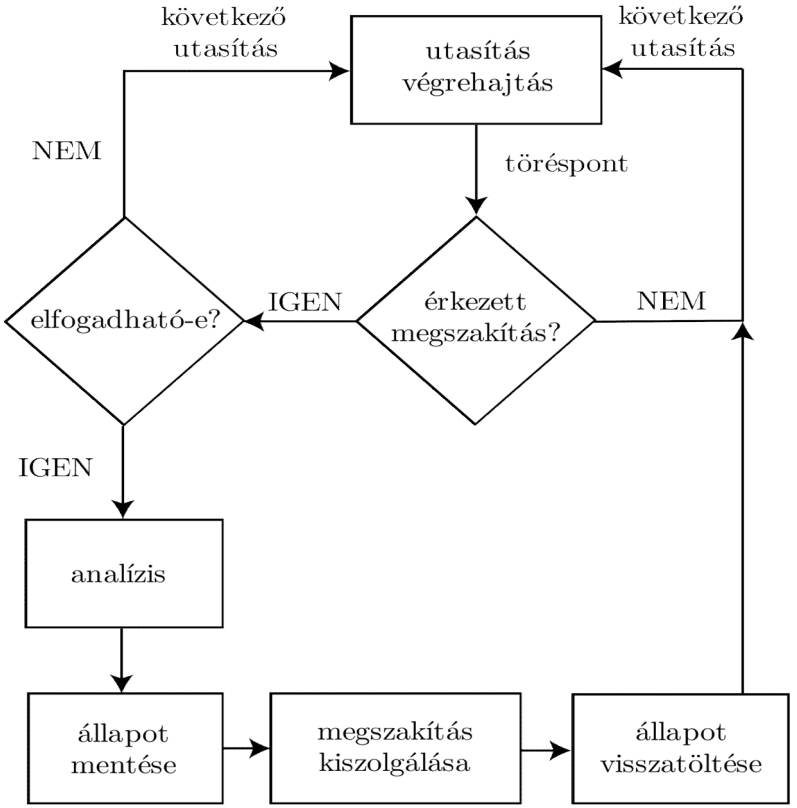
Megszakítási rendszer:

Megszakítás definíciója:
A feldolgozás szempontjából váratlannak tekinthető események kezelésére szolgáló művelet.
Célja nem csak a reagálás, hanem a folyamatosan változó körülmények között az optimális működés biztosítása.

Folyamata:



A megszakítás lényege, hogy csak akkor foglalja le a CPU figyelmét amikor arra tényleg szükség van!

Történeti áttekintés:

CPU és I/O-ból érkező adatok összehangolása céljából jött létre.

Kezdetben az I/O adatátvitel vezérlésére alkalmazták először, mivel a CPU rengeteg felesleges munkát végez azzal, hogy várakozik az I/O eszközökre a válaszra, mivel azok lényegesen lassabbak.
Ezért az I/O tevékenység befejeződése egy megszakításkérő jelet generál és a következő utasítás megkezdése helyett helyett egy vezérlés-átadást hoz létre a megszakítás konkrét okától függő címre (az aktuális megszakító rutin kezdőcíme)

Később rájöttek, hogy a megszakítások belső rendszerek vezérlésére is alkalmasak.
(pl.: határérték túllépés esetén jelzés azonnal, és beavatkozás!)

Megszakítás esetén a kontextus el kell menteni, majd betölteni a megszakítási rutin kontextusát, végre kell hajtani az megszakítást, majd visszaadni a vezérlést. Ez a kontextus tárolása gyakorlatban annyit tesz, hogy például a PC-be nem a következő utasítás címe, hanem a megszakítási rutin kezdőcíme kerül, majd PC eredeti állapotának visszatöltődik.

Megszakítási okok és források:

Prioritási sorrendben:

1. Géphibák: ezek a megszakítások nem letilthatók
   Például: automatikus hibafigyelő áramkörök jelzései (hőszenzorok, energiaellátás, …).
   Egy részük hibajelző kódok segítségével tárja fel a hibákat (paritás, watch dog, …).

2. I/O források: ide tartozik minden perifériával kapcsolatos állapotjelzés

3. Külső források
   Például: reset gomb, hálózati kommunikáció

4. Programozási források:

a. Szándékos: amikor egy program megszakítást kér. Például: rendszerhívás, BIOS hívás

b. Nem várt (hibakezelés): mindig valamilyen utasítás végrehajtása vagy a végrehajtás megkísérlése során alakul ki

Programozási okok miatt fellépő megszakítások:

A következő hibáknál léphet fel:

1. Memóriavédelem megsértése

   Alapértelmezetten minden program mikor fut, lefoglalódik számára egy memória terület.

   Ezt védeni kell a „véletlen” felülírástól! Amennyiben egy program a futása során „idegen” címre hivatkozik, az alkalmazott hardver memóriavédelem működésbe lép.

2. Tényleges tárkapacitás túlcímzése

   Adott implementációnál amennyiben a fizikai tárkapacitás kisebb, mint az utasítás végrehajtása során elméletileg kiadható legnagyobb cím, előfordulhat véletlen túlcímzés.

   ⟶ Megszakítás következik be.

3. Címzési előírások megsértése

   Példa: a címek 32 bitesek és a címzés byte-osan történik. Ebben az esetben csak minden negyedik byte címezhető. Ha program a címzésnél ezt megsérti, megszakítás történik.

4. Aritmetikai logikai végrehajtás közben fellépő hibák

   Például: overflow, underflow, 0-val való osztás

Megszakítások csoportosítása:

1. Megkülönböztetünk **szinkron** és **aszinkron** megszakításokat.

a) Szinkron megszakítás: adott programnak ugyanazon paraméterével történő futtatása során mindig ugyanott jelentkezik a megszakítás (pl.: integer túlcsordulás)

b) Aszinkron megszakítás: véletlenszerűen lépnek fel

- várható: pl.: I/O egység által kért, vagy DMA által kért
- nem várható: pl.: hardverhibák

2. Utasítások **végrehajtása között** vagy utasítások **végrehajtása közben** fellépő megszakítások

a) Között: egy utasítás végrehajtásának eredményeképpen következik be. Utasítás végrehajtás után azonnal elkezdődhet a kezelés. A program folytatása a kezelés eredményétől függ. Például: overflow, tárvédelmi hiba, …

b) Közben: egy utasítás végrehajtása alatt következik be. Nincs szinkronban a végrehajtási ciklussal. Például: hardvermegszakítások

3. Felhasználó által **kért**, illetve **nem kért** megszakítások

a) Kért megszakítás például: OS rutinok, BIOS rutinok, debug, …
b) Nem kért például: overflow, I/O egység által kért, hardverhiba, …

4. A megszakított program a megszakítás után **folytatódik** vagy **befejeződik**

a) Folytatódik például: I/O megszakítás, OS rutinok
b) Nem folytatódik például: hardverhiba esetén

5. Maszkolható vagy nem **maszkolható** megszakítások

a) Maszkolható: a megszakítást le lehet tiltani. Prioritás rendelhető hozzá és bizonyos esetekben a megszakítás nem lép érvénybe. Például: I/O kérés, debug, …
b) Nem maszkolható: pl.: súlyos hardverhiba (túlmelegedés), paritásbit jelzések

A megszakítás kiszolgálás általános folyamata:

Egy megszakítás segítségével egy adott program leállítható és tetszőleges program lefuttatása után folytatható.

