Heute: Kleiner Ausflug in die Rechnerarchitektur

Kap. 8: Ca..

Cage ?

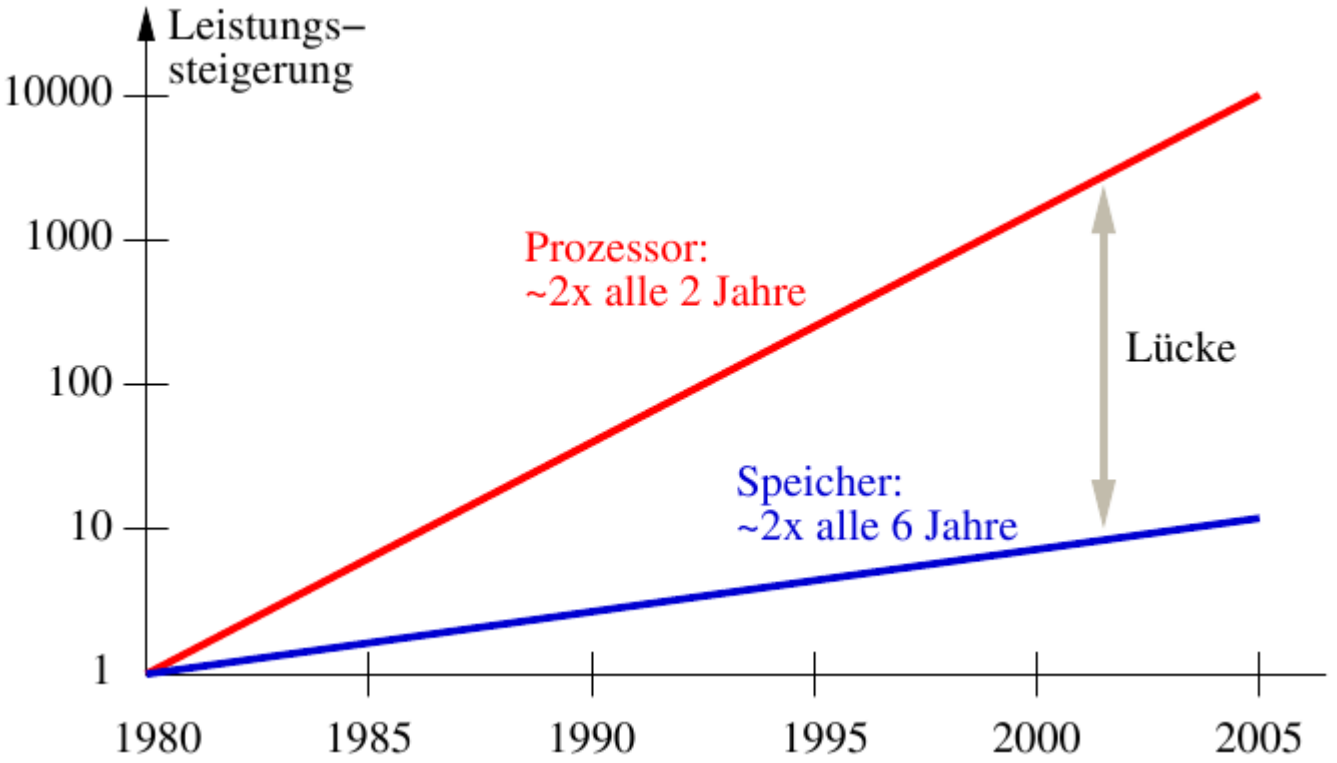
Cash ?

Cache !

Wozu Caches? (1)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Prozessorgeschwindigkeit hat sich in den letzten 30 Jahren mehr als vertausendfacht
- Speicherzugriffsgeschwindigkeit ist zwar auch gestiegen, aber deutlich weniger:



Wozu Caches? (2)

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Prozessortakt > 10 x Speichertakt ist heute keine Seltenheit → Speicher wird zum „Flaschenhals“
- Skalare Prozessoren: 1 Befehl pro Takt (*Super*skalare: noch mehr …)
- Wie soll das gehen?
- Ein Maschinenbefehl besteht mindestens aus
  - einem Speicherzugriff (Opcode fetch)
  - + evtl. auch noch mehrere Operanden
- Lösung: schneller, dafür kleiner Zwischenspeicher zw. Prozessor und Hauptspeicher: Cache(1)

(1) Engl. *Cache*: Depot, geheimes Versteck

Motivation

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Cache gehört zur Mikroarchitektur, nicht zur ISA, ist also aus Programmierersicht transparent
- Warum sich hier damit beschäftigen?
- Gründe:
  - Erzielte/erzielbare Performance hängt von Lokalitätseigenschaften (s.u.) der Programme ab
    - Durch geschicktes/ungeschicktes Coden kann die Performance u.U. um Größenordnungen variieren
  - Im Bereich der Betriebssysteme sind Caches manchmal doch nicht so ganz transparent
    - DMA, Debuggen, Laden von Programmen
- → Grundlegende Kenntnisse über die Arbeitsweise von Caches sind erforderlich

Typisches Programmverhalten (Heuristik)

