Korszerű számítógép architektúrák I.

Készítette: Simon Péter [1]

2021. május 16.

[1] Hallgatói jegyzet Durczy Levente előadásai alapján

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. A tárgy célja

A tárgy célja a párhuzamos számítógép architektúrák ismertetése.

1.2. Történeti áttekintés

Az iparban először a Ford T-modell gyárása során alkalmazták a párhuzamos végrehajtást (futószalag). A számítástechnikában is hasonlóan működnek a párhuzamos architektúrák. Fő cél a fejlesztés során a feldolgozás gyorsítása, ezért a nagy számítógépeknél már a 60-as években megjelent a párhuzamos végrehajtás. A mikroszámítógépekben ugyanez a 80-as években történt meg.

1.3. A párhuzamosság csoportosítása

1.3.1. Funkció szerint

- Rendelkezésre álló párhuzamosság (feladatokban rejlő párhuzamosítási lehetőségek).
- Kihasználható párhuzamosság (valóban használható párhuzamosság).

1.3.2. Típus szerint

- Adat párhuzamosság.
- Funkcionális párhuzamosság.

1.3.3. Elhelyezkedés szerint

- Időbeli párhuzamosság (futószalag): Ford T-modell gyártósor analógia szerint az egyik helyen fényeznek, másik helyen beszerelik a motort egyidőben.
- Térbeli párhuzamosság (több, azonos típusú végrehajtó egység egyidejű működése): előző analógia szerint két helyen két különböző autót fényeznek egyszerre.

1.4. Adat párhuzamosság

Hasznosítása kétféleképpen lehetséges:

- Adatpárhuzamos architektúrákkal (adatelemeken párhuzamos, vagy futószalag elvű műveletek végrehajtása).
- Funkcionális párhuzamossággá alakítással (adatelemeken történő műveletek ciklus formájában történő megfogalmazása).

1.5. Funkcionális párhuzamosság

A funkcionális párhuzamosság a feladat logikájából következik. Az architektúrák, operációs rendszerek és a fordítóprogramok is igyekeznek kihasználni.

1.5.1. Szintjei

Különböző szinteken értelmezhetjük (alacsonytól magas felé haladva):

- Utasítás szintű párhuzamosság (ILP - Instruction Level Parallelism): program utasítások párhuzamos végrehajtása.
- Ciklus szintű párhuzamosság: egymást követő iterációk párhuzamos végrehajtása (függőségek akadályozhatják).
- Eljárás szintű párhuzamosság: mértéke leginkább a feladat jellegétől függ.
- Program szintű párhuzamosság: egymástól független programok párhuzamos végrehajtása.
- Felhasználó szintű párhuzamosság: több, egymástól független felhasználó kiszolgálása (pl. szerverek, időosztásos rendszerek).

1.5.2. Kihasználása

Utasítás szintű párhuzamosság esetén

- Utasítás szinten párhuzamos architektúrákkal.
- Erre a célra szolgáló compiler segítségével.

Ciklus- és eljárás szintű párhuzamosság esetén

Szálak (folyamatok) segítségével. A szál vagy folyamat a tárgykód legkisebb önállóan végrehajtható része. Szálakat létrehozhat:

- Programozó, párhuzamos nyelveket használva (fork, join...).
- Operációs rendszer, ami támogatja a többszálas végrehajtást.
- Párhuzamos fordítóprogram.

Szemcsézettségi szintek

A funkcionális párhuzamosság kihasználásához különböző szemcsézettségi szintek rendelhetők. A finom szemcsézettségtől a durváig haladva a párhuzamossági szintek:

- utasításszint
- szál szint
- folyamat szint
- felhasználói szint

A finom szemcsézettségű párhuzamosság alacsony szinten, közvetlenül kihasználható. A magasabb szintű (durvább szemcsézettségű) párhuzamosság kihasználásához nem elég a hardver és a compiler támogatása, hanem OS-ek alatti konkurens végrehajtás szükséges.

1.6. Compilerek

A compilerek a magas szintű nyelveken írt programokat fordítja le a processzor által értelmezhető kódra.

1.6.1. Feladatai

- analizálni a forrásnyelvű program karaktersorozatát,
- szintetizálni (létrehozni) a tárgykódot.

1.6.2. Analízis részei

- Lexikális elemzés (konstansok, változók, operátorok).
- Szintaktikai elemzés.
- Szemantikai elemzés.

1.6.3. Szintézis részei

- Kódgenerálás (assembly kód vagy gépi kód).
- Kód optimalizálás (pl. független részek keresése párhuzamos végrehajtáshoz).

1.7. Párhuzamos architektúrák osztályozása

1.7.1. Klasszikus modell

A klasszikus osztályozási modell J. Flynn amerikai mérnök nevéhez kötődik. Ez a rendszer a vezérlő és a feldolgozási egységek számán alapul. Az osztályozáshoz bevezetett új fogalmak:

- SI (Single Instruction stream) - egyszeres utasításfolyam: az architektúra egy vezérlőegységgel rendelkezik.
- MI (Multiple Instruction stream) - többszörös utasításfolyam: az architektúra több utasításfolyamot tud egyidőben végrehajtani.
- SD (Single Data stream) - egyszeres adatfolyam: egy CPU egy adatfolyamot dolgoz fel.
- MD (Multiple Data stream) - többszörös adatfolyam: több végrehajtó egység egy időben dolgoz fel több, egymástól független adatfolyamot.

Ezek alapján a következő architektúra osztályok határozhatók meg:

- SISD (Single Instruction Single Data): Neumann modell, szekvenciális végrehajtás.
- SIMD (Single Instruction Multiple Data): multimédiás feldolgozás, ugyanazon műveletek végrehajtása sok adaton.
- MISD (Multiple Instruction Single Data): elméleti, nem használják a gyakorlatban.
- MIMD (Multiple Instruction Multiple Data): teljes párhuzamos feldolgozás.

A Flynn modell egyszerű és átlátható, de nem utal a párhuzamosság fajtájára, szintjére és módjára.

1.7.2. Új modell

Az új osztályozás megkülönböztet adatpárhuzamos és funkcionálisan párhuzamos architektúrákat.

Adatpárhuzamos architektúrák

Az adatpárhuzamos architektúrák típusai:

- verktor
- asszociatív (neurális)
- SIMD
- szisztorikus

A gyakorlatban leggyakrabban a SIMD architektúrákat használják, a tárgy is csak erre tér ki.

Funkcionálisan párhuzamos architektúrák

A funkcionálisan párhuzamos architektúrák típusai:

- utasítás szinten párhuzamos (ILP)
  - futószalag (időbeli párhuzamosság)
  - VLIW (térbeli+időbeli párhuzamosság)
  - szuperskalár (térbeli+időbeli párhuzamosság)
- szál szinten párhuzamos (SMT - Simultaneous multithreading), pl. Intel Hyper Threading
- folyamat szinten párhuzamos
  - elosztott memória használatú
  - közös memória használatú

1.8. Történeti áttekintés

A desktop gépekben először a 80-as években jelent meg a párhuzamosság, futószalag elvű processzorokkal. Ezek voltak az első ILP processzorok, skalár CPU-nak is hívták őket. A szuperskalár processzorok a 90-es évek elején jelentek meg, majd később kiegészültek multimédiás kiterjesztéssel. A szuperskalár processzorok dinamikus utasítás ütemezéssel rendelkeztek, így elérték a sorrenden kívüli kibocsátást és a kibocsátási párhuzamosságot. A 2000-es évektől a fejlődésnek két iránya jelent meg: evolúciós és revolúciós.

Evolúciós fejlődés: a logikai architektúra változatlan, viszont a feldolgozási szélesség 32-ről 64 bitre nőtt. Ennek eredménye az x86-os architektúra bővülése x86_64-re.

Revolúciós fejlődés: ez egy teljesen új logikai architektúrát és utasításkészletet eredményezett. Ez az Intel és a HP által kifejlesztett VLIW elvű, IA64 architektúra.

Eredmény: a fejlődés eredményeként megjelentek a többmagos processzorok, a mai napig ez az irány a meghatározó.

1.9. Utasítások felbontása

A processzor által végrehajtott utasítások négy elemi műveletre oszthatók:

- Fetch
- Decode
- Execute
- WriteBack

1.10. Kibocsátás és végrehajtás

A kibocsátás a processzor dekódoló egységéből történik, ezeket a kibocsátott utasításokat hajtják végre a végrehajtó egységek. Ha a CPU képes párhuzamosan több utasítás végrehajtására, a kibocsátási kapacitást is növelni kell. Ha a CPU a dekódoló egységéből óraciklusonként 1-nél több utasítás kibocsátására képes, megvalósul a kibocsátási párhuzamosság.

Következmények:

- Az utasítások párhuzamos végrehajtása során minden ILP CPU-nak figyelembe kell vennie az utasítások között fennálló függőségeket.
- Meg kell őrizni a soros végrehajtás konzisztenciáját (a programozó soros végrehajtású programot ír, ennek párhuzamos végrehajtás esetén is ugyanúgy kell viselkednie).

2. fejezet

Függőségek

2.1. Bevezetés

Függőség alakul ki, ha egy programban az egymást követő utasítások függenek egymástól. A függőségek gátolják a párhuzamos végrehajtást. Következmény pl. a futószalag processzoroknál, hogy a teljesítmény nem növelhető egy bizonyos ponton túl a fokozatok növelésével a függőségek és az egyes lépéseknél felmerülő időveszteségek miatt.

2.2. Típusai

- adat
- vezérlés
- erőforrás

2.3. Adat függőségek

Probléma: az utasítás végrehajtásához egy előző utasítás eredményére van szükség.

2.3.1. Csoportosítása

Jellege szerint

- utasítás szekvenciában (lineáris feldolgozás)
  - valós függőség - nem teljesen megszüntethető (RAW - Read After Write)
    - műveleti adatfüggőség
    - behívási adatfüggőség
  - ál függőség - teljesen megszüntethető
    - WAR - Write After Read
    - WAW - Write After Write
- ciklusban

Operandus típusa szerint

- regiszter
- memória

2.3.2. Műveleti adatfüggőségek

Probléma felvetés: feltélezzük, hogy

- a processzor 3 operandusos utasításokat használ
- 4 fokozatú futószalagos végrehajtás van (Fetch, Decode, Execute, WriteBack).

Ezekkel a feltételekkel két számot szeretnénk összeszorozni, az eredményt pedig megduplázni. Az utasításaink:

```
; I1
MUL r3, r2, 1      ; r3 = r1 * r2
; I2
SHL r3             ; r3 * 2
```

Az utasítások végrehajtásának időbeli sorrendje:

	t_1	t_2	t_3	t_4	t_5
I_1	F_{MUL}	$D_{r1,\ r2}$	E_{MUL}	W/B_{r3}	
I_2		F_{SHL}	D_{r3}	E_{SHL}	W/B_{r3}

A probléma, hogy I_2 végrehajtása során, a dekódolási fázisban (t_3 időpillanat) szükség lenne az r3 regiszter értékére, viszont az csak t_4 időpillanatban áll elő (I_1 végrehajtásának writeback fázisában). Tehát a futószalagos módszerrel párhuzamosított végrehajtás során műveleti adatfüggőség keletkezett, mivel az utasítások lehívása és végrehajtása között átfedés van. Ilyenkor a műveletek elakadnak.

Megoldás: egy speciális utasítás, a NOP (No Operand) használata az alábbi módon:

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F_{MUL}	$D_{r1,\ r2}$	E_{MUL}	W/B_{r3}			
I_2		F_{SHL}	NOP	NOP	D_{r3}	E_{SHL}	W/B_{r3}

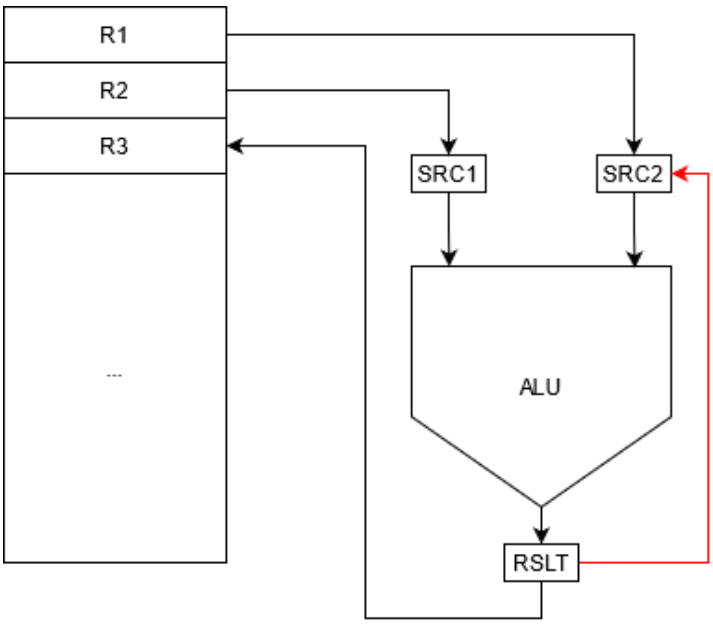
Következmény: a műveleteknek várakozniuk kell egymásra, két óraciklus késés keletkezik a futószalagon. Az ezeket követő utasításokhoz is be kell szúrni két NOP-ot, mivel a dekóder foglalt. Ezt a jelenséget teljesen nem lehet megszüntetni, viszont a fékező hatást csökkenthetjük.

Kezelés: operandus előrehozásával csökkenthető a fékező hatás, ez viszont extra hardvert igényel (hardveres, azaz dinamikus megoldás). Extra hardver nélkül csak szoftveresen, azaz statikusan kezelhetjük a problémát. Ilyenkor a compiler oldja meg a függőségek kezelését. Általában előnyösebb a dinamikus megoldás.

Dinamikus megvalósítás: az ALU-hoz tartozó rejtett regisztereket és az adatutakat a 2.1 ábra mutaja. Alapesetben a MUL utasítás végrehajtása során az adat az r_1 és r_2 regiszterekből az src_1, illetve src_2 regiszterekbe kerül, majd a művelet elvégzése után a rslt rejtett regiszteren keresztül visszaírásra kerül r_3-ba. Az adatút rövidítésének érdekében, extra hardver segítségével az rslt regiszter tartalmát közvetlenül visszavezethetjük az ALU egyik forrásregiszterébe. Ennek útja látható az ábrán pirossal. Ekkor az utasítások végrehajtása az alábbi módon valósul meg:

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F_{MUL}	$D_{r1,\ r2}$	E_{MUL}	W/B_{r3}			
I_2		F_{SHL}	D	E_{SHL}	W/B_{r3}		

Mivel már t_3 időpillanatban is rendelkezésre áll az SHL utasítás operandusa, két óraciklussal hamarabb kezdhető meg a művelet végrehajtása. Ezzel megszüntettük a késést. Ezt a megoldást minden modern CPU használja.

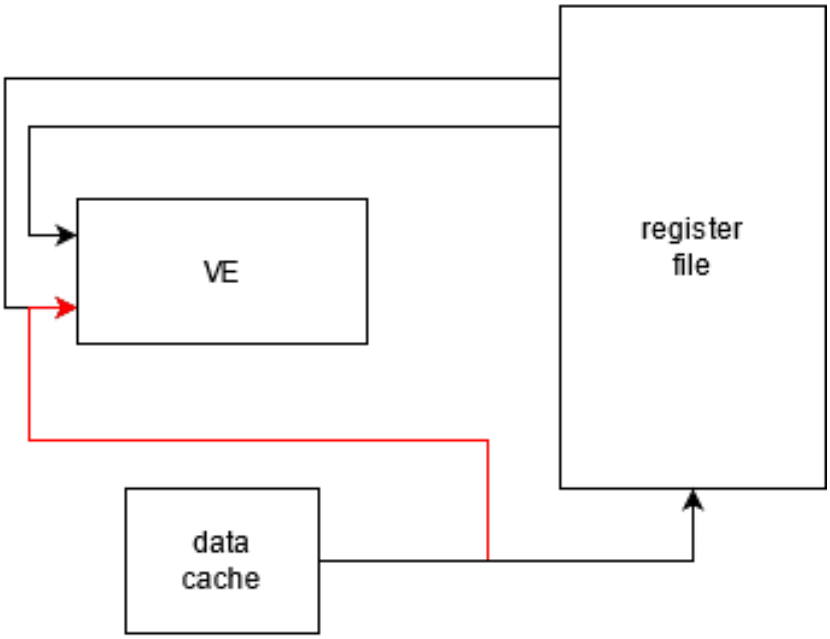


2.1. ábra. Az eredmény visszavezetése a forrás regiszterbe

2.3.3. Lehívási adatfüggőség

Probléma: a regiszterekbe az operatív tárból (cache) töltjük be a szükséges adatokat, majd ezután a regiszterekből hívja le a végrehajtó egység (ALU). A cache elérése viszont sok időt vesz igénybe. Ennek látható az általános adatútja a 2.2 ábrán, fekete vonallal jelölve.

Kezelés: a folyamat gyorsítására extra hardvert alkalmazunk, amivel a cache-ből történő lehíváskor egyúttal a végrehajtó egységbe is betöltjük az adatot (piros vonal). Így egy óraciklust megspórolhatunk.



2.2. ábra. A lehívott adat bevezetése a műveletvégző egységbe

2.3.4. WAR - Write After Read

Probléma felvetés: egy MUL utasítást egy ADD követ az alábbi módon:

```
; I1
MUL r3, r2, 1     ; r3 = r1 * r2
; I2
ADD r2, r4, r5    ; r2 = r4 + r5
```

A szorzás (MUL) sokkal lassabb, mint az összeadás (ADD), ezért előfodulhat, hogy a párhuzamos végrehajtás során I_2 hamarabb lefut, mint hogy I_1 betöltse a forrás operandust. Mivel I_2 módosította I_1 bemeneti operandusát, a MUL utasítás hibás eredményt fog adni. Következménye, hogy sérül a szekvenciális konzisztencia.

Megoldás: r_2 tartalmát egy ideiglenes regiszterbe irányítjuk (pl. r_{23}). Ekkor az assmebly utasítások így néznek ki:

```
; I1
MUL r3, r2, 1     ; r3 = r1 * r2
; I2
ADD r23, r4, r5   ; r23 = r4 + r5
```

Az $r_{23} \rightarrow r_2$ hozzárendelést nyilvántartjuk, majd amikor a MUL utasítás végzett, visszaírjuk r_{23} tartalmát r_2-be. Az átmeneti (átnevezési) regiszterek tulajdonságai:

- új, önálló, de rejtett,
- saját címtartománnyal rendelkezik,
- a programozó számára traszparens,
- extra hardvernek számít.