Kiszolgálás előkészítése:

- Ha valamelyik egység megszakítás kérést bocsát ki, az aktiválja az INTR vezérlő vonalat
- A CPU befejezi az aktuális utasítást. A vezérlő minden utasítás töréspontban megnézi, van-e megszakítás kérés.

  (A processzoroknak van egy magszakítás bemenetük, ezen érzékelik a megszakítást

  Régebben ez egyszintű volt : van vagy nincs megszakítás

  Jelenleg megszakítás áramkör vezérli: tudja kezelni a prioritásokat

  ⟶ ez van rákötve a CPU INTR bemenetére)
- Ha igen:
- Első kérdés: elfogadható-e a megszakítás?
- Mikor juthat érvényre egy megszakítás? 3 feltétele van:

  Megszakítható az aktuális folyamat (program vagy megszakítás)

  Megfelelő a prioritás nagysága

  Az adott megszakítás nincsen maszkolva

- INTACK vezérlő vonal aktiválása a CPU által, ha a megszakítás el van fogadva
- a periféria erre INTR vonalat deaktiválja

A CPU feladata a megszakítás kiszolgálásakor eltárolni az adott folyamat kontextusát egy programtól független tárolóban. Betölti a PC-be a megszakítási rutin kezdőcímét és ha szükséges az állapotinformációkat is.

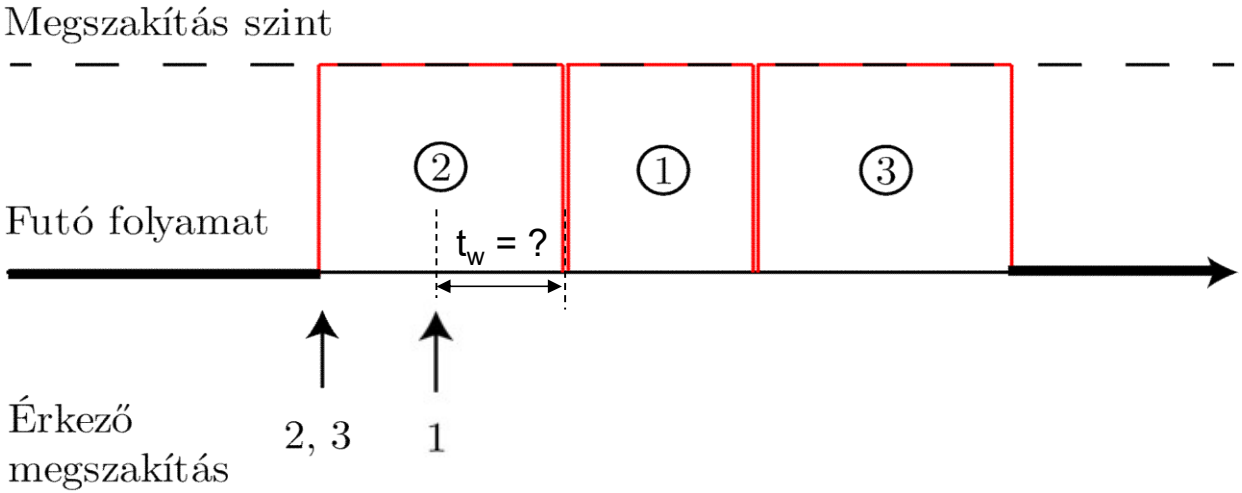
Különbség a közönséges program és megszakítás között, hogy a megszakított programtól teljesen függetlenül fut le egy folyamat.

Megszakítási rendszerek szintjei:

1. Egyszintű: A megszakítás nem megszakítható, vagyis, ha egy megszakítás létrejön addig nem kezdődhet másik megszakítás, amíg vissza nem térünk a normál állapotba.

Példa: legyen egy három szintű megszakítási rendszerünk.
Prioritási szintek: 1 > 2 > 3.

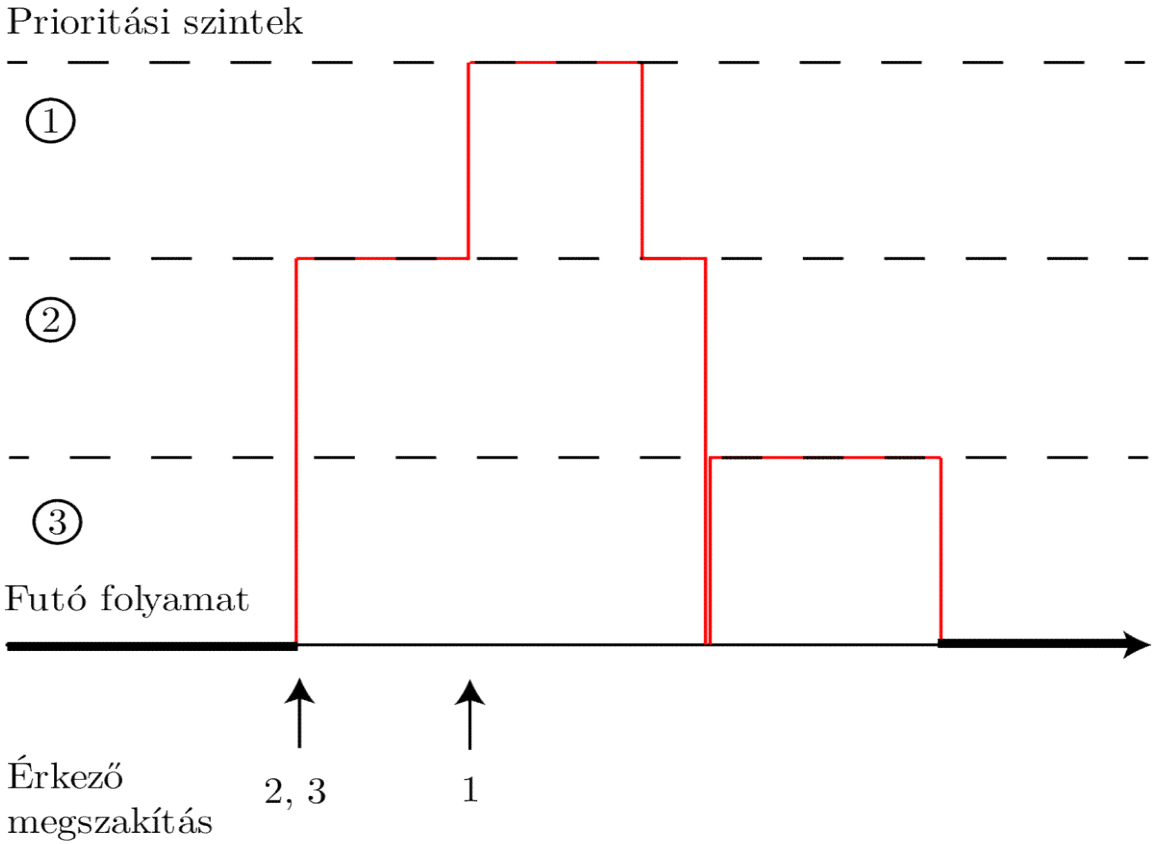
Egyszintű megszakítás:



