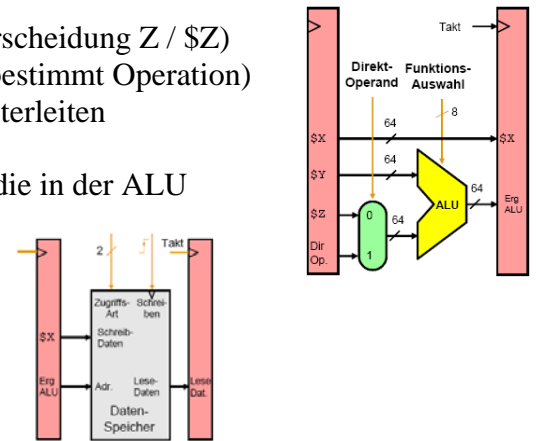
SP: MMIX-Prozessor greift auf Datenspeicher zu (Adresse die in der ALU berechnet wurde)

Datenspeicher: Adressbus: 64bit

Datenbus: 64bit (Ausgang + Eingang)

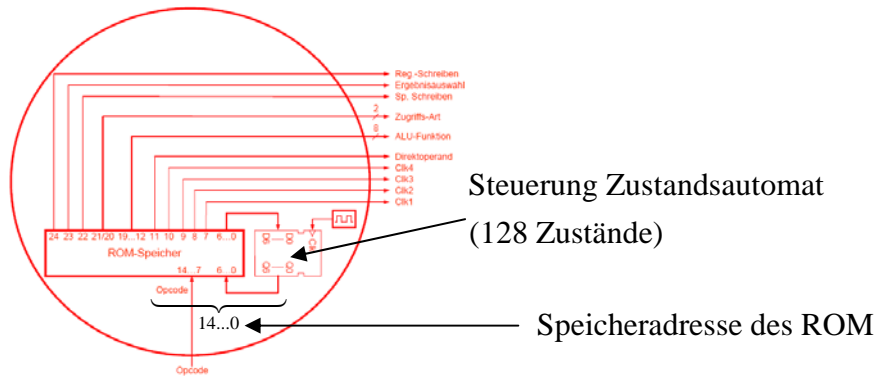
Steuerbus: 2bit (Byte, Wyde, Tetra)

Taktleitung um Daten am Eingang zu übernehmen



ES: Entweder Ergebnis AF-Phase oder in SP-Phase eingelesenes Datenwort in Registerblock schreiben

ROM-basierte Steuerung für MultiCycle Datenpfad:



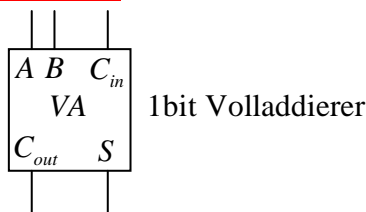
Rechenwerk-ALU (Hardwarerealisierung):

ALU Befehle:

- Arithmetische (Addition, Subtraktion,...) und logische Funktionen (AND, OR,...)
- Weitere Befehle (Vergleichsoperationen, Prüfen auf Null)

Hardwarerealisierung:

Addition:



Subtraktion:

- Addition des 2er Komplements
- C_{in} beim LSB auf „1“ setzen (Addition von 1 zur konjugierten Zahl)
- Steuersignal B_{invert} auf „1“ setzen

Vergleichsoperationen:

- Alle bits gleich „0“ bis auf LSB →
$$\begin{cases} LSB = 1, A - B < 0 \\ LSB = 0, A - B \geq 0 \end{cases}$$

Realisierung über „L“-Leitung, die beim LSB mit SET der Overflow-Detection Unit des MSB verbunden ist, bei allen anderen ist L=0 gesetzt.

Prüfen auf Null:

$$a - b = 0 \Rightarrow a = b \quad Zero = \overline{(S_0 + S_1 + \dots + S_{63})}$$

→ ALU Rechenwerk für 64bit durch Kaskadierung von 64 1Bit Rechenwerken (Ripple-Carry-Addierer). Sehr einfach und flächengünstig aber sehr lange Laufzeiten.

Gatterlaufzeiten:
$$\begin{cases} Sub.: 3n + 3 \\ Add.: 3n + 2 \end{cases} \Rightarrow \text{Zeitkomplexität: } O(n)$$

Implementierung mit 2 stufiger Logik → Mehr Kosten, jedoch geringere Laufzeit!

Schneller Übertrag: Rekursiv: $C_{i+1} = (A_i \cdot B_i) + (A_i + B_i) \cdot C_i = g_i + p_i \cdot C_i$

Generierung: $g_i = A_i \cdot B_i \quad g_i = 1 \Rightarrow C_{i+1} = 1$ (unabhängig von C_i)

Propagation: $p_i = A_i + B_i \quad g_i = 0 \quad p_i = 1 \Rightarrow C_{i+1} = C_i$

→ Realisierung über 2stufigen Carry Lookahead mit vier 4bit ALUs (16bit Carry Lookahead ist ca. 6mal Schneller als 16bit Carry Ripple)

Propagation und Generierung wird jeweils für einen Block berechnet!

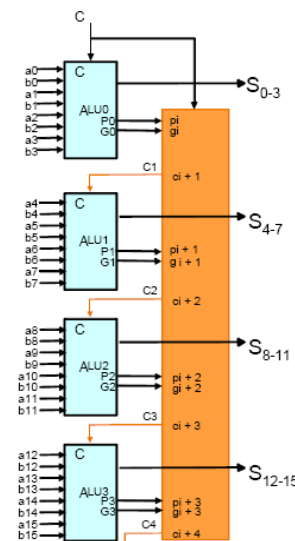
→ Propagation für einen Block tritt genau dann auf, wenn alle Bits des Blockes propagieren

→ **Generierung** für einen Block tritt genau dann auf, wenn im Block ein Carry-Signal entsteht, das hinaus transportiert wird

| | |
|---|---|
| $P_0 = P_3 \cdot P_2 \cdot P_1 \cdot P_0$ | $G_0 = g_3 + (p_3 \cdot g_2) + (p_3 \cdot p_2 \cdot g_1) + (p_3 \cdot p_2 \cdot p_1 \cdot g_0)$ |
| $P_1 = P_7 \cdot P_6 \cdot P_5 \cdot P_4$ | $G_1 = g_7 + (p_7 \cdot g_6) + (p_7 \cdot p_6 \cdot g_5) + (p_7 \cdot p_6 \cdot p_5 \cdot g_4)$ |
| $P_2 = P_{11} \cdot P_{10} \cdot P_9 \cdot P_8$ | $G_2 = g_{11} + (p_{11} \cdot g_{10}) + (p_{11} \cdot p_{10} \cdot g_9) + (p_{11} \cdot p_{10} \cdot p_9 \cdot g_8)$ |
| $P_3 = P_{15} \cdot P_{14} \cdot P_{13} \cdot P_{12}$ | $G_3 = g_{15} + (p_{15} \cdot g_{14}) + (p_{15} \cdot p_{14} \cdot g_{13}) + (p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12})$ |

