Beter, Sneller, Mooier

Processoren

12 januari 2015

Beter! Sneller!

- Krachtigere CPU:
  - maak instructies die meer doen
- Snellere CPU:
  - pipeline, out-of-order execution
- Sneller RAM:
  - cache

meer mogelijkheden...

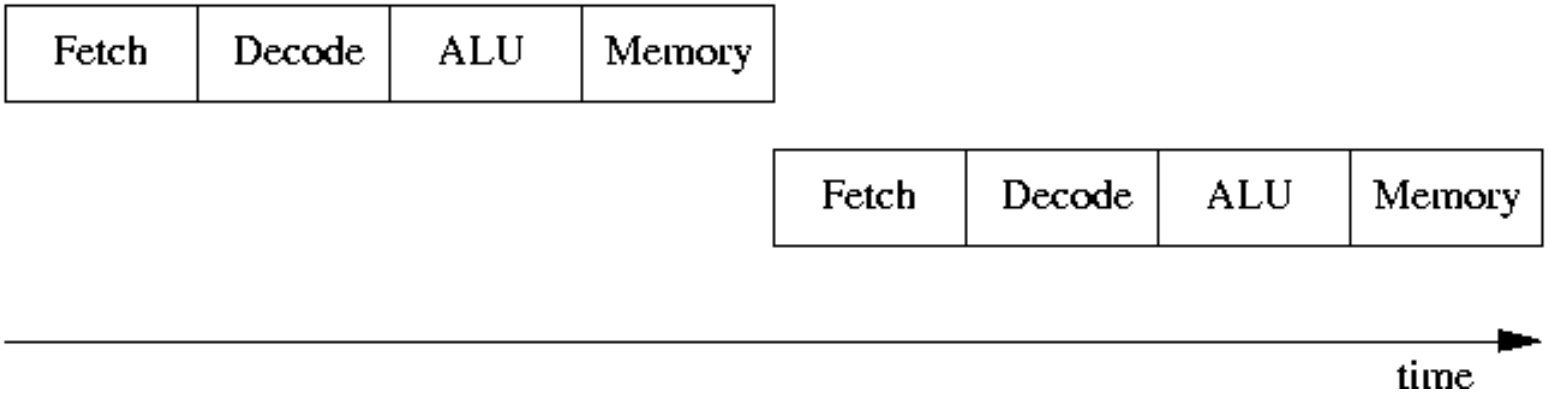
- Welke extra's kan processor-bouwer bieden?
  - kleinere programma's
    → instructies die meer doen
  - snellere programma's
    → hogere klokfrequentie
    → pipelining (*), out of order execution
    → caching (*)

- vandaag: twee technieken (*) daarvoor
  - die niet in de practicumprocessor gebruikt worden

Instructies die meer doen...

- bv. PUSH en POP bij de practicum-processor
  - schrijft naar geheugen/leest uit geheugen
  - verandert stack pointer
  - kunnen we dit binnen onze instructieformaten makkelijk coderen?
  - welke implementatie problemen krijgen we als we dit willen doen?