Legnagyobb probléma az 1-es prioritású megszakítás érkezésének időpontja, mert ez a legnagyobb prioritású kérés, és mégis várakozni kényszerül.

Többszintűmegszakítási rendszer:

A megszakítások is megszakíthatóak, ezért fontossági sorrendben futnak le.



Hátránya: a valóságban nagyon sok megszakítás van, nem lehetséges mindegyikhez külön prioritási szintet rendelni.

Többszintű, többvonalú megszakítási rendszer:

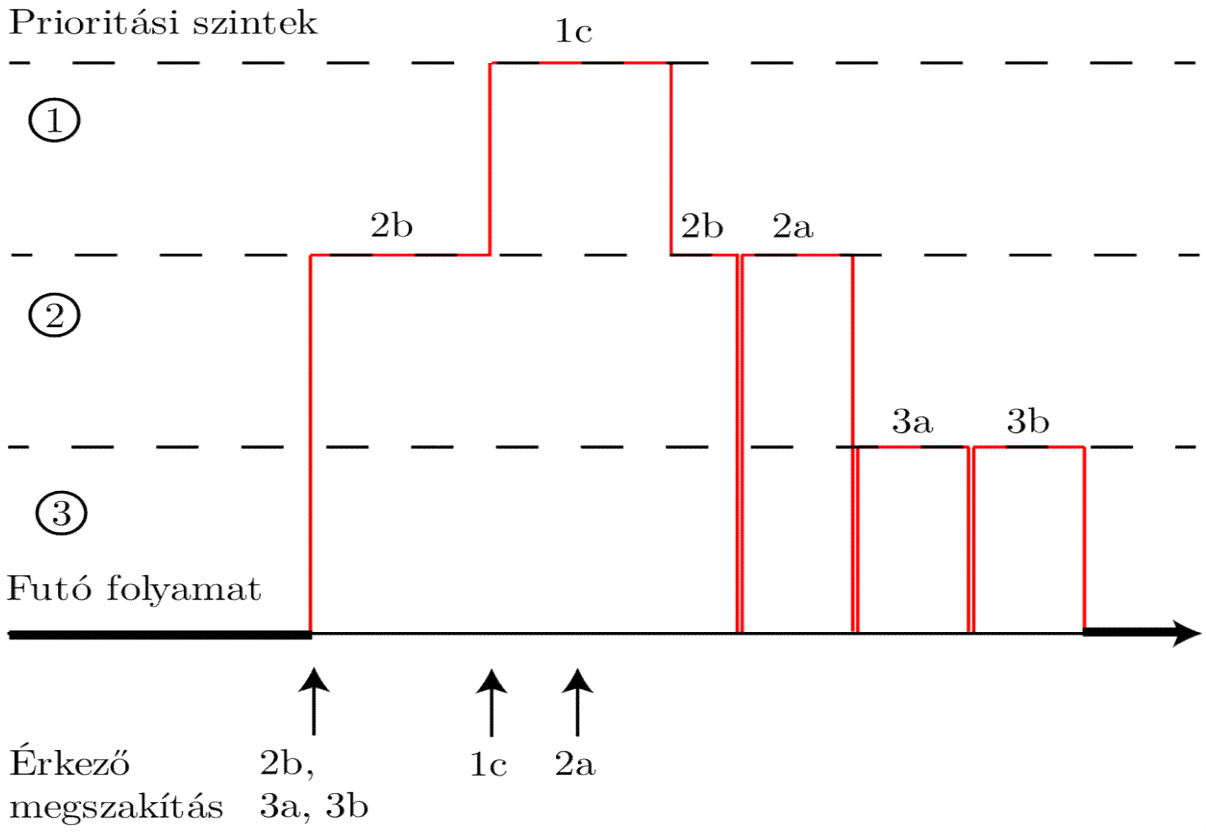
Elve, hogy a megszakítási forrásokat/okokat osztályozzák. A megszakítási osztályokhoz prioritási szinteket rendelnek.

Osztályok: 1, 2, 3

Prioritás: 1, 2, 3

Osztályokon belül megszakítás típusok: a, b, c, d ...

Osztályok között többszintű, osztályon belül egyszintű a megszakítás rendszer, de ezen belül is vannak prioritási szintek!



Gyorsító tárak (Cache)

A gyorsítótárak feladata az adatforgalom gyorsítása és egyenletessé tétele.

Önálló tároló szerepe nincsen, mindig az operatív tár bizonyos részeinek másolatát tartalmazza.

A cache működése transzparens, önállóan nem címezhető, így a kezelését a programozó helyett a hardver végzi!

Elhelyezkedése: legtöbbször a processzor lapkáján található.

Az adatátvitel a cache és a memória között mindig blokkos formájú!

Definíció: A cache az adatok és utasítások átmeneti tárolására szolgáló gyors működésű, a programozó számára **nem** elérhető tároló.

A cache elterjedése és fejlődése körülbelül 1980-tól kezdődött, előtte még terveztek processzorokat cache nélkül.