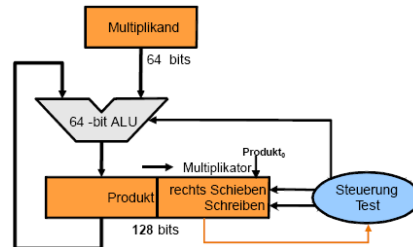
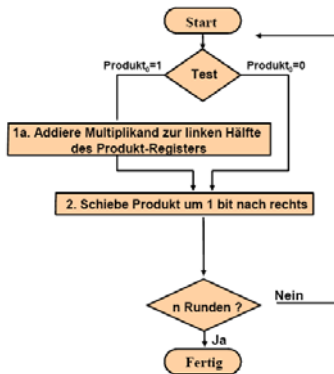
| |
|---|
| $C_1 = G_0 + (P_0 \cdot C_0)$ |
| $C_2 = G_1 + (P_1 \cdot G_0) + (P_1 \cdot P_0 \cdot C_0)$ |
| $C_3 = G_2 + (P_2 \cdot G_1) + (P_2 \cdot P_1 \cdot G_0) + (P_2 \cdot P_1 \cdot P_0 \cdot C_0)$ |
| $C_4 = G_3 + (P_3 \cdot G_2) + (P_3 \cdot P_2 \cdot G_1) + (P_3 \cdot P_2 \cdot P_1 \cdot G_0) + (P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0)$ |

→ Zeitkomplexität: $O(\log n)$

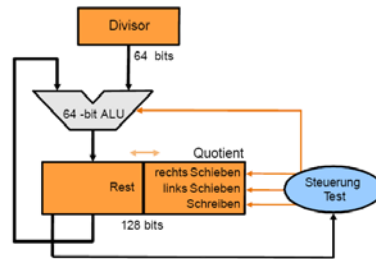
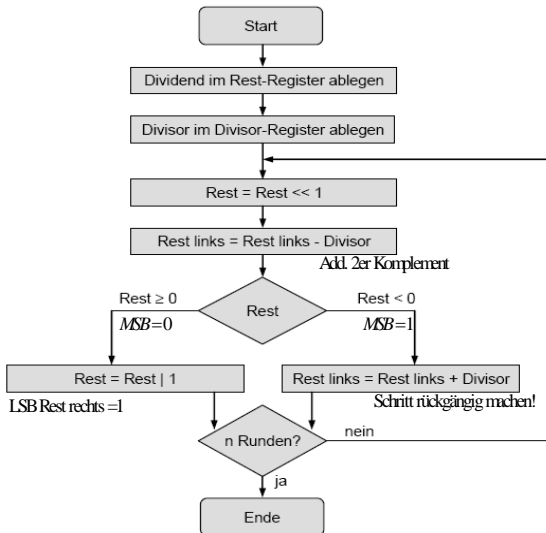


Implementierung arithmetischer Operationen:

Multiplikationsalgorithmus:



Divisionsalgorithmus:

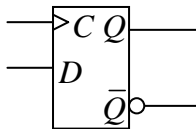


Registerblock:

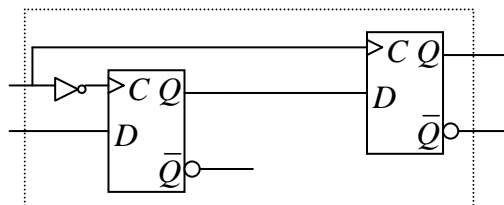
- Schneller Zwischenspeicher
- Element des Datenpfads
- MMIX hat 256 Register á 64bit (Realisierung: 256 * 64 flankengest. 1bit D-FlipFlops)

1bit D-FlipFlop:

D-FlipFlop

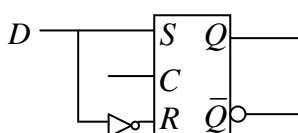


Master / Slave

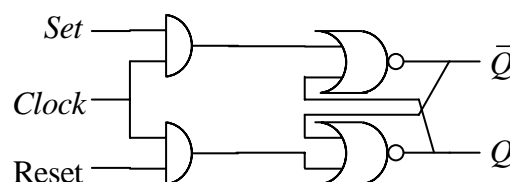


Durch Kaskadierung von zwei D-FlipFlops wird Taktflankensteuerung erreicht (steigende Flanke)

Taktgest. RS-FlipFlop



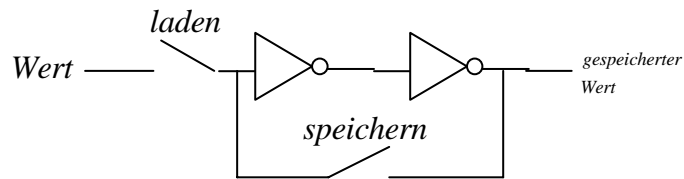
Gatter-Ebene



Sequentielle Schaltungen:

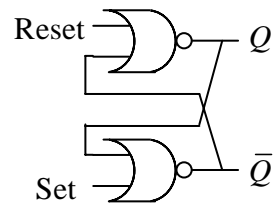
(kombinatorische Schaltungen + Feedback) → Grundlage für Datenspeicherung

Zwei Inverter als statische Speicherzelle:



Speicher mit verkoppelten NOR-Gattern:

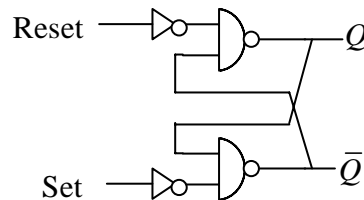
| S | R | $Q(t)$ | $Q(t+\Delta)$ | |
|---|---|--------|---------------|----------------|
| 0 | 0 | 0 | 0 | hold |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 0 | reset |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | set |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 0 | X | nicht erlaubt! |
| 1 | 1 | 1 | X | |



$$Q(t+\Delta) = S + \bar{R} \cdot Q(t)$$

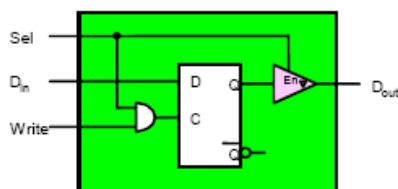
Speicher mit verkoppelten NAND-Gattern:

| \bar{S} | \bar{R} | $\bar{Q}(t)$ | $\bar{Q}(t+\Delta)$ | |
|-----------|-----------|--------------|---------------------|----------------|
| 0 | 0 | 0 | X | nicht erlaubt! |
| 0 | 0 | 1 | X | |
| 0 | 1 | 0 | 0 | reset |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | set |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 0 | 0 | hold |
| 1 | 1 | 1 | 1 | |

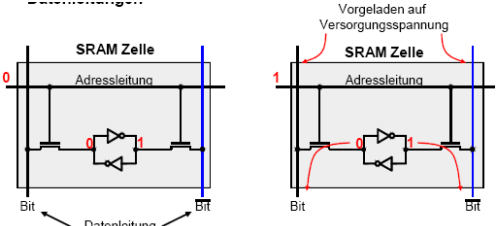
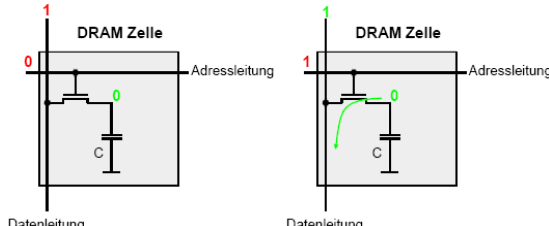
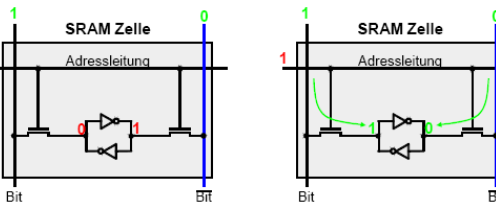
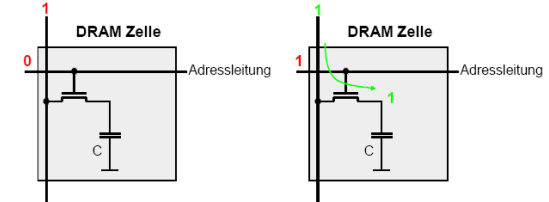


$$Q(t+\Delta) = S + \bar{R} \cdot Q(t)$$

Speicherzelle für 1bit:



Speicherzellen:

| SRAM (Static Random Access Memory) | DRAM (Dynamic Random Access Memory) |
|--|--|
| <ul style="list-style-type: none"> • Speicherung durch Rückkopplung • 6 Transistoren je bit • schnell, teuer • schnelle Speicher (Register, Cache) | <ul style="list-style-type: none"> • Daten als Kondensatorladung • → Auffrischung erforderlich • schwierig in Logik IC zu integrieren • langsam, billig |
| <p><u>Lesevorgang:</u></p> <ul style="list-style-type: none"> • Datenleitungen vorladen auf Versorgungsspg. • Adressleitung auswählen • Speicherzelle zieht Datenleitung auf Masse (LOW) → Diff.verstärker detektiert Spg.differenz auf Datenleitungen  <p>The diagram shows two SRAM cells during a read operation. In the first, the address line is 0 and the data line is 0. In the second, the address line is 1 and the data line is 1. The text 'Vorgeladen auf Versorgungsspannung' indicates the data lines are precharged to the supply voltage.</p> | <p><u>Lesevorgang:</u></p> <ul style="list-style-type: none"> • Datenleitung vorladen auf Versorgungsspg. • Adressleitung auswählen • Detektion und Verstärkung der Ladungsänderung • Zurückschreiben des gelesenen Wertes! • Auffrischen → Dummy Lesevorgang!  <p>The diagram shows two DRAM cells during a read operation. In the first, the address line is 0 and the data line is 0. In the second, the address line is 1 and the data line is 1. A capacitor 'C' is shown in each cell, and a green arrow indicates the charge being read from the capacitor to the data line.</p> |
| <p><u>Schreibvorgang:</u></p> <ul style="list-style-type: none"> • Datenleitungen beschalten • Adressleitung auswählen  <p>The diagram shows two SRAM cells during a write operation. In the first, the address line is 0 and the data line is 1. In the second, the address line is 1 and the data line is 0. Green arrows show the data being written into the cell.</p> | <p><u>Schreibvorgang:</u></p> <ul style="list-style-type: none"> • Datenleitung beschalten • Adressleitung auswählen  <p>The diagram shows two DRAM cells during a write operation. In the first, the address line is 0 and the data line is 1. In the second, the address line is 1 and the data line is 1. A capacitor 'C' is shown in each cell, and a green arrow indicates the data being written to the capacitor.</p> |

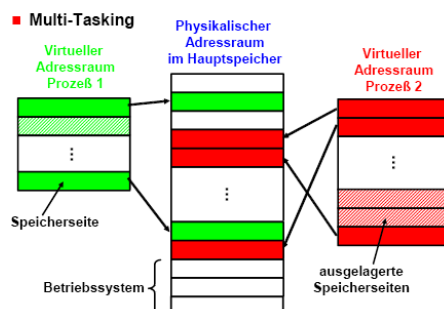
Virtueller Speicher:

Es gibt oft viel mehr adressierbaren Speicher, als physikalisch vorhanden ist.

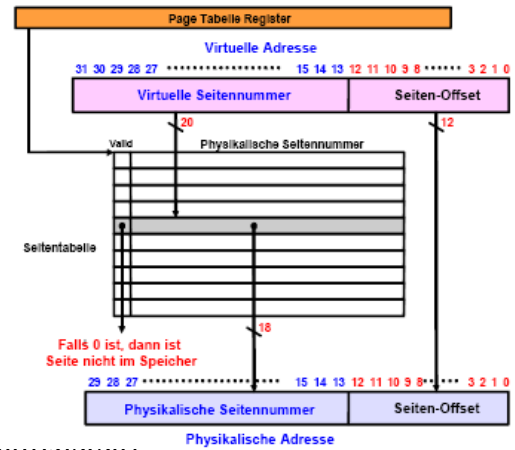
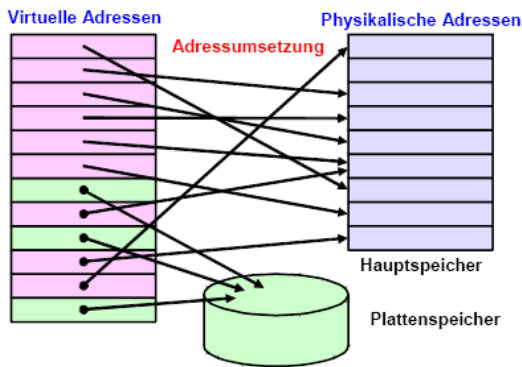
→ Virtueller Speicher

- Nutzt großen Adressraum
- Verfügbarer Adressraum bestimmt Größe des virtuellen Speichers
- Bildet großen Adressraum auf kleinen physikalischen Speicher ab
- Abbildung und Management übernimmt Betriebssystem
- Programme und Daten passen nicht in den Hauptspeicher
- Prozess darf Daten anderer Prozesse nicht verändern (je Prozess eigener Adressraum)
- Teil des virtuellen Speichers liegt im physik. Speicher, der andere auf der Festplatte
- Virtuelle Adressen müssen in physikalische umgesetzt werden
- Virtueller Speicher wird in Pages aufgeteilt (Größen: 512B, 1KB, 4KB, ...)

Multi-Tasking:



Adressumsetzung:



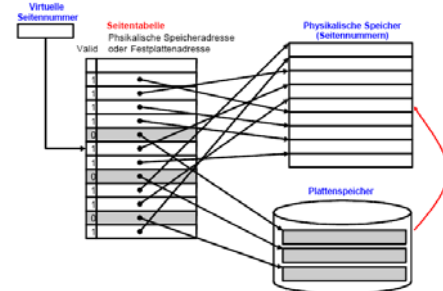
Virtueller Adressraum: 2^{20} Seiten (20 Bit um Seite zu identifizieren)

Physikalischer Adressraum: 2^{18} Seiten (18 Bit um Seite zu identifizieren)

Seite hat $4K = 2^{12}$ Speicherelemente (12 Bit um Speicherelement innerhalb der Seite zu identifizieren, Seiten-Offset)

Seitenfehler (Zugriff auf Seite, die nicht im physikalischen Speicher liegt):

1. Systemaufruf mit Seitenfehler (page fault)
2. Wenig genutzter Seitenrahmen wird ausgelagert und Betriebssystem lädt Seite, die Fehler ausgelöst hat, in den frei gewordenen Seitenrahmen
3. Seitentabelle wird angepasst und Befehl nochmals ausgeführt



Seitenersetzung:

Bei Seitenfehler muss eine Seite aus dem Hauptspeicher entfernt werden, um für nachgeladene Seite Platz zu machen. Dabei zeigt das Valid-Bit an, ob eine Seite im physikalischen Speicher liegt oder nicht. Das Dirty-Bit zeigt an, ob die Seite modifiziert wurde, wenn ja, muss die Seite vor dem überschreiben auf der Festplatte gesichert werden.

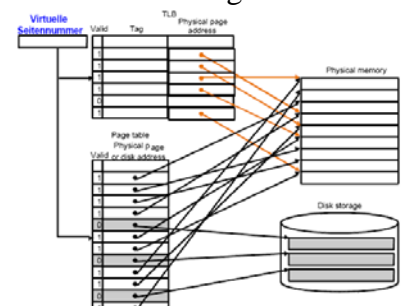
Translation Lookaside Buffer (TLB):

Ermöglicht schnellen Zugriff auf Seitentabelle. Kleiner Cache, der die letzten benutzten Adressabbildungen speichert.