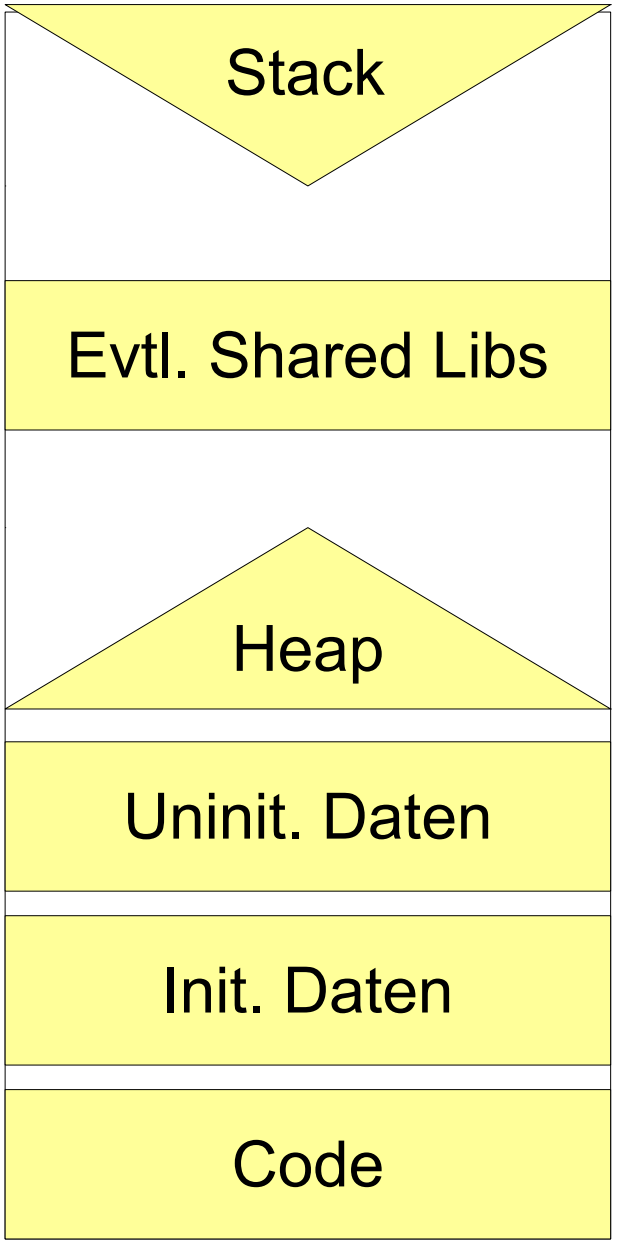
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- **Räumliche Lokalität:** Zugriffe häufig auf Adresse in der Nähe bereits zuvor benutzer Adressen
- **Zeitliche Lokalität:** Zugriffe auf dieselbe oder benachbarte Adressen zeitlich nahe beieinander
- Trifft zu auf:
  - Befehlszugriffe: meistens (ausser bei Sprüngen)
  - Datenzugriffe: Programmabhängig
- Caches nutzen diese Eigenschaften:
  - Nachgefragte Daten und der sie umgebende Speicherblock (Cache-Zeile) werden möglichst lange zwischengespeichert
  - Wird das gleiche (oder ein benachbartes Datum) erneut nachgefragt, wird es aus dem Zwischenspeicher geliefert
- → geht wesentlich schneller

Begriff: Working Set

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- ***Working set***: Gesamtheit der Speicher-objekte, auf die ein Prozess während eines gegebenen Zeitintervalls zugreift
- **Ziel**: möglichst viel vom Working Set eines Prozesses im Cache halten
- Programme arbeiten i.d. R. auf verschiedenen „Sektionen“:
  - Programmcode (`.text`)
  - Initialisierte Daten (`.data`)
  - Uninitialisierte Daten (`.bss`)
  - Heap (z.B. von `malloc()`)
  - Stack
  - Evtl. shared Libraries
- ➔ Working Set besteht aus >5-6 „Regionen“



Begriffe: Hit und Miss

- Programm greift auf ein Objekt (z.B. Variable) zu
- Zwei Möglichkeiten:

1. ***Cache Hit***: Daten sind bereits im Cache → schnell
2. ***Cache Miss***: Daten sind nicht im Cache, Hauptspeicherzugriff erforderlich → langsam

Begriffe: Konsistenz und Kohärenz

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Der Cache hält **Kopien** von im Speicher liegenden Objekten → Gefahr von Inkonsistenzen
- Cache ist **konsistent** → Alle Cache-Kopien und Original im Hauptspeicher haben gleichen Wert
- Cache ist **kohärent** → Cache und Hauptspeicher liefern für das gleiche Objekt den gleichen Wert
- Kohärenz betrifft das korrekte Arbeiten von Programmen
  - Inkohärenz führt zu Fehlern → vermeiden!
  - (Vorübergehende) Inkonsistenz ist tolerabel, solange sie nicht zu Inkohärenz führt
- Es gibt verschiedene Verfahrensweisen um Kohärenz sicherzustellen (sog. Kohärenzprotokolle)
  → hier nur Grundlagen behandelt

Kohärenzprotokoll: Fallunterscheidung

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

A) Bei **Lese**zugriff:

- ***Cache Hit*** → Daten aus dem Cache liefern, keine weiteren Aktionen
- ***Cache Miss*** → Daten werden aus dem Speicher geliefert, dabei in den Cache kopiert

B) Bei **Schreib**zugriff:

- ***Cache Hit*** → zwei Möglichkeiten:
  1) ***Write Trough***: Geschriebene Daten werden zugleich im Cache und im Speicher gespeichert (einfach aber langsam)
  2) ***Copy-Back***: Datum wird zunächst nur im Cache gespeichert, zugehörige Cache-Zeile als „dirty“ markiert
     → System wird vorübergehend inkonsistent
- ***Cache Miss*** → auch zwei Möglichkeiten:
  1) ***No Write Allocate***: Datum wird nur in Speicher geschrieben
  2) ***Write Allocate***: Datum wird in Speicher und Cache geschrieben (zusammen mit umliegender Cachezeile)

Cache-Effektivität (statistisch)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Effektive Wartezeit T_{eff} hängt ab von:
  - Wartezeit bei Zugriff auf Hauptspeicher (Cache miss): T_{miss}
  - Wartezeit bei Zugriff auf Cache (Cache hit): T_{hit}
  - Wahrscheinlichkeit eines Cache-Hit (*Trefferrate*): H

$$T_{eff} = H * T_{hit} + (1 - H) * T_{miss}$$

- Beispiel: H = 80 %, T_{hit} = 0, T_{miss} = 20ns:
  - => T_{eff} = 0,8 * 0 + 0,2 * 20ns = **4ns**

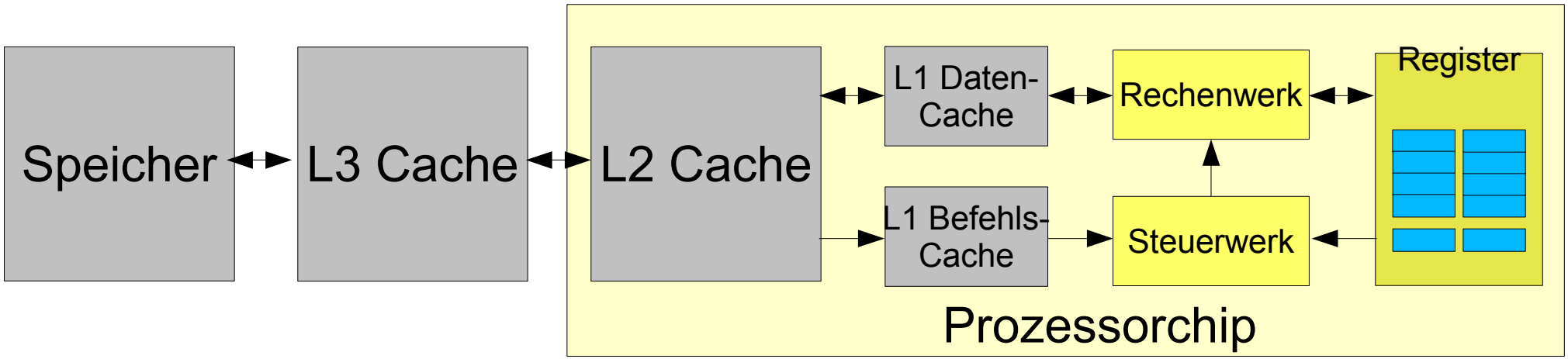
Mehrstufiger Cache (1)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- z.B. Zweistufiger Cache (Level 1 und Level 2):

$$T_{eff} = H_{L1} * T_{L1hit} + (1 - H_{L1}) * (H_{L2} * T_{L2hit} + (1 - H_{L2}) * T_{miss})$$