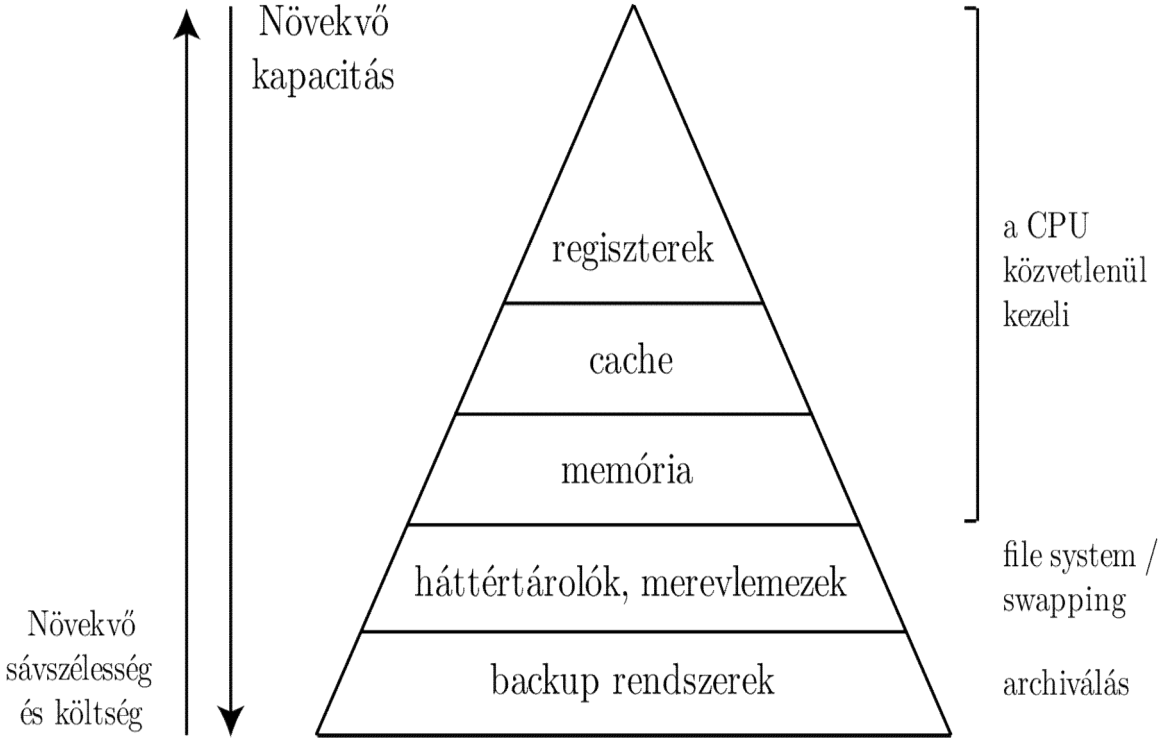
1980-tól 2000-ig körülbelül milliószorosára nőtt a processzorok teljesítménye,

míg a memória elérése csak százszorosára.

Ez tette szükségessé a gyorsítótárak fejlesztését.

ÓBUDAI EGYETEM

A tárolók piramisában a regiszterek és memóriák között helyezkedik el:



Felosztás:

A tapasztalat azt mutatta, hogy érdemes az utasításokat és az adatokat külön cache-ben tárolni, ezért a modern processzorok már ezt a felépítést alkalmazzák. Például az Intel már 1993-tól használ külön adat- és utasítás cache-t.

Okai:

- Az adatok és utasítások általában eléggé függetlenül kezelhetőek egymástól
- Ha egy programrészlet sok adattal dolgozik, akkor közös cache esetén előfordulhat, hogy a sok adat miatt kikerülnek a cache-ből a programhoz tartozó utasítások, ami lassuláshoz vezetne.
- Utasítás cache-nél csak az olvasást kell gyorsítani, míg adatoknál az írást és az olvasást is
  → eltérő technológiákkal lehet ezeket optimálisan kihasználni.

A modern, három szintű cache-t használó rendszerekben a következő típusú gyorsítótárak jelennek meg:

- Utasítás cache – Level 1
- Adat cache – Level 1
- Mixed cache – Level 2, 3

Cache szintek:

A cache tervezésének alapvető dilemmája, hogy el kell dönteni melyik a fontosabb:
a sebesség vagy a találati arány.
A gyorsítótárakban **az adatokat keresni kell!** ⟶ minél nagyobb a kapacitása,
annál több időbe telik a keresett adatot megtalálni.
Kompromisszumra van szükség: gyors legyen **ÉS** nagy legyen a kapacitás ⟶ Ellentmondás!
Ezért vezették be a többszintű cache-t.

Az L1 cache a regiszter után a leggyorsabb tároló, általában a CPU órajelén üzemel. Mérete kicsi, nagyságrendileg 2x32 vagy 2x64 kbyte. Késleltetése alacsony, körülbelül 1.3-1.5 ns (3-4 óraciklus). A késleltetések a többi cache szint között a következőképpen alakul:

	Elérési idő	
	nanosec	ciklus
L1	1.3-1.5 ns	3-4
L2	4.5-8	10
L3	12-20	20-40
RAM	60-80	50-200

Példák a többszintű gyorsítótárakra:

- Intel 80486: L1: 128 kB
- Intel P1/P2: L1: 128 kB L2: 512 kB
- Intel P4: L1: 256 kB L2: 1MB
- Intel Core 2: L1: 512 kB L2: 2-8 MB
- Intel Core i: L1: 2x32 kB L2: 256 kB L3: 3-8MB

Lényeges szempont a cache tároló és az operatív tár azonos részei tartalmának az egyezőségét biztosítani. Tehát ha egy, a cache-ben tárolt operandus értéke megváltozik, azt vissza kell írni az operatív tárba is!

Az adatátvitel a RAM és a cache között mindig blokkos formában történik, mivel nagy valószínűséggel az utasítások és adatok felhasználása is az egymás utáni tároló címekből történik. A processzor és a cache közötti adatátvitelt ezzel szemben byte szintű is lehet.

Fontos, hogy a cache kisebb mérete miatt mindig dönteni kell, hogy a memóriából mely adatokat töltse be a rendszer.

A cache-ben a memória egyes, egymást követő rekeszeinek **tartalmát tároljuk**, de ezek mellé el kell tárolni az **adatok memóribeli címét** is!

A memória címnek csak akkora részét kell tárolni a cache-ben, amelynek alapján közvetlenül (a tárolt értékből), vagy közvetve (a tárolt értékből és annak cache-beli helyéből, sorából - cache line) a blokk kezdő címe egyértelműen meghatározható.

A címnek azt a részét, amelyet a cache-ben elhelyez a CPU és ami alapján a kiválasztás történik, **TAG**-nek nevezzük.

