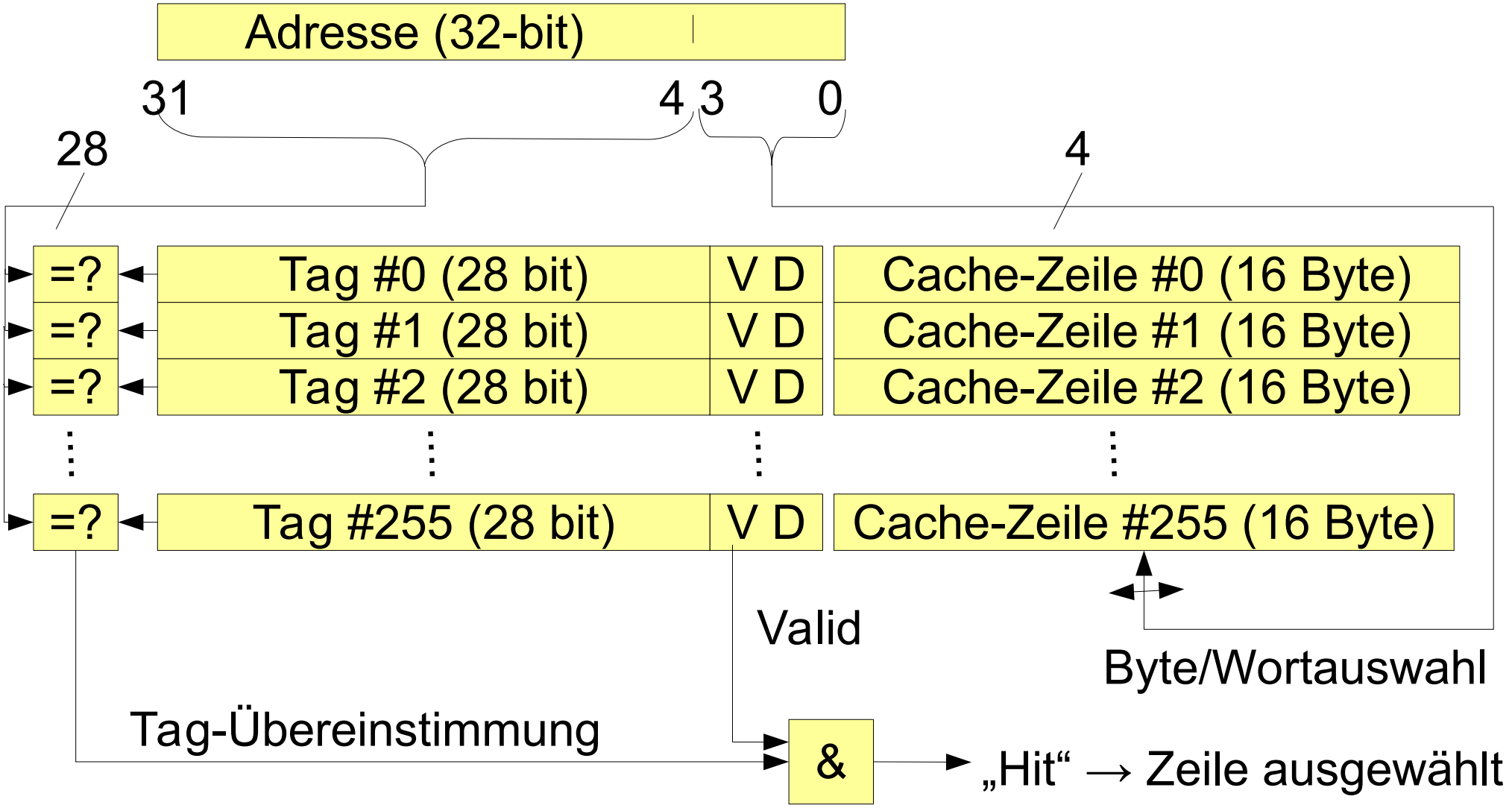
Cache-Organisationsformen

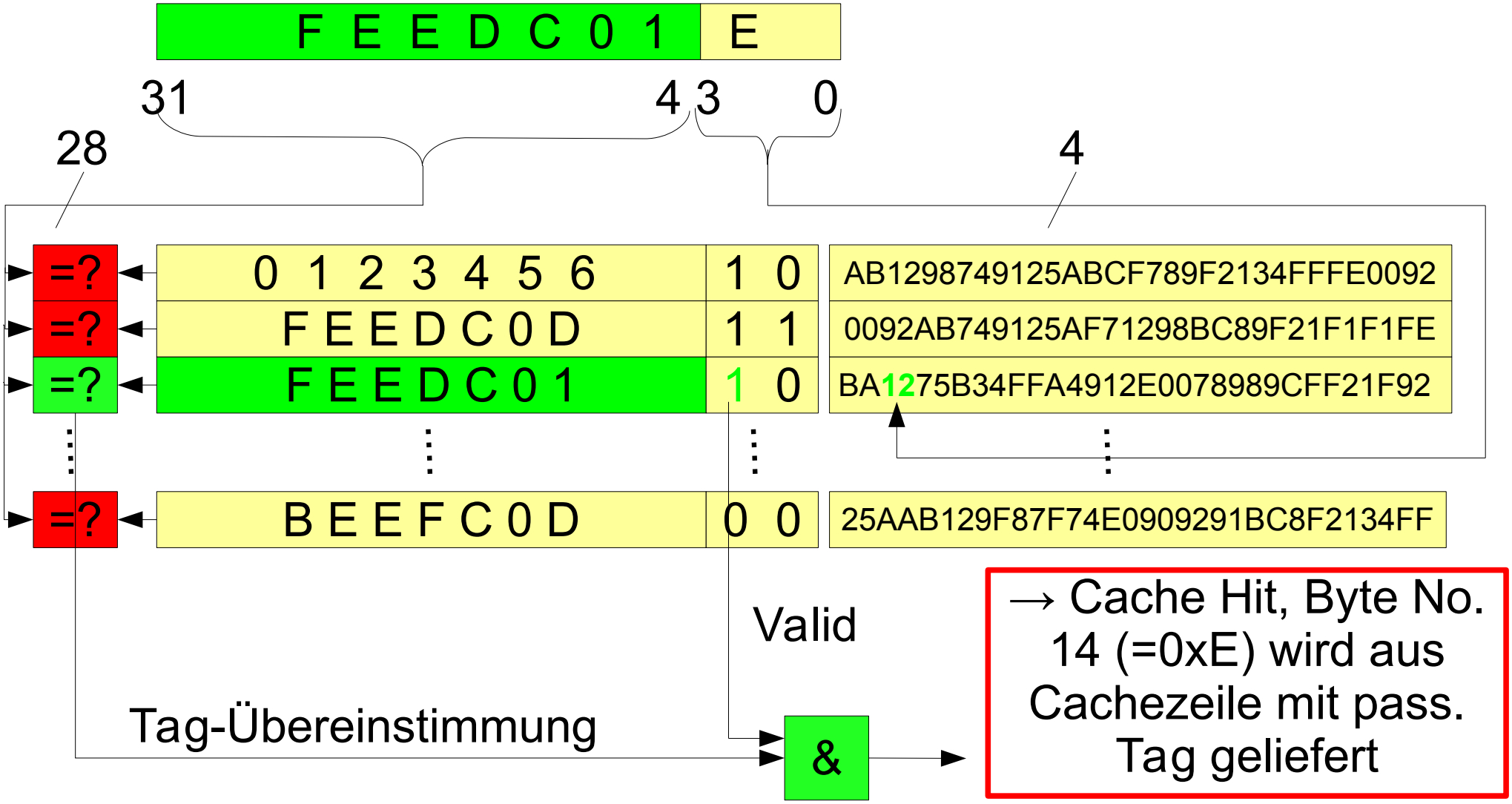
- Cache-Organisationsformen:
  1. Vollassoziativ: *fully associative*
  2. Direkt abbildend: *direct-mapped*
  3. Mehrfach assoziativ: *N-way set associative*
- Annahmen bei den folgenden Beispielen:
  - 32-bit Adressierung
  - 4K (4096) Byte Cache
  - Cache-Zeilengröße: 16 Byte
    (→ 4096 : 16 = 256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