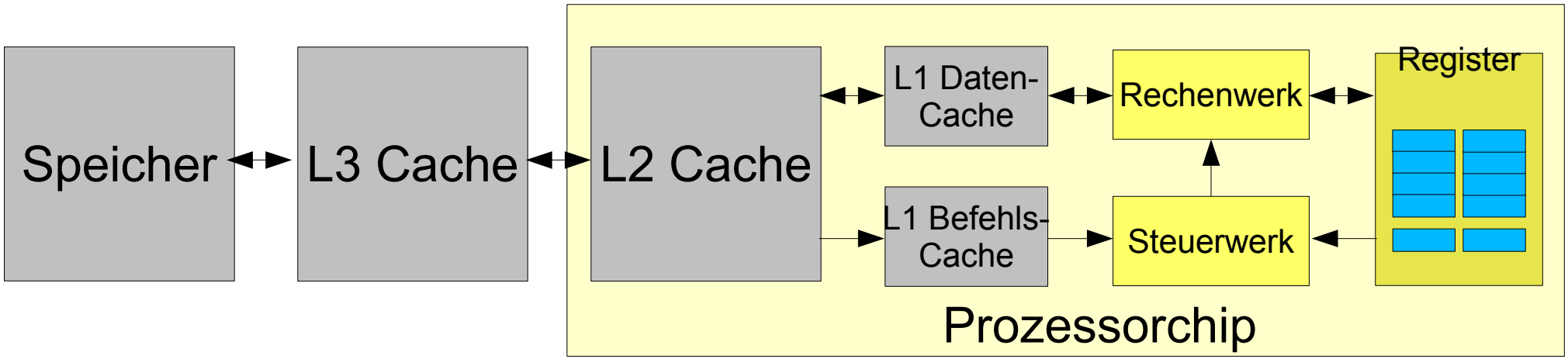
- Beispiel: H_{L2} = 90 %, T_{L2hit} = 4ns, T_{miss} = 20ns:
  => T_{eff} = 0,8 * 0 + 0,2 * (0,9 * 4ns + 0,1 * 20ns) = **1,12ns**
- Caches heute meist mehrstufig:



Mehrstufiger Cache (2)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

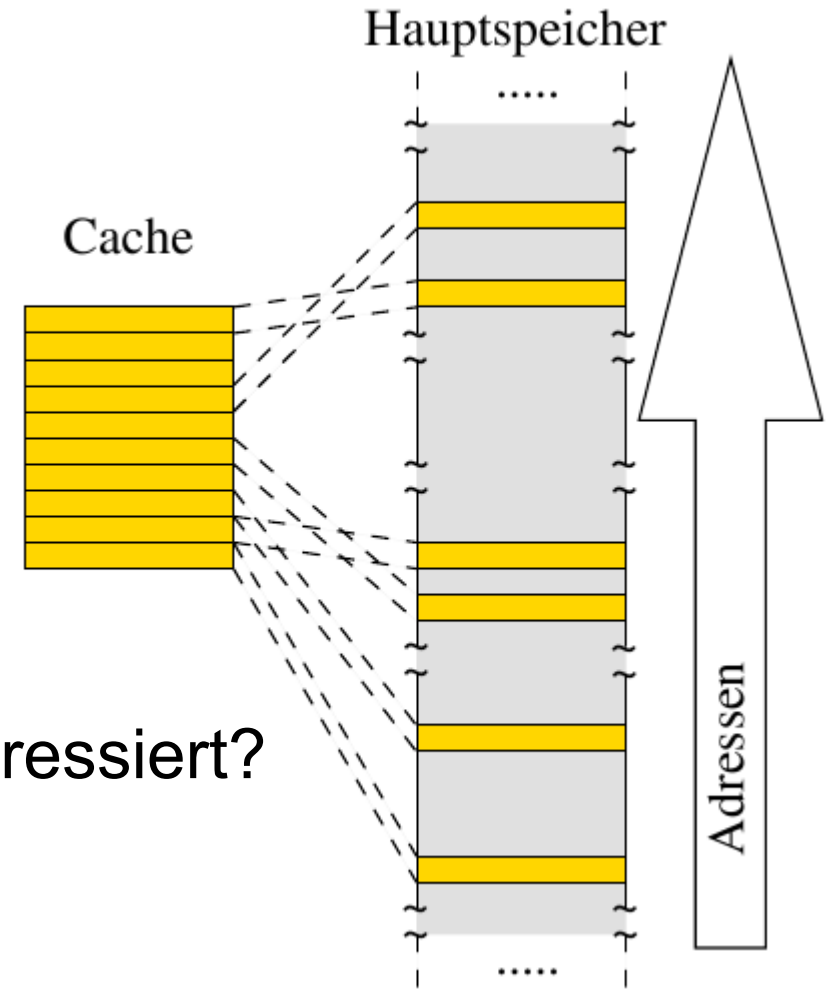
- Heute bei PCs gängige Größen
  - L1 Cache: On-Chip, getrennt für Befehle und Daten, je 16-64kB
  - L2 Cache: On-Chip, gemeinsam für Befehle und Daten, 256kB bis 4MB
  - L3 Cache: Optional, Off-Chip
    (z.B. schnelles ECL-RAM mit 5ns Zugriffszeit)



Assoziativspeicher (1)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Bisher betrachtete Speicher ordnen Adressen → Daten zu
- Cache: hält Kopien relativ kleiner(1) Hauptspeicher-Ausschnitte
- Zuordnung zwischen Speicher-objekt und Cache-Zeile ergibt sich aus der Zugriffsreihenfolge
- **Kein** Zusammenhang zwischen Adresse eines Objekts und dem Ort seiner Cache-Kopie
- Adresse (bzw. Index) einer Cachezeile ist ohne Bedeutung
- Wie aber werden Cache-Zeilen adressiert?