Megjegyzés: a regiszterkészletek csoportosítása:

- architekturális: programozó használja,
- átnevezési: a vezérlés használja az álfüggőségek feloldására.

2.3.5. WAW - Write After Write

Probléma felvetés: egy MUL utasítást egy ADD követ az alábbi módon:

```
; I1
MUL r3, r2, 1     ; r3 = r1 * r2
; I2
ADD r3, r4, r5    ; r3 = r4 + r5
```

A szorzás (MUL) sokkal lassabb, mint az összeadás (ADD), ezért előfodulhat, hogy a párhuzamos végrehajtás során I_1 később fut le, mint I_1. Mivel az eredményt ugyanabba a regiszterbe írják, ebben az esetben I_1 felülírja I_2 eredményét az r_3-ban. Ezzel sérül a szekvenciális konzisztencia.

Megoldás: r_3 átirányítása egy átnevezési regiszterbe, az előbb leírt módon.

2.3.6. Ciklusbeli függőség

Probléma felvetés: egy ciklusban az előző iterációban kiszámolt adatot használunk fel, például:

for $i = 2$ to n **do**
  $X_i \leftarrow A_i * X_{i-1} + B_i$
end for

Kezelés: ez egy erős függőség, hardveresen nehezen feloldható. Megoldás az algoritmus áttervezése.

2.4. Vezérlés függőségek

Elágazások esetén léphetnek fel. Itt a statikus és dinamikus kezelésnek eltérő jelentése van, mint az adatfüggőségeknél. A statikus kezelés itt egy állandó, mindig alkalmazható eljárást jelent, míg a dinamikus kezelés az adott programtól függ.

2.4.1. Feltétlen ugrásnál

Probléma felvetés: az alábbi utasítássorozatban a feltétlen ugrás (JMP) egy SHL utasításra mutat:

```
DIV
MUL
JMP ; SHL-re mutat
ADD
...
SHL
```

Ekkor a kritikus utasítások így követik egymást időben:

	t_1	t_2	t_3	t_4	t_5	t_6
MUL	F_{MUL}	D	E	W/B		
JMP		F_{JMP}	D	E	W/B	
ADD			F_{ADD}	D	E	W/B

A JMP utasítás az Execute fázisban állítja át a Program Countert, ezzel végzi el az ugrást. A futószalag végrehajtás miatt azonban ekkorra már a következő utasítás, az ADD is lehívásra került, sőt, előfordulhat, hogy az azt követő utasítás is. Ezek viszont fölösleges lépések. Ritkább esetekben a JMP-t követő utasítás be is fejeződhet, mire az ugrás végrehajtásra kerül, ami veszélyezteti az architekturális regisztertartalmakat.

Megoldás: a probléma kezelése statikus, dinamikus, vagy spekulatív (branch prediction) módon történhet.

Kezelés utasítások átrendezésével (dinamikus): compiler segítségével történő optimalizálás. A compiler megpróbálja átrendezni az utasítások sorrendjét. Az előző kódrészlet optimalizált változata:

```
JMP ; SHL-re mutat
MUL
DIV
ADD
...
SHL
```

Az optimalizálás utáni végrehajtási sorrend:

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
JMP	F_{JMP}	D	E	W/B			
MUL		F_{MUL}	D	E	W/B		
DIV			F_{DIV}	D	E	W/B	
SHL				F_{SHL}	D	E	W/B

A sorrend megváltoztatásával elértük, hogy amíg az ugrás végre nem hajtódik (t_3), csak olyan utasításokat hívunk le, amiknek még az ugrás előtt kell lefutniuk. Mire az ADD utasításhoz elérnénk, felülíródik a PC és a megfelelő utasítás hívódik le (SHL). A módszer hátránya, hogy hatékonysága a futószalag fokozatok számának növelésével rohamosan csökken.

Kezelés NOP utasításokkal (statikus): a JMP utasítás mögé egy vagy több NOP utasítás kerül be. Ez a futószalag várakoztatását jelenti, amíg elő nem áll az ugráshoz szükséges PC. Ez az ún. ugrási buborék, nagysága $n-1$, ahol n a futószalag fokozatok száma.

2.4.2. Feltételes elágazásnál

Kezelése csak dinamikusan, a végrehajtás során történik (spekulatív elágazáskezelés - branch prediction). A feltételtől függ az, hogy ugrás vagy soros folytatás következik.

2.5. Erőforrás függőségek

Akkor lép fel, ha több utasítás ugyanazt az erőforrást használná. Ilyenkor várakoztatni kell az egyiket. Erőforrások lehetnek például regiszterek, pufferek, végrehajtó egységek, stb.

Példa: a logikai futószalagok különböző célokra dedikált végrehajtó egységekben vannak megvalósítva. Ilyen pl. a lebegőpontos vagy a fixpontos végrehajtó egység. Ha sok olyan utasítás van, ami lebegőpontos végrehajtást igényel, előfordulhat, hogy a lebegőpontos végrehajtó egységnél sorban állnak az utasítások, míg a fixpontos kihasználatlanul várakozik.

Megoldás: úgy kell tervezni a processzort, hogy ez ne okozzon szűk keresztmetszetet. Ezt az erőforrások többszörözésével érhetjük el. Fontos szempont a hatékonyság, 70-80%-os kihasználtság az általános. A mai processzorokban magonként kb. 6 végrehajtó egység van.

2.6. Szekvenciális (soros) konzisztencia megőrzése

2.6.1. Soros konzisztencia típusai

- utasítás feldolgozás soros konzisztenciája
  - utasítás végrehajtás soros konzisztenciája (processzor konzisztencia)
  - memória hozzáférés soros konzisztenciája (memória konzisztencia)
- kivételkezelés soros konzisztenciája
  - pontatlan kivételkezelés
  - pontos kivételkezelés

2.6.2. Processzor konzisztencia

Probléma felvetés: párhuzamos végrehajtás esetén az alábbi assembly kódban előfordulhat, hogy az ADD utasítás hamarabb lefut, mint a DIV.

```
; I1
DIV r3, r2, r1
; I2
ADD r5, r6, r7
; I3
JZ cimke
```

Mivel a JZ utasítás mindig a legutoljára végzett utasítás eredményét használja fel a feltételes ugrás eldöntéséhez, ha I_1 később végez, mint I_2, JZ a DIV eredménye alapján fog ugrani, ami hibás működéshez vezethet.

Megoldás: a hardvert úgy kell tervezni, hogy ilyen hiba ne fordulhasson elő.

2.6.3. Pontatlan kivételkezelés (gyenge konzisztencia)

Probléma felvetés: tegyük fel az alábbi kódrészletben, hogy az ADD túlcsordul, de a MUL még nem végzett.

```
;I1
MUL r3, r2, r1
;I2
ADD r5, r6, r7 ;tulcsordul -> INT
;I3
JZ cimke
```

Az ADD utasítás túlcsordulása miatt megszakítást kell kezelnünk, ilyenkor a processzor a regiszterek állapotát (program kontextust) elmenti egy verem regiszterbe. Miután a kivételt lekezeltük, a veremből visszatöltődik a kontextus és folytatódik a végrehajtás. Ebben az esetben viszont nem fogjuk tudni, hogy a MUL utasítás végzett-e már, így az r_3 regiszter definiálatlan állapotba kerül. Ez hibákhoz vezethet.

Megoldás: a korai szuperskalár architektúráknál (első Pentium CPU-k) megoldatlan volt a probléma, pl. kék halált is okozhatott. A probléma megoldása a pontos kivételkezelés.

2.6.4. Pontos kivételkezelés (erős konzisztencia)

Minden mai CPU a megszakítás kéréseket csak az utasítások eredeti sorrendjében fogadja el. Az előző példában a processzor csak akkor fogadja el a megszakítás kérést, ha a MUL utasítás végzett. Megvalósítása történhet

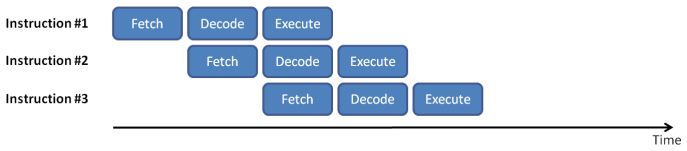
- átrendező puffer segítségével (pl. Intelnél ROB - ReOrder Buffer) vagy
- címkézéssel.

3. fejezet

Időbeli párhuzamosság: futószalag CPU-k

3.1. Bevezetés

A futószalagos (pipeline) végrehajtás lényege, hogy egy utasítást több részre osztunk (általában fetch, decode, execute, writeback), majd ezeket a részeket külön, egymással párhuzamosan hajtjuk végre (3.1 ábra). Az utasítások n részre osztásával elméletileg a sebesség n-szeresére növekszik.



3.1. ábra. Futószalagos végrehajtás

A teljesítmény gátjai: a gyakorlatban nem mindig valósul meg a fokozatok számának növekedésével arányos gyorsulás. A végrehajtást a függőségek (adat, vezérlés, erőforrás) lassítják. A függőségek oka a sok párhuzamosan futó utasítás ($n+1$, ahol n a fokozatszám).

A hatékonyság maximalizálása: a tapasztalat szerint a hatékonyság kb. 15-30 fokozatú futószalag esetén maximalizálható, efölött a függőségek miatt már csökken a teljesítmény. Ez az általános célú alkalmazásokra igaz, a mai általános processzorokban kb. 20 fokozat van. Speciális feladatokra (ahol kevés a függőség) használható superpipeline CPU, ami akár 200 fokozatú is lehet.

3.2. Történeti áttekintés

- Intel 80486: 3 fokozat, de már külön lebegőpontos futószalag
- Intel Pentium (P5): 5 fokozat
- Intel Pentium III (P6): 11-17 fokozat
- Intel Pentium IV (Netburst): 20-31 fokozat
- Intel Core 2 (újratervezett P6, több mag): általában 14 fokozat
- Intel Core i: 16-20 fokozat

3.3. Gyakorlati példa - az Intel Atom CPU

Az Intel Atom processzor a 2000-es években jelent meg, 16 fokozatú futószalagot használ. A processzor CISC (Complex Instruction Set Computing) architektúrájú, azaz egy utasításon belül nem csak a regiszterekből, hanem a memóriából, vagy a gyorsítótárból is képes adatot lehívni.

Az Intel Atom fokozatai:

- 1-3. fokozat: instruction fetch (IF)
- 4-6. fokozat: instruction decode (ID)
- 7-8. fokozat: instruction dispatch (SC - Switch Context, IS - Instruction Schedule)
- 9. fokozat: source operand read (IRF - Instruction Register File)
- 10-12. fokozat: data cache access, CISC architektúrához szükséges (AG - Address Generation, DC_1 - Data Cache 1, DC_2 - Data Cache 2)
- 13. fokozat: execute
- 14-15. fokozat: exception + multitask handling (FT_1 - Fault Tolerant 1, FT_2 - Fault Tolerant 2)
- 16. fokozat: commit, ez a visszaírás (W/B vagy DC_1)

Következmény: ez egy tisztán futószalag elvű processzor, ami teljesítményben visszalépést jelentett a korábbi architektúrákhoz képest. Előnye az alacsony fogyasztás, ezért főleg mobil eszközökbe használták az Atom CPU-kat.

3.4. Futószalagos feldolgozás előfeltételei (2 fokozat esetén)

Az ideális futószalag megvalósítható, ha

- a számítógép 2 db egymástól független végrehajtó egységgel rendelkezik,
- az egyik fokozat kimenete a másik fokozat bemenete,
- mindkét fokozat végrehajtási ideje azonos,
- a fokozatok szinkronizáltak, órajelre kapják az inputot, és egyetlen óraciklus alatt elvégzik a feladatukat.

Ekkor $t = \frac{T}{2}$, ahol T a szekvenciális végrehajtási idő és t a futószalagos végrehajtási idő.

3.5. Függőségek kezelése

- Operandus előrehozással:
  - Minden architektúránál használják.
  - Részletesen: 2. fejezet.
- Újrafeldolgozással:
  - Leggyakrabban az execute fokozat egymás után többszöri végrehajtását jelenti.
  - Pl. szorzásnál az ismétlődő összeadásokhoz használható.
  - A futószalag feldolgozást lassítja, de összességében jobb teljesítményt biztosít.

3.6. Típusai

1. Előlehívás (overlapping)
2. Vektor CPU-k (60-as évek)
3. Teljes pipeline

3.6.1. Előlehívás

A visszaírás során történik meg a következő utasítás lehívása:

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F	D	E	W/B			
I_2				F	D	E	W/B

Előnyök:

- 4 óraciklus helyett csak 3 kell egy utasításhoz, így a teljesítmény 25%-al nő, valamint
- nincsenek függőségek, mivel a forrás operandus beolvasásakor (t_5) már megvan az előző utasítás eredménye (t_4).

Hátrány: nem túl nagy mértékű gyorsulás.

3.6.2. Vektor CPU

Csak az execute fokozat működött futószalagszerűen.

3.6.3. Teljes pipeline

A futószalagos feldolgozás kiterjesztése a teljes folyamatra:

	t_1	t_2	t_3	t_4	t_5
I_1	F	D	E	W/B	
I_2		F	D	E	W/B

3.7. Logikai futószalagok

Az eltérő utasítások eltérő felépítésű futószalagokat igényelnek, ezért egy processzor több futószalagot is tartalmaz. A cél a funkcionális kialakítás. Példák különböző funkciókat ellátó futószalagokra:

- aritmetikai: F, D, E, W/B
  - fixpontos
    - egyszerű: +, -, léptetés, ...
    - összetett: *, /, ...
  - lebegőpontos
- ugró (branch): F, E
- LOAD / STORE

Az utasítások értelmezése: az utasításokat két szinten értelmezhetjük, pl. a fetch utasítás két szintje a következő:

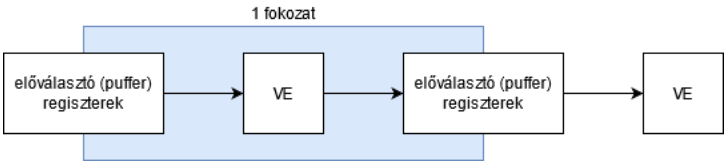
1. Fetch
2. MAR ← PC
   MDR ← [MAR]
   IR ← MDR
   PC ← PC+1

3.8. Fizikai megvalósítás

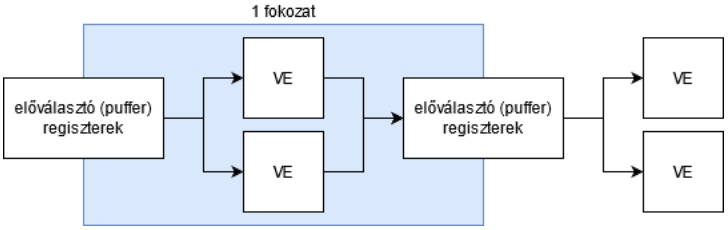
Alkalmazásuk alapján megkülönböztetünk univerzális és dedikált futószalagokat. Az univerzális minden művelet elvégzésére alkalmas, míg a dedikált speciális műveletekre képes. Hardveres szempontból az univerzális futószalag előnytelen, mivel sok tranzisztorra van szükség, a kialakítása bonyolult és drága, ráadásul a végrehajtás lassú. Ezért általában a 3.7. részben leírt dedikált (egy adott funciót ellátó) futószalagokat építenek a processzorokba. Az eredmény kevesebb logikai kapu, így gyorsul a végrehajtás (a bemenettől a kimenetig gyorsabban átérnek az elektronok).

Megjegyzés: a futószalag sebességét általában a leglassabb fokozat sebessége határozza meg, tehát a tervezési cél a megközelítőleg azonos sebességű fokozatok létrehozása.

Fokozatok kialakítása: a fokozatok előtt előválasztó (puffer) regiszterek vannak. Ezek a felhasználó számára láthatatlanok, az adat először ezekbe töltődik be. Ezekből a regiszterekből kerül aztán a végrehajtó egységbe az adat, majd az utasítás elvégzése után szintén puffer regiszterekbe kerül a kimenet. A puffer regiszterek szükségesek, mivel a gyakorlatban egy fokozat nem mindig végez egy óraciklus alatt. Az ebből adódó várakozás során ezekben a regiszterekben tárolódik az adat. Ennek a kialakítása látható a 3.2. ábrán. A későbbiekben a végrehajtó egységekből egymás mellé többet is helyeztek, így megvalósítva a térbeli párhuzamosságot (3.3. ábra).



3.2. ábra. A fokozatok felépítése

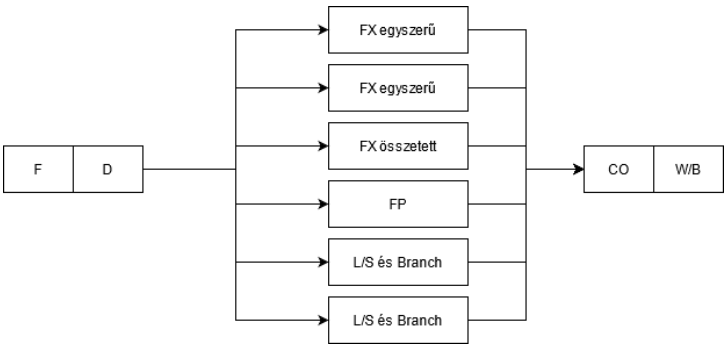


3.3. ábra. A végrehajtó egységek párhuzamosítása

3.8.1. Példa: a PowerPC 604

A PowerPC 604-es processzorban egymással párhuzamosan több dedikált futószalag is működött (3.4 ábra). A fetch és decode fokozatok minden óraciklusra lehívtak egy utasítást, és a megfelelő futószalagba töltötték. Így a CPU képes volt egymással párhuzamosan több utasítást is végrehajtani (akár 4-et is). Mivel előfordulhatott, hogy a később lehívott és betöltött utasítás végzett hamarabb (pl. elsőként egy FP, másodikként egy FX utasítás → FX hamarabb végez), szükséges volt egy konzisztencia fokozat (CO) bevezetése. Az utasítások címkézésre kerültek, a sorrendet a CO biztosította. A párhuzamosság miatt szükség volt a fokozatok közötti várakoztatásra, ez az interlock funkció.

Megjegyzés: az Intel 80486 és az első Pentium csak 2 utasítás futószalaggal rendelkezett. A mai Core i architectúrák magonként általában 6-8 futószalagot alkalmaznak.