Vollassoziativer Cache: Aufbau

Hochschule RheinMain



Vollassoziativer Cache: Beispiel

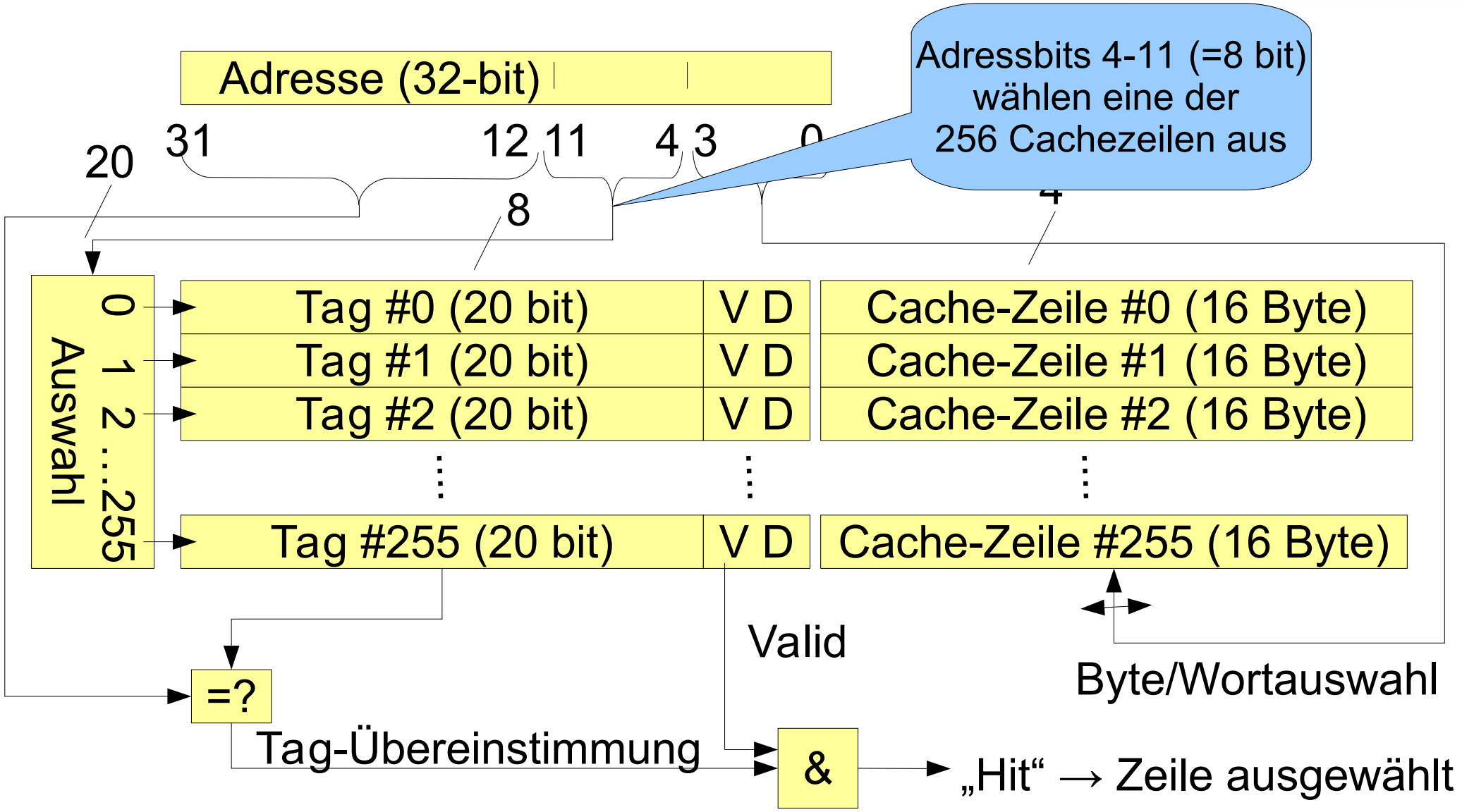


Vollassoziativer Cache

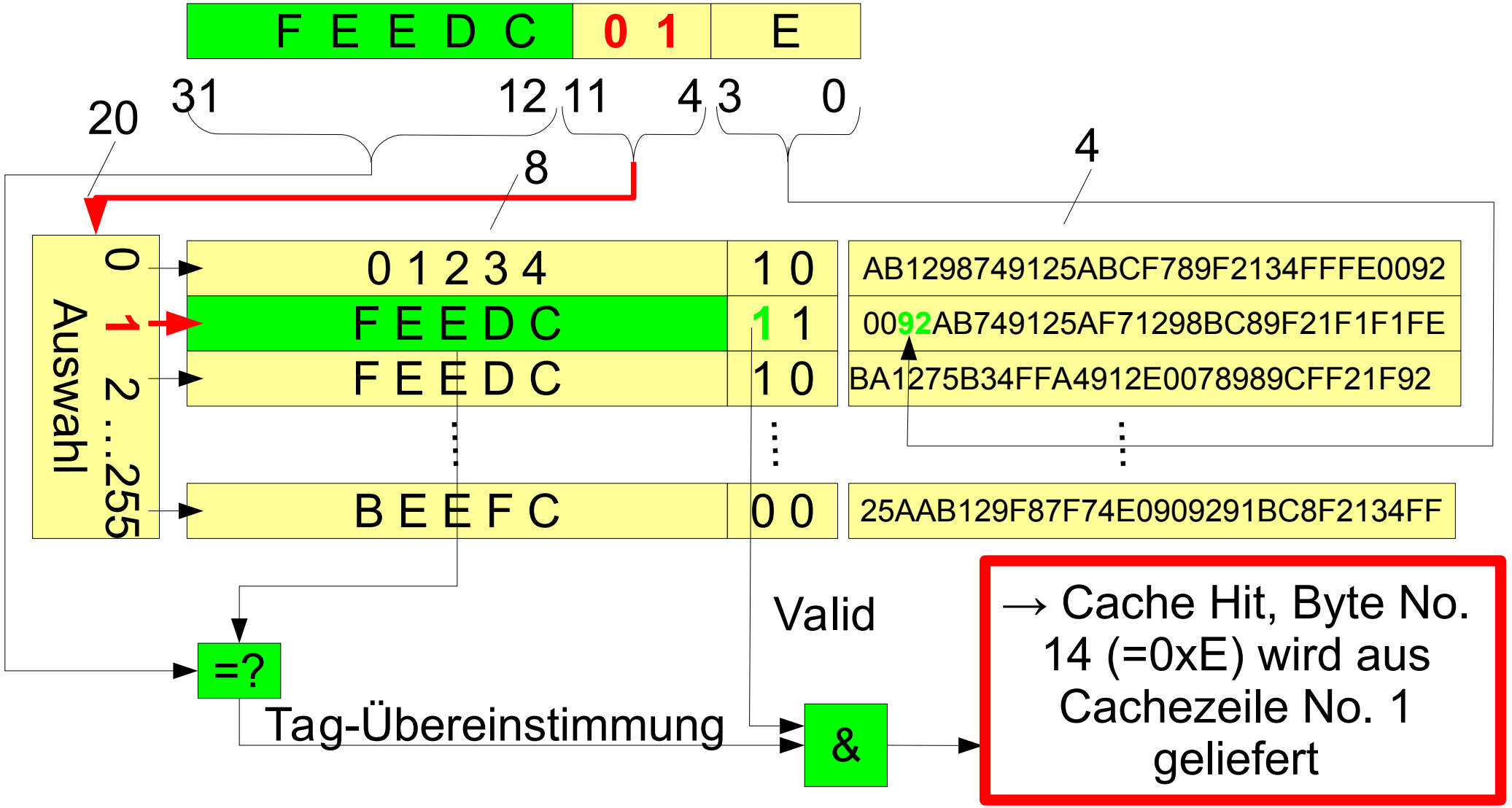
- Ein Objekt kann in eine beliebige Cache-Zeile kopiert werden
  → Freie Auswahl eines freien / zu verdrängenden Eintrags
- Identifikation der Zeile ausschließlich anhand des Tags
  → Konsequenzen:
  - Tags **aller** Zeilen müssen mit Adresse verglichen werden
  - Vergleich muss **gleichzeitig** auf 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