Ez származhat fizikai vagy virtuális címből attól függően, hogy a cache a CPU és az MMU (Memory Management Unit), vagy az MMU és a RAM között helyezkedik el. Az első esetben a fizikai, a másodikban a virtuális címekből történik a tag származtatása.

A virtuális címek használatának hátránya:

- A tárolandó TAG nagyobb, mivel a virtuális címtér is nagyobb és hosszabbak a címek.
- A virtualizációból adódó helyettesítéseket kezelni kell.

Előny ugyanakkor, hogy a virtuális tag csökkenti a cache hiba késleltetést.

A virtuális tagelés hátrányai miatt általában a mai architektúrák fizikai TAG-eket használnak.

A visszakeresés módja az úgymond tartalom szerinti asszociatív keresés

(CAM – Content Address Memory), ami azt jelenti, hogy a vizsgált adatnak a cache-ben tárolt adattal való egyezőségét vizsgálja a CPU kiolvasáskor és kereséskor.

A vizsgálat a keresett adat **címének összehasonlítását** jelenti a cache-ben tárolt címekkel, vagy azok egy részével.

A cache akkor működik hatékonyan, ha a keresett adat a kiválasztások többségében a cache-ben és nem a memóriában található. Ha a keresett adat a cache-ben megtalálható, **cache hit-ről**, ellenkező esetben **cache miss-ről** beszélünk.

Cache hit: A találatok aránya függ a cache méretétől és szervezési módjától. A modern rendszerekben a találati arány közelít a 100%-hoz, az elvárt hibaarány 1-2%-os.

Cache miss: Ha a keresett adat nem található meg a gyorsítótárban, a CPU a RAM-ból olvas, viszont a regiszteren kívül a cache-be is betölti!

Helyettesítési stratégia:

A cache tároló tartalmának cseréjekor a találati arány fenntartása érdekében lényeges a megfelelő a helyettesítése stratégia (replacement policy) kiválasztása.

Alapértelmezés szerint a cache tele van (!!!), ezért, ha más adatokra lenne szükség, mint amit jelenleg tárol, bizonyos részeit ki kell cserélni. Ezeknek a cseréknek a módját határozza meg a replacement policy.

Típusok: FIFO – legrégebben betöltött blokkot írja felül

LIFO – legutoljára betöltött blokkot írja felül

LFU (Least Frequently Used) – legritkábban használt blokkot írja felül

LRU (Least Recently Used) – legrégebben használt blokkot írja felül

A cache-ben az adaton és a tagen kívül az adatok állapotára vonatkozó információt is tárolni kell.

Ezek a vezérlést és a helyettesítési eljárást kiszolgáló bitek.

A legfontosabb vezérlő bitek:

- **D (dirty) bit:** A dirty bit egy blokk valamely részének felülírását, módosítását jelzi. Az ilyen blokk helyére nem lehet újat betölteni, előbb a módosított adatokat ki kell írni az operatív tárba.

- **V (valid) bit:** a cache tartalmának érvényességét jelzi a cache sorra vagy blokkra vonatkozóan. Ha be van állítva, az adat a megadott című tárolóhelyhez tartozik és aktuálisan érvényes. Például törlés (flush cache line) esetén a V bit 0-ra lesz állítva, ezzel jelzi a CPU számára, hogy az adott terület szabadon írható. Minden sorhoz legalább egy V bit tartozik.

A cache-eket jellemző paraméterek:

- Méret: néhány kB – közel 100MB
- Elhelyezkedés: on chip – lapkán, off-chip – különálló
- Blokk méret: a fő tár és a cache között egy egységben mozgatott adatmennyiség. Általában 4-64 byte. Adatnál és utasításnál eltérő lehet.
- Sorméret (line size): az az adatmennyiség, amely egy összehasonlításnál maximálisan kijelölhető. Általában a blokk mérettel azonos, de kisebb is lehet.
- Helyettesítési stratégia (előző oldal)

- Adat aktualizálási módszer: két típust különböztetünk meg:

  write through – az adat változása esetén azonnal visszaírásra kerül az operatív tárba. Biztos az adatok tárolása, nem léphet fel inkonzisztencia a cache és memória között.

  write back – csak az adott cache line felülírása előtt aktualizál. Gyorsabb és gyakrabban használt, mert a módosításokat nem kell azonnal kiírni. Viszont áramszünet esetén adatvesztés léphet fel.
- Koherencia mechanizmus: ez a módszer, amely biztosítja a fő tár és a cache tárak tartalmának egyezőségét
- Átlagos elérési idő (average access time):

  AAT := „Hit time” + („Miss rate” x „Miss penalty”) ⟵ memóriához fordulás ideje

Cache Típusok

Teljesen asszociatív cache (full associative):

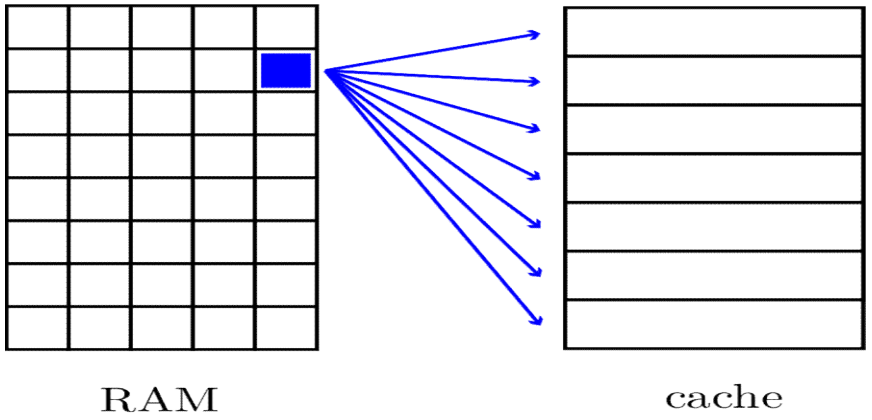
Egy beolvasott blokk a cache-ben bármelyik sorba kerülhet. Az elhelyezés sorát a helyettesítési stratégia határozza meg. Keresésnél a CPU a tageket vizsgálja minden sorban egyszerre, mivel az adat bárhol lehet.

Ennek megvalósításához szükséges **n darab** párhuzamos összehasonlító áramkör

⟶ drága, bonyolult és nagyobb fogyasztás.

Full associative cache:

Előnye a nagy találati arány és
a rugalmasság, mivel mindig
a legoptimálisabb elrendezése van.