(1) Cache-Zeilen Größe: einige zehn bis hundert Byte

Assoziativspeicher (2)

- **Assoziativspeicher** (auch „inhaltsadressierter Speicher") (engl. *Content Adressable Memory* – CAM)
- **Assoziation von Wertepaaren**
  (hier: Speicheradresse und Cache-Zeile)
  (Auch der Mensch „speichert" assoziativ)
- Ein Cache-Eintrag besteht aus:
  - Einem Identifikationsteil, dem *Tag* (Etikett)
  - Einem Datenbereich (Cache-Zeile),
    der die eigentliche Kopie des Speicherinhaltes enthält
  - Einem Verwaltungsteil: z.B. Bits „V" (Valid) und „D" (Dirty)
    - V=1 → Cache-Zeile ist in Benutzung
    - D=1 → Inhalt der Cache-Zeile wurde verändert → ist inkonsistent

Assoziativspeicher (3)

- Arbeitsweise:
  - Bei Zugriff auf ein Speicherobjekt: Adresse wird **gleichzeitig** mit den Tags **aller** Cache-Einträge verglichen
  - Existiert ein Eintrag mit übereinstimmendem Tag und V = 1 → *Cache Hit*: Gültige Kopie ist im zugehörigen Datenbereich
  - Existiert kein solcher Eintrag → *Cache Miss*

Assoziativspeicher (4)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Modellierung in C:

```
struct cache_eintrag {
  void *tag;
  int V, D;
  unsigned int cache_zeile[C
};

struct cache_eintrag cache[ANZ_CA

unsigned int *finde_eintrag(void *A    sse)
{
  for(i = 0; i < ANZ_CACHEZL; i++) {
    if(cache[i].tag == Adresse && cache[i].V)
      / * cache hit * /
      return(&cache[i].cache_zeile[0]);
  }
  return(NULL); / * cache miss * /
}
```

In Wirklichkeit: **parallele** Suche in Hardware

Verdrängungsstrategien

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Wenn alle Cachezeilen belegt sind und eine Kopie eines Objekts in einer Zeile neu angelegt werden soll, so muss eine bestehende Cachezeile überschrieben („verdrängt“) werden
- Nach welcher Strategie erfolgt diese Auswahl?
- Hier besprochene Möglichkeiten:
  - **Random**: Ersetzung nach dem Zufallsprinzip
    - Einfach, überraschend gutes Ergebnis
  - **FIFO** (first in first out): Die am längsten im Cache befindliche Zeile wird ersetzt
    - Immer noch einfach, aber keine gute Idee
  - **LRU** (least recently used): Die am längsten nicht mehr verwendete Zeile wird ersetzt
    - Gute Ergebnisse, aber aufwändig
  - **LFU** (least frequently used): Die am wenigsten häufig verwendete Zeile wird ersetzt
    - Aufwand und Ergebnis ähnlich LRU (d.h. gut aber aufwändig)

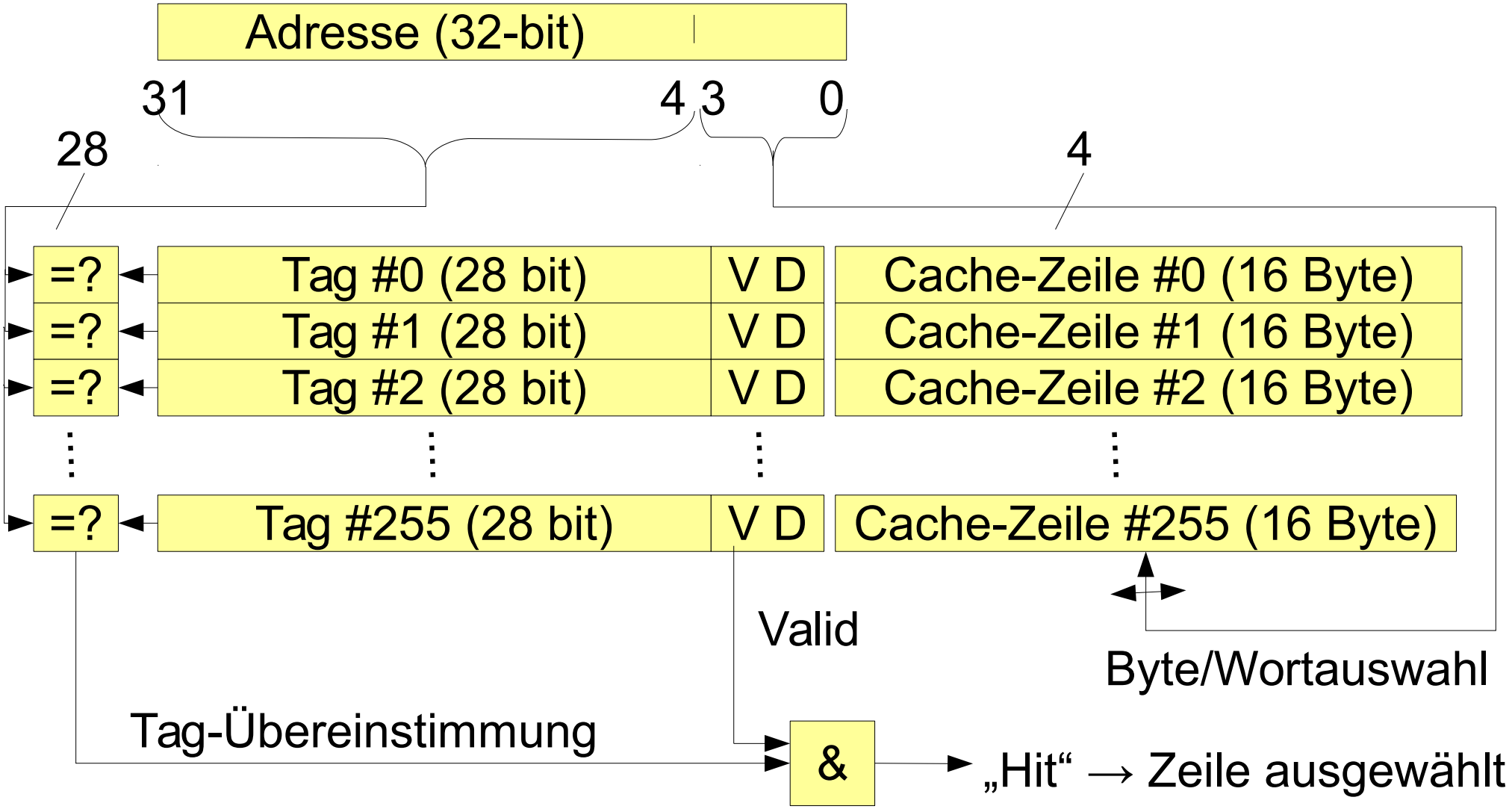
Cache-Organisationsformen

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Cache-Organisationsformen:
  1. Vollassoziativ: *fully associative*
  2. Direkt abbildend: *direct-mapped*
  3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
  - 32-bit Adressierung
  - 4K (4096) Byte Cache
  - Cache-Zeilengröße: 16 Byte
    (→ 4096 : 16 = 256 Cache-Einträge)