Direkt abbildender Cache: Beispiel



Direkt abbildender Cache

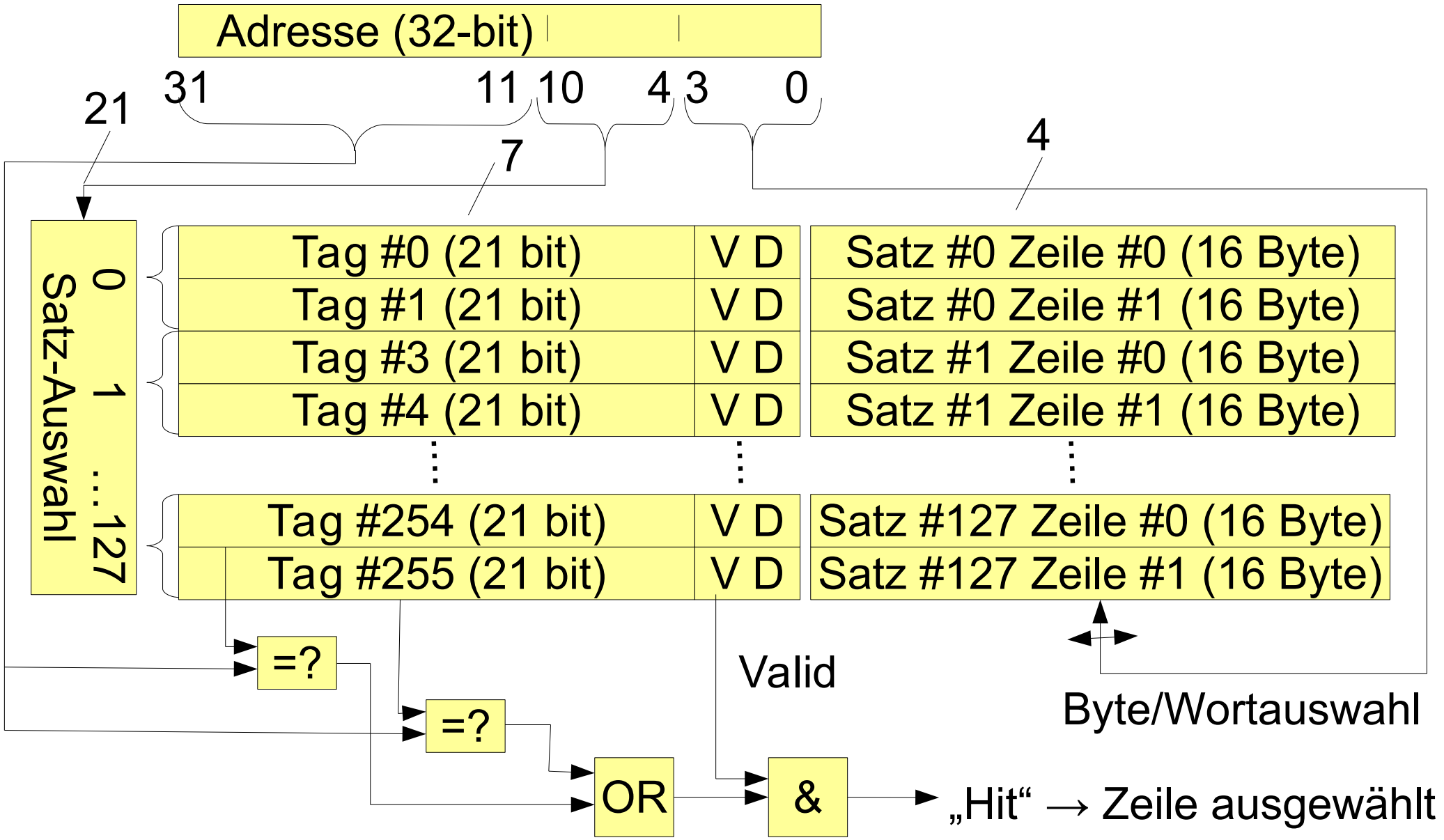
- Ein Teil der Adresse (im Beispiel: Bits 4 bis 11) wählt die Cachezeile aus
- Eindeutige Zuordnung ohne Wahlfreiheit, keine alternativen Verdrängungsstrategien möglich (aber auch keine nötig)
- Tag dient allein zur Hit/Miss Entscheidung
- **Konsequenzen**:
  - (+) Einfacher Aufbau (nur ein Vergleicher 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