Direct mapping (1 way set associative cache):

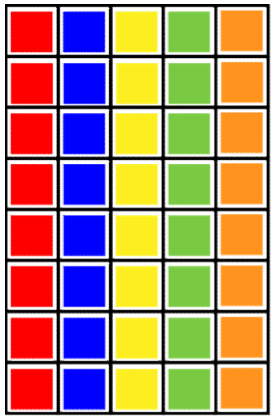
Ennél a megoldásnál minden memória blokk csak egy bizonyos cache line-ba tölthető.

A blokk helyét a TAG határozza meg. Mivel a cache line-hoz több memória blokk is hozzá van rendelve, előfordulhat, hogy gyakran cserélni kell a cache tartalmát, ami teljesítmény csökkenést okozhat.

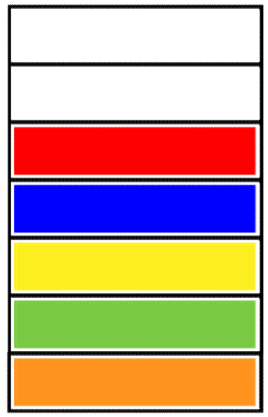
Így a megoldás rugalmatlan és kisebb a találati arány (hátrány).

Előnye viszont a gyors visszakeresés és az olcsó megvalósítás, mivel **csak egy darab** összehasonlító áramkör kell.

Direct mapping (1 way set associative cache):



RAM



cache

N-way associative cache:

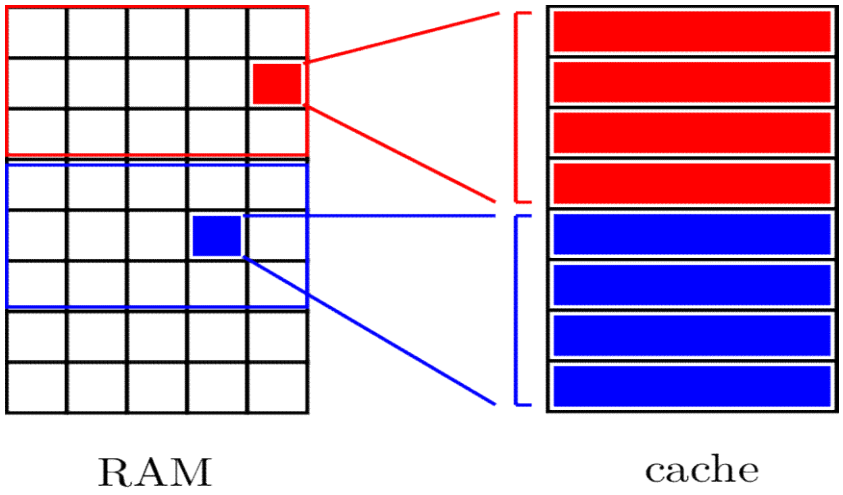
Az előző két módszer közötti kompromisszumot jelenti ez a módszer. N jellemzően 2, 4, 8 vagy 16. Az n változó meghatározza, hogy az adott blokk hány cache line-ba kerülhet.

Például egy 4 utas asszociatív cache-nél egy blokk 4 cache line-ba kerülhet. Eredménye, hogy a CPU a csoport index alapján 4 darab cache line-ra tudja szűkíteni a keresést, tehát csak 4 db összehasonlító áramkör kell.

Előny: rugalmasabb és nagyobb találati arányú, mint a direct mapping, valamint olcsóbb, mint a teljesen asszociatív, mert kevesebb áramkör szükséges.

Az Intel általában 8 utas asszociatív cache-t használ.

N-way associative cache:



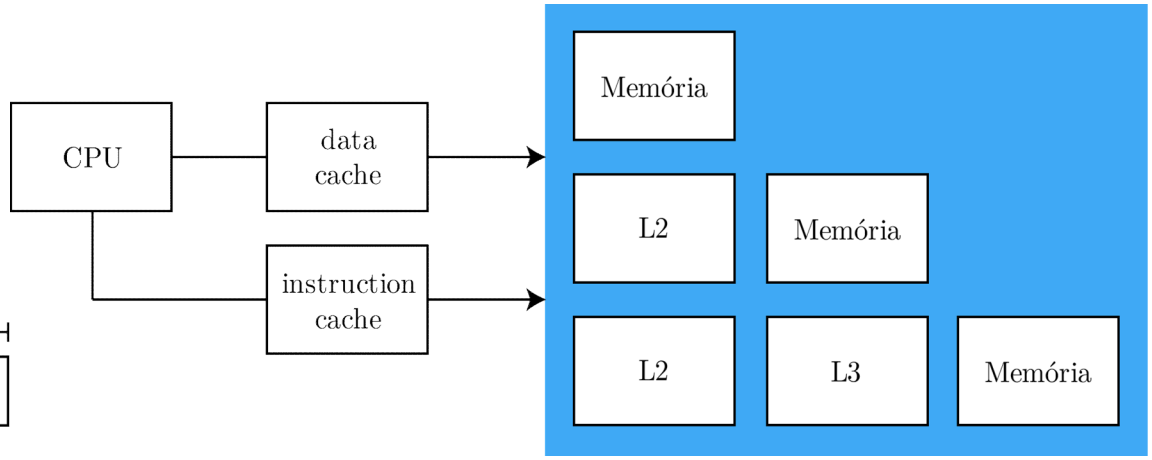
Sector mapping cache:

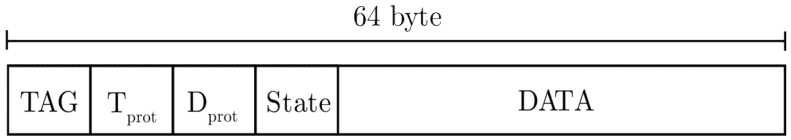
A gyakorlatban nagyon ritkán használt. Itt a csoport bárhová kerülhet, viszont a blokknak a helye a csoporton belül kötött.

Hierarchiák:

Mai rendszerekben szinten mindenhol megtalálható a data és az instruction cache (L1).
Innentől több felépítés lehetséges:

L3-at szokás LLC-nek is nevezni
(„last level cache”)





Cache line felépítése:
Például egy 64 byte-os cache line esetén a felépítés:

- Directory Entry: TAG és egyéb állapotjelzők
- Tprot: TAG protection bitek: biztosítják az tag egyezőséget az elírások ellen
- Dprot: data protection bitek: biztosítják az adat egyezőséget az elírások ellen
- State: állapot bitek (V bit, D bit, …)
- Data: maga az adat

Adatszervezési módok:

Többszintű cache-ek esetén megkülönböztetünk inclusive és exclusive gyorsítótárakat.