3.4. ábra. A PowerPC 604 futószalagja

3.9. RISC és CISC architektúrák

Az utasításkészlet (tervezési stratégia) alapján kétféle architektúrát különböztetünk meg:

- RISC: Reduced Instruction Set Computing - csökkentett utasításkészletű architektúra és
- CISC: Complex Instruction Set Computing - bővített utasításkészletű architektúra.

Mindkettő használatban van napjainkban.

3.9.1. Történeti áttekintés

Az első CPU-k kevés utasítással rendelkeztek (RISC), majd a 70-es években az egyre több funkció és bonyolultabb utasítások miatt ez a szám növekedett (CISC). A 80-as években rájöttek, hogy a sok utasítás ugyan megkönnyíti a programozást, de a címzés bonyolultsága miatt káros hatással van a teljesítményre. Ez vezetett a RISC architektúrák újbóli megjelenéséhez.

3.9.2. RISC

Példa: a mobil eszközök ARM (Advanced RISC Machine) processzorai RISC architektúrájúak.

Tulajdonságai:

- Kis számú (50-150) utasítással rendelkezik → címzési módok egyszerűsödése.
- Nincs olyan utasítás, ami a LOAD/STORE-t aritmetikával kombinálja (nem lehet egyszerre betölteni az adatot és végrehajtani a műveletet).
- Minden műveletvégző utasítás regisztereket használ → memóriából vagy gyorsítótárból nem lehet dolgozni.
- Memória vagy cache elérés csak LOAD/STORE utasításokkal történhet.
- Nagy számú regiszterkészlet (mivel minden művelethez regiszterekre van szükség).
- Általában 3 operandusos utasítások → az eredmény nem írja felül a bemeneti regisztert, hanem külön regiszterbe kerül.
- Minden utasítás hossza egyforma (pl. 128 bit) → könnyebb a futószalagos feldolgozás.
- A fordítóprogramok bonyolultabbak a kevés utasítás miatt.
- Általában huzalozott (hardveres) az utasítás feldolgozás (decode).
- Utasítás végrehajtás általában egy óraciklust vesz igénybe (cél az egyforma ciklusidő).

Előnye: általában gyorsabb végrehajtás a CISC architektúrákhoz képest.

Hátránya: a bonyolultabb feladatokat instrukció szekvenciákkal kell megoldani. Ez a fordításnál okoz problémát, növelheti a program méretét.

3.9.3. CISC

Példa: a 80-as években elterjedt Intel 80386-os egy tisztán CISC architektúra.

Tulajdonságai:

- Nagy számú utasításkészlet (több száz).
- A sok utasítás nagy belső mikroprogramtárat igényel.
- Sokféle címzési mód (tartalmaz típus címzési módot is) és sokféle utasítás.
- Változó méretű (akár összetett) utasítások → a dekódolónak nem csak dekódolni kell az utasítást, hanem azonosítani is az utasítás végét (tudnia kell, hogy hol fejeződik be). Ezt hívják utasítás határra illesztésnek, plusz hardvert és időt igényel.
- Közvetlen memória elérés lehetsége → a második operandus lehet memória vagy cache cím is.
- Két operandusos utasítások → az első operandus felülíródik az eredménnyel.
- Az előző kettőből következik, hogy az első operandus nem lehet memória/cache cím, mivel az eredmény memóriába írása nagyon lelassítaná a működést.
- Az utasítások feldolgozása több ciklusidő lehet → bonyolultabb feldolgozás.
- Egyszerűbb a gépi kódú programozás a sokféle utasítás miatt (egyszerűbb fordítóprogramok).
- Egy utasításban több elemi műveletet is végre tud hajtani.
- Az utasítások folyamatosan bővültek, így a régi programokkal visszafelé kompatibilis maradt.
- A futószalag fokozatok között sebesség különbség lehet → feloldására interlock funkciót használnak (részletesen: 3.8.1).
- Általában a memória elérés miatt +2 fokozat szükséges a RISC-hez képest (AG - címszámítás és cache elérés).

Előnyei: kompatibilitás a régi programokkal, egyszerűbb compilerek.

Hátránya: bonyolultabb, lassabb végrehajtás.

3.9.4. Hibrid CISC

Példa: az x86-os architektúra ISA-t (Instruction Set Architecture) használ, ami egy hibrid CISC architektúra.

Tapasztalat: megfigyelték, hogy a CISC processzorok az idő 80%-ában az utasítások mindössze kb. 20%-át használják.

Optimalizálás: a feldolgozás gyorsítása érdekében kialakítottak a CISC architektúrán belül egy RISC magot. Ez a megoldás minden mai (Core i) architektúrában megjelenik.

3.9.5. Hibrid RISC

A mai ARM processzorok sem tisztán RISC architektúrák, hanem CISC jellegű utasításokkal vannak kibővítve (pl. az ARM Thumb, ami egy tömörített utasításkészlet).

3.10. Következmények

A futószalagos feldolgozás következményei:

- Jelentősen felgyorsult az utasítások lehívása és az operandusok betöltése.
- Amíg a processzorok feldolgozási sebessége jelentősen nőtt, a memória sebessége kevésbé (ez a jelenség a sebességolló) → cache bevezetése (IBM - 1968, Intel - 80-as évek). A cache előnye, hogy a gyakran használt operandusok gyorsan elérhetők.
- Maximális végrehajtási sebesség: 1 utasítás / ciklus (további növekedés kibocsátási párhuzamossággal vagy utasításon belüli párhuzamossággal lehetséges).
- A vezérlés átadási utasítások kifinomult technikája szükséges (függőségek miatt).
- Az elágazás kezelés bonyolódik.

3.10.1. Elágazások kezelése

Korai RISC gépek

Kezelés ugrási buborékkal.

Korai CISC gépek

A dekódoló fokozatba építették a címszámító és a logikai komparáló egységet, így a dekódolási ciklus végére előáll az ugrási cím.

Későbbi CISC gépek (futószalag CPU-k és első generációs szuperskalár architektúrák)

Kezelés fix előrejelzéssel, pl. mindig ugrik. Ezzel a megoldással az ugrási cím előre kiszámításra kerül és megkezdődik az ugrási címen lévő utasítások lehívása. Ha mégse kell ugrani, az utasításokat visszatörli és az eredeti helyen folytatódik a végrehajtás. Előnye, hogy kiküszübölte az ugrási buborékot, növekszik a teljesítmény. Korlátja, hogy ha nagy látenciával rendelkező műveletet kellett végrehajtani az ugrási feltételben, az blokkolta a kibocsátást. Ilyen megoldást használ például az Intel 80486-os CPU.

Második generációs szuperskalár architektúrák

A CPU a dekódolási folyamatok egy részét már akkor elvégzi, amikor az utasításokat az L1 cache-be írja. Az előre elvégzett feladatok:

- utasítás típusazonosítás
- ugrások felismerése
- utasításhossz meghatározása (szükséges, mivel CISC-nél változó az utasításhossz)
- spekulatív elágazásbecslés
  - regiszterátnevezéssel
  - átrendező puffer (Intel: ROB - ReOrder Buffer)

3.11. Összegzés

A fejezetben felsorolt módszerekkel elérték a futószalag elvű processzorok teljesítményének korlátait. A további gyorsítás érdekében párhuzamos kibocsátást kellett alkalmazni, így jöttek létre a szuperskalár processzorok.

4. fejezet

Szuperskalár architektúrák (párhuzamos kibocsátás)

4.1. Bevezetés

A futószalag architektúrájú processzorok óraciklusonként csak egy kimenetet tudtak előállítani, ami korlátozta a teljes CPU sebességét. Erre jelent megoldást a szuperskalár architektúra (1990 környékén jelent meg), aminek alkalmazásával elérhető a párhuzamos kibocsátás. A szuperskalár architectúrákat első, második és harmadik generációra oszthatjuk. A második és harmadik generációban már multimédiás (utasításon belüli párhuzamosság) képességek is megjelentek.

4.2. Közös jellemzőik

- A dekódoló egységből képes óraciklusonként több utasítást kibocsátani (ennek száma a kibocsátási ráta, első generációnál 2-3, másodiknál 4-6).
- Időbeli és térbeli párhuzamosság (több futószalag párhuzamosan).
- Maguk küzdenek meg a függőségekkel (dinamikusan, extra hardverek segítségével, pl. adattípusonként külön regisztertár).
- Kompatibilitás: evolúció, azaz kompatibilis marad a korábbi futószalag architektúrákkal. Így a régi programok is futhatnak rajta.

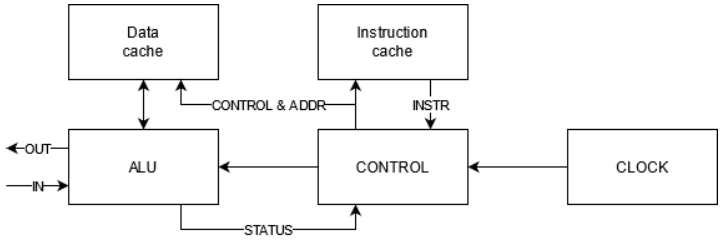
4.3. Harvard architektúra

1944-ben írták le, lényege, hogy az adat és a programkód fizikailag elkülönített útvonalakon mozog. Következménye, hogy párhuzamos adatutak jönnek létre, így növekszik a teljesítmény. A szuperskalár architektúrák az L1 cache-nél alkalmaznak egy módosított Harvard architektúrát, ahol az utasítás és az adat külön tárolódik. Az egyik eltérés az eredeti Harvard architektúrától, hogy képes programkódot is adatként betölteni. Az L2 és L3 gyorsítótárakban megosztott tárhely van az adatok és utasítások részére. Ezekből is látható, hogy a mai processzorok tervezésénél felhasználják mind a Harvard, mint a Neumann elveket.

4.3.1. Vezérlési vázlat

A Neumann elvű processzoroknál egy óraciklus alatt vagy adat, vagy utasítás lehívása történhetett meg. A Harvard architektúráknál ezzel szemben egy vezérlőegység (CONTROL) hívja le az utasítást az instruction cache-ből (INSTR adatút), majd ez alapján jelet küld a data cache-nek, hogy az ALU-ba milyen címen lévő adat kerüljön (CONTROL&ADDR). Ezzel egyidőben az instruction cache felé is küld CONTROL&ADDR jelet, tehát a következő órajelre az adat és a következő utasítás egyszerre hívódik le. A CONTROL egység felel az ALU irányításáért is, az ALU az IN és OUT adatutakon kommunikálhat

a perifériákkal, a STATUS adatúton pedig visszacsatolást biztosít a vezérlőegység számára. Az egész működést a CLOCK által kibocsátott órajel irányítja. Ennek a vázlata látható a 4.1. ábrán. Ezt a felépítést használják a modern processzorok is.



4.1. ábra. Harvard architektúra vezérlési vázlata

4.3.2. Előnyei

- Képes párhuzamosan adatot és utasítást olvasni, vagy írni cache nélkül is.
- Az adat és utasítás tárolók különálló címtartománnyal rendelkeznek, amikben a címek különböző hosszúak is lehetnek (pl. utasítás címek 32 bit, adat címek 64 bit).

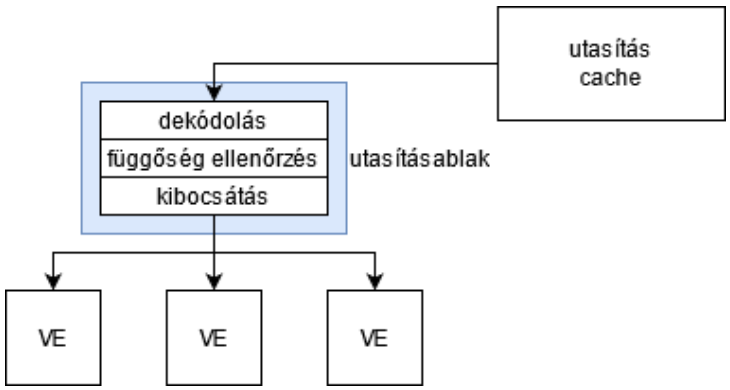
4.4. Első generációs szuperskalárok (keskeny szuperskalárok)

4.4.1. Jellemzői

- Közvetlen, vagyis nem pufferelt kibocsátás: a CPU a dekódolt utasítást közvetlenül küldi a végrehajtó egységhez.
- Statikus elágazásbecslés. Az egyik ilyen megoldás a fix (mindig ugrik) elágazásbecslés, a másik lehetőség a programkód bizonyos tulajdonságai alapján létrehozott statikus elágazásbecslés. A statikus elágazásbecslést a fetch alrendszer végzi, így nincs ugrási buborék.
- Gyorsítótár a memória lassúságának kiküszöbölésére, itt már megjelentek a kétszintű gyorsítótárak is: L1 cache a processzorlapkán, L2 különálló lapkán.
- Különálló adat és utasítás az első szintű gyorsítótárban (ütközik a Neumann elvekkel, mivel az utasítás és az adat külön operatív tárban tárolódik → Harvard architektúra).
- Az operatív tár és az L2 cache közös az adat és utasítás számára → Neumann architektúra.
- Az előző két pontból következik, hogy az L1 cache elérése során Harvard elvűként, a memória és az L2 cache elérésekor pedig Neumann elvűként működik a processzor. Ezt a megoldást használják ma is, pl. az x86-os architektúra.

4.4.2. Utasításablak

Az utasításablak egy olyan puffer, amely az óraciklusonként kibocsátott utasításokat tartalmazza (4.2. ábra). Az utasításablak pufferből kerül feltöltésre, utasítás kibocsátáskor kiürül. Itt történik a dekódolás és a függőség ellenőrzés. A végrehajtható utasítások kibocsátásra kerülnek, tehát közvetlenül a végrehajtó egységbe kerülnek. A végrehajtható utasítás olyan utasítás, aminek nincs függősége. A függőséggel rendelkező utasítások addig maradnak az utasításablakban, amíg a függőség meg nem szűnik.



4.2. ábra. Az utasításablak működése

Utasítás pótlásának módjai:

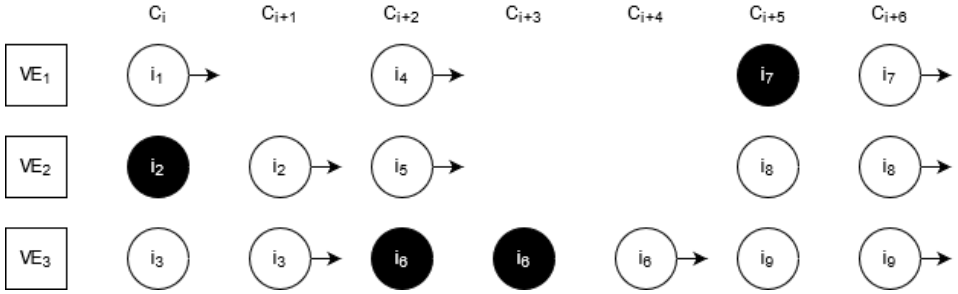
- A kibocsátott utasításokat egyenként pótoljuk.
- A kibocsátott utasításokat egyszerre pótoljuk (megvárjuk, hogy az összes utasítás kiürüljön).

Utasítás végrehajtás és kibocsátás módjai:

- sorrendben
- sorrenden kívül

Alkalmazása szuperskalár processzorokban: az első generációs szuperskalár architektúrák az utasításablakot egyszerre töltötték fel és sorrendi utasítás kibocsátást alkalmaztak. A sorrendi kibocsátás hátránya, hogy egy függő utasítás a függőség megszűnéséig blokkolja a kibocsátást, így a végrehajtó egységek kihasználatlanul álltak. Ez volt az első generációs szuperskalárok legnagyobb problémája.

Példa egyszerre pótló és sorrendben kibocsátó utasításablakra: 3 végrehajtó egységünk van, az utasításablak kezdeti tartalma pedig két független és egy függőséggel rendelkező utasítás. Jelöljük körrel a független utasításokat és teli körrel a függő utasításokat. Az adott óraciklusban kibocsátott utasításokat jobbra mutató nyíl jelzi. Az utasításablak tartalma a 4.3. ábrán látható. Az első óraciklusban (C_i) az I_2 utasításnak függősége van, így még nem bocsátható ki, a sorrendiség miatt pedig I_3 sem. I_2 és I_3 a második óraciklusban (C_{i+1}) kerül kibocsátásra. Ezután új utasításokat hívunk le (I_4, I_5 és I_6). I_4-et és I_5-öt rögtön ki is tudjuk bocsátani, mivel nincs függőségük, és a sorrendiség miatt I_6 függősége se akadályozza őket. I_6 függőségének feloldására viszont két óraciklust kell várnunk, ezért csak a C_{i+5} óraciklusban bocsátható ki. Mivel megint kiürült az utasításablak, újabb három utasítást hívunk le. Ebben az óraciklusban viszont egyetlen utasítást se tudunk kibocsátani, mivel a sorrendiség miatt I_7 függősége I_8-at és I_9-et is akadályozza. Ezen utasítások kibocsátása csak a C_{i+6} óraciklusban lehetséges.



4.3. ábra. Az utasításablak tartalma egyszerre történő feltöltés és sorrendbeli kibocsátás esetés

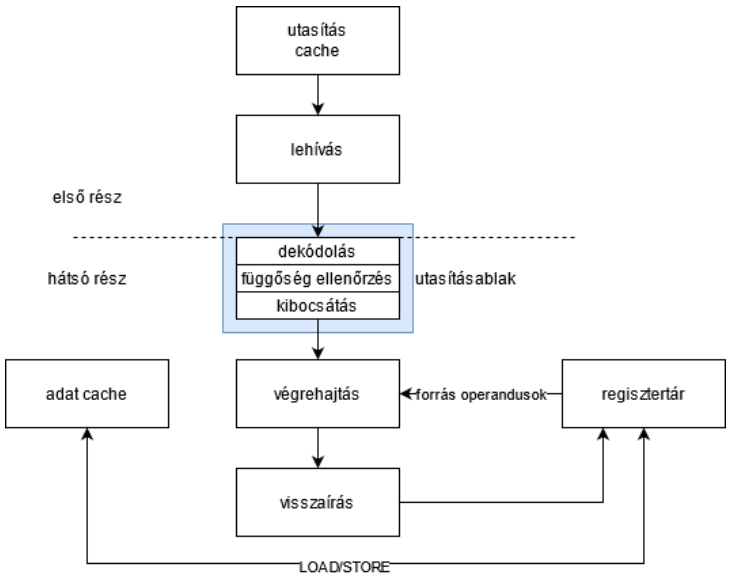
Egyszerre pótló és sorrendben kibocsátó utasításablak értékelése: a tapasztalat azt mutatja, hogy az ilyen elven működő processzor kibocsátási rátája közelít az 1 utasítás/ciklushoz. Tehát a jóval több tranzisztor és bonyolultabb felépítés ellenére nem sokkal gyorsabb, mint a futószalag processzorok. A keskeny szuperskalár név innen ered.