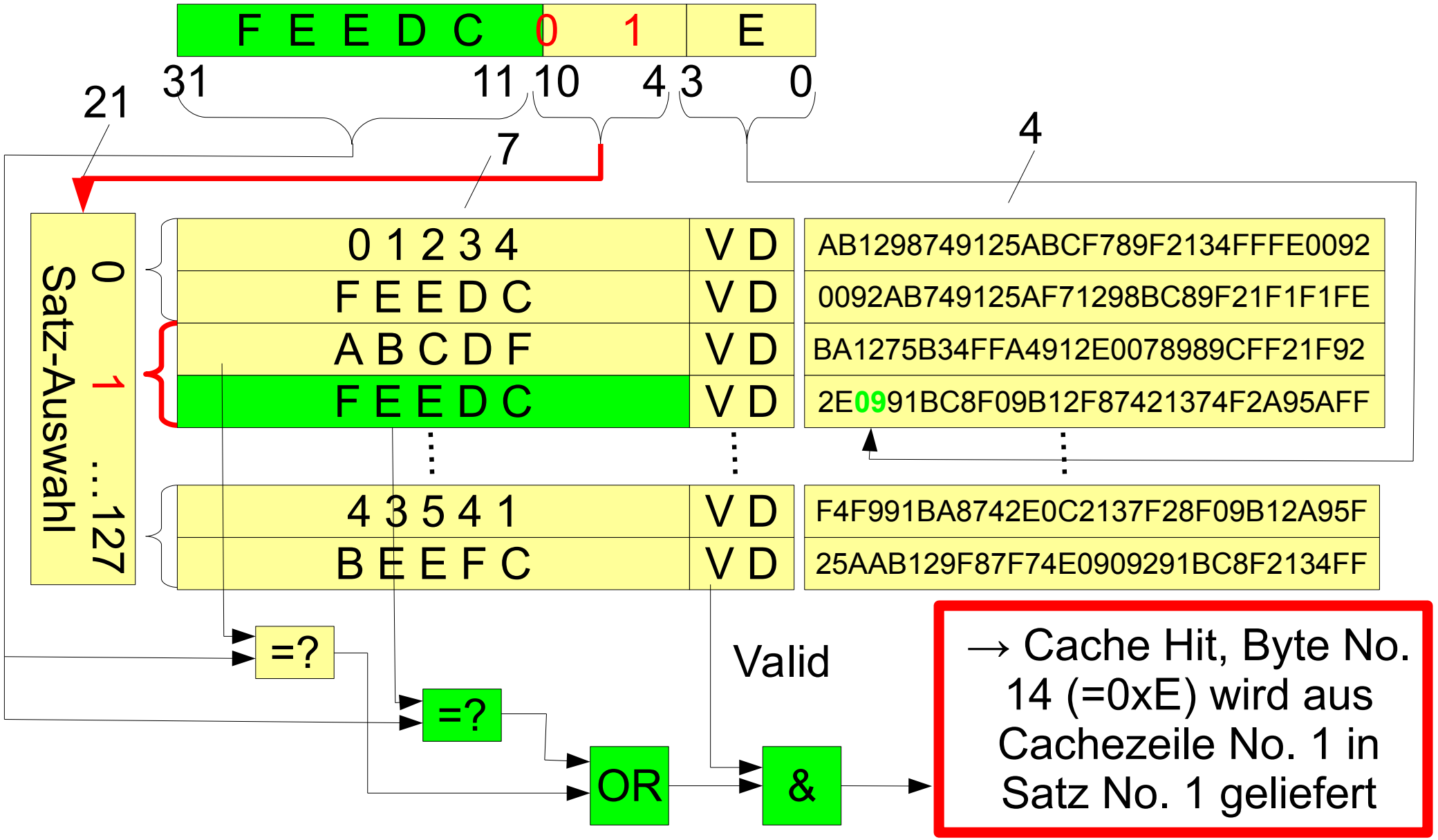
- Vollassoziativer Cache: Hohe Trefferquote, aber aufwändig
- Direkt abbildender Cache: Geringer Aufwand, aber schlechte Trefferquote (neigt zu „thrashing")
- Kompromiss: Mehrfach assoziativer Cache(*)
  - Zusammenfassen von je N (N = 2, 4, 8, ...) Cache-Zeilen zu einem „Satz" (engl. „s*et*")
  - Ein 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