Exclusive cache:

Exclusive cache esetében a tárolók egymástól függetlenek, nem tartalmazzák egymás adatait, így ugyanaz az adat nem lehet jelen egyszerre több tárban. Több CPU vagy mag esetén már komplikált.

Az adatok betöltése kétféleképpen történhet:

- Először L3-ba és ha szükséges akkor onnan L2-be, L1-be.
- Először az L1-be töltjük be az adatokat, majd, ami nem fér bele, azt az L2-ba, és ami oda se, azt L3-ba. Ilyenkor nevezzük az L3-at victim cache-nek is (ez a jellemzőbb).

Intel „smart cache”: L3-at dinamikusan megosztja a magok között

Inclusive cache:

Inclusive cache-eknél a magasabb szintű tároló tartalmazhatja kisebb szintű tároló adatait.

Hátránya, hogy kétszer kell írni és csökken az alacsonyabb szintű cache mérete.

Előnye:

A magasabb szintű cache sora szabadon cserélhető, mert az alacsonyabban is megtalálható az adat, ahonnan majd kiírásra kerülhet.

Több mag esetén, amikor a másik mag cache-ében kell keresni az adatot, a duplikálás miatt elég az alacsonyabb szintűben (pl.: L2-ben).

Mivel manapság elég nagy méretű cache-eket használnak, az előnyök miatt inkább az inclusive cache-t alkalmazzák főleg többmagos processzoroknál. (L1, L2)

Viszont a közösen használt cache-ek esetén a cache koherencia biztosítása bonyolultabb inclusive cache esetén, ezért L3 jellemzően nem tartalmazza L2-t.

Data prefetch logic:

További gyorsítási lehetőség, általában az L2-ben található. Az L1 cache adatelérési sémáit elemzi, és ha szekvenciális lekérést talál, akkor előre behúzza az adatokat a memóriából L2-be vagy L3-ba. Tulajdonképpen előre gondolkodik, így a vezérlés és biztosítani tudja, hogy az adatok legalább 99%-a rendelkezésre álljon a gyorsítótárban, amikorra azokra szükség van.

Cache koherencia mechanizmus:

Több CPU-s vagy többmagos rendszerek esetében mindig figyelni kell arra, hogy az egyes CPU-k (vagy magok) gyorsítótáraiban megegyező adatok legyenek!

Meghatározza azt a módszert, amellyel biztosítani lehet a főtár és a cache tárak tartalmának egyezőségét!

Fontos a gyorsítótár egyezőség fenntartása több CPU vagy mag esetén. Ezek megoldására léteznek az alábbi cache koherencia protokollok:

- Snoopy
- Snorf
- Könyvtár alapú
- MESI
- MOESI

Cél, hogy a módosított adat a lehető leggyorsabban bekerüljön az összes processzor gyorsítótárába, mielőtt a többi esetlegesen műveletet végezne rajta. Az adat változás érvényesítése kétféleképpen történhet:

- Invalidáció: érvényteleníti az összes cache tárban az adott cache line-t azáltal, hogy a V bitjét 0-ra állítja.

- Write back alapú rendszerekben előfordulhat, hogy a helyes eredmény még nem íródott vissza az operatív tárba! Ezért nem elég invalidálni, hanem el kell kérni a helyes adatot a másik CPU-tól vagy magtól.
- Felülírás az új állapottal:

  a CPU vagy mag direktben küldi el a változott adatot a többi CPU-nak vagy magnak, mely felülírja az adott cache line-t.

  A gyakorlatban inkább az invalidációt alkalmazzák, mivel egy invalidálási üzenet sokkal kisebb, mint egy teljes cache line tartalma, így nem generál akkora forgalmat (nem terheli a belső buszt), viszont biztosítani kell a visszaírást is.

Koherencia protokollok:

Snoopy protokoll:

A vezérlés folyamatosan figyeli, „szaglássza” a közvetítő réteget olyan tranzakciók után, melyek hatással vannak önmagára. Erre minden CPU vagy mag saját vezérlése figyel. Például, ha egy cache olyan írási művelettel találkozik, aminek a TAG-je olyan, amit ő is tárol, azonnal invalidálja azt a cahce line-t és elkéri az aktuális adatot a másik magtól.

Előnye az egyszerű kiépíthetőség.

Hátránya, hogy sok CPU vagy mag esetén terheli a buszrendszert.

Könyvtár alapú protokoll:

A megosztott adatok egy közös könyvtárban vannak elhelyezve. A könyvtár egy szűrőként működik, amelyen keresztül a processzornak engedélyt kell kérnie, hogy betölthessen egy adatot az operatív tárból. Ha a bejegyzés változik, akkor a könyvtár vagy felülírja, vagy invalidálja a többi cache tartalmát.

Ezzel a megoldással nincs szükség külön buszra. Elsősorban NUMA rendszerekben használatos.

MESI protokoll:

A **Modified Exclusive Shared Invalid** rövidítése. Állapotjelzőket vezet be:

- Modified: **egy** tár valid, a többi invalid ⟶ egyik tárban módosult az adat, de még nem került visszaírásra ⟶ ez az egy szabadon módosítható!
- Exclusive: érvényes az adat ⟶ egyezik a fő tárral és csak itt van és nem módosult még (nyugodtan módosítható, nem kell máshol frissíteni)
- Shared: az adat a fő tárral egyezik, de több tárolóban is megtalálható
- Invalid: az adat (blokk) érvénytelen

MOESI protokoll:

„Modified - Owned – Exclusive - Shared - Invalid”

Hasonló az előzőhöz, egy kivétellel:

- Owned: van egy ötödik „tulajdonban lévő” állapot, amely módosított és megosztott adatokat is képvisel. Ezzel elkerülhető, hogy a módosított adatokat visszaírjuk a fő memóriába a megosztás előtt.

  Ahhoz, hogy ez lehetséges legyen, lehetővé kell tenni az adatok közvetlen gyorsítótár-gyorsítótár közötti átvitelét, így a módosított állapotú adatokkal rendelkező gyorsítótár el tudja juttatni ezeket az adatokat egy másik olvasóhoz anélkül, hogy azokat a memóriába továbbítaná.

  az Owned állapot lehetővé teszi a processzor számára, hogy a módosított adatokat közvetlenül a másik processzornak továbbítsa. Ez akkor előnyös, ha a kommunikáció két CPU között lényegesen jobb, mint a memóriával.