4.4.3. Végrehajtási modell RISC architektúrák esetén

A teljes rendszer feldolgozási sebességét az alrendszerek átbocsátási képessége határozza meg. Az alrendszerek jellege és száma az adott mikroaarchitektúrától függ. A végrehajtási modell három részből áll:

- első rész: feladatai az utasítás lehívás és az utasításablak feltöltése
- utasításablak
- hátsó rész: feladata az utasításablak kiürítése (dekódolás, függőség ellenőrzés, kibocsátás), végrehajtás és a visszaírás.

Ennek az egyszerűsített modellje látható a 4.4. ábrán. Mivel az ábra RISC architektúrát mutat be, az adat cache közvetlenül nem érhető el, csak a regisztertár. A regisztertár és az adat cache közötti adatmozgatást LOAD/STORE utasítások segítségével érhetjük el.



4.4. ábra. Az első generációs szuperskalár RISC architektúrák végrehajtási modellje

Szélesség: az alrendszerek átbocsátási képességét nevezzük szélességnek. A teljes rendszer szélességét a legkeskenyebb alrendszer szélessége határozza meg.

4.4.4. Szűk keresztmetszetek

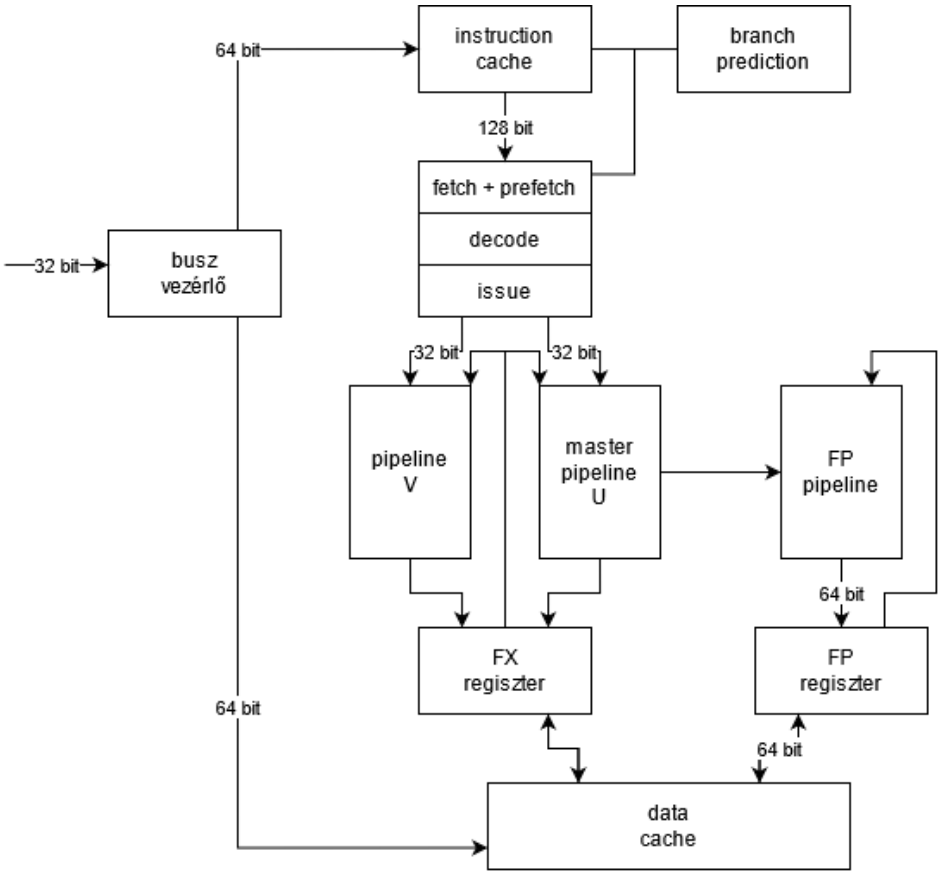
Az első generációs szuperskalároknál a következő szűk keresztmetszetek lépnek fel:

- kibocsátás,
- memória,
- elágazásfeldolgozás és
- függőségek.

A memória szűk keresztmetszetét cache bevezetésével, az elágazását pedig statikus becsléssel csökkentették. Nem tudták viszont kezelni a RAW függőségek és az adatfüggőségek által létrehozott szűk keresztmetszeteket. Első generációs szuperskalároknál még az ál adatfüggőségek is blokkolnak. A kibocsátási szűk keresztmetszet oka a közvetlen kibocsátás volt, így ezt se lehetett kezelni. Következmény, hogy a végrehajtás általános célú alkalmazások esetén korlátozódik, és maximum 2 utasítás/ciklusra korlátozódik (RISC-nél max 3 utasítás/ciklus). A gyakorlatban viszont még ezt se érte el, kb. 1 utasítás/ciklust eredményezett.

4.4.5. Gyakorlati példa: Intel Pentium I

A Pentium I-es CPU keskeny szuperskalár, CISC architektúrájú processzor volt. Újdonság, hogy belül 64 bites, kívül pedig 32 bites buszokkal kapcsolódott (ez a megoldás nem egyedi, az Intel 8088 CPU kívül 8, belül 16 bites). A nagyobb utasításokat így két óraciklusra tudta betölteni. A processzor elvi működése a 4.5. ábrán látható.



4.5. ábra. Az Intel Pentium I processzor működése

Futószalagok: 2 futószalaggal rendelkezett, ebből egy dedikált (V) és egy univerzális (U vagy master). A dedikált futószalag képes volt FX, L/S és Branch (elsősorban 1 ciklusos) utasítások végrehajtására, az univerzális pedig minden x86-os utasítást el tudott végezni (lebegőpontos számításokat is). A két futószalag csak akkor dolgozott egyszerre, ha mindkettő egyszerű utasításokat dolgozott fel. Mindkét futószalag 5 fokozatú volt (Fetch, Decode, Address Generation, Execute, Writeback). Az U futószalag ezen kívül még 3 fokozatot tartalmazott, amit csak a lebegőpontos műveletekhez használt.

Ugrás előrejelzés: Újdonságnak számított még, hogy ugrás előrejelzést alkalmazott, ehhez 2 db prefetch buffert használt.

Cache: Az adat és az utasítás cache is 8 kbyte-os volt.

4.5. Második generációs szuperskalárok

Az első generációs szuperskalár CPU-k kibocsátási szűk keresztmetszete miatt az átbocsátóképesség növeléséhez át kellett tervezni az architektúrát. Így születtek meg a második generációs szuperskalár processzorok. Kibocsátási rátájuk kb. 4 utasítás/ciklus RISC és 3 utasítás/ciklus CISC architectúrák esetén, tehát az első generációhoz képest jelentős teljesítménynövekedés történt.

4.5.1. Feltételei

Egy CPU második generációs szuperskalárnak nevezhető, ha az alábbiak teljesülnek:

- dinamikus utasítás ütemezés
- regiszter átnevezés
- elágazások előrejelzése dinamikus előrejelzéssel, ugrástörténet figyelembe vételével (ez kb. 90-95%-os pontosságot biztosított)
- kifinomult és kibővített gyorsítótár alrendszer
- sorrenden kívüli kiküldés

A dinamikus utasítás átnevezés és a regiszter átnevezés segítségével sikerült kiküszöbölni a közvetlen kibocsátási szűk keresztmetszet.

4.5.2. Példák

- Intel Pentium Pro
- AMD K6
- PowerPC 604

4.5.3. Dinamikus elágazásbecslés

Az elágazások történetét történet bitek formájában írja le. A dinamikus becslés lehet 1, 2 vagy 3 bites.

1 bites: a történet bit értéke 0 vagy 1, attól függően, hogy ugrás vagy soros folytatás történt (a bitek jelentése architektúránként változik). A következő elágazásnál a történet bit alapján döntötte el a CPU, hogy ugrás vagy soros végrehajtás következik.

2 bites: a történet biteket 2 bittel ábrázoljuk. Az értékek a következők lehetnek:

- 11 - határozott elágazás (mindenképp ugrik, általában ez a kezdeti állapot)
- 10 - gyenge elágazás
- 01 - gyenge soros folytatás
- 11 - határozott soros folytatás

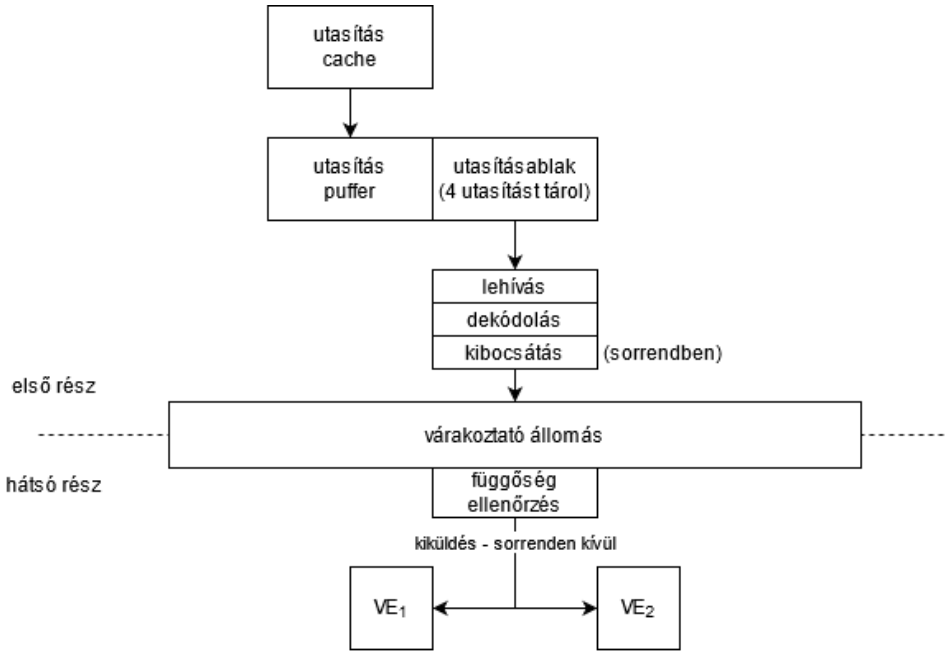
Mivel legtöbbször az elágazásnál ugrás történik, ezért a történet bitek értéke kezdetben 11. Amennyiben a következő elágazásnál mégis hibás volt az ugrás, a történeti bitek értéke 10-ra, gyenge elágazásra változik, tehát az ezt követő alkalommal megint meg fogja próbálni az ugrást. Ha harmadszorra is hibás volt az ugrás, átkerül 01 állapotba, azaz gyenge soros folytatásra.

4.5.4. Dinamikus utasítás ütemezés (várakoztatás, vagy pufferelt utasításkibocsátás)

Lényege, hogy az utasítások kibocsátása pufferelt, sorrendi módon történik, a kiküldés viszont sorrenden kívüli. Ez jelentősen megnöveli a mikroarchitektúra elejének átbocsátóképességét, valamint kiküszöböli a kibocsátási szűk keresztmetszetet.

4.5.5. Sorrenden kívüli kiküldés

A második generációs szuperskalároknál kibocsátáskor nincs függőségvizsgálat, ezért a lehívás, a dekódolás és a kibocsátás nominális rátával működhet, ami ebben az időben általában 3-4 utasítás/ciklus volt. Az utasítások kibocsátáskor a kibocsátási pufferbe kerülnek, ez a várakoztató állomás. A kibocsátási puffer lehet közös is, de előfordulhat, hogy a különböző típusú utasításoknak (FX, FP, stb.) külön pufferek vannak fenntartva. A kibocsátási puffernek köszönhetően a vezérlő óraciklusonként akár több tucat utasítás közül is választhat, hogy mit küldjön ki végrehajtásra. A döntés az állapot bitek alapján történik, a vezérlő ezek segítségével dönti el az utasításokról, hogy azok függők vagy függetlenek. A CPU a független utasításokat küldi ki végrehajtásra. Ez a működés látható a 4.6. ábrán.



4.6. ábra. A sorrenden kívüli kiküldés működése RISC architektúrák esetén

4.5.6. Regiszter átnevezés

A dinamikus utasítás ütemezés ugyan növeli az átbocsátóképességet, viszont a függő utasításokat nem küszöböli ki, azok továbbra is lassítják a végrehajtást. Erre jelent megoldást a regiszter átnevezés módszere, ami kiküszöböli az ál adatfüggőségeket. Ez a WAR és WAW függőségeket küszöböli ki, azokat még kibocsátás előtt megszünteti (még mielőtt a várakoztató állomásba bekerül). A függőség okát és a módszer leírását lásd részletesen: 2.3.4. és 2.3.5. részek. A gyakorlatban ennek eléréséhez a CPU minden regiszterhez rendel egy átnevezési (piszkozat) regisztert. Ilyenkor az átnevezési logika követi az aktuális regiszter allokációkat, átnevezik a forrás regisztereket is, így az architekturális regisztereket elég csak a visszaíráskor figyelembe venni. Az allokációt a CPU az utasításokhoz, és nem az architekturális regiszterekhez köti. A forrás operandus megfelelő helyről való beolvasását a forrás regiszter átnevezés biztosítja. Ha megszűnik a függőség, az utasítás végrehajtásra kerül és az eredmény visszaíródik az architekturális regiszterbe, majd az átnevezési regiszterek felszabadítása következik. Ez a megoldás a dinamikus elágazásbecslés számára is előnyös, mivel ha kiderül, hogy a program mégse azon az ágon folytatódik, amihez az adott utasítást végrehajtottuk, az eredmény még csak a piszkozat regiszterben van jelen, ahonnan könnyen törölhető.

4.5.7. Végrehajtási modell RISC architectúrák esetén

A második generációs szuperskalár RISC architektúráknál az utasítások lehívása 128 bites buszon keresztül történik. Ezután a processzor első részének feladata a dekódolás és az átnevezés. A lehívás és a dekódolás általában 4 utasítás/ciklus szélesek. A kibocsátás egy várakoztató állomásba történik (itt több tucat, ma már az x86_64 architektúránál akár több száz utasítás is tárolódhat). A függőség ellenőrzés

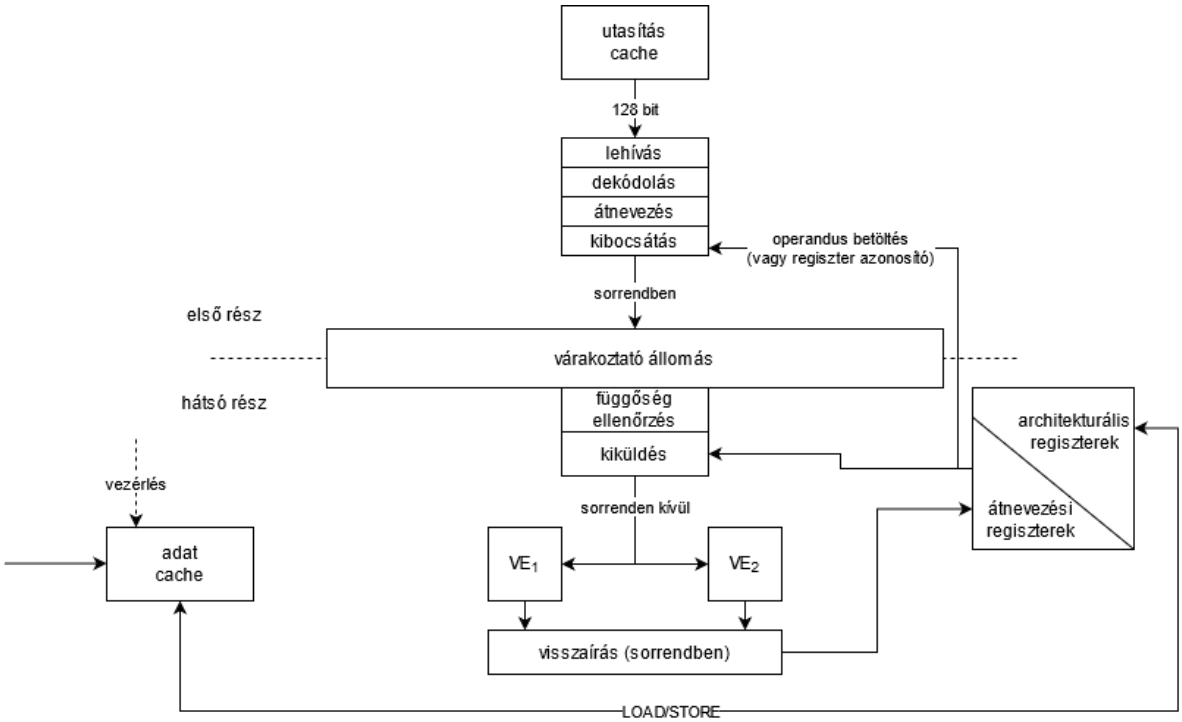
itt, a várakoztató állomásban történik meg, ahonnan a végrehajtó egységek felé sorrendben kerülnek kiküldésre az utasítások. Az operatív tár elérése a RISC architektúra miatt egy adat cache-en keresztül valósul meg, ami az architekturális regiszterekkel kétirányú kommunikációt folytat. Az utasítások eredménye visszaírásnál az átnevezési regiszterekbe kerül, ahonnan ha minden rendben volt, visszaíródhat az architekturális regiszterekbe. Az operandusok betöltése kétféleképpen történhet:

- kibocsátáskor, vagy
- kiküldéskor.

Ha egy valós adatfüggőség miatt kibocsátáskor még nem áll rendelkezésre az operandus, az operandus helyére annak az átevezési regiszternek az azonosítója kerül, ahol majd a függőség feloldásához szükséges eredmény elő fog állni. Az operandusok és a regiszter azonosítók megkülönböztetésére a CPU állapot biteket iktat a műveleti kódba az alábbi módon:

$$\mathrm{MK} \mid \mathrm{OP}_1 \mid \mathrm{A}_1 \mid \mathrm{OP}_2 \mid \mathrm{A}_2 \mid \mathrm{OP}_3 \mid \mathrm{A}_3$$

Az állapot bitek értéke lehet 0, azaz operandus nem áll készen, vagy 1, ha az operandus készen áll. 0 értékű állapot bit esetén az utasítás a várakoztató állomásban van addig, amíg az operandus rendelkezésre nem áll. Ekkor a kiküldés előtt az átnevezési regiszterből betöltődik az operandus, az állapot bit értéke pedig 1-re változik, az utasítás pedig végrehajtásra kerül.