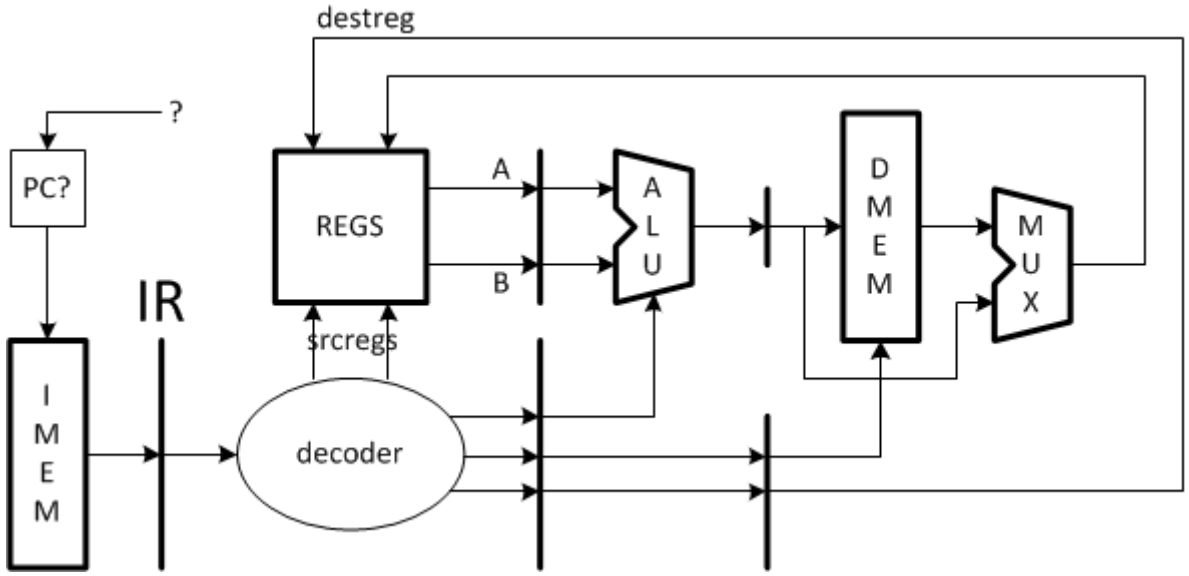
Huidige practicum processor



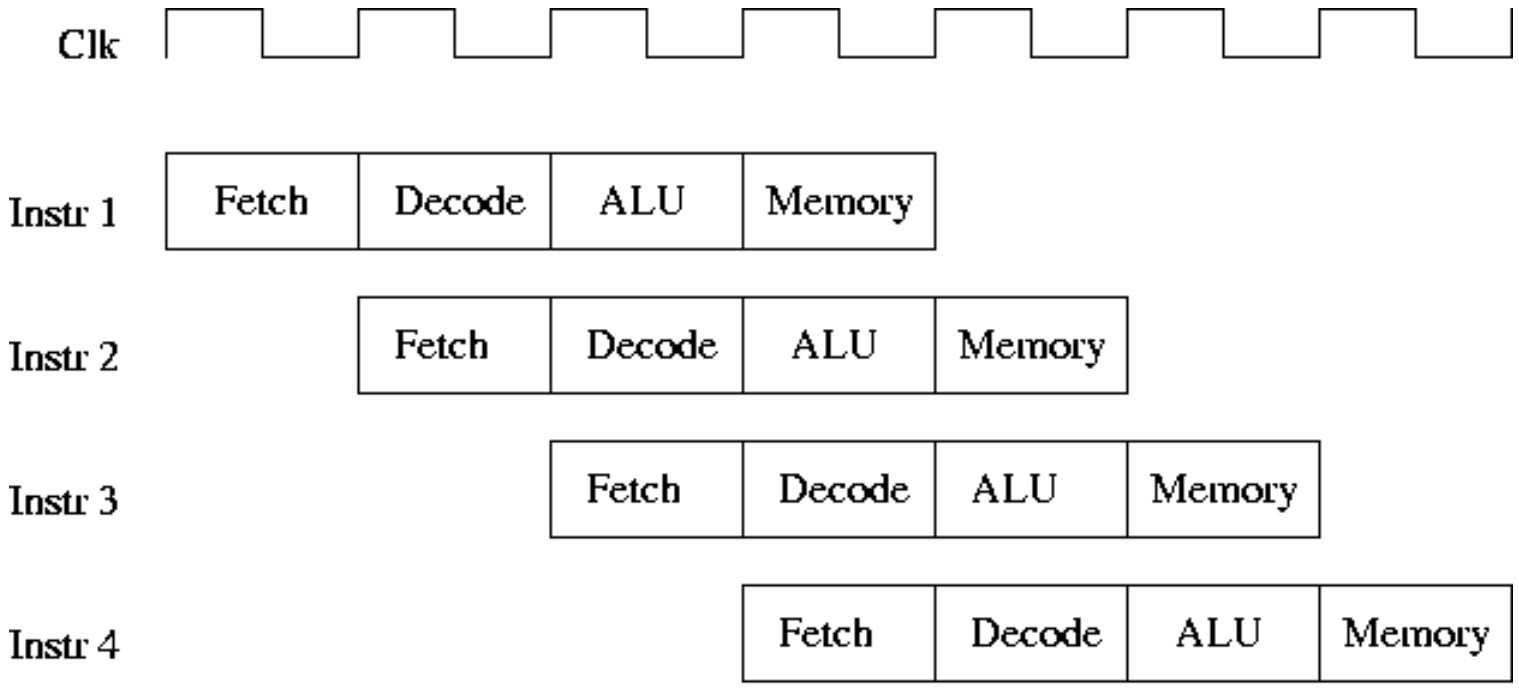
- Twee klokken per instructie
- In de praktijk kost decodering en de ALU net zoveel tijd als geheugen-access. Dit wordt in Hades niet gemodelleerd.