  (pl.: többmagos CPU-k, magonkénti L2 gyorsítótárral)

Két cache esetén a gyorsítótár-sorok megengedett állapotai a következők:

	M	O	E	S	I
M	✗	✗	✗	✗	✓
O	✗	✗	✗	✓	✓
E	✗	✗	✗	✗	✓
S	✗	✓	✗	✓	✓
I	✓	✓	✓	✓	✓

MESIF protokoll:

„Modified - Exclusive - Shared – Invalid - Forward”

- Forward: hasonló az Owned-hez de itt egy nem módosult (tiszta) cache line van kinevezve arra, hogy ha egy másik CPU, vagy mag igényli ezt a blokkot, akkor nem a memóriából tölti be, hanem innen!

Lehet a kettőt kombinálni is (MOESIF)

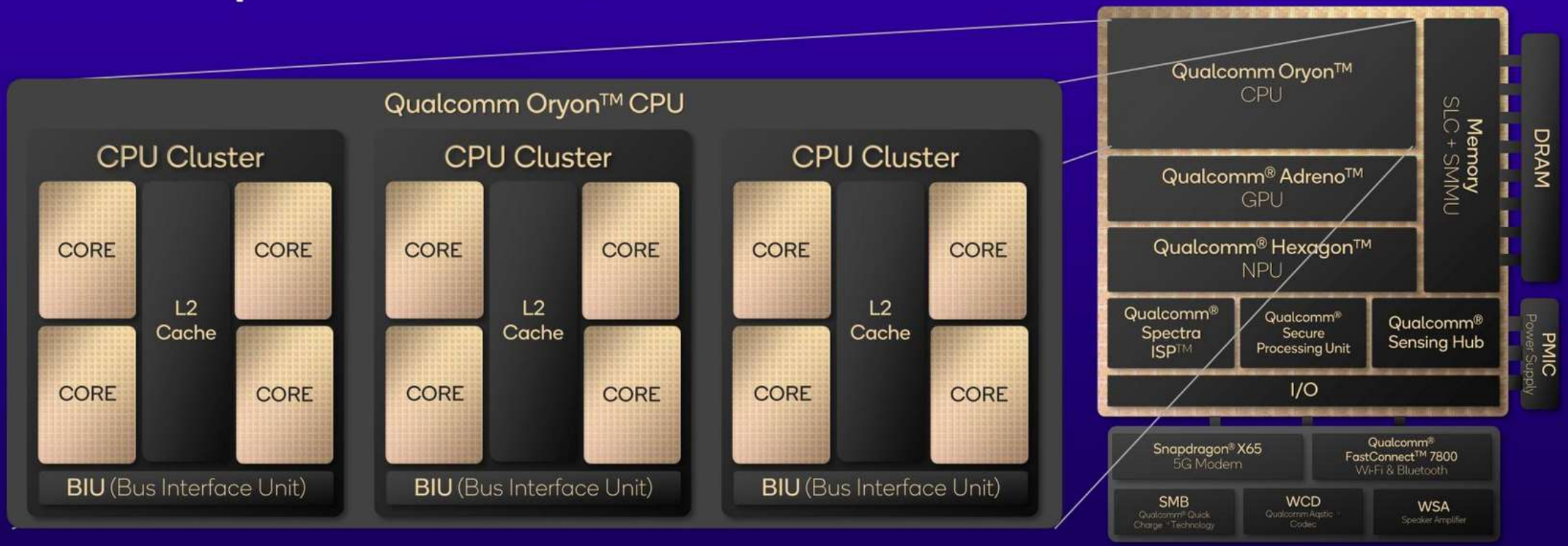
	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

Példa az egyik legújabb CPU architektúrára:

Qualcomm Oryon CPU:

Eltér az Intel megoldásaitól, CPU mag klasztereket használ (3 x 4 mag).
Nincsenek teljesítményre és hatékonyságra optimalizált magok!
Minden klaszter külön órajelezhető és akár le is kapcsolható a terhelés függvényében.
Egy klaszter (4 mag) osztozik egy 12 MB-os L2 cache-en (nem magonként van, mint az Intelnél ~ 1MB / mag) és ez a cache egy 12 utas asszociatív cache (összesen 36 MB).
Az L3 cache mindössze 6 MB, viszont gyors! (victim cache).
A gyorsítótárak koherenciáját MOESI protokoll segítségével tartják fenn .
Az L1 utasítás cache mérete 192KB (háromszor akkora, mint az Intelnél), 6 utas asszociatív.
Így ciklusonként akár 16 utasítást tud lekérni (Fetch), és 8 utasítást tud dekódolni ciklusonként (Intel 6-ot, AMD 4-et jellemzően).
Az L1 adat cache 96 KB (kb. kétszer akkora, mint az Intelnél), 6 utas asszociatív.
650 bejegyzéses várakoztató állománnyal rendelkezik! Ebből választ a vezérlés, hogy mely utasításokat küldi ki végrehajtásra.
800 db átnevezési regiszterrel és 14 futószalaggal rendelkezik.

12 brand-new custom CPU cores power the SoC



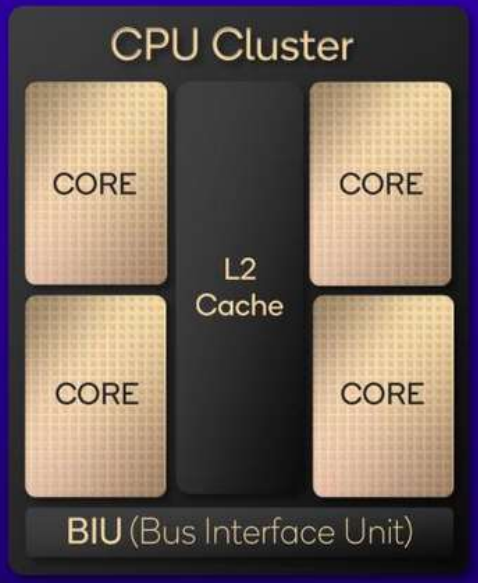
Shared L2 Cache

Shared L2-Cache per CPU Cluster

- Fully coherent 12MB 12-way shared cache
- MOESI protocol
- Operates at full Core frequency
- Full 64B reads, writes, evictions, and fills to/from L1 caches
- L1 Inclusion policy for energy efficient operation

Multiple Outstanding Low Latency Requests

- Optimized for L1 data accesses to L2
- Average latency of 17 cycles for L1 miss to L2 hit
- Over 50 in-flight requests per Core into the system,
  - And over 220 memory transactions in-flight in the L2Cache.
- Optimized snoop operations
  - Core-to-Core and Cluster-to-Cluster



ÓBUDAI EGYETEM

Köszönöm a figyelmet!