4.7. ábra. A második generációs szuperskalár RISC architektúrák végrehajtási modellje

4.5.8. Értékelés

A második generációs szuperskalárok kibocsátási rátája kb. 3-4 utasítás/ciklus, a kiküldési ráta pedig általában 5-8 utasítás/ciklus. Ezt nevezzük tölcsérszerű elvnek, azaz a processzor hátsó része felé haladva az átbocsátási képesség nő (szélesedik). Ez a felépítés segít a szűk keresztmetszetek kiküszöbölésében.

4.5.9. Kiküldéshez kötött operandusbetöltés

A későbbi generációknál rájöttek a mérnökök, hogy fölösleges a kibocsátáskor betölteni az operandusokat. Ezért úgy módosították az architektúrát, hogy a kibocsátásnál ugyan megmaradt a gépi kód struktúrája, de az állapot bitek minden esetben 0-ra voltak állítva, az operandusok helyére pedig a regiszter azonosító

került. A függőség ellenőrzés és az operandusok betöltése így mindig a kiküldés előtt történik, ami egyszerűsíti és gyorsítja a működést.

4.5.10. CISC feldolgozás

Újdonság, hogy a CISC utasításokat RISC-szerű utasításokká konvertálták. Ez azért volt előnyös, mert a tapasztalat szerint a processzor általános felhasználás során az idő 80%-ában az utasítások mindössze 20%-át használja. A CISC utasítások hossza általában 1-17 byte volt (Intel), ezeket konvertálták egységesen 128 bites, RISC-szerű utasításokká, amik aztán a fentebb látható módon kerültek végrehajtásra. Az átalakítás a dekódolási fázisban, a RISC-CISC visszaalakítás pedig a visszaírás során történt. Ez a felépítés (kívül CISC architektúra, belül RISC mag) máig jellemző az Intel processzorokra.

Utasítások aránya: átlagosan egy CISC utasítás 1,2-1,5 RISC utasítássá konvertálható (az egyszerűbb CISC utasítások akár egy RISC-el is leírhatók, de a bonyolultabbakhoz akár 3-4 RISC utasítás is szükséges).

Kibocsátási ráta: a konverzió miatt a CISC utasításokban mért kibocsátási ráta valamelyest csökken, nagyjából 3 utasítás/ciklusra.

4.5.11. Függőségek kezelése

Összefoglalva a második generációs szuperskalárok a következőképp kezelik a függőségeket:

- Erőforrás függőség: több végrehajtó egység alkalmazása.
- Ál adatfüggőségek: regiszter átnevezés (100%-os megoldás).
- Valós adatfüggőség: még mindig blokkol, részleges kezelés a várakoztató állomással, a sorrenden kívüli kiküldéssel és a spekulatív elágazáskezeléssel.
- Vezérlés függőség: spekulatív elágazáskezeléssel részleges kezelés.

4.5.12. Utasítások végrehajtási sorrendje

Egy utasítás akkor kerül kiküldésre a várakoztató állomásból, ha a bemenő operandusai rendelkezésre állnak → adatvezérelt (mohó/stréber) végrehajtási modell.

4.5.13. Gyakorlati példa: Intel Pentium Pro

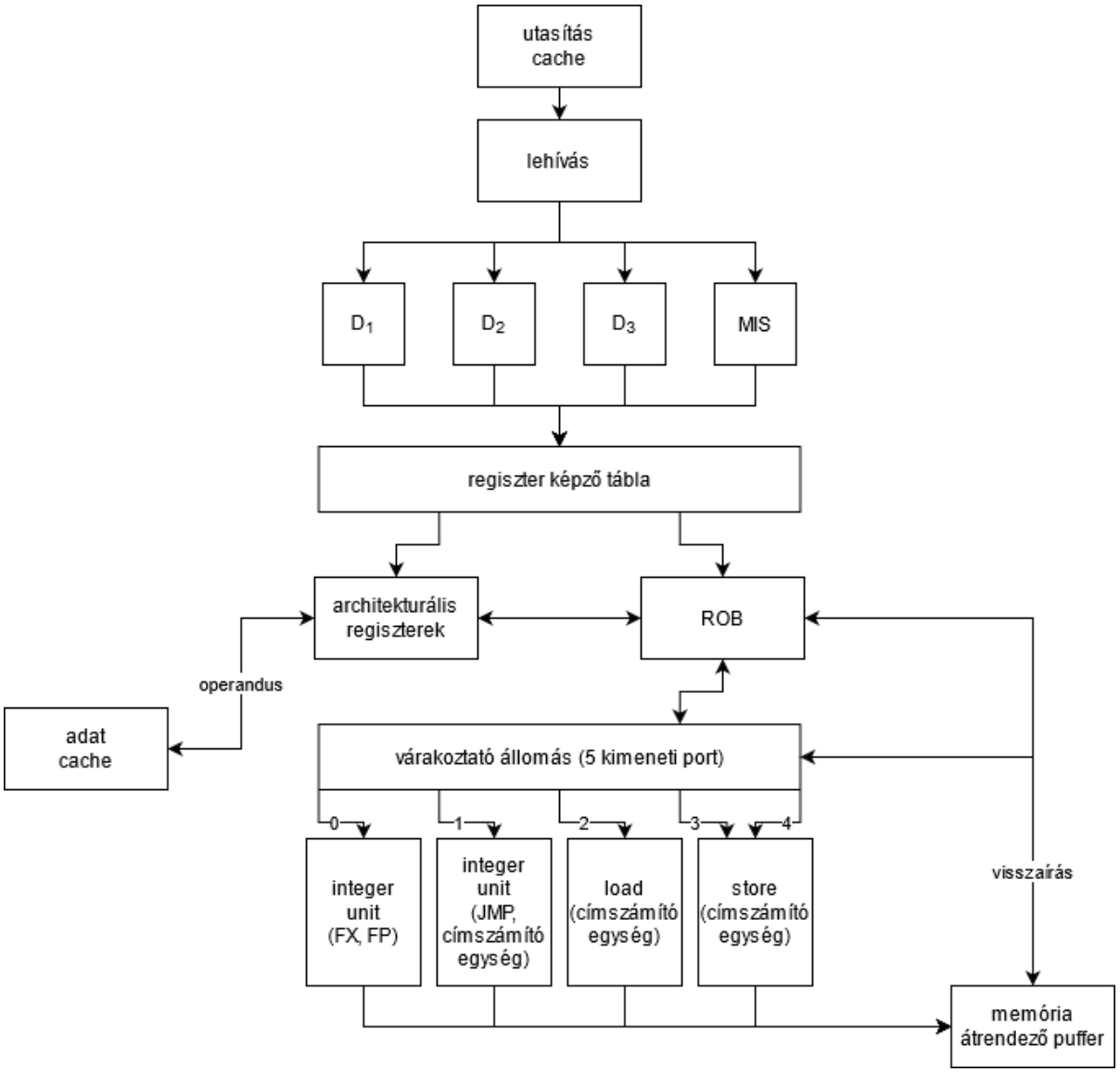
Tulajdonságai

- Kezdeti frekvencia: 133 MHz
- 14 fokozatú FX futószalag (nagy ugrást jelentett)
  - hátrány: több függőség
  - előny: a rövidebb fokozatok miatt növelhető a frekvencia
- RISC-szerű utasításokat használ, ez is segített a rövidebb fokozatok elérésében.
- 20 bejegyzéses várakoztató állomás (vegyes, azaz egy helyen tárolta az FX, FP, stb. utasításokat)
- Szigorúan soros konzisztencia, ezt a ROB (ReOrder Buffer) biztosítja.
- A ROB végzi a regiszterátnevezést is.
- A lehívás 128 bites, itt fontos az utasítás határra illesztés (CISC miatt).
- A dekódolás során a CISC utasításokat RISC-be konvertálja, óraciklusonként 3 CISC utasítás hajt végre.
- A dekóder négy egységből áll, részei:

- D_1 - legfeljebb 4 RISC műveletté alakítható CISC utasítások dekódolása
- D_2 és D_3 - egy RISC műveletté alakítható CISC utasítások dekódolása (egyszerű dekódolók - összeadás, kivonás)
- MIS - 4-nél több RISC utasítássá alakítható utasítások dekódolása

- A várakoztató állomás több független utasítás esetén mindig az idősebbet küldi ki végrehajtásra.
- A kívülről CISC architektúrának köszönhetően megmaradt a kompatibilitás, ugyanakkor a RISC mag miatt jelentősen javult a teljesítmény.

Működése



4.8. ábra. Az Intel Pentium Pro processzor működése

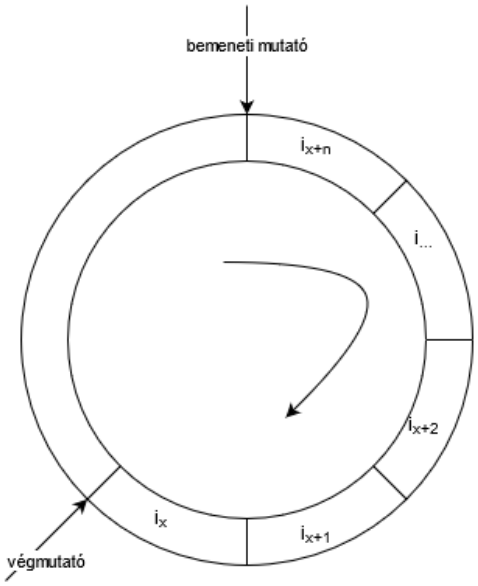
4.5.14. A reorder buffer működése

A reorder bufffer tartalmazza az átnevezési regisztereket, vezérli a várakoztató állomást és a memória átrendező puffert. A ROB-ot folyamatosan frissíteni kell a függőségek mielőbbi feloldása érdekében (tehát fontos, hogy az átnevezési regiszterazonosítók helyére bekerüljenek az operandusok). Amikor egy

eredmény előáll, a processzor asszociatív keresést végez (regiszterazonosító alapján), hogy a várakozó utasítások között van-e olyan, aminek operandusa a most előállt eredményt igényli. Ha talál ilyet, az eredményt betölti a regiszterazonosító helyére és az állapot bitet 1-re állítja. Ha mindkét forrás operandus állapot bite 1 lesz, a ROB kiküldi az utasítást a végrehajtó egységek felé.

Szemléletése

A működése egy kör alakú puffer segítségével ábrázolható (4.9. ábra). A várakoztató állomás a bemeneti és a végmutató között látható puffer regiszterekből áll. Itt sorakoznak az i_n-el jelölt utasítások. Az utasítások a bemeneti mutatónál töltődnek be, a ROB elvégzi a regiszterátnevezéseket és beírja a forrás operandusokat, majd később az eredményt is az átnevezési regiszterekbe. A bemeneti és végmutató között bármelyik független utasítás kiküldhető a végrehajtó egységek felé. Ha egy utasítás végrehajtásra került, a bemeneti és a végmutató is egyel elmozdul, az ábrán az óramutatú járásával ellentétesen. Ekkor már az i_{x+n+1}-edik utasítás is része lesz a várakoztató tárnak.



4.9. ábra. A reorder buffer szemléltetése

Az utasítások sorrendisége

Tegyük fel, hogy az ábrán látható utasítások közül i_{x+1} és i_{x+2} független utasítások és már megtörtént a végrehajtásuk, viszont i_x függő utasítás és még nem történt meg a végrehajtása. Ekkor az utasítások sorrendiségének biztosíítása érdekében i_{x+1} és i_{x+2} eredményei még nem írhatók ki a célregiszterekbe, azzal meg kell várniuk i_x befejeződését, így a pufferregiszterek sem szabadulhatnak fel addig. Ezzel látszólag i_x végrehajtása ugyanúgy blokkoló hatású, viszont a teljesítmény mégis jelentősen növekszik, hiszen i_x végrehajtásakor i_{x+1} és i_{x+2} eredményei már készen állnak, így csak ki kell írni őket. Míg a végrehajtás akár 9-10 óraciklust is igénybe vehet, kiíráskor egy óraciklus alatt akár 5-6 utasítás eredménye is kiírható, ezáltal jelentős teljesítménynövekedést érünk el.

Spekulatív bit

A ROB minden utasításhoz rendel egy spekulatív bitet. Azoknál az utasításoknál, amiknél még nem biztos a végrehajtás szükségessége (olyan elágazás részei, ahol a feltétel még nem került kiértékelésre), a spekulatív bit értékét 1-re állítja a ROB. Amíg a spekulatív állapot fennáll, az eredmény nem írható ki, így biztosítva az utasítások sorrendiségét. Ha az ugrási feltétel kiértékelésre került, két eset lehetséges. Egyik, ha a kérdéses utasítást végre kell hajtani: ekkor a spekulatív bit 0-ra változik, az eredmény kiírásra kerül, a végmutató pedig elmozdul. Mésik esetben, ha mégse kell végrehajtani az utasítást, az utasítás törlésre kerül a ROB-ból, az átnevezési (piszokzat) regiszter értéke felszabadításra kerül, a végmutató elmozdul, a helyes irányba történő utasítás pedig lehívásra kerül.

Rekonverzió (RISC → CISC)

Mivel egy CISC utasítás több RISC utasítássá konvertálódik a végrehajtásra, de a rendszer kívülről CISC architektúrájú, az egy CISC utasításhoz tartozó RISC utasítások eredményei csak akkor írhatók ki, ha már mindegyiknek előállt az eredménye. Így az eredmények egyszerre kerülnek kiírásra és megtörténhet a rekonverzió.

4.6. Harmadik generációs szuperskalárok

A második generációs szuperskalárok a fent említett módszerekkel elérték az architektúra teljesítménybeli korlátait, így más irányba kellett fejleszteni. Így jelent meg 1997 környékén az utasítás szintű párhuzamosság, amivel elérkeztek a harmadik generációs szuperskalár CPU-k. Egyesek szerint nincs haramadik generáció, hanem ez csak a második generáció kibővítése multimédiás/3D képességekkel.

4.6.1. Utasításon belüli párhuzamosság típusai

- Duál műveleti utasítások (pl. multiply-add: $y = ax + b$. Elsősorban numerikus feldolgozást segíti, általános alkalmazásoknál ritka).
- SIMD utasítások (Single Instruction Multiple Datastream - több operanduson ugyanazon utasítás elvégzése. Ilyen pl. az Intel MMX PADDW utasítása, ami 4 fixpontos összeadást végez el egy utasítás alatt).
- VLIW architektúrák

4.6.2. SIMD jellemzői

- A SIMD utasítások egy kibővített utasításkészletet alkotnak (multimédiás utasításokkal).
- A logikai architektúra kibővítését igényli.
- 8 bemenő operandus szükséges → sokszorosára nő a memóriaigény.
- A növekvő memória sávszélesség miatt az L2 cache közös lapkára kerül a CPU-val (ez korábban, pl. a Pentium II-nél külön foglalatba került, ez volt a slot 1).
- A gyorsabb megjelenítés érdekében a buszrendszer is bővült (pl. AGP - Advanced Graphics Port, PCI express).
- 3D alkalmazásokat, játékokat tett lehetővé.
- Először az Intel Pentium processzorban (1997) jelent meg MMX (MultiMedia eXtensions) néven, ami egyelőre csak fixpontos multimédiás utasításokat támogatott.
- Az MMX technológia kibővítésre került a Pentium III processzorban (1999), SSE (Streaming SIMD Extensions) néven, ami már lebegőpontos utasításokat is tartalmazott.
- A processzor általános teljesítménye nem nőtt jelentősen a SIMD bevezetésével, viszont a multimédiás alkalmazásokat (kép, videó, 3D játék) jelentősen meggyorsította, mivel ezeknél nagy mennyiségű, függőség nélküli adattal kell dolgozni.

4.6.3. Fixpontos SIMD feldolgozás

A fixpontos SIMD utasítások elsősorban a hang, és a pixeles megjelenítést használó multimédiás alkalmazásokat gyorsítják. Egy utasításban 2-8 operandus lehetséges. (A lebegőpontos SIMD utasítások a vektoros és 3D alkalmazásokat gyorsítják, utasításonként 2-4 operandussal rendelkeznek.)

Hangfeldolgozás

A hangot a feldolgozásához először digitalizálni kell. Ezt az analóg jelből történő, meghatározott időközönkénti mintavételezéssel érhetjük el. Ebben a formában már tárolható és visszajátszható. A minőséget befolyásolja a mintavételezés sűrűsége (felbontás), ennek növelése jobb minőséget, de nagyobb erőforrásigényt jelent. Példa: 50 kHZ-es mintavételezés → 50 ezer minta másodpercenként (kb. DVD minőség). A hangfeldolgozás másik jellemzője, hogy egy adott mintát hány biten tárolunk (minta nagysága). 8 bit esetén 256 féle hangmagasságot tudunk tárolni. A cél, hogy minnél jobban hasonlítson a digitális jel az eredeti, analóg függvényhez.

Hang tömörítése

Ebben az időben még nem állt rendelkezésre annyi tároló kapacitás, hogy jó minőségű jeleket lehessen tárolni, ezért született meg a tömörítés. A különböző tömörítési formátumok (AVI, MP3) használatával nagyrészt minőség romlás nélkül tudták csökkenteni az adatok mennyiségét. A SIMD utasítások segítségével ezeket a tömörített formátumokat valós időben lehet lejátszani.

Képfeldolgozás

A képek esetén képpontokat tárolunk el, minden képponthoz hozzá kell rendelni a színét és a fényességét. A színek három értékből állnak össze: piros, zöld és kék. A színek előállításához használjuk a színfordítási táblázatot, amiben minden színhez egy számérték van hozzárendelve. Ez alapján áll elő a színskála kódolása. Kezdetben 1 biten tárolták az értékeket (világos/sötét), aztán a színes kijelzők megjelenésével ez 8, később 16 (hicolor), végül 24 (truecolor) bitre bővült. A 32 bites színskálán 24 biten ábrázolják a színeket, a többi 8 bit pedig az alfa csatorna, ami az effekteket tárolja (pl. átlátszósági mutató).

Probléma

A hang és kép feldolgozása során tehát a problémát a nagy mennyiségű adat tárolása és feldolgozása okozza.

Megoldás

- Kezdetben célhardverrel, multimédiás kártyákkal.
- Később a CPU multimédiás bővítésével.