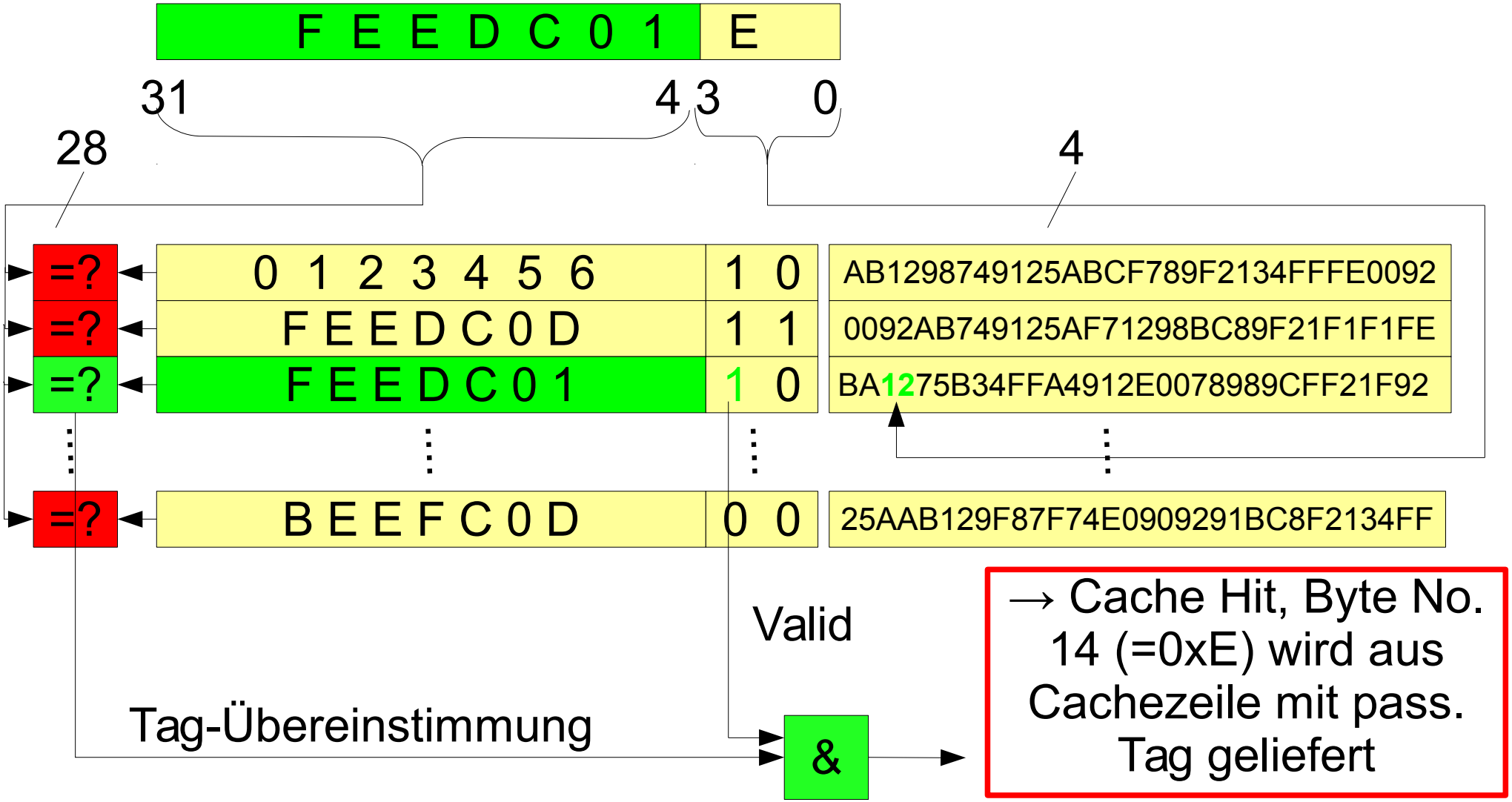
Vollassoziativer Cache: Aufbau

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Vollassoziativer Cache: Beispiel