Organisation eines TLB: Tag-Feld (oberer Teil der virtuellen Adresse), Datenfeld (zugehörige aktuelle physikalische Adresse), Verwaltungsbits (valid, reference, dirty)

Speicherzugriff mit TLB:

1. Adressieren des TLB mit der virtuellen Adresse
2. HIT: Verwendung des TLB-Dateneintrags zur Adressierung des Hauptspeichers
- MISS: - TLB-Miss: Seite ist im Speicher; Adressabbildung muss aus Seitentabelle in TLB geladen werden; Bei Ersetzung müssen Verwaltungsbits in die Seitentabelle zurück kopiert werden (write back)
- Seitenfehler: Seite ist nicht im Hauptspeicher; Betriebssystem wird aktiviert, um die Seite zu laden; Dabei neu entstehende Adressbildung wird in den TLB geladen



Performance:

Quantitative Bestimmung der Prozessorleistung (Maß für die Leistungsfähigkeit)

Quantitative Bewertung:

- Benutzer: Antwortzeit (wie schnell wird Aufgabe gelöst)
- Rechenzentrum: Durchsatz (wieviel Arbeit pro Tag erledigt)
- Allgemein: Execution Time (Ausführungszeit für ein Programm) [User Execution Time, CPU Execution Time, System Execution Time]

Performance Maße:

CPU-Performance-Gleichung:
$$Zeit = \underbrace{\frac{\text{Befehle}}{\text{Programm}}}_{\text{Software/Compiler}} \times \underbrace{\frac{\text{Taktzyklen}}{\text{Befehle}}}_{\text{Architektur (ISA)}} \times \underbrace{\frac{\text{Zeit}}{\text{Taktzyklus}}}_{\text{Implementierung}}$$

Übliche Leistungskennzahlen:

MIPS (Millionen Instruktionen pro Sekunde)
MOPS (Millionen Operationen pro Sekunde)
FLOPS (FloatingPoint Operationen pro Sekunde)

Wenig aussagekräftig, da Leistungsfähigkeit der Instruktionen unbekannt.

→ Sinnvolle Leistungsaussagen nur möglich durch Vergleich von identischen Programmen auf verschiedenen Rechnern → Benchmarks (Reale Applikationen, Kernel, Toy-Benchmarks, Synthetische Benchmarks)

Verbesserung der Prozessor-Leistung:

- Erhöhung der Taktrate
- Verbesserung der Prozessororganisation zur CPI Reduzierung
- Compilerverbesserungen, die eine geringere Instruktionszahl oder eine Verwendung von mehr Instruktionen mit geringerer CPI nutzen

Quantifizierung von Verbesserungen:

Amdahls Gesetz:

$$Ausführungszeit_{neu} = Ausführungszeit_{alt} \times \left((1 - Anteil_{verbessert}) + \frac{Anteil_{verbessert}}{Verbesserung} \right)$$
$$Verbesserung = \frac{Ausführungszeit_{alt}}{Ausführungszeit_{neu}} = \frac{1}{(1 - Anteil_{verbessert}) + \frac{Anteil_{verbessert}}{Verbesserung}}$$

Leistungsverbesserung hängt vom Anteil an Ausführungszeit, die ein bestimmter Task mit der verbesserten Hardware verbringt, und der Verbesserungsrate, die durch die verbesserte Hardware erreicht wird, ab.

Aufwand für Verbesserungen lohnt sich nur, wenn verbesserte Strukturen häufig genutzt werden. (Make the common case fast)

Pipelining:

Pipelining verbindet SingleCycle und MultiCycle Datenpfad. → Taktzyklus wird kurz gehalten (→ hohe Taktrate). In jedem Taktzyklus kann ein Befehl abgearbeitet werden. Die Ausführung eines Befehls kann bis zu 5 Zyklen dauern, jedoch nicht alle Befehle benötigen 5 Zyklen.

| Befehl | Erforderliche Zyklen | | | | | Takte |
|-----------|----------------------|----|----|----|----|-------|
| Vergleich | BH | BD | AF | | | 3 |
| Arithm. | BH | BD | AF | ES | | 4 |
| LDW | BH | BD | AF | SP | | 4 |
| STW | BH | BD | AF | SP | ES | 5 |

Der Datenpfad erzeugt einen Aktionsfluss durch die einzelnen Hardware Einheiten → Während eines Taktes ist nur ein Teil der Hardware beschäftigt. → Optimierung! Komponenten sollen gleichzeitig aktiv sein (→ Gleichzeitige Bearbeitung mehrerer Befehle) → Pipelining (Fließbandverarbeitung)

Wiederverwenden von unbenutzten Komponenten (Befehlsspeicher ist frei während Befehlsdekodierung → Lesen des 2. Befehls, während 1. Befehl dekodiert wird, Sobald 1. Befehl ausgeführt wird, kann der 2. Befehl dekodiert und der 3. Befehl geholt werden...

- Maximale Nutzung der Hardware durch überlappende Ausführung mehrerer Befehle
- Jede der 5 Ausführungsphasen verwendet eine andere Funktionseinheit
- Prinzipiell 5 Befehle gleichzeitig (einer in jeder der 5 Phasen)
- Voraussetzung: Alle Befehle dauern in etwa gleich lang

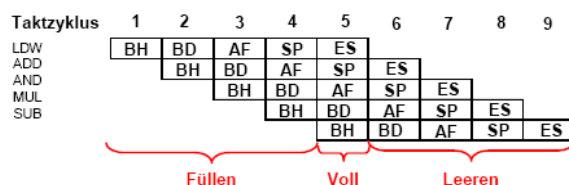
Pipeline-Diagramm:

(Befehlsfolge vertikal von oben nach unten, Taktzyklen horizontal von links nach rechts, Befehle in einzelne Phasen zerlegt)

Pipeline-Tiefe = Zahl der Stufen

Wie lange benötigen die 5 Befehle ?

- 40 ns für Single-Cycle Datenpfad mit einer Zykluszeit von 8 ns
- Mit Pipelining braucht man 9 Zyklen oder 18 ns
- 42 ns mit Multi-Cycle Datenpfad



Die meisten Taktzyklen werden verwendet, um die Pipeline zu füllen bzw. zu leeren.

- Nahezu eine Befehlsausführung je Taktzyklus
- Je mehr Befehle, desto mehr Taktzyklen mit Vollausslastung → Bessere Beschleunigung!
- CPI ist nahezu 1 (SingleCycle) + Jeder Taktzyklus ist sehr kurz (2ns) (MultiCycle)
- **Ideale Beschleunigung gegenüber SingleCycle entspricht der Pipeline-Tiefe!**

Tiefe Pipelines erhöhen Durchsatz, aber es ergeben sich mehr Konflikte!

Pipelining-Paradox:

Pipelining verbessert nicht die Ausführungszeit eines einzelnen Befehls, Pipelining erhöht nur den Durchsatz (d.h die Menge an Befehlen, die pro Zeiteinheit abgearbeitet werden)

→ Schnellere Abarbeitung einer Folge von Befehlen

Anfangsphase: Füllen der Pipeline

Betriebsphase: gleichzeitige Bearbeitung der Befehle

Endphase: Leeren der Pipeline

Bei einer idealen Pipeline (Pipeline bleibt immer gefüllt, wird nicht angehalten; jede Stufe benötigt genau einen Takt) mit k Stufen und Programm mit n Befehlen, die k Takte zur Ausführung benötigen, sind in der Betriebsphase k Befehle gleichzeitig in Bearbeitung. Es werden $k - 1$ Takte zum Auffüllen benötigt, dann wird in jedem Takt ein Befehl fertig.