Bit-blokk átvitel és pakolt adattípusok

A multimédiás megjelenítés tipikus művelete a bit-blokk átvitel. Lényege, hogy minden megnyitott ablakot egy bit-blokkként kezelünk, így nem egyesével dolgozunk az operandusokkal. Ez szükséges, mivel például egy 1920x1080-as kijelző 24 bites színskála esetén kb. 6 MB adatot jelent, ami azt jelenti, hogy 30 FPS megjelenítés esetén 180 MB/sec feldolgozási sebességre lenne szükség. Ez a sebesség még ma is soknak számít, ezért a megoldására egyrészt tömörítést alkalmaztak, másrészt pedig architekturális megoldást vezettek be. Az architekturális megoldás egy új fixpontos adattípus bevezetése, a pakolt adattípus. A pakolt adattípus az Intel MMX processzorokban jelent meg először, típusai:

- pakolt byte (8 bit hossz * 8 db = 64 bit)
- pakolt félszó (16 bit hossz * 4 db = 64 bit)
- pakolt szó (32 bit hossz * 2 db = 64 bit)

A SIMD utasításoknak köszönhetően ezeket az adattípusokat egy utasítás alatt össze tudták adni az Intel MMX CPU-k. Ezek az új utasítások a következőek:

- 4 alapművelet
- logikai műveletek mindhárom pakolt adattípusra

A multimédiás alkalmazások nagyrészt az internet elterjedése miatt lettek szükségesek.

Fizikai architektúra

A pakolt adattípusok feldolgozásához az architektúra fizikai módosítására is szükség volt:

- 64 bites regiszterek (új regiszterek helyett a lebegőpontos regiszterek használata, amik 80 bitesek voltak, így megfeleltek a 64 bit tárolására)
- 64 bites buszrendszer (először belső, majd külső buszok is)
- Pentium II-es processzorban már két MMX futószalag került beépítésre, ami tényleges gyorsítást jelentett a függőségek hiánya miatt

4.6.4. Lebegőpontos SIMD feldolgozás

A lebegőpontos SIMD utasítások a vektoros képfeldolgozást segítik, aminek használatával a feldolgozandó adat mennyisége csökkenthető.

Vektoros képfeldolgozás

Egyenesekkel vagy görbékkel határolt obejktumok geometriai jellemzőkkel leírhatók. Példa: egy egyenes ábrázolása Full HD kijelzőn keresztben 1920 képpontot jelent fixpontosan, ennyi adatot is kell tárolnunk. Vektorosan azonban elég a két végpontot eltárolni, ami lényegesen kevesebb adatot jelent. Kör esetében elégséges a középpont és a sugár tárolása. A módszer eredménye, hogy jóval kevesebb adatot kell tárolnunk, viszont a megjelenítéshez sokkal többet kell számolni. Képek tárolása is lehetséges így, ha azokat geometriai alakzatokra bontjuk fel, majd ezeknek az adatait tároljuk. Az első 3D játékok is ilyen elven működtek. Előnye, hogy a képek könnyen mozgathatók, kicsinyíthetők és nagyíthatók. A lebegőpontos ábrázolás szükséges, mivel a geometriai műveletek során törtekkel is kell számolni. Hátránya, hogy 2D ábrázolásnál az éles határok miatt kevésbé hasonlít a valóságra, ennek kiküszöbölésére textúrákat alkalmaznak.

3 dimenziós ábrázolás

3D ábrázolásnál értelmezni kell egy harmadik dimenziót a 2D-s képen. Az élethű megjelenítéshez biztosítani kell, hogy a párhuzamosok a végtelenben összetartanak, hogy a közelebbi obejktumok nagyobbnak hassanak, valamint az atmoszférikus hatást (a távolabbi obejktumok halványabbak és kékesebbek). A 3D-s filmeknél minimum 15-30 FPS-t kell előállítani. Ehhez kb. 500 000 objektum megjelenítése szükséges másodpercenként → nagy mennyiségű lebegőpontos adat feldolgozására van szükség.

Példa - az Intel Pentium III

Ilyen lebegőpontos feldolgozásra először az Intel Pentium III-as processzora volt képes, az SSE kiterjesztéssel. A CPU 500 MHz-en működött és nagyjából 2 GFLOPS/sec számítási kapacitással bírt. Ez azt jelenti, hogy másodpercenként 2 milliárd lebegőpontos műveletet tudott végrehajtani. A mai processzorok kb. 12-20 GFLOPS/sec teljesítményt biztosítanak.

Logikai architektúra

Logikai oldalról nézve mindez egy új utasításkészletet jelentett, ez az SSE (ld. feljebb). Kb. 70 db új utasítást tartalmazott, amik a lebegőpontos pakolt adattípusokkal tudtak műveleteket végezni. Az adatok 128 bites pakolt típusok voltak:

- 4 * 32 bit - egyszeres pontosság, vagy
- 2 * 64 bit - kétszeres pontosság.

Ezek megfelelnek az IEEE 754-es szabványnak, ami a lebegőpontos számokra vonatkozik.

Fizikai architektúra

Itt már szükség volt ténylegesen új regiszterek létrehozására, így 8 db új 128 bites regiszter került a processzorokba. Ezek mentéséről és töltéséről az operációs rendszer vagy az adott szoftver gondoskodik. Az első szoftveres támogatás a Windows 98-ban jelent meg. Az új regiszterek bevezetésével 1985 óta először változtatta meg az Intel a processzorok regiszterkészletét. Következménye, hogy a megszakítás rendszert is át kellett tervezni, és ez az OS gyártójával való együttműködést igényelte.

5. fejezet

VLIW architektúrák

5.1. Bevezetés

A fejezetben nem tárgyaljuk részletesen a VLIW architektúrákat, mivel a fejlesztésüket néhány éve leállították. Ennek ellenére egy előremutató technológia volt, aminek hasznos lehet a működését nagyvonalakban megismerni. Fejlesztése az 1970-es években kezdődött.

5.2. Működési elv

A VLIW architektúrájú CPU-k működési elve összefoglalható az alábbi pontokban:

- időbeli és térbeli párhuzamosság (hasonlóan a szuperskalár architektúrákhoz)
- a függőségek kezelését a fordítóprogram végzi
- teljesen új utasításformátummal rendelkezik → nem kompatibilis egyetlen korábbi architektúrával sem
- 1 db, több (egymástól független!) utasítást tartalmazó utasításszó
- a hardver teljesen párhuzamosított kódot kap végrehajtásra (statikus kezelés)
- minden utasítás egy-egy végrehajtó egységet közvetlenül vezérel
- az utasításhossz függ a végrehajtó egységek számától (akár 1024 bit)

5.3. Előfeltételek

A fent leírt működés biztosításához szükséges a következő előfeltételek biztosítása:

- függőségek kezelése és
- utasítások hatékony ütemezése.

Míg a szuperskalár architektúráknál ezek kezelése dinamikus módon (hardveresen) történik, addig a VLIW elvű rendszereknél ez a szoftver (compiler) feladata. Következmény, hogy a VLIW architektúrák egyszerűbb felépítésűek lehetnek, kevesebb tranzisztor szükséges.

5.4. Típusai

Két típusú VLIW architektúrát különböztetünk meg:

- korai (széles) VLIW architektúrák
- késői (keskeny) VLIW architektúrák

5.5. Előnyök és hátrányok

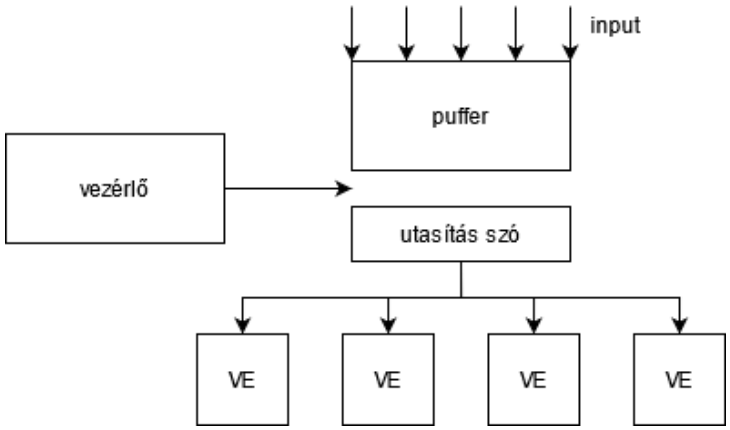
A VLIW architektúrák előnyei a szuperskalárokhoz képest:

- ugyanolyan fokú párhuzamosság mellett jóval egyszerűbb felépítés → kevesebb tranzisztor
- azonos számú tranzisztor mellett jóval nagyobb feldolgozási szélesség valósítható meg

Hátrányai:

- Statikus ütemezés (a compiler felelőssége minden egyes függőség típus kezelése) → a compilernek nagyon pontosan kell ismernie a fizikai architektúrát pl. végrehajtó egységek száma és típusa, ismétlési és késleltetési idők. Ez bonyolult és erősen architektúra függő.
- Elvárás, hogy a compiler agresszív párhuzamos optimalizálást hajtson végre.
- Minden új konfigurációhoz új compilert kell írni.
- Teljesen új utasításkészlet (ISA - Instruction Set Architecture), így nem kompatibilis a korábbi programokkal.

5.6. Logikai ábrázolás



5.1. ábra. A VLIW architektúra logikai ábrázolása

5.7. Első generáció - széles VLIW-ek

A széles VLIW architektúrákat a 70-es években fejlesztették, nagyjából 10-20 végrehajtó egységgel rendelkeztek. Az utasítá szavak 256-1024 bit hosszúak volt. A compilerek ezt még nem tudták kihasználni a kezdetleges függőség kezelés miatt (nem tudták ellátni a CPU-t elegendő független utasítással).

5.8. Második generáció - keskeny VLIW-ek

A 90-es években keskeny VLIW-eket kezdett fejleszteni az Intel és a HP, Itanium Project néven. Ezek a processzorok 2-8 végrehajtó egységgel rendelkeztek, ami lényegesen nagyobb teljesítményt biztosított a fejlettebb compilerek segítségével, mint a korabeli futószalag processzorok. A digitális feldolgozás szükségessége szintén segített a keskeny VLIW architektúrák elterjedésében, mivel ilyen feladatoknál nagy mennyiségű független utasítást kell végrehajtani. Végül a 2000-es évek elején jelentek meg az első ilyen processzorok, 4-8 utasítást tartalmazó utasítás szóval. Az Intel ezeket EPIC (Explicitly Parallel Instruction Computing) processzoroknak nevezte.

Jellemzői

- Fordítási időben történő ütemezés → nagy mennyiségű tranzisztor szabadul fel.
- A plusz tranzisztorokat extra regiszterek kialakítására használták fel.
- Regiszterekben gazdag architektúra.
- Tervezési elv, hogy a lehető legtöbb utasítás fusson párhuzamosan, akkor is, ha egy részük feleslegesen hajtódik végre.

A tervezési elv előnyei

- Vezérlés függőség (pl. feltételes elágazás) esetén mindkét ág végrehajtásra kerül → csökken az elágazási késletetés.
- Spekulatív adatbetöltés (optimalizáló compiler funkció - a compiler úgy állítja össze a gépi kódot, hogy amíg pl. egy adat betöltésére várakozik, addig más utasításokat közben végrehajt).
- Alkalmas nagy megbízhatóságú szerverekben történő alkalmazásra.

A tervezési elv hátrányai

- Energiapazarló, mobil eszközökben nehezen alkalmazható a nagy fogyasztás miatt. Következmény, hogy elsősorban szerverekben terjedt el.

5.9. Az IA-64 architektúra

Az Intel a VLIW architektúra továbbfejlesztéseként megalkotta az "új ISA", azaz az IA-64 architektúrát. Ez volt az Intel első 64 bites architektúrája, a tervek között szerepelt az x86 kivezetése is, viszont az IA-64 nem volt kompatibilis az x86-ra írt szoftverekkel. Bár a Windows XP-t még megírták IA-64-re is, a drága gyártásnak és a többmagos, 64 bites, de visszafelé kompatibilis x86_64-es processzorok megjelenésének köszönhetően a fejlesztést 2016-ban végleg leállították. Maga a VLIW architektúra egy modern, előre mutató és megbízható rendszer volt, a fejlesztés leállításának okai részben üzletiek voltak (amíg az Intel és a HP az IA-64-et fejlesztette, az AMD megalkotta az x86_64 architektúrát, ami kompatibilis volt az x86-os szoftverekkel). Nem kizárható, hogy a jövőben még visszatérnek a VLIW CPU-k.

6. fejezet

Az Intel Netburst architektúra

6.1. Bevezetés

A második generációs szuperskalárok fejlesztése során a 90-es évek második felében elérték az architektúra teljesítménybeli korlátait. A hatékonyság növelésének extenzív forrásai kimerültek. A sebesség további fokozásához a következő módszerekkel keresték a megoldást:

- új architektúrák, pl. VLIW (ld. 5. fejezet)
- párhuzamosság egyéb forrásai (szál vagy folyamat szintű párhuzamosság)
- frekvencia erőteljes növelése

Az Intelnél a frekvencia erőteljes növelésével próbálkoztak, de az akkori legfejlettebb architektúrájuk, a P6 (Pentium III alapja) nem volt alkalmas 1333 MHz fölötti működésre. A fejlesztés során a 10 GHz elérését tűzték ki célul, így új architektúra kellett. Ez lett a Netburst architektúra, amire a Pentium IV processzorcsalád épült. Az architektúra nem jelentette a második, illetve harmadik generációs szuperskalárok gyökeres újratervezését, a célkitűzés mindössze a magasabb frekvenciás működés volt.

6.2. A frekvencia növelése

A frekvencia növeléséhez a következő változtatásokat vezették be:

- Gyártási csíkszélesség csökkentése (180 nm → 65 nm), általában egy lépésben 0.7-szeresére csökkentik (így fele akkora helyen fér el ugyanannyi tranzisztor, mint a korábbi technológiánál, tehát a tranzisztorok száma megduplázható → Moore-törvény).
- Futószalag fokozatok hosszának csökkentése (ennek egyik módja a fokozatok kisebb részekre bontása, viszont ekkor a fokozatok száma növekszik, nő a párhuzamosan végrehajtott utasítások száma és a függőségek száma is → csökkenhet a hatékonyság).

6.3. Jellemzői

- CISC architektúra (1-17 byte hosszú utasítások)
- belső RISC mag
- hosszabb futószalagok (több függőség, de magasabb frekvencia)

6.4. Újdonságok

6.4.1. Execution Trace Cache

Az Execution Trace Cache az L1 utasítás cache-nek felel meg, de CISC helyett már dekódolt RISC utasításokat tartalmaz a végrehajtásuk feltételezett sorrendjében.

6.4.2. Hyper futószalag

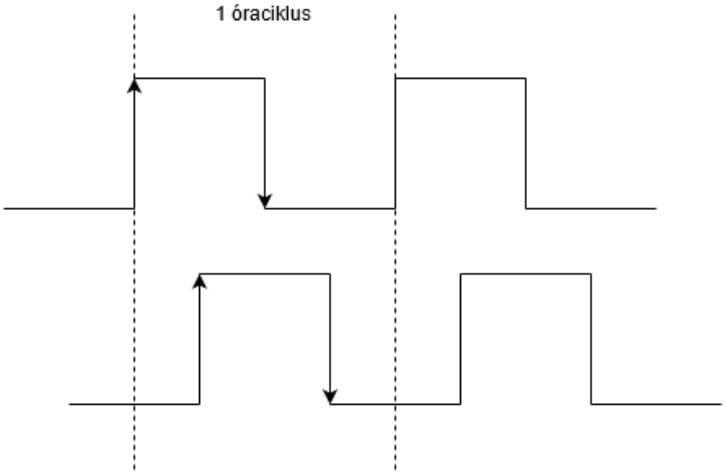
A Hyper futószalag lényege, hogy a dekódoló fokozat nincs benne. A dekódolás futószalagon kívül történik, azért, hogy az L1 cache-ben (Execution Trace Cache) már dekódolt és átalakított utasítások szerepelhessenek. Oka, hogy a CISC utasítások átalakítása lassú, akadályozta volna a nagy frekvenciás végrehajtást, ezért külön került a futószalagtól. Újdonság még, hogy a korábbi 10-15 fokozathoz képest 20-31 fokozatot tartalmazott. Ennek hátránya, hogy hibás becslés esetén nagyobb a büntetés, több fokozatot kell törölni a futószalagból → fajlagos teljesítmény csökkenés.

6.4.3. Enhanced Branch Prediction

Ez egy továbbfejlesztett elágazásbecslő logika, 94-97 %-os hatékonysággal (PIII-hoz képest kb. 33 %-al csökkent a hibás becslések száma).

6.4.4. Quad Data Rate Bus

A Quad Data Rate Bus egy belső rendszerbusz, gyorsítja az adatelérést az L1 és L2 gyorsítótárak felé. A buszfrekvencia négyszeresén továbbítja az adatokat (a külső rendszerbusz sebességét nem lehetett növelni, mivel a memória nem gyosult jelentősen). Ennek eléréséhez kettő órajel generátort alkalmaz, 90°-os fáziseltolással, valamint a felfutó és lefutó élen is történik adattovábbítás. Így óraciklusonként négy adattovábbítás történik. Ezt a működést mutatja be a 6.1. ábra.



6.1. ábra. A Quad Data Rate Bus kettős órajele

6.4.5. Rapid Execution Engine

A Rapid Execution Engine az egyszerű FX műveletek gyors végrehajtására szolgáló végrehajtó egység. Az órajel felfutó és lefutó élére is képes műveletvégzésre, így a végrehajtási idő akár egy fél ciklusra is csökkenhet. A gyorsaság érdekében kevesebb kaput tartalmaz, csak az alapműveletek elvégzésére képes. Ez sikeres lett, nőtt a hatékonyság, később kettőt is beépítettek a CPU-kba.

6.4.6. Replay System

Probléma, hogy a sok fokozatú futószalag sok függőséggel jár, ami az utasítások várakozását és kihasználatlanságot eredményezhet. Ennek megoldására az ütemező megbecsüli az utasítások végrehajtási idejét, így tudja, hogy az utasítás végrehajtásának kezdetétől mennyi idő múlva fogja igényelni a bemenő operandust. Ezért a függő utasítást már ennyi idővel a szükséges függőség teljesülése előtt kiküldi, hogy pont mire felhasználná a bemenő operandust, az már éppen előállt. Ha rossz volt a becslés és hamarabb szükség lenne a forrás operandusra, mint ahogy az előállt, az utasítás egy speciális sorba, a Replay Queue-ba kerül, ahonnan később újra kiküldésre kerül. Bár a hatékonyságot mindez csökkenti, elkerülhető a futószalag leállása, így biztosítja a végrehajtó egységek optimális kihasználását.