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



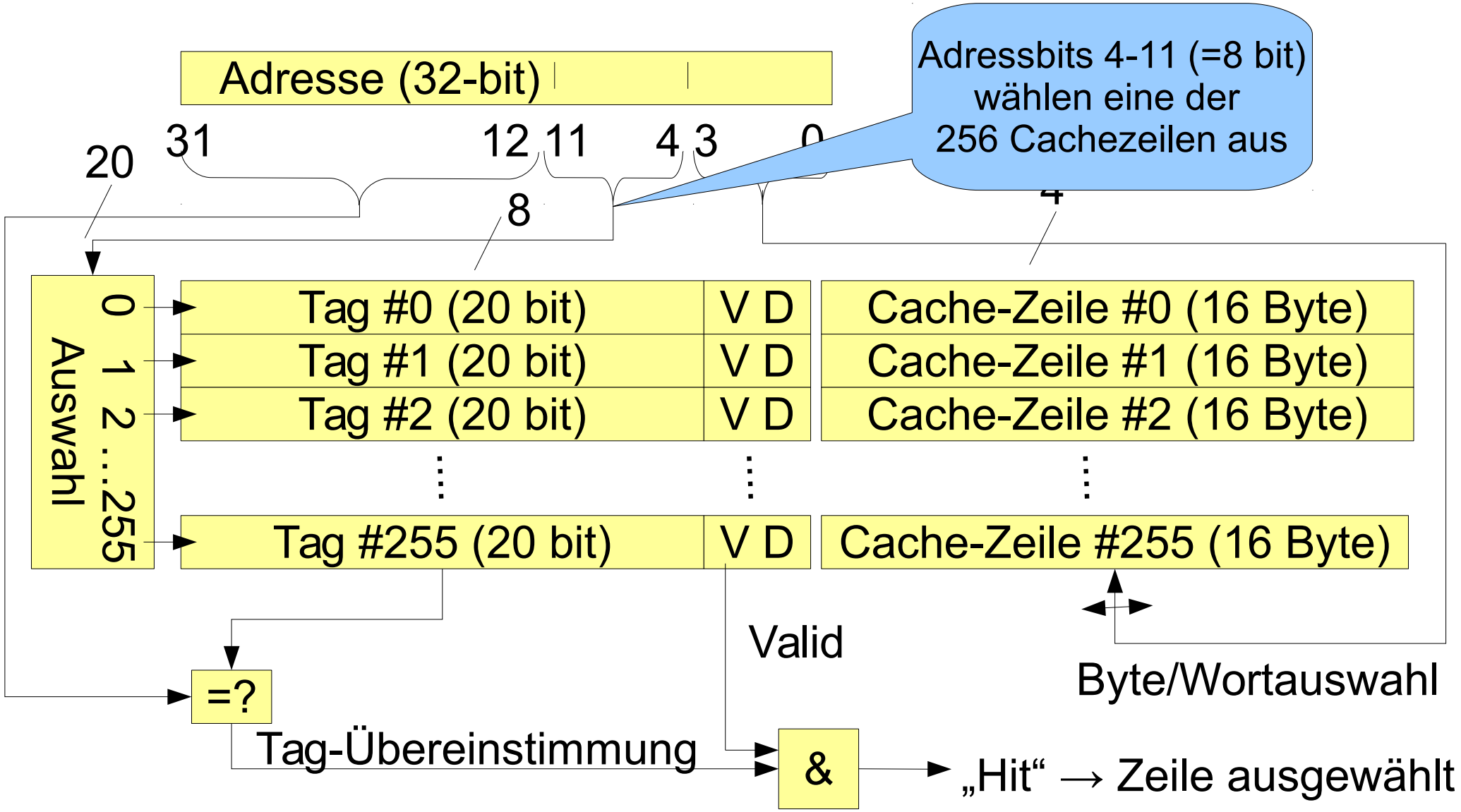
Vollassoziativer Cache

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
  → Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
  → Konsequenzen:
  - Tags **aller** Zeilen müssen mit Adresse verglichen werden
  - Vergleich muss **gleichzeitig** auf allen Zeilen erfolgen
  - Für jede Zeile wird ein eigener Vergleicher benötigt
  - Jedes Tag darf maximal ein Mal vorkommen
- Erreicht höchste Trefferquote (wg. Eintrags-Wahlfreiheit)
- Große Anzahl an Vergleichern (Im Beispiel: 256 für einen 4K Cache) → sehr hoher Hardwareaufwand
- Beispiel:
  - TLB-Cache des MIPS R3000 / R4000: Vollassoziativer Cache mit 64 / 128 Einträgen

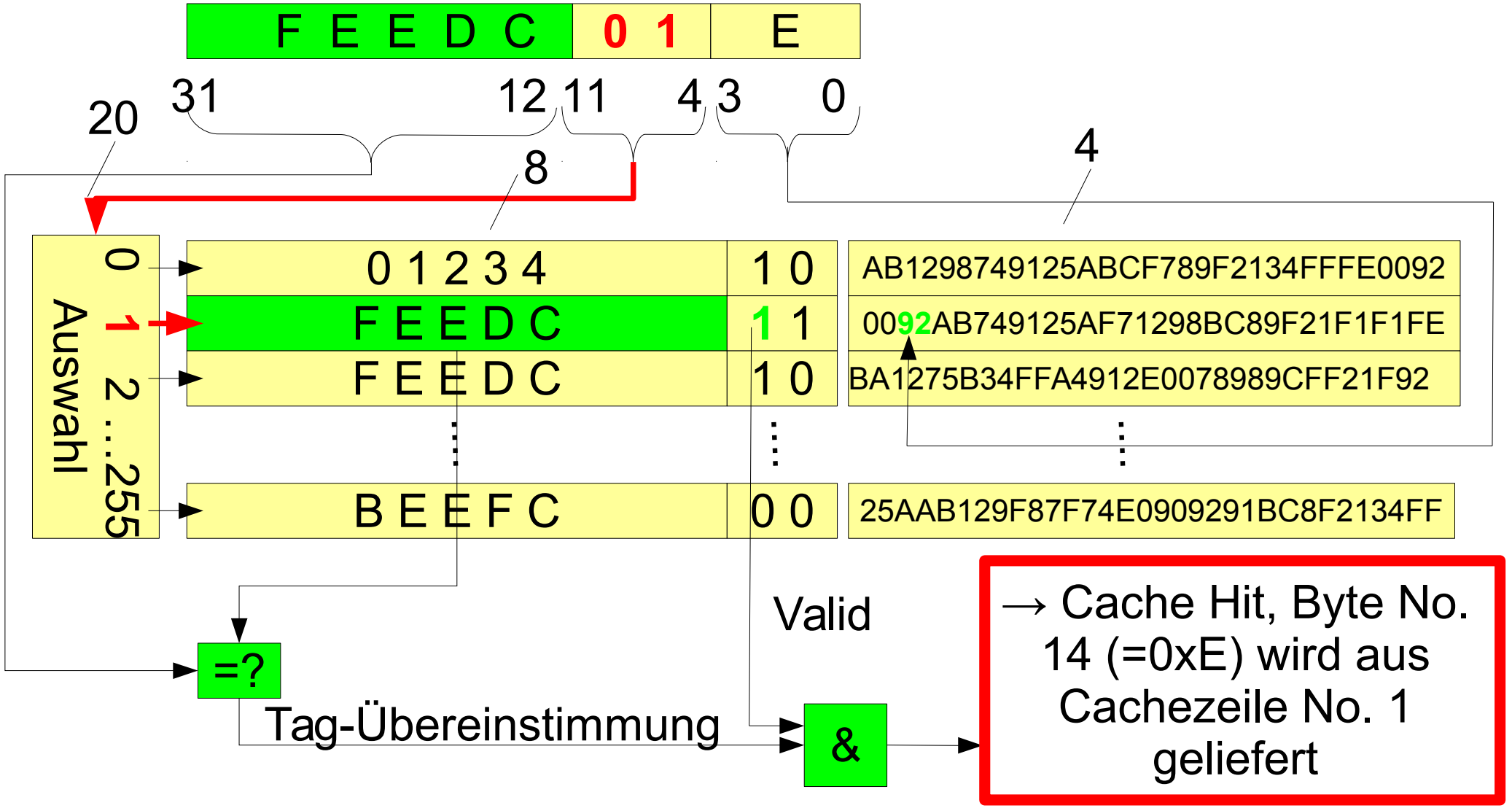
Direkt abbildender Cache: Aufbau

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Direkt abbildender Cache: Beispiel

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Direkt abbildender Cache

- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen**:
  - (+) Einfacher Aufbau (nur ein Vergleicher erforderlich)
  - (-) Schlechte Trefferquote
- **Beispiel**:
  - Ein Programm arbeitet in einer Schleife mit zwei Objekten, die in verschiedenen Cache-Zeilen liegen, deren Adressen sich aber in Bits 4 bis 11 nicht unterscheiden → Objekte verdrängen sich permanent gegenseitig (sog. „*Cache Trashing*“)