→ Gesamte Ausführungszeit: $n + k - 1$ Takte

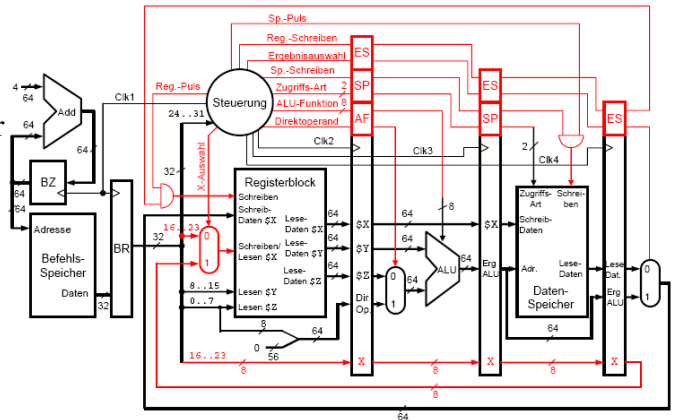
Latenz L : Zeit, die ein Befehl zum Durchlaufen der Pipeline benötigt (ideal: $L = \text{"Stufen"}$)

Durchsatz T : Anzahl Befehle, die pro Takt die Pipeline verlassen. $T = \frac{n}{n+k-1}$

Beschleunigung S : $S = \frac{n \cdot k}{n+k-1} = \frac{k}{1+(k-1)/n}$

Steuerung für Pipeline Datenpfad:

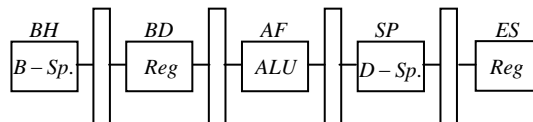
Steuerinformation muss mit dem Datenfluss über die Pipeline-Stufen transportiert werden
 → neue Register



Grenzen und Probleme beim Pipelining:

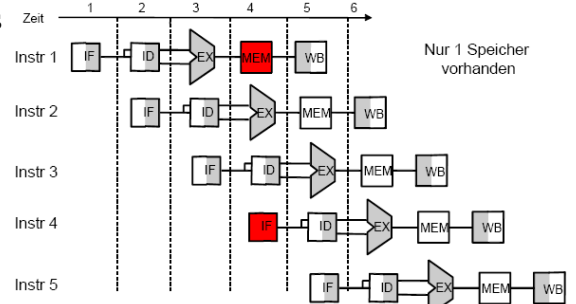
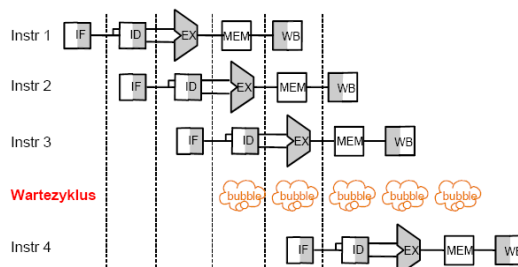
Konflikte (nächster Befehl kann nicht im geplanten Taktzyklus ausgeführt werden) → Unterbrechung des taktsynchronen Durchlaufs eines Befehls durch die einzelnen Stufen der Pipeline → Pipeline nicht mehr vollständig gefüllt → Wartezyklen (eine/mehrere Pipeline-Stufen sind untätig)

Konfliktarten (Hazards):



- **Struktureller Konflikt** (Hardware kann Kombination von Befehlen nicht unterstützen) z.B.: Zwei Speicherzugriffe in einem Taktzyklus

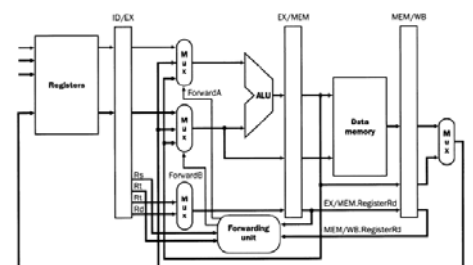
Lösung: Pipeline anhalten → Wartezyklus



- **Datenkonflikt** (Befehl benötigt Ergebnis von vorherigem)

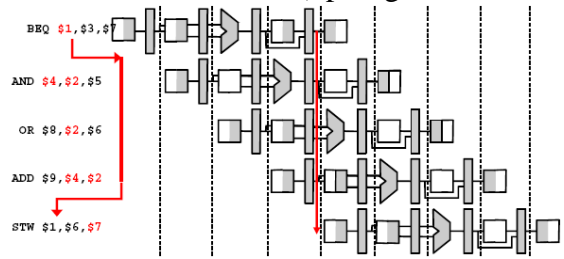
Lösung: - Compiler muss Datenkonflikt erkennen und auflösen (Pipeline anhalten, Wartezyklus)
 - Forwarding (Daten sind bereits in Pipeline-Register verfügbar (Hardwareaufwand, zusätzliche Datenpfade 'Bypass' erlauben Verwendung von ALU-Ergebnissen in AF-Phase Des nächsten, übernächsten Befehls)
 ABER: Bei Lade-Befehl trotzdem Datenkonflikt → Wartezyklus trotz Forwarding

- **Read After Write (RAW)**
 - Befehl₂ versucht Daten zu lesen bevor Befehl₁ sie geschrieben hat
 - echte Datenabhängigkeit
 - Befehl₁: $a = b + c$
 - Befehl₂: $b = a * x$
- **Write After Read (WAR)**
 - Befehl₂ versucht Daten zu schreiben bevor Befehl₁ sie gelesen hat
 - Gegenabhängigkeit
 - Befehl₁: $a = b + c$
 - Befehl₂: $b = a * x$
- **Write After Write (WAW)**
 - Befehl₂ versucht Daten zu schreiben bevor Befehl₁ sie geschrieben hat: Befehl₁ erhält falsche Daten
 - Anti-Abhängigkeit
 - Befehl₁: $a = b + c$
 - Befehl₂: $a = y * x$



Vermeidung von Datenkonflikten in der Software durch Vertauschung von Befehlen → Guter Compiler!

- Steuerungskonflikt (Der in der Pipeline folgende Befehl ist nicht der, der als nächstes ausgeführt werden soll – bei Sprungbefehlen)
Tritt nach Laden eines Sprungs auf, da vor Bestimmung der Zieladressen nicht fest steht von wo die nächsten Befehle geladen werden müssen (Sprungziel erst in ES-Phase bekannt) → 3 Wartezyklen



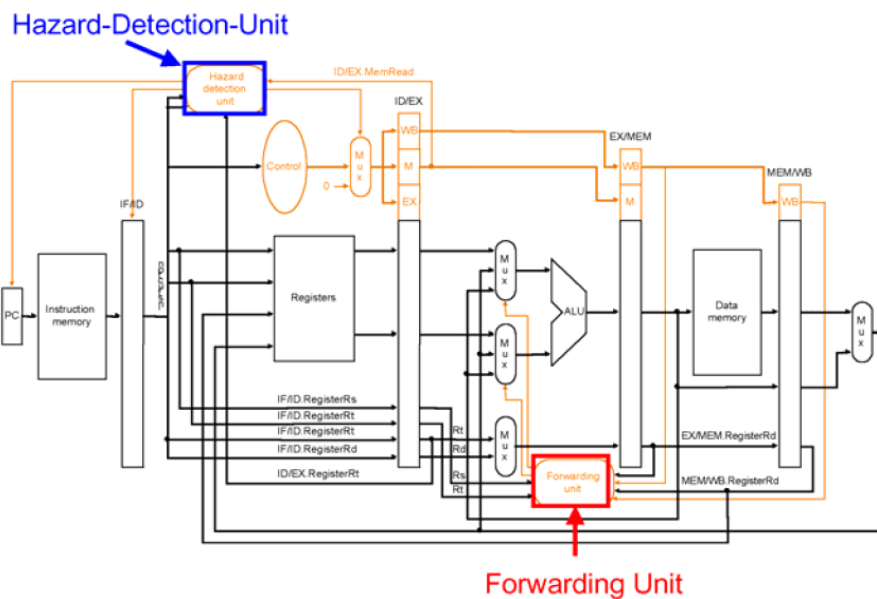
Lösung: - immer Wartezyklen

- Predict Branch Not Taken (Unmittelbar folgende Befehle ausführen, falls doch Sprung, müssen Befehle annulliert werden)
- Predict Branch Taken (Befehle die am Sprungziel stehen ausführen → Während BH-Phase muss Verzweigungsbefehl und Sprungzieladresse bekannt sein)
- Delayed Branch (Verzweigung wird in jedem Fall nach fester Anzahl von weiteren Zyklen ausgeführt; in den Zwischenzyklen werden von der Verzweigung unabhängige Befehle eingebracht; meist 1 DelaySlot)