Pipelining: het idee

- Ontkoppel de fasen van een instructie, zodat elk in parallel gedaan kan worden.
- Gevolg: elke fase heeft inputregisters.
- Poging:



Gepipelinede practicum processor



- Elke klokslag wordt nu een instructie gestart!

Pipelining: het idee 2

- Probleem: fetch en data access moeten dan parallel =>

- Maak een Harvard-architectuur:
  - Apart geheugen voor instructies en data.
  - Aparte adres-, data- en control bussen voor fetch en data access
- In de praktijk (MIPS, Arm) wordt dit gedaan door elk een eigen I- of D-cache te geven die op hun beurt gemeenschappelijk geheugen aanspreken.

- De PC is een eigen register in de fetcher (MIPS) of wordt met een eigen pad naar de registerbank bijgewerkt (Arm) (en heeft een eigen incrementer)

Problemen met pipelining: Hazards

Voorbeelden:

a)
```
ADD     R1,R2,R3
SUB     2,R3,R4
```

b)
```
READ    [R2+20],R5
SUB     R5,R4,R6
```

c)
```
SUBf    R1,R2,R0
JUMP.NZ Loop
ADD     3,R4,R5
```

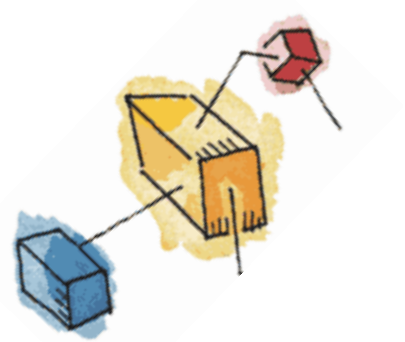
Hazards: oplossingen

- Laat de assembler een warning geven (MIPS1)
- Register forwarding logic: voer output van een stage naar een vorige pipeline stage om een hazard op te lossen (lost a op).
- Stalling: stop NOP bubbles in de pipeline totdat je verder mag (lost b op).
- Voor control hazards: evalueer condities tijdens decoderen en sta toe dat de instructie na de jump nog uitgevoerd wordt of flush de pipelijn (lost c op).

Jumps zijn lastig

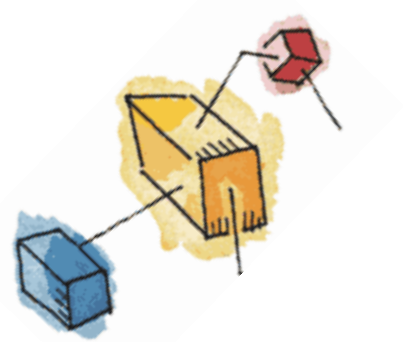
- Bij conditionele sprong-instructies zijn er twee mogelijke voortzettingen van het programma.
- Welke van de twee moet de processor nemen? Pas bekend als de vorige instructie uitgerekend is.
- Pijplijn leeg gooien en weer vullen, als de sprong genomen wordt (Arm7) → inefficiënt

- Bij SPARC en MIPS: sprongen werken met één instructie vertraging door apart pad van decoder naar fetcher.