Mehrfach assoziativer Cache

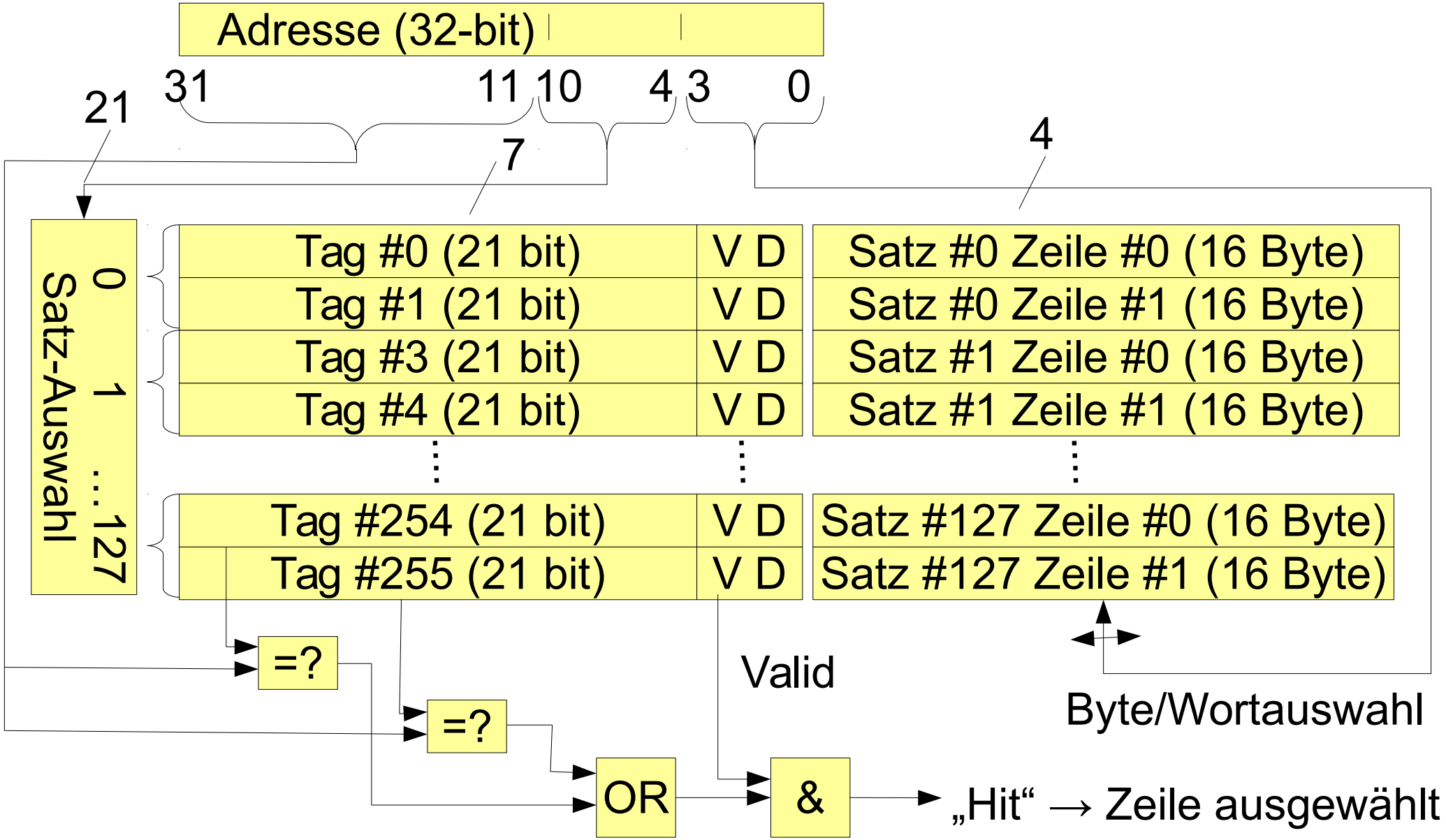
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing")
- Kompromiss: Mehrfach assoziativer Cache(*)
  - Zusammenfassen von je N (N = 2, 4, 8, ...) Cache-Zeilen zu einem „Satz" (engl. „*set*")
  - Ein Teil der Adresse dient als Satznummer
  - Innerhalb eines Satzes gibt es N mögliche Cache-Zeilen („Wege", engl. „*ways*"), die anhand ihres Tags unterschieden werden
- ➔ Für die Auswahl eines freien bzw. zu verdrängenden Cache-Eintrags stehen N Alternativen zur Verfügung
- Verdrängungsstrategien können –wenn auch eingeschränkt– umgesetzt werden

(*) engl. *N-way set associative cache*

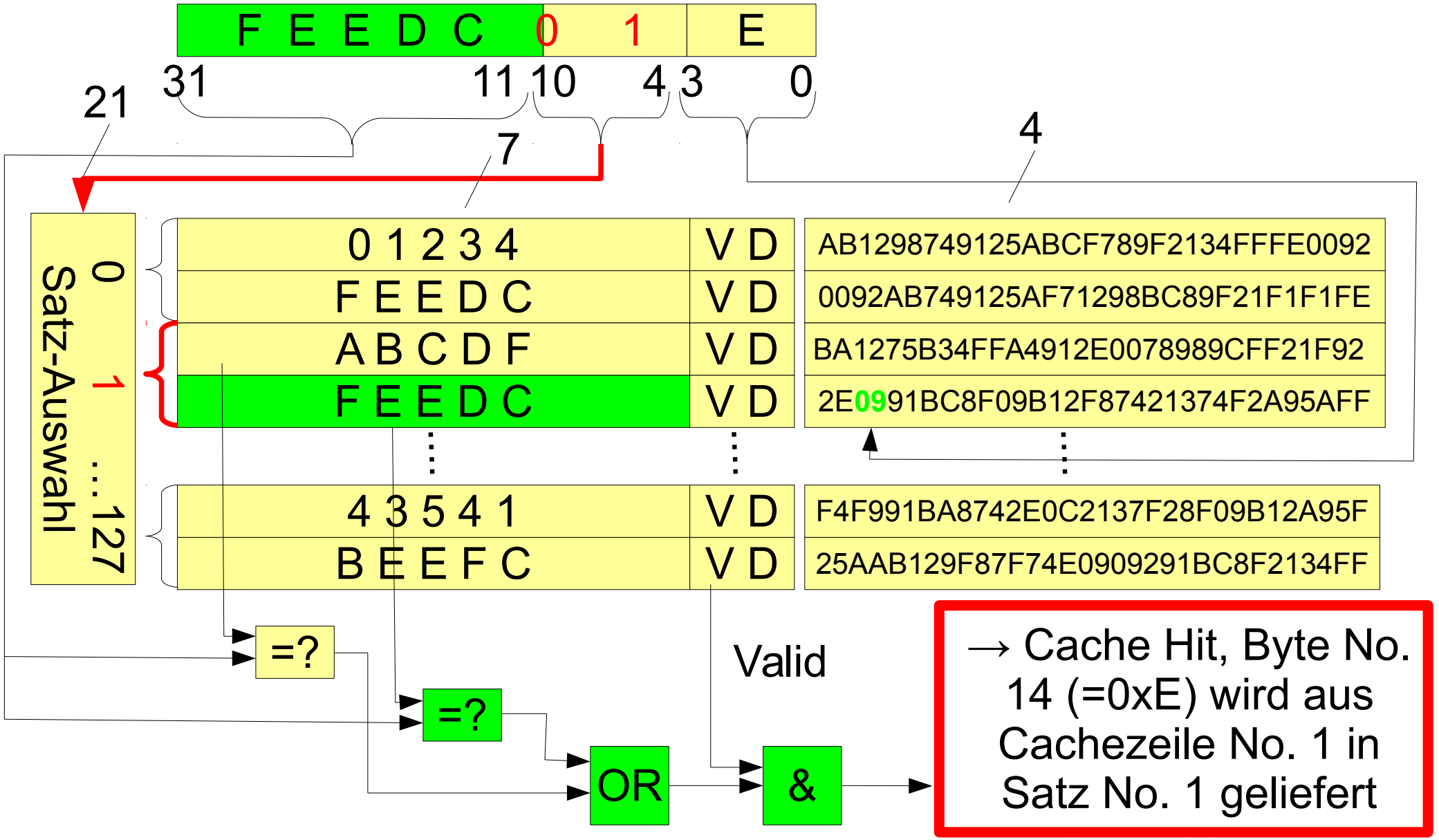
z.B. zweifach assoziativer Cache: Aufbau