6.5. Következmény

A fajlagos teljesítmény, azaz az IPC (Instructions Per Cycle) csökken. Ez összességében nem jelenti viszont a teljesítmény csökkenését, mivel az órajel magasabb lett (több ciklus időegység alatt).

6.6. Fejlődési korlátok

6.6.1. Statikus disszipáció

Problémát jelentett a szivárgási áram exponenciális növekedése a frekvencia emelésével (statikus disszipáció). A szivárgási áram létrejön, mivel a tranzisztorokon akkor is folyik valamennyi áram, amikor az kikapcsolt állapotban van. A szivárgási áram fajtái:

- Gate $\rightarrow$ Drain
- Source $\rightarrow$ Drain

A statikus disszipáció miatt növekedett a hőtermelés, 3.8 - 4 GHz környékén bekövetkezett a hőkatasztrófa (1 cm^2-en 100 W disszipáció). Ennek kezelése hagyományos eszközökkel nem megvalósítható, csak vízhűtéssel, ami viszont nem gazdaságos. A statikus disszipáció kiszámítása:

$$D_{\mathrm{s}} = V * I_{\mathrm{leak}},$$

ahol V a feszültség, I_{leak} pedig a szivárgási áram, ami a frekvencia növekedésével exponenciálisan nő.

6.6.2. Dinamikus disszipáció

Az a disszipáció, amit az aktív tranzisztorokon átfolyó áram hőtermelése okoz. A dinamikus disszipáció kiszámítása:

$$D_{\mathrm{d}} = A * C * V^2 * f_{\mathrm{c}},$$

ahol A az aktív kapuk száma, c a kapuk összesített elosztott kapacitása, v a tápfeszültég, f_{c} pedig a magfrekvencia. Látszik, hogy a dinamikus disszipáció négyzetesen függ a feszültségtől, így a frekvencia növelése a feszültség csökkentésével ellensúlyozható.

6.6.3. Teljes disszipáció

A teljes disszipáció a statikus és a dinamikus összege. A frekvencia növelésével egyre jelentősebbé vált a dinamikus disszipáció, arányaik változása:

	D_{s}	D_{d}
1995	1	10000
2005	1	1

Ezért jelentek meg az új tranzisztor technológiák.

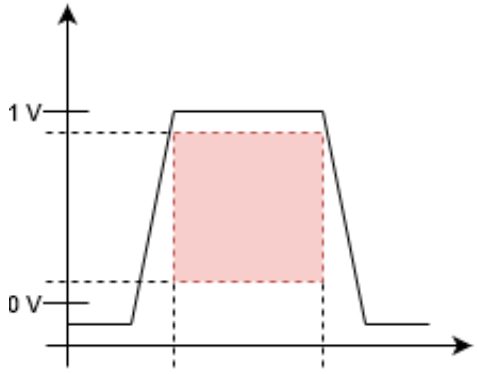
6.6.4. Párhuzamos buszok frekvencia korlátja

A buszokon logikai 1-esek és 0-k közlekednek. Annak, hogy az érzékelők ezeket a jeleket meg tudják különböztetni, időbeli és feszültségbeli fetételei vannak. Ezt írja le a Data Valid Window fogalma: ahhoz, hogy az érzékelő egy jelet érvényesnek tekintsen, a jelnek egy bizonyos feszültség intervallumba kell esnie és egy bizonyos ideig fenn kell állnia. A DVW a 6.2. ábrán látható, itt a pirossal ábrázolt területbe nem szabad beleesnie a jelnek, hogy érvényes legyen. Ez gátolja a feszültség csökkentésének lehetőségeit, mivel nagyon kis feszültség esetén a kisebb zavarok is azt eredményezhetik, hogy a jelet ne tekintse érvénytelennek az érzékelő. A frekvencia is csak bizonyos pontig növelhető, mivel:

- nagy frekvencián a párhuzamos vezetékeken érkező jelek elcsúsznak egymástól (delay skew, megoldás soros busszal),
- jel visszaverődések, azaz reflexiók léphetnek fel (kezelése hullám impedancia lezárással lehetséges),

- fázisbizonytalanság (jitter) jelenhet meg, azaz a zavarok elmossák a jelek felfutó és lefutó éleit és zavarja a jelszintek stabil állapotát. A jitter okozói a vezetékek közötti áthallás és a külső vagy belső elektromágneses interferencia (megoldás: LVDS - low voltage differential scaling, tehát a jelet két vezetéken, két különböző feszültségszinten továbbították, a logikai 0 és 1 értékeket a feszültségek különbségéhez rendelik. Pl. PCIe, QPI, DMI).

A feszültség növelésével viszont tovább növelhető a frekvencia, mivel a felfutási idő röviebb lesz, meredekebb a felfutási szög, így könnyebb megfelelni az időbeli követelményeknek. A feszültség (fogyasztás) és a frekvencia (teljesítmény) tehát egymással ellentétes célok, ezért a processzorok gyakran dinamikusan változtatják a feszültség és frekvencia értékeiket: alacsony terhelésen kis feszültség és frekvencia, nagy terhelésen magasabb feszültség és frekvencia.



6.2. ábra. A Data Validation Window

6.6.5. Egyéb korlátok

Megjelentek a párhuzamosság (ILP) hatékonysági korlátai.

6.7. A disszipáció csökkentése

A DVFS (Dynamic Voltage and Frequency Scaling) a dinamikus feszültség és frekvencia szabályozást jelenti (skálázás). Ez menet közben meghatározza, hogy az adott feladathoz milyen teljesítményre van szükség, majd ehhez igazítja a processzor feszültségét és frekvenciáját. Példa ilyenre, amikor a mobiltelefon lekapcsolt képernyővel készenléti módban van, majd felvesszük és játszani kezdünk rajta. Működése:

1. szükséges teljesítmény meghatározása
2. frekvencia hozzáillesztése a szükséges teljesítményhez
3. órafrekvencia fenntartásához szükséges minimális feszültség beállítása
4. később kiegészült az AVS-el (Adaptive Voltage Scaling), ez a következő félév anyaga

6.8. Az Intel Pentium 4

A Pentium 4-es CPU 2000-től 2008-ig volt kapható, de a fejlesztését már 2004-ben befejezték, amikor a frekvenciát már nem tudták tovább növelni a statikus disszipáció növekedése miatt. Kevésbé volt hatékony, mint a korabeli AMD CPU-k, viszont a magasabb órajel miatt nagyobb teljesítményre volt képes. Míg a Pentium III órajele 500 és 1333 MHz között volt, a P4 1.5 GHz-ről indult és a Prescott architektúrára (2004) elérték a 3.2 GHz-et.

6.8.1. Thermal Monitor

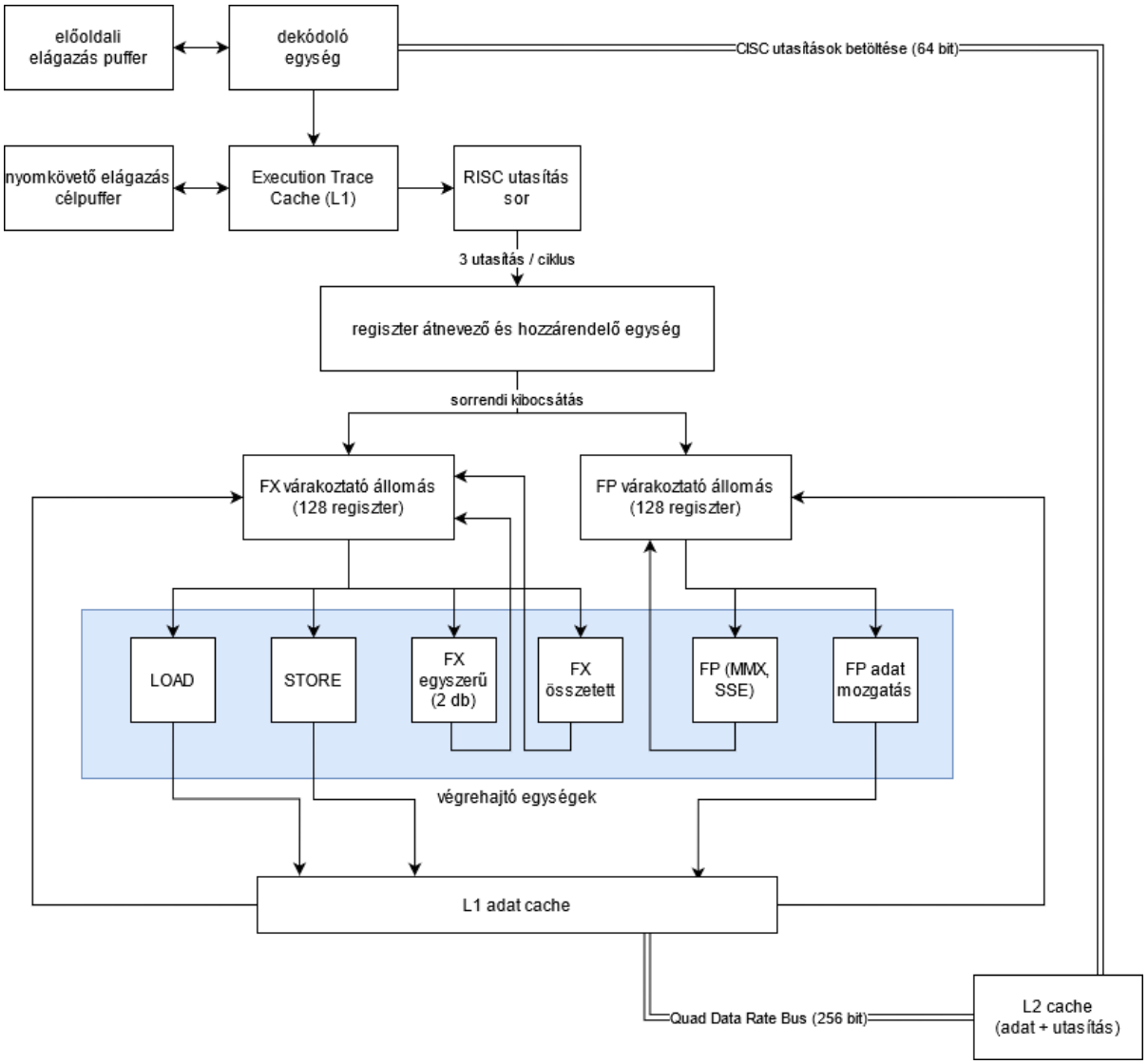
A magasabb frekvenciát a versenytársakéhoz képest jobb hővédelemmel sikerült elérni, ez volt a Thermal Monitor. Ez egy órajel moduláción alapuló megoldás, ha az érzékelő túlmelegedést észlelt, lekapcsolta (későbbiekben csökkentette) az órajelet.

6.8.2. Multimédia

A P3-hoz képest fejlesztették a multimédiás feldolgozást, 144 új utasítás került be az SSE (FX) és MMX (FP) utasításkészletekbe.

6.8.3. Felépítés

A processzor működési vázlata a 6.3. ábrán látható.



6.3. ábra. A Pentium 4 processzor működési vázlata

7. fejezet

Szál szinten párhuzamos architektúrák

7.1. Bevezetés

Az Intel 80386-os processzora óta bevezetett újítások a fogyasztás és a lapkaméret növekedésével jártak, viszont ehhez képest a teljesítmény csak kisebb mértékben javult. Tehát a teljesítmény növekedése nem állt egyenes arányban a komplexitás növekedésével. Ez elsősorban az egy magos processzorokra igaz. Következmény, hogy egyre több tranzisztor kell és nő a fogyasztás. Ennek megoldásához el kellett térni a klasszikus tervezési elvektől, ahol csak utasítás szinten használták ki a párhuzamosságokat. A fejlődés következő lépcsőjét a szál szintű párhuzamosság kihasználása jelentette.

7.2. A szál

A szál a program legkisebb önállóan végrehajtható része → párhuzamosan futtatható. Míg az utasítás szintű párhuzamosság felderítésére önmagában képes a hardver, a szálak kihasználásához az operációs rendszer támogatására is szükség van. A párhuzamosság lehet

- implicit: a programozó szekvenciális programot ír, a hardver és a szoftver párhuzamosítja a kódot,
- explicit: a programozó kifejezetten párhuzamos programot ír.

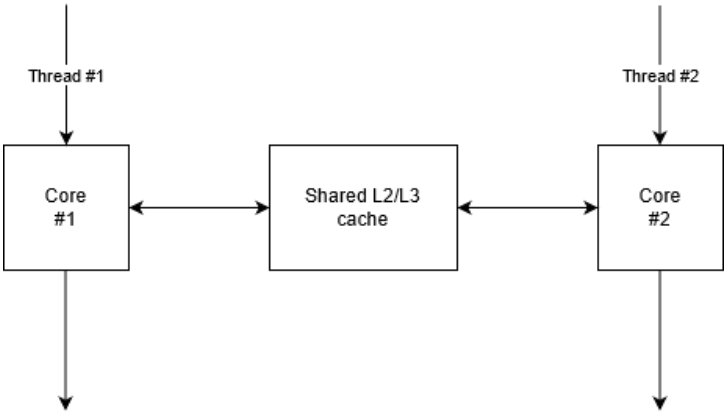
Fontos, hogy egy szálon belül nincs párhuzamosság, ezért a teljesítmény növeléséhez több szál létrehozása szükséges. Ez egy magasabb szintű párhuzamosság, így összetettebb is.

7.2.1. Szálak származtatása

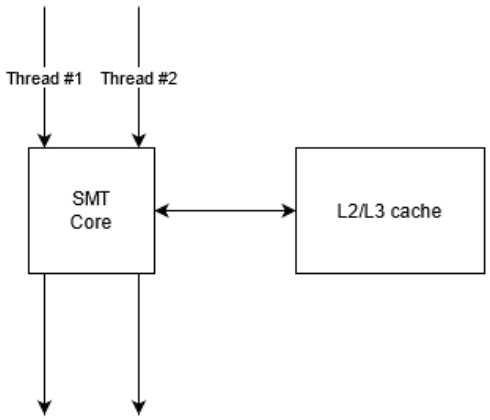
- Különböző alkalmazásokból (multiprogramozás).
- Ugyanabból az alkalmazásból:
  - multitasking
  - multithreading

7.2.2. Többszálúság csoportosítása

- Szoftveres: többszálú alkalmazás futtatása egyszálú CPU-n, pl. időosztással.
- Hardveres: többszálú alkalmazás futtatása többszálú CPU-n. Többszálú processzorok:
  - SMP - Symmetric MultiProcessing (7.1. ábra).
  - SMT - Simultaneous MultiThreading (7.2. ábra). Kb. 5% komplexitás növekedéssel 0-30%-os teljesítmény növekedés érhető el a segítségével.



7.1. ábra. SMP CPU



7.2. ábra. SMT CPU

7.3. Több szál futtatása P4 CPU-n (Hyper Threading nélkül)

A futószalag bonyolult, sok fokozatú, ezért sok függőség lép fel. A függőségek miatt egy feladat befejezése előtt gyakran szükséges egy másik elkezdése, azonban egy új szál futtatásánál az aktív szál kontextusának mentésével kell kezdeni. Ezután a másik szál kontextusát be kell tölteni és folytatni a végrehajtást. A kontextus váltás akár 2-3 ezer óraciklus is lehet, tehát nagyon erőforrás igényes. A multimédiás alkalmazások és a több alkalmazást futtató felhasználó miatt még gyakrabban volt szükség kontextus váltásra. A kontextus váltások csökkentése miatt jelentek meg a többszálú architektúrák.

7.4. Operációs rendszerek szálkezelése

A modern operációs rendszerek minden folyamatra külön lap táblát tartanak nyilván, ez a TLB (Translation Lookaside Buffer). Ezekben tárolódik az adott folyamat kontextusa, ami a memóriában vagy a háttértáron tárolódik. Tehát a szálkezelés hardveres és szoftveres támogatás is igényel.

7.5. Szál szinten párhuzamos architektúrák osztályozása

7.5.1. Finoman szemcsézett

A CPU 2 vagy több szálat futtat, órajelenként vált a szálak között. A gyakori kontextus váltások miatt az ilyen processzorok több regisztert tartalmaznak, így azokban több szál adatai is elférnek (nem kell várni a memóriából betöltésre). Pl. két szálú CPU esetén mindkét szál kontextusát tároljuk a regiszterekben

és egy kontextus kapcsoló segítségével történik a váltás. Következmény, hogy nincs késleltetés a váltások között. A teljesítmény növekszik, pl. ha az egyik szál valamilyen függőségre vár, az nem blokkolja a másik szálat (cél az üresjáratok feltöltése). 0-20%-os teljesítmény növekedést eredményezhet.

7.5.2. Eseményvezérelt (SoEMT - Switch on Event MultiThreading)

Amennyiben az egyik szál futása megakad (pl. függőség miatt), akkor vált a másikra. Probléma, hogy a szál megakadását érzékelni kell, ami általában 1-2 óraciklus időigényű.

7.5.3. SMT (Simultaneous MultiThreading)

Az elvét 1968-ban írták le, gyakorlati megvalósítása viszont csak a 2000-es években kezdődött el. Ennek egyik gyakorlati megvalósítása az Intel Hyper Threading. Lényege, hogy az összes szál futtatása párhuzamosan történik.

Hardveres megvalósítás

Az SMT-t támogató processzorok elsősorban szuperskalár architektúrák. A párhuzamosság megvalósításához az alábbi harveres követelményeknek kell megfelelniük:

- Számos erőforrást meg kell többszörözni (pl. szálankénti program counter, regiszter tároló).
- Az erőforrásokat meg kell tudni osztani a szálak között (szükség esetén egyesíteni is, hogy egy szál megakadása esetén az aktív szál a teljes processzort tudja használni).

7.6. Az Intel Hyper Threading

Az Intel Hyper Threading technológia egy két szálas SMT architektúra. Először a Northwood alapú Pentium 4-es processzorokba került be. A két szál utasításait egyszerre hajtja végre, out of order kiküldéssel. Egy mag az operációs rendszer számára két külön logikai magnak látszódik, ezért úgy ütemezi az utasításokat, mintha két processzoros rendszeren futna. Tehát egy CPU egyszerre két logikai CPU utasításait hajtja végre. A processzorban ekkor egyszerre két architekturális állapot van jelen.