Cache Memory

- Invisible to the OS
  - Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
  - Processor speed faster than memory access speed
- Exploit the principle of locality with a small fast local memory



Principle of Locality

- More details later but in short …
- Data which is required soon is often close to the current data
  - If data is referenced, then its neighbours might be needed soon.

Cache and Main Memory

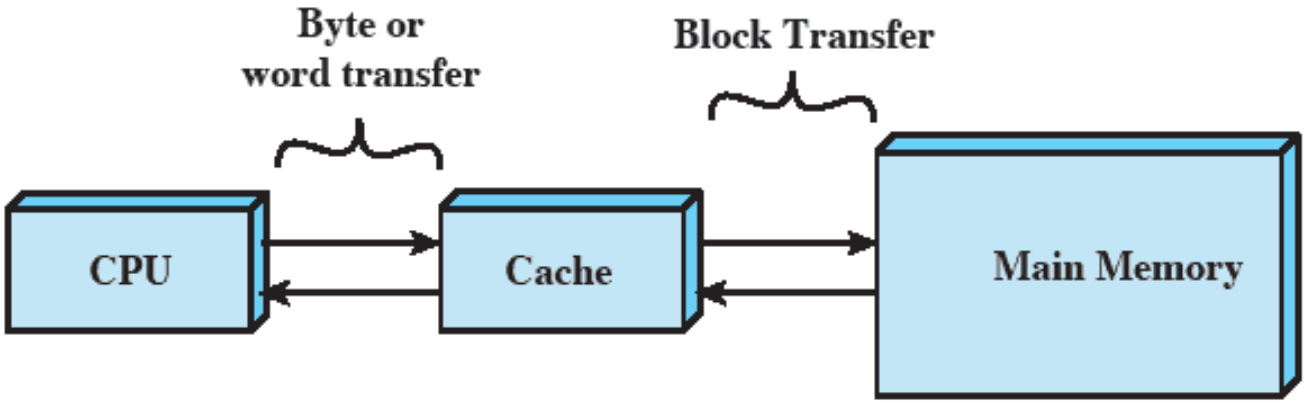
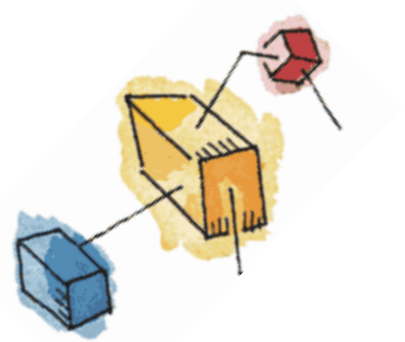


Figure 1.16 Cache and Main Memory



Cache Principles

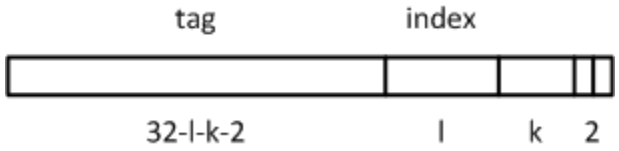
- Contains a copy of a portion of main memory
- Processor first checks cache
  - If not found, read block of memory into cache
  - Advantage: the DRAM of main memory likes to be read/written as blocks of memory.
- Because of locality of reference, likely future memory references are in that block

Cache: hoe werkt het?

- We verdelen het geheugen in blokken van elk $K=2^k$ woorden van bv. 32 bits groot.

  (Pentium 2 L1 cache kiest k=3, dus K = 8)

- Verdeel elk adres in 4 stukken:



- Hierin is $C = 2^l$ het aantal cache lines.

- Met de index adresseren we nu het cache ram.