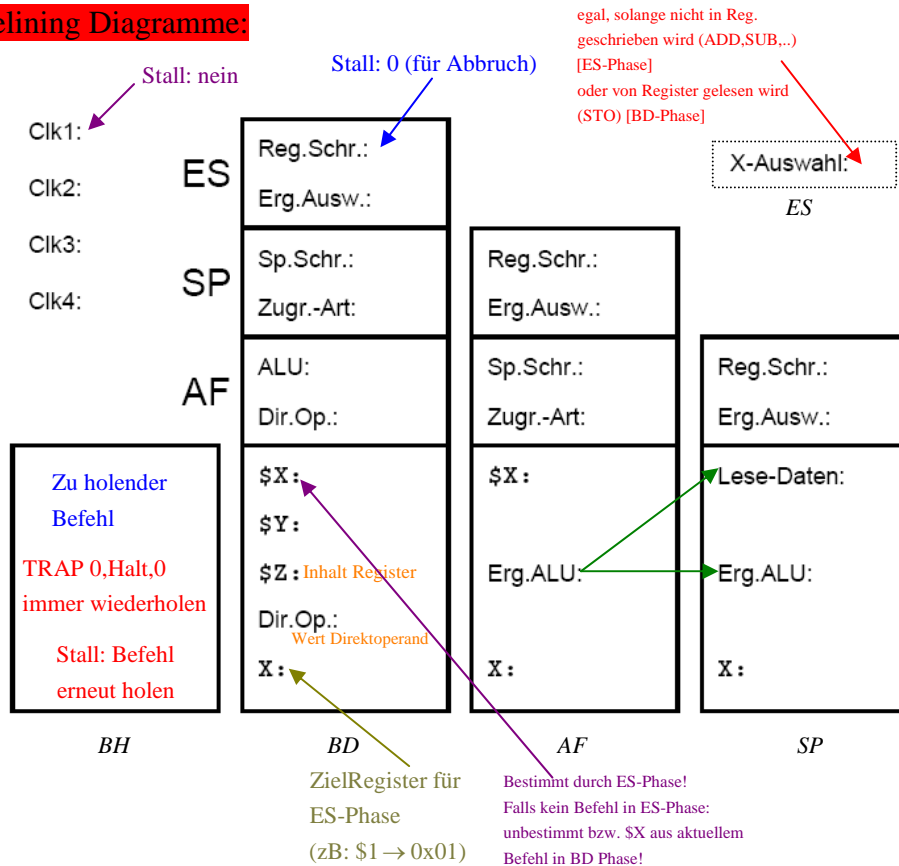
Software Maßnahmen zur Vermeidung von Konflikten:

- Code Movement: Umordnen von Befehlen, damit datenabhängige Befehle möglichst weit auseinander stehen
- Loop Unrolling: Konflikte treten häufig bei Schleifen auf → Schleifencode mehrmals hintereinander kopieren (→ Code Movement)

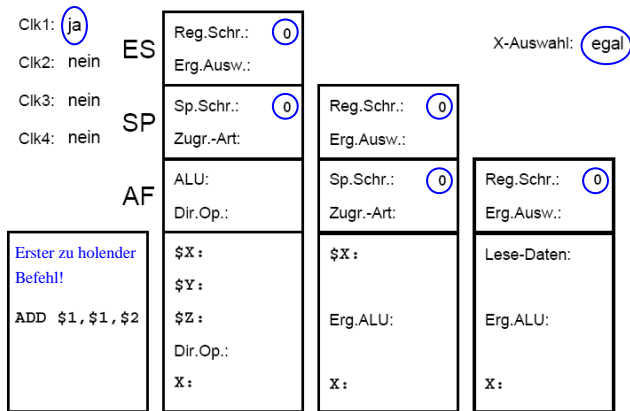
Datenpfad mit Hazard-Detection-Unit:



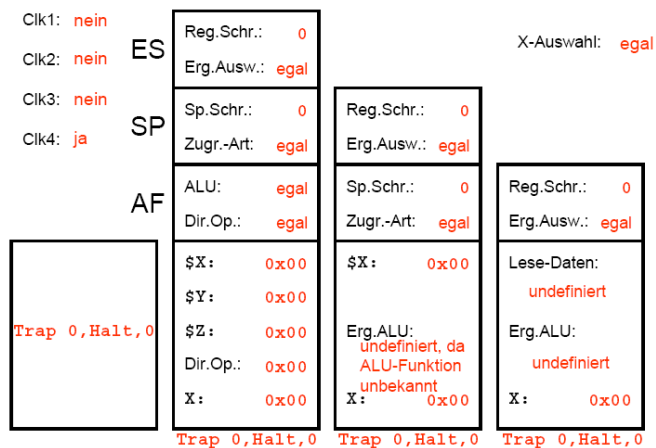
Pipelining Diagramme:



Zustand bei Holen des ersten Befehls:



Zustand am Ende des Programms:



Cache:

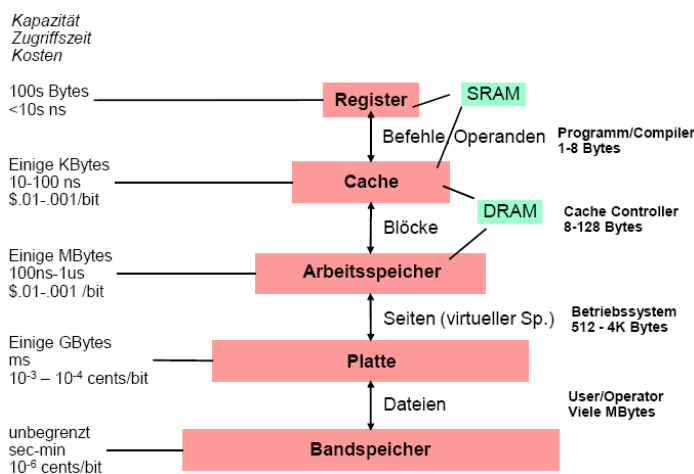
Grundprinzip-Lokalität:

Zeitliche Lokalität: Wenn ein Datum referenziert wird, dann wird es wahrscheinlich bald wieder referenziert

Räumliche Lokalität: Wenn ein Datum referenziert wird, dann werden Daten deren Adressen in der Nähe liegen wahrscheinlich auch bald referenziert

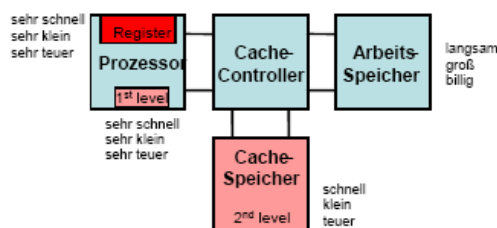
Speicherhierarchie: Nutzt zeitliche und räumliche Lokalität; es müssen nicht immer alle Daten zu allen Zeiten verfügbar sein

Speicherhierarchie:

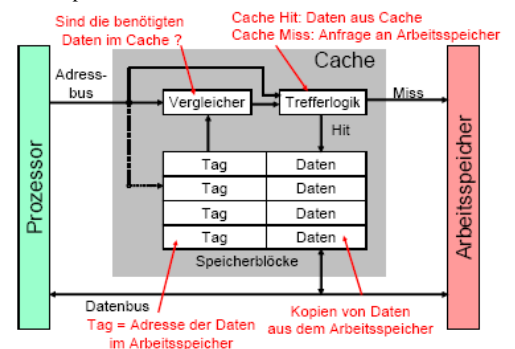


Cache-Speicher:

Kleiner und schneller Speicher; sitzt zwischen Prozessor und Hauptspeicher; enthält Teile der gerade vom Prozessor bearbeiteten Daten und Instruktionen.