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



zweifach assoziativer Cache: Beispiel

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim



Mehrfach assoziativer Cache: Fazit

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Deutliche Verbesserung der Trefferquote gegenüber direkt abbildendem Cache
- N Wege → N Vergleicher werden benötigt
- Für N = 1 „degeneriert" er zum direkt abbildenden Cache
- Für N = <Anzahl der Cache-Zeilen> „degeneriert" er zum vollassoziativen Cache
- Für Zwischenwerte von N: guter Kompromiss zwischen Aufwand und Trefferquote
- Heute der am meisten verwendete Cache
- (s.o.) Working Set üblicher Programme besteht aus > 5-6 Regionen
- N sollte >= Anzahl der Regionen sein (sonst → Thrashing)

Mehrfach assoziativer Cache: Beispiel

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

- Pentium 4:
  - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
  - L1 Befehlscache: 8-fach assoziativ
  - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

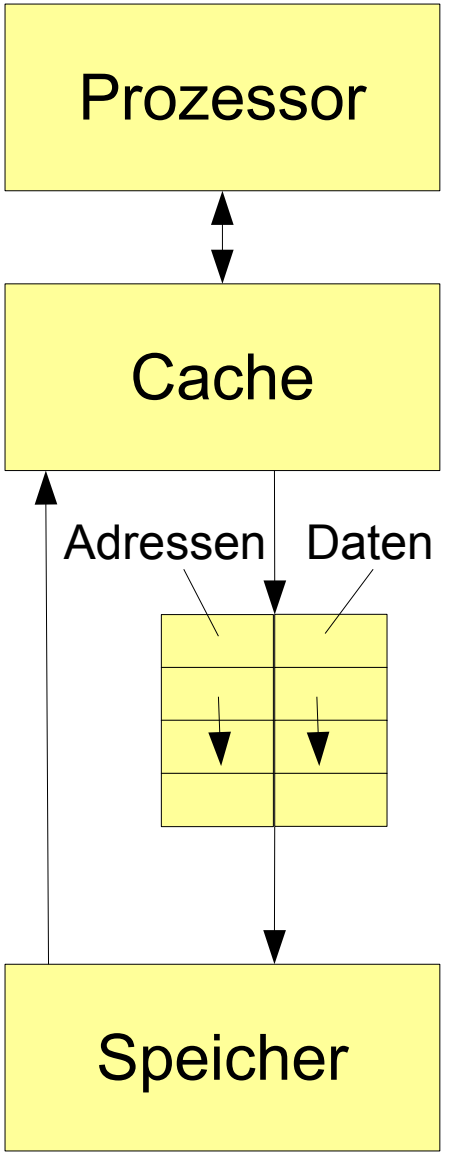
Fallstricke

- Potenzielle Probleme im Zusammenhang mit Caches
  - „zerklüfteter“ Working Set oder ungünstige Adresslage von Variablen kann zu *Thrashing* führen → drastischer Performance-Einbruch
  - **Multitasking**: Prozesswechsel bedeutet i.d.R. auch kompletten Wechsel des Working Set
  - ➔ Nach Prozesswechsel ist der Prozessor langsamer (u.U. bis Faktor 30!)
  - DMA und Schreibzugriffe auf Codespeicher: evtl. explizites *flush* & *invalidate* erforderlich (s.o.)
  - **Multicore:** Vielfach gemeinsamer L2/L3 Cache: → gegenseitiges „ausbremsen“ der Cores, wenn auf verschiedenen Working sets gearbeitet wird.

Schreib-Pufferspeicher (*Write Buffer*)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Charakteristisches Verhalten von –z.B.– C-Programmen:
  - Etwa 10% „store”-Befehle, d.h. speichern von Daten
  - Solche Schreibzugriffe kommen häufig in schneller Folge („Bursts”) vor (z.B. wenn zu Beginn eines Unterprogramms Register gerettet werden)
- Insbesondere bei einem *write through* Cache muss der Prozessor hier auf den langsamen Hauptspeicher warten
- Abhilfe durch *Write Buffer*:
  - Ausstehende Schreibzugriffe (Adressen **und** zu schreibende Daten) werden in einen FIFO-Puffer zwischengespeichert
  - → Prozessor kann sofort weiterarbeiten
  - → Zwischgespeicherte Speicherzugriffe werden parallel dazu abgearbeitet



Schreib-Pufferspeicher (*Write Buffer*)

Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

- Write Buffer finden sich z.B. bei ARM, PowerPC und MIPS-Prozessoren
- Potenzielles Problem: Lesezugriffe können Schreibzugriffe „überholen“
- z.B. bei Ein-/Ausgabe:
  - Gerät löst Interrupt aus, obwohl der bereits (per Schreibzugriff) abgeschaltet wurde
- Lösungswege:
  - **Software**: Puffer explizit „flushen“ (spezieller Maschinenbefehl)
  - **Hardware**: Jeder Lesezugriff wartet, bis der Puffer leer ist