7.6.1. Üzemmódok

ST (single task)

Egy szál végrehajtása történik, a megosztott erőforrások egyesítésre kerülnek. Két állapota lehet, attól függően, hogy melyik logikai CPU aktív:

- ST0
- ST1

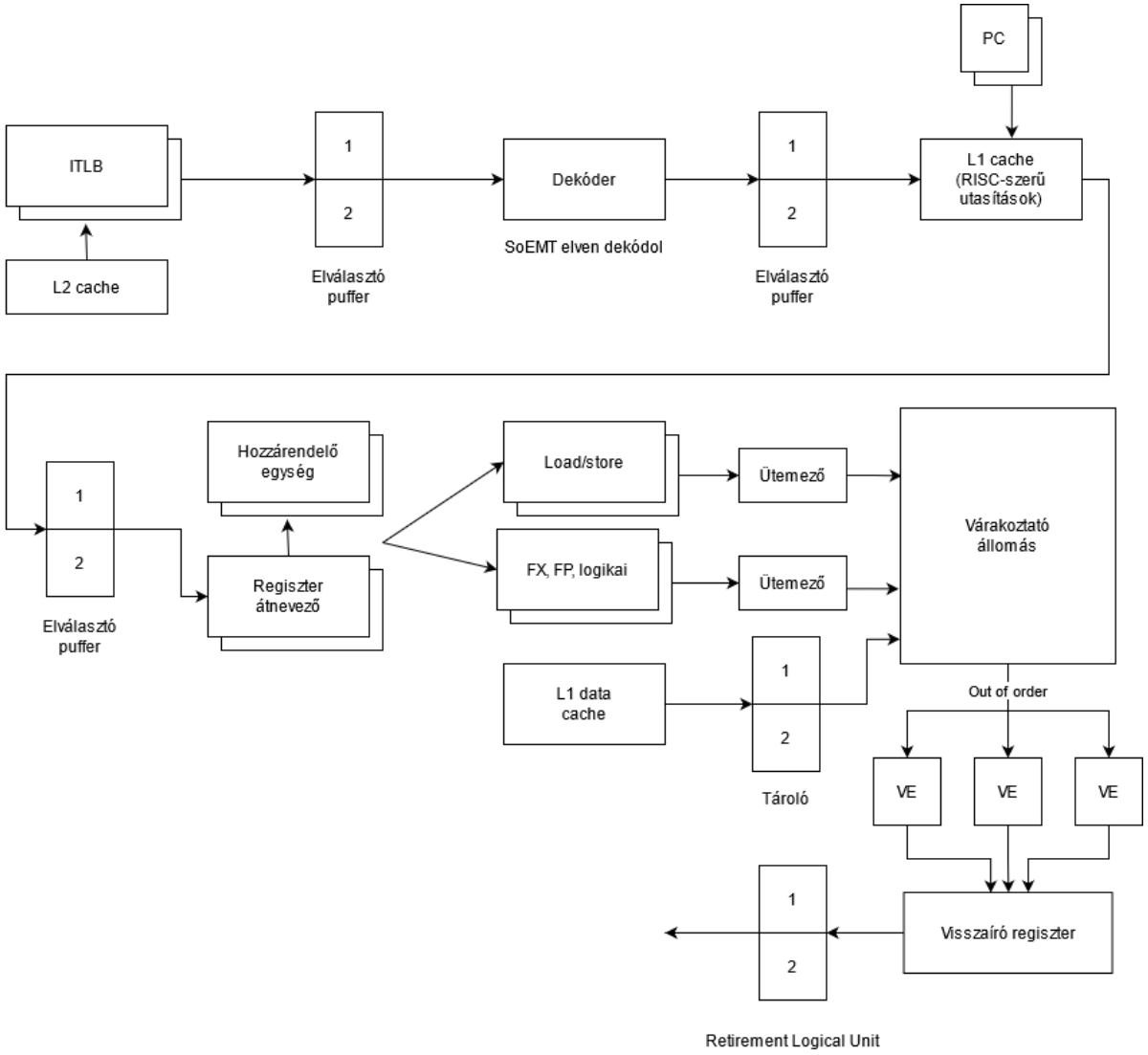
MT (multi task)

Több szál végrehajtása történik párhuzamosan.

Váltás

A processzor először MT módban indul. Ha az egyik szál megakad, ST módba vált a processzor. A függőség megszűnésével a CPU visszakerül MT üzemmódba. Az üzemmódok közötti váltás a HALT utasítás segítségével történik, ami megszakítja a CPU futását és energiatakarékos állapotba helyezi. Ezt csak az OS, vagy más, alacsony szintű alkalmazás adhatja ki.

7.6.2. Működési vázlat (Intel Pentium 4 HT)



7.3. ábra. Az Intel Hyper Threading működése

7.7. Az SMT megvalósítási céljai, összegzés

- Kis magméret növekedés.
- Egy szál várakozása esetén a másik szál gond nélkül futhasson, folytathassa a műveletek végrehajtását.
- Egy szál futása esetén a végrehajtás ugyanolyan gyors legyen, mint egy egy szálas CPU-n.
- A technológia nagyon jól bevált, hatékony, a mai processzorokban az AMD és az Intel is gyakran használja.
- Főleg szerverek esetén biztonsági kérdések merülnek fel, elképzelhető, hogy kártékony módon is felhasználható a szálkezelés.

8. fejezet

Folyamat szinten párhuzamos architektúrák

8.1. Bevezetés

A folyamat szinten párhuzamos architektúrák MIMD típusúak, multiprocesszoros rendszerek.

8.2. Fejlesztési motivációk

A multiprocesszoros, folyamat szinten párhuzamos rendszerek kifejlesztését az alábbi tényezők tették szükségessé:

- Utasítás szintű párhuzamosság korlátozott lehetőségei:
  - Branch prediction soha nem 100%-os.
  - Általános célú alkalmazásoknál a programokban lévő utasítás szintű párhuzamosság sokszor ahhoz sem elegendő, hogy 4-6 független végrehajtó egységet ellásson utasításokkal.
- Energiahatékonyság: egyes CPU-knál kb. 10-15% számítási teljesítmény növekedéshez akár 50% fogyasztás emelkedés is szükséges lehet. Következmény, hogy hatékonyabb lehet n darab CPU-t használni egységnyi órajelen, mint 1 darab CPU-t n-szeres órajelen (jól párhuzamosítható feladatok esetén).
- Költséghatékonyság: n darab olcsóbb és lassabb CPU összekapcsolása hatékonyabb lehet, mint 1 darab gyors és drága processzor.
- Egyszerű bővíthetőség: ha megfelelően fel van készítve, a bővítése könnyebb lehet egy multiprocesszoros rendszernek.
- Hibatűrés: ha az egyik processzor meghibásodik, a többi átveszi a feladatait.

8.3. Csoportosítás

8.3.1. Memória használat szerint

- Közös memória használatú: van olyan, minden CPU számára látható közös memória, melyen keresztül a folyamatok kommunikálni tudnak egymással.
- Elosztott memória használatú: a folyamatok üzenetek segítségével kommunikálnak egymással. Ezek az üzenetküldésen alapuló multiprocesszoros rendszerek.

8.3.2. Memória elérés ideje szerint

- UMA (Uniform Memory Access): a memória elérés ideje azonos minden CPU számára. Következmény, hogy nincs jelentősége annak, hogy a memóriában hova írjuk az adatot.
- NUMA (Non-Uniform Memory Access): a memória műveletek ideje nem azonos. Ezeknél a rendszereknél fontos, hogy egy adat az őt író és az őt olvasó CPU-hoz is közel legyen.

8.4. Korlátok

- Szoftver: vezérlés áramlásos rendszerekben a párhuzamosságok felderítését szoftveresen kell megoldani. Megoldás:
  - olyan compilert írunk, ami képes felfedezni a szoftverek párhuzamosítható részeit és ennek megfelelő kódot generál, vagy
  - a párhuzamosítást a programozóra bízzuk (főleg ez a működőképesebb).

8.5. Amdahl törvénye

Annak eldöntéséhez, hogy érdemes-e folyamat szintű párhuzamosítást alkalmazni, felhasználhatjuk Amdahl törvényét. Segítségével kiszámolható, hogy az n darab CPU-ból álló rendszer mekkora teljesítmény növekedést tud nyújtani. Ehhez fontos tudni, hogy milyen mértékben párhuzamosíthatók a folyamatok. Egy valós program tartalmaz szekvenciális és párhuzamosan futtatható részeket is, ezek alapján pedig feltételezhetjük, hogy egy program P-ed része párhuzamosítható. Ebből következik, hogy 1-P rész szekvenciálisan futtatható. Legyen a futási idő 1 darab CPU-n 1 egység. Ekkor N db CPU esetén a futási idő ideális körülmények között:

$$(1-P)+\frac{P}{N}\ .$$

Ebből a teljesítmény növekedés:

$$S_{\mathrm{p}}(N)=\frac{1}{1-P+\frac{P}{N}}\ .$$

Például, ha azt szeretnénk, hogy egy 100 processzoros rendszeren 50-szeres teljesítmény növekedést kapjunk az 1 CPU-hoz képest, akkor

$$P=\frac{S_{\mathrm{p}}(N)-1}{S_{\mathrm{p}}(N)}\cdot\frac{N}{N-1}\approx 0.9898\ ,$$

tehát a programunk majdnem 100%-ának párhuzamosíthatónak kell lennie. Amdahl törvényéből következik, hogy a gyorsulásnak véges határértéke lesz végtelen processzor esetén is, mivel

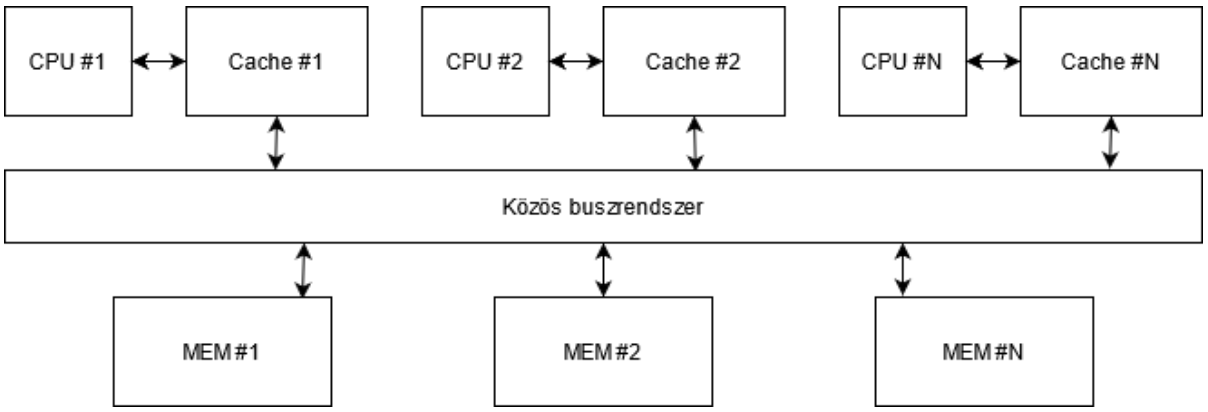
$$\lim_{N\to\infty}S_{\mathrm{p}}(N)=\frac{1}{1-P}\ ,$$

így például egy 95%-ban párhuzamosítható programmal legfeljebb 20-szoros gyorsulás érhető el. Következmény: a processzorok számát nem gazdaságos egy bizonyos határon túl emelni.

8.6. UMA rendszerek

8.6.1. Felépítés

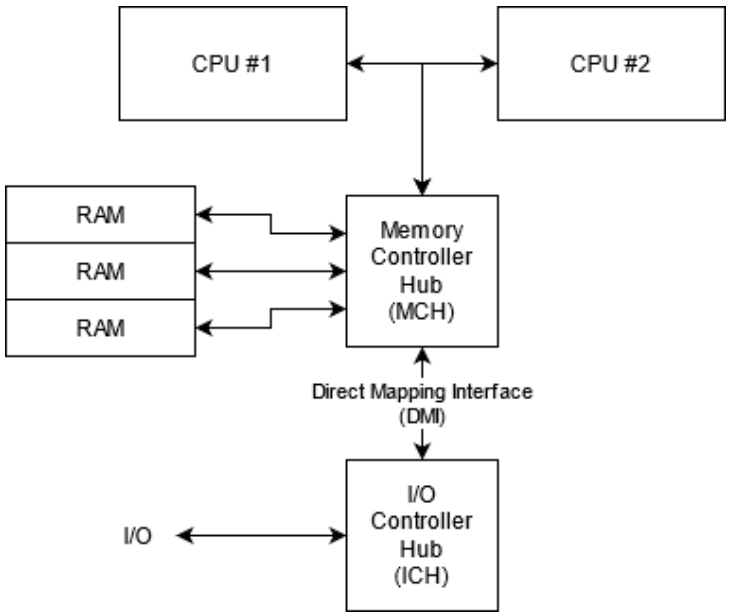
Az UMA, vagy más néven SMP (Symmetric Multiprocessing) rendszereknél a processzorok egy közös buszon keresztül érik el a memóriát (8.1. ábra). Probléma, hogy a közös buszrendszeren keresztül továbbítódik minden adat, így az szűk keresztmetszetet képez. Ezért a megoldás nem jól skálázható, kb. 8-16 CPU köthető össze a buszrendszer túlterhelése nélkül.



8.1. ábra. Az UMA rendszerek memória elérése

8.6.2. Gyakorlati példa

A kezdeti többmagos architektúráknál alkalmazott FSB (Front Side Bus) felépítés UMA elven működik (8.2. ábra).

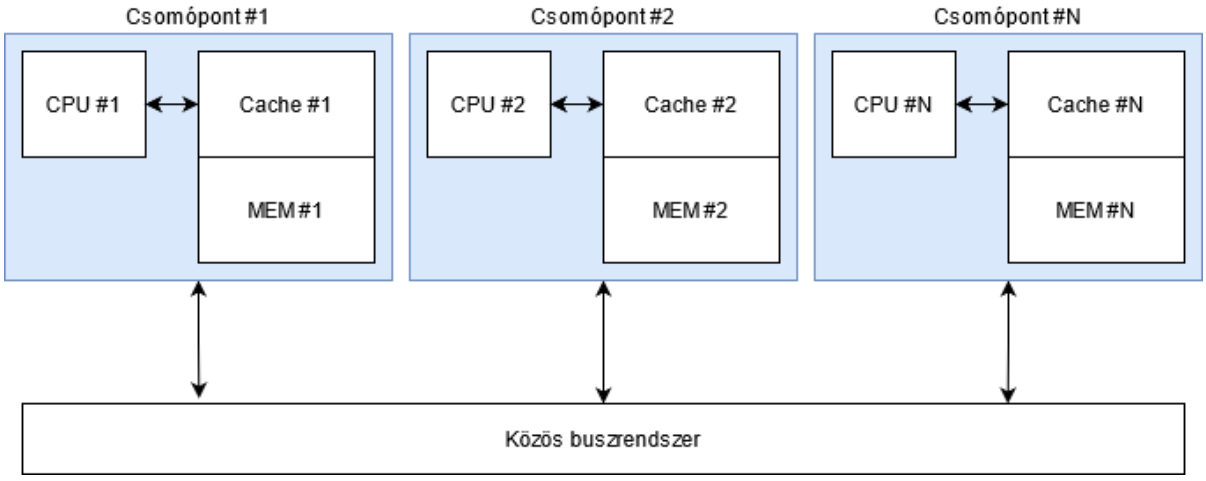


8.2. ábra. A tradícionális FSB rendszer felépítése

8.7. NUMA rendszerek

8.7.1. Felépítés

Az UMA rendszerek problémáinak kiküszöbölésére fejlesztették a NUMA architektúrákat, amik közös, de tartományokra bontott memória címtérrel rendelkeztek (8.3. ábra). A tartományok megfelelői az ábrán a csomópontok. Eredmény, hogy a buszrendszerre kisebb terhelés esik, így a megoldás jól skálázható.



8.3. ábra. A NUMA rendszerek memória elérése

8.7.2. Csoportosítás

A NUMA rendszereket két csoportra oszthatjuk:

- CC NUMA: cache coherent
- NC NUMA: non cache coherent

8.7.3. Cache coherent

A cache koherens rendszereknél az adatok legfrissebb példányát a gyorsítótárakban nyilvántartják és frissítik. Ez a gyakorlatban azt jelenti, hogy ha két CPU ugyanazzal az adattal dolgozna, és mindkettő betölti a gyorsítótárába, majd az egyik módosítja az adatot, a másik CPU gyorsítótárában lévő adatot invalidálja egy figyelő rendszer és az adat frissítésre kerül. A módszer hátránya, hogy ez komoly adminisztrációt igényel, ami a buszrendszert terheli. A CC NUMA rendszereket ezért szokás hívni az UMA rendszerek skálázható kiterjesztésének is. Ez a megoldás kb. 32 CPU-ig skálázható, viszont az UMA-hoz képest a memória műveletek késleltetése 2-7-szeresére növekszik.

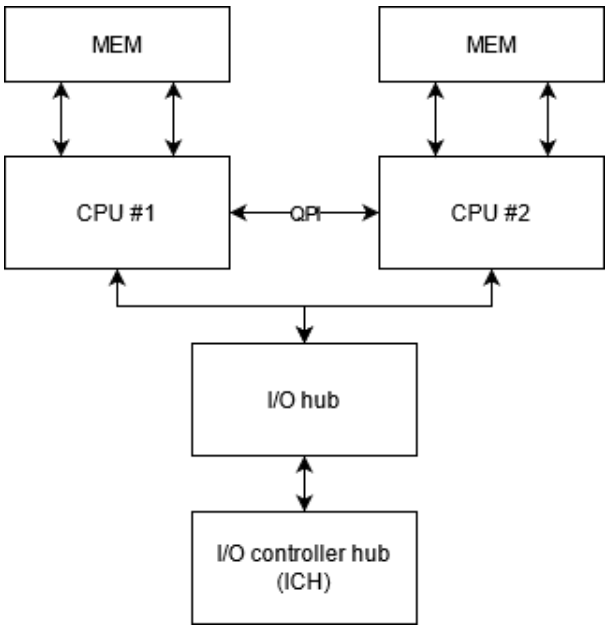
8.7.4. Non cache coherent

A nem cache koherens rendszerek nem garantálják, hogy a proceszor mindig az adat legfrissebb verziójával dolgozik, az ebből adódó problémás helyzeteket szoftveresen kell detektálni és kezelni. Ez a legjobban skálázható megoldás, tervezése és építése könnyű, viszont a programozása sokkal nehezebb. Általában sok gépes (több összekapcsolt szerver) rendszerekben használják. Ilyen processzor például