Cache/Main-Memory Structure

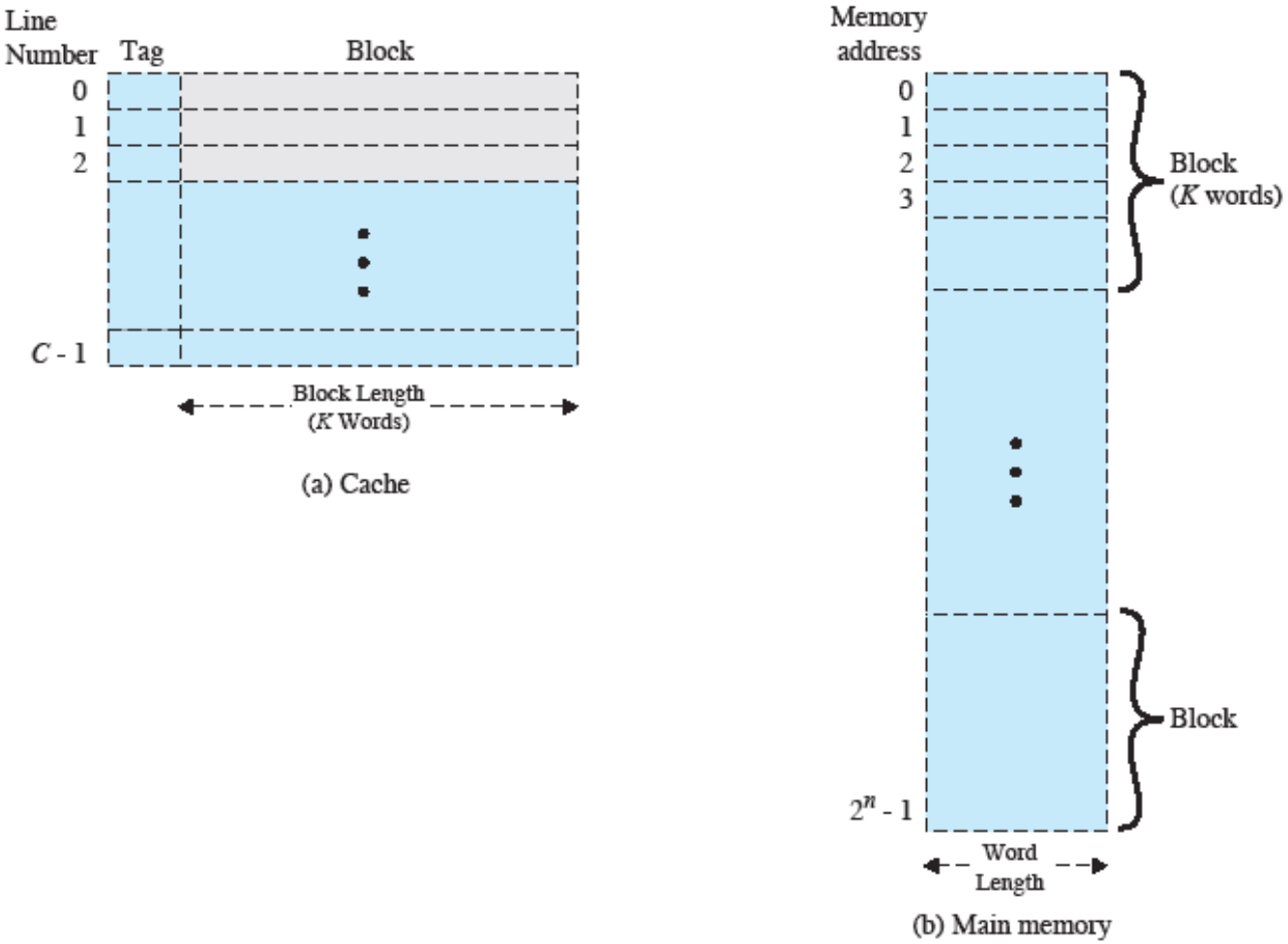


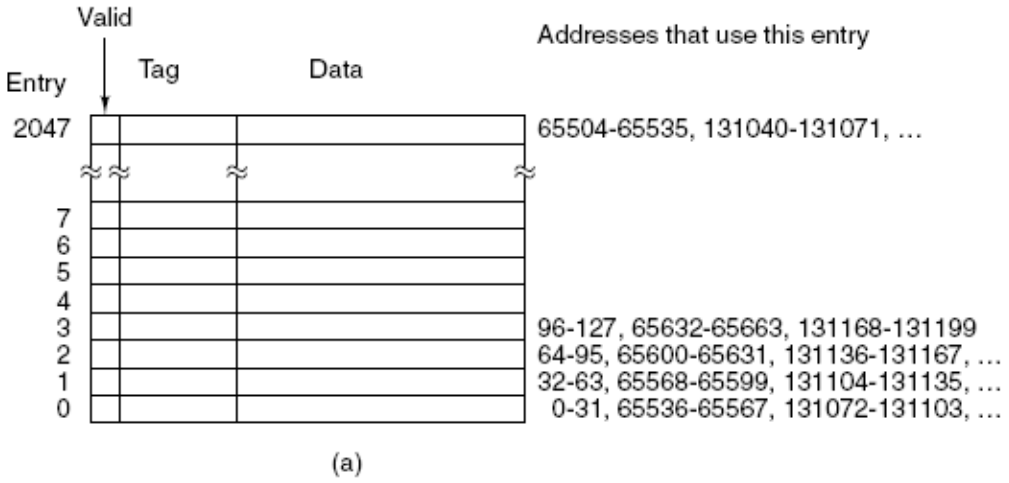
Figure 1.17 Cache/Main-Memory Structure

direct-mapped cache

(a) cache-structuur

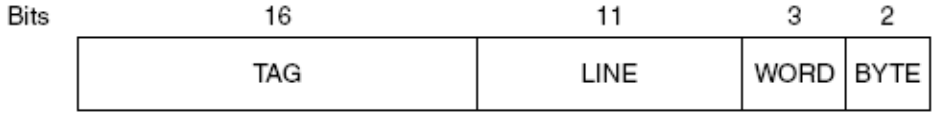
Elk data-veld bevat 8 words
van 4 bytes, dus 32 bytes.

In elke cache-lijn kan één blok
van 32 bytes opgeslagen worden.
De tag geeft aan welke.



(a)

(b) bitpatroon van een adres (32 bits)



(b)

In een adres geeft “line” aan
in welke 'regel' van de cache
de CPU moet zoeken.

(c) In de praktijk houden we cache tags
(en vlaggen) en cache data in twee
verschillende RAM geheugens bij.

Cache hit en cache miss

- Als de tag uit de cache gelijk is aan het tag gedeelte van het adres en het valid bit is gezet, matcht deze cache lijn (Cache hit) en mogen we de k+2 overige bits uit het adres gebruiken om de data uit deze cache lijn op te halen.

- Als deze ongelijk is (Cache miss), halen we dit blok op uit het geheugen, schrijven het in deze cache lijn, en schrijven we het tag gedeelte van het adres in de tag van de cache lijn en zetten het valid bit.

- Bij een reset zetten we alle valid bits op 0.

Cache thrashing

- Wat doet deze cache als er op 0x40 een klein loopje staat dat een call doet naar 0x30040?
- Deze 2 blokken hebben dezelfde cache index (nl. 2). Hun tags zijn echter 0 en 3.
- Bij de call moet de code op 0x30040 in deze lijn ingeladen worden en na elke return moet de code op 0x40 weer ingeladen worden.
- Dit fenomeen heet cache thrashing.