Prinzipieller Aufbau:



Ein Speicherzugriff führt auf Suche nach Daten im Cache:

- **Cache-Hit:** gesuchte Daten werden gefunden
Hit Rate h : Wahrscheinlichkeit für Cache-Hit
- **Cache-Miss:** gesuchte Daten werden nicht gefunden
→ Daten müssen aus Arbeitsspeicher in Cache geladen werden
Miss Rate $1-h$: Wahrscheinlichkeit für Cache-Miss

Direkt abgebildeter Cache:

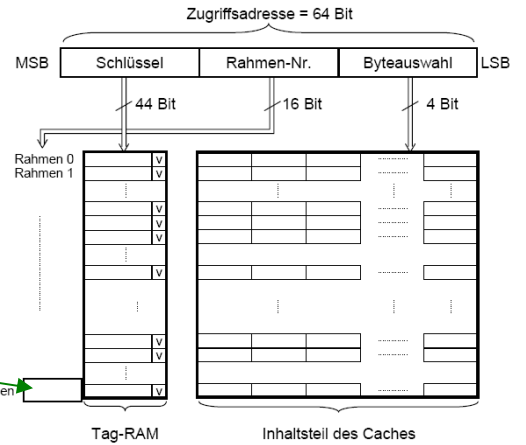
Jeder Adresse im Arbeitsspeicher ist eine feste Adresse im Cache zugeordnet.

Schlüssel(Tag)-Feld:

Jeder Cache-Block kann von mehreren Stellen des Arbeitsspeichers stammen; Daten selbst geben keine Auskunft über Herkunft → Oberer Teil der Arbeitsspeicheradresse wird als Schlüssel bzw. Tag verwendet

$GrößeCacheSpeicher = AnzahlRahmen \cdot AnzahlByteproRahmen$

$AnzahlRahmen = 2^{RahmenNr.}$ $AnzahlRahmen - 1$



Ist das erforderliche Datum nicht im Cache, so muss es aus dem Hauptspeicher geholt werden → Steuerung muss Prozessor anhalten bis gesuchte Daten im Cache vorhanden sind. Nachladen dauert ein Vielfaches des Cache-Zugriffs (Nachladezeit (Miss-Penalty) = Zugriffszeit des Arbeitsspeichers t_{ASP})

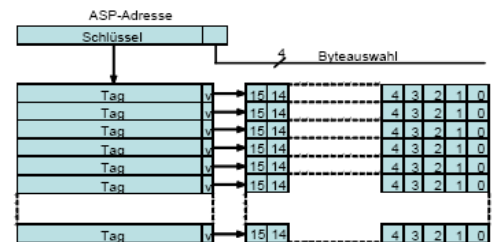
Gesuchte Daten brauchen freien Platz im Cache → Bereits vorhandene Daten müssen überschrieben werden (evtl. wieder nachladen) → Assoziativer Cache (keine feste Zuordnung der Speicherstellen und Cache-Blöcke)

Leistungsgewinn durch Cache: $t_{eff} = h \times t_{Cache} + (1-h) \times t_{ASP}$

- ♦ t_{eff} : effektive Speicherzugriffszeit
- ♦ t_{Cache} : Zugriffszeit des Cache-Speichers
- ♦ t_{ASP} : Zugriffszeit des Arbeitsspeichers

Voll-assoziativer Cache:

Speicherblock kann in beliebigem Cache-Block abgelegt werden → Kein CacheIndex → Schlüssel mit allen CacheTags vergleichen (Conflict-Miss = 0)

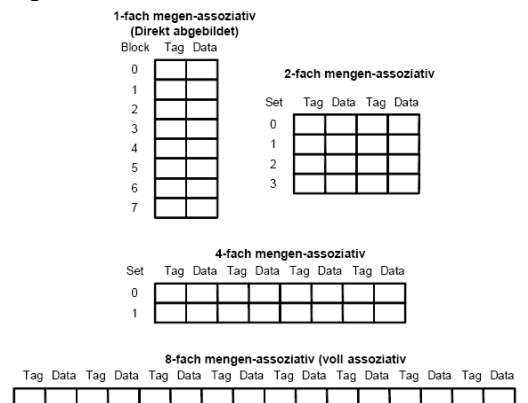


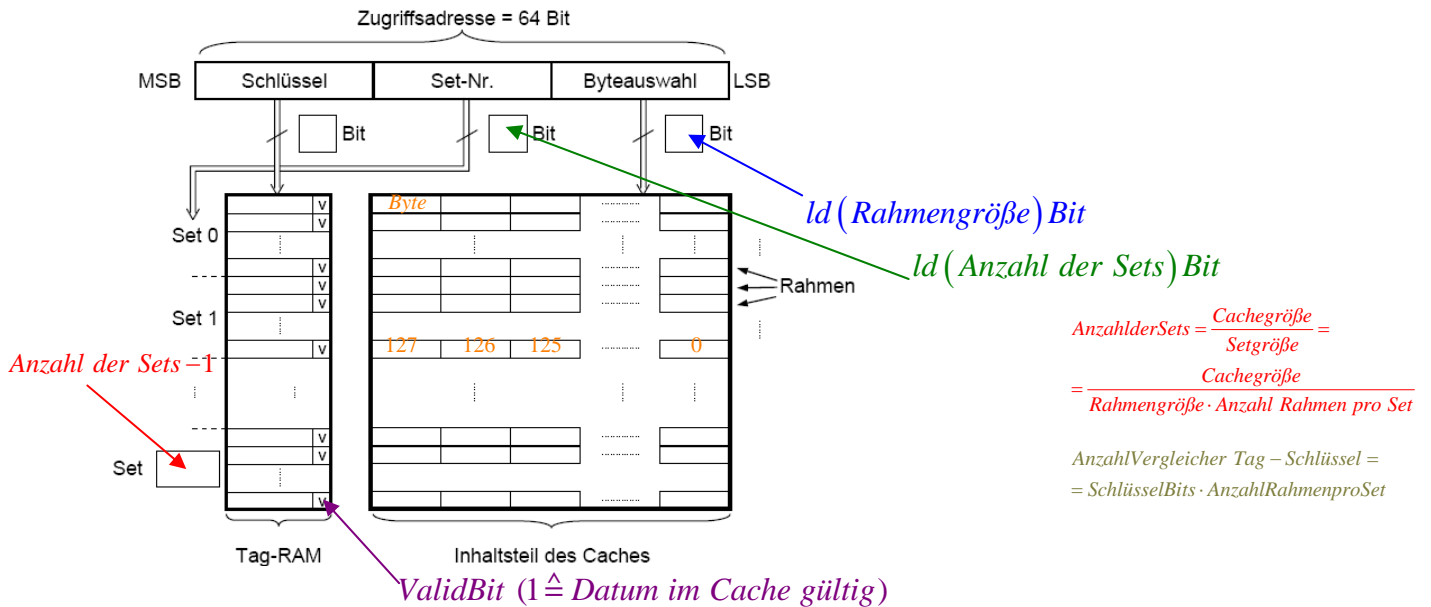
Mengen-assoziativer Cache:

Feste Menge von CacheBlock Partitionen; Speicherblock wird fest einer Partition zugewiesen; innerhalb einer Partition gilt das assoziative Prinzip

→ N-fache Mengen-Assoziativität:

- N Einträge für jeden CacheIndex
- N direkt abgebildete Caches arbeiten parallel





Im Mengen-assoziativen Cache werden N Vergleichsoperationen benötigt (Direkt abgebildeter Cache nur eine Vergleichsoperation). Zusätzlich tritt eine Multiplexerverzögerung für die Daten auf und die Daten kommen erst nach einer Hit/Miss Entscheidung (in direkt abgebildetem Cache ist CacheBlock schon vor Hit/Miss Entscheidung verfügbar, falls Miss vorliegt wird Problem später behoben). Beim direkt-abgebildeten Cache werden im Cache befindliche Speicherzellen schneller verdrängt

Cache Schreiben:

Write-Through: Neuer Wert wird sofort in Arbeitsspeicher kopiert; zusätzlicher Schreibpuffer um zusätzliche Wartezyklen zu vermeiden

Write-Back: Neuer Wert wird erst dann in den Arbeitsspeicher kopiert, wenn Cache-Block verdrängt wird (Miss); → keine Schreibpuffer notwendig, aber komplexere Steuerung

Cache-Performance:

$$\text{CPU-Zeit} = (\text{CPU-Taktzyklen} + \text{Speicherlade-Zyklen}) \times \text{Taktzykluszeit}$$

$$\text{Speicherlade-Zyklen} = \text{Speicherzugriffe} \times \text{Miss-Rate} \times \text{Miss-Penalty}$$

Durchschnittliche Speicherzugriffszeit (AMAT)

$$\text{AMAT} = \text{Hit-Zeit} + \text{Miss-Rate} \times \text{Miss-Penalty}$$

Verbesserung der Performance:

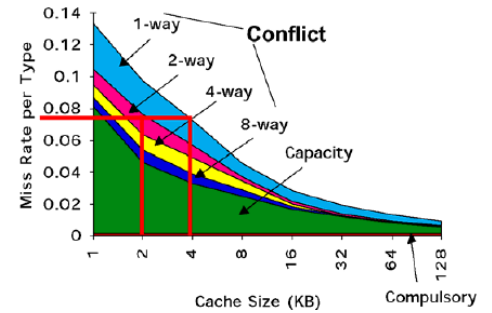
Reduzierung der Miss-Rate:

$$\text{CPUZeit} = \text{IC} \times \left(\text{CPI} + \frac{\text{Speicherzugriffe}}{\text{Instruktionen}} \times \text{MissRate} \times \text{MissPenalty} \right) \times \text{Taktzykluszeit}$$

- Compulsory:** Erster Zugriff auf einen Block trifft nicht den Cache, Block muss erstmals geladen werden (Kaltstart-Miss, First-Reference-Miss)
- Capacity:** Cache hat nicht genug Platz, um alle Blöcke des aktuellen WorkingSet zu enthalten → Entfernen und späteres Zurückladen (Kapazitäts-Miss) [vor allem voll-assoziative Caches]
- Conflict:** In nicht voll-assoziativen Caches werden Blöcke auf Grund von Adresskonflikten überschrieben (Collisions-Miss, Interferenz-Miss) [mengen-assoziative, direct-mapped Caches]

Erhöhung der Assoziativität (2:1 Cache Regel):

**Miss-Rate eines direkt abgebildeten Cache der Größe n
≈ Miss-Rate eines 2-fach mengen-assoziativen Cache der
Größe n/2**



Prefetching von Daten & Instruktionen:

Nicht nur aktuelle Speicherzelle aus dem Arbeitsspeicher in den Cache laden, sondern auch mehrere unmittelbar folgende Speicherzellen. → Bei Zugriff auf aufeinanderfolgende Adressen liegen Daten bereits im Cache (typisch für Befehlsspeicher)

Optimierung durch den Compiler:

Instruktionen: Umordnung im Speicher, um Conflict-Misses zu reduzieren

Profiling (Konfliktvermeidungsmaßnahme unter Nutzung von Informationen über die Ablaufcharakteristik)

Daten: Array-Merging (Verbesserung der räumlichen Lokalität durch Verbundarray)

Schleifenvertauschung (Datenzugriff in Speicherordnung erzwingen)

Schleifenfusion (Zusammenfassen von Schleifen mit überlappenden Variablen)

Blocking (Zusammenfassen von Daten zu kompakten Blöcken)

Miss-Penalty:

Bei Cache-Misses kann Write Through mit Write Buffern zu RAW-Konflikten beim Lesezugriff auf den Hauptspeicher führen

Ablauf bei Write Back: Block aus Cache in Write Buffer kopieren; neuen Block aus Speicher lesen; anschließend Schreiben aus Write Buffer

Reduzierung der Miss-Penalty:
$$\text{CPUZeit} = \text{IC} \times \left(\text{CPI} + \frac{\text{Speicherzugriffe}}{\text{Instruktionen}} \times \text{MissRate} \times \text{MissPenalty} \right) \times \text{Taktzykluszeit}$$

Subblock-Platzierung:

Bei Miss keine ganzen Blöcke laden; Block in Subblöcke mit eigenen Valid-Bits aufteilen

→ Bei Miss müssen nur Subblöcke ersetzt werden (bei geringer Speicherbandbreite weniger Zeit)

Early Restart/Critical Word First:

CPU setzt Arbeit fort, bevor Lesevorgang vollständig abgeschlossen ist.

Early Restart: Sobald das angeforderte Wort des zu ladenden Blockes ankommt, wird es an die CPU weitergeleitet → CPU kann weiterarbeiten

Critical Word First: Anforderung des fehlenden Wortes zuerst und Senden an die CPU, sobald es verfügbar ist; CPU setzt Arbeit fort, während die restlichen Werte des Blockes geladen werden

Non-blocking Cache:

Erlaubt weiteren lesenden Zugriff auf den Cache während einer Miss-Behandlung

- hit under miss: reduziert effektive Miss-Penalty durch Fortsetzung der CPU Arbeit während eines Miss
- miss under miss: Unterstützung überlappender Misses (Erhebliche Erhöhung der Komplexität des CacheControllers, mehrere Speicherbänke nötig, da sonst keine parallelen Speicherzugriffe möglich sind)

2.Cache Ebene:

$$\text{AMAT} = \text{Hit-Zeit}_{L_1} + \text{Miss-Rate}_{L_1} \times \text{Miss-Penalty}_{L_1}$$

$$\text{Miss Penalty}_{L_1} = \text{Hit-Zeit}_{L_2} + \text{Miss-Rate}_{L_2} \times \text{Miss-Penalty}_{L_2}$$

$$\text{AMAT} = \text{Hit-Zeit}_{L_1} + \text{Miss Rate}_{L_1} \times (\text{Hit-Zeit}_{L_2} + \text{Miss-Rate}_{L_2} + \text{Miss Penalty}_{L_2})$$

Lokale Miss-Rate— Misses in diesem Cache / Gesamtzahl Speicherzugriffe **auf diesen Cache**
(Miss-Rate_{L_1})

Globale Miss-Rate—Misses in diesem Cache / Gesamtzahl Speicherzugriffe **durch die CPU**
($\text{Miss Rate}_{L_1} \times \text{Miss Rate}_{L_2}$)

Die **globale Miss-Rate** ist der interessante Faktor.