zweifach assoziativer Cache: Beispiel



Mehrfach assoziativer Cache: Fazit

- 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

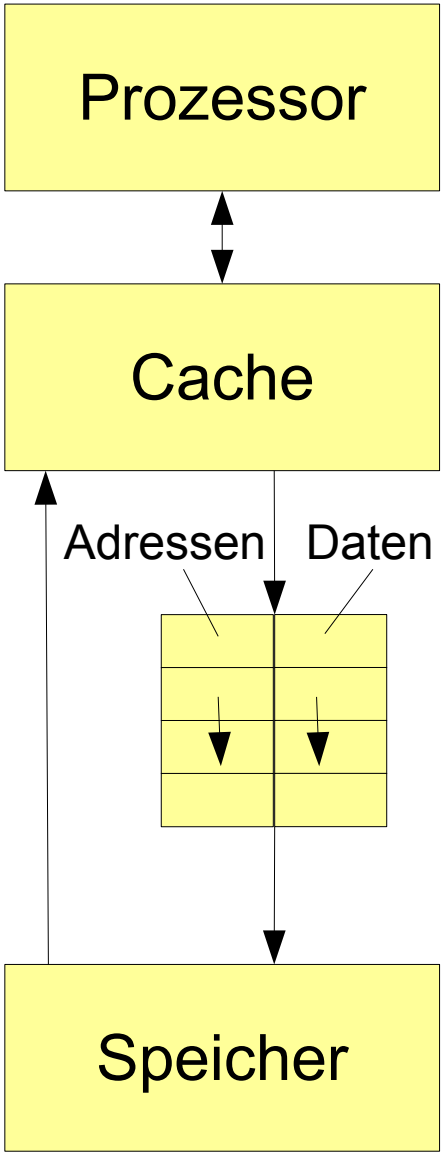
- Pentium 4:
  - L1 Datencache: 4-fach assoziativ (64 Byte Zeilengröße)
  - L1 Befehlscache: 8-fach assoziativ
  - L2 Cache: 8-fach assoziativ (64 Byte Zeilengröße)

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*)

- 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*)

- 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