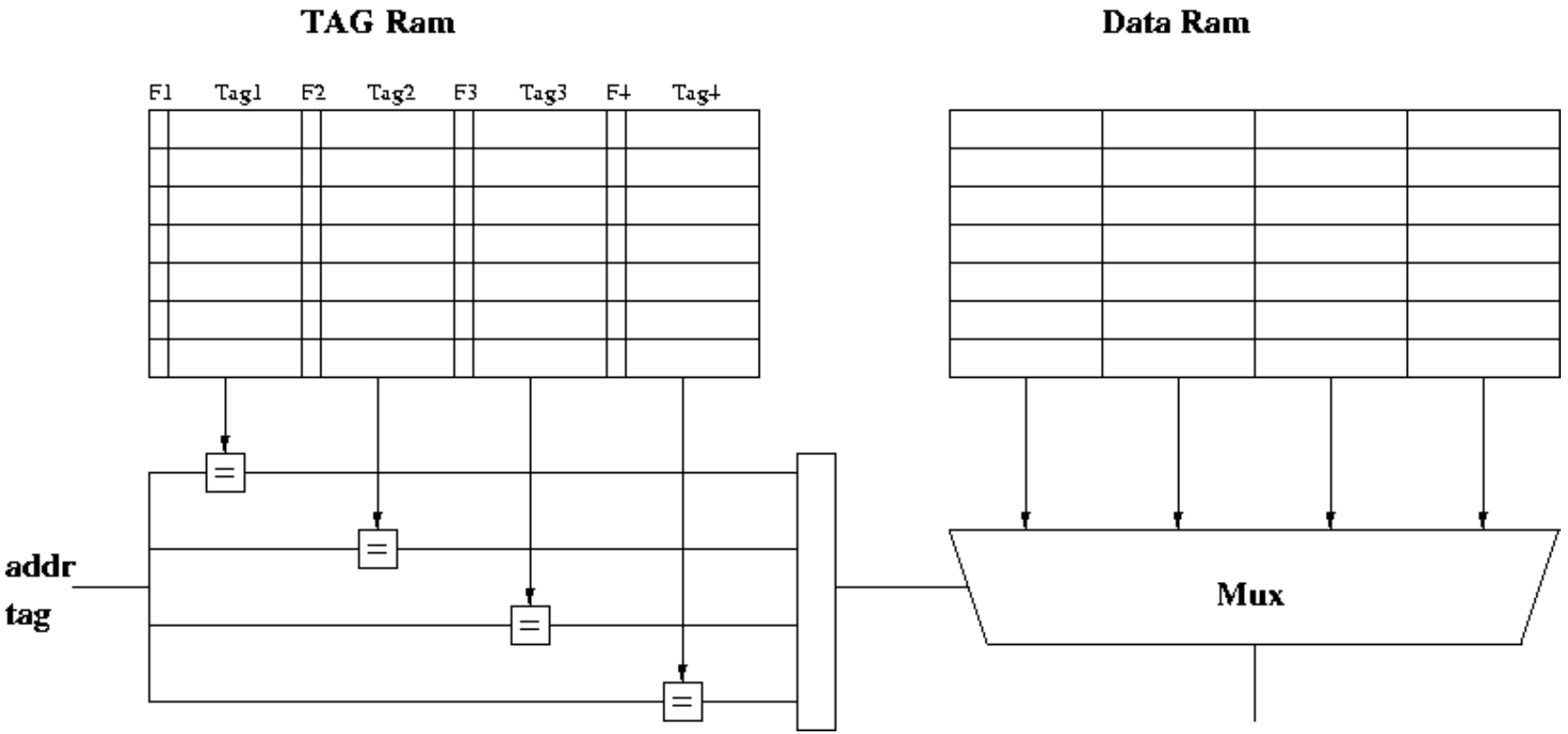
Multiple cache ways

- Om cache thrashing te vermijden, bewaren we verschillende blokken bij dezelfde cache index en dus ook met verschillende tags.

- De vergelijking voor een cache hit vindt dus plaats met meerdere tags, maar dat kan in parallel.

- Dit geeft dus de volgende memory structuur voor de cache:

Four way cache



Cache policies

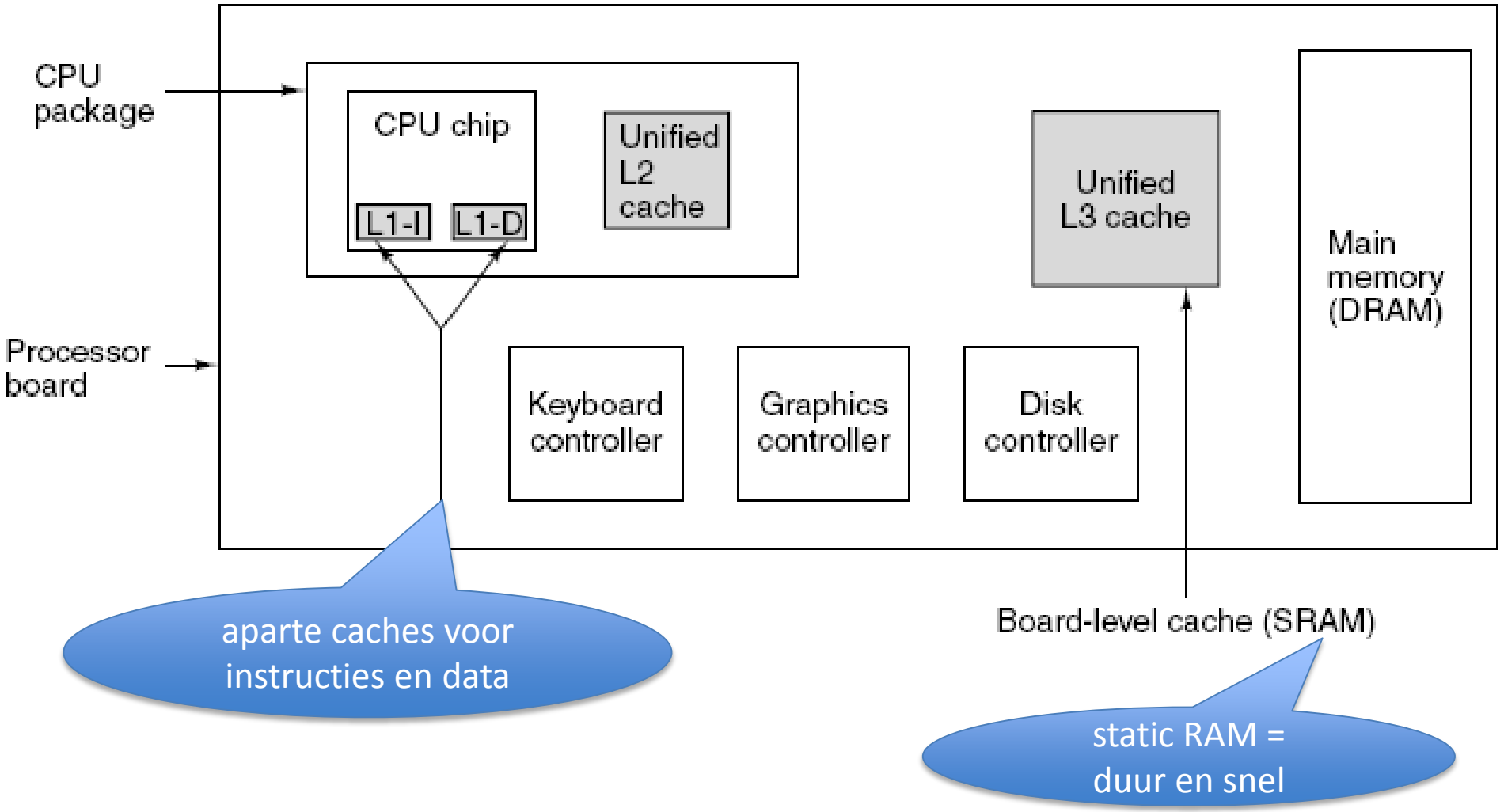
- Over cache valt nog veel meer te vertellen:
  - Hoe kiezen we bij een multiway cache welke van de set vervangen wordt (Replacement policy)?
  - Wat als er geschreven wordt? De cache data wordt geschreven maar hoe gaat dat naar main memory (Write policy)?
  - Hoe zit het met parallellisme? Moeten caches naar elkaar luisteren (Cache coherency)?

Cache

- Probleem van hoge frequentie:
  moederbord kan niet sneller worden

- Oplossing:
  cache memory op, in of direct naast CPU

Voorbeeld: 3 cache levels



Samenvatting

- Pipelining
  - problem: hazards
- Caching
  - problem: thrashing, policies
- Elke verbetering geeft ook problemen om op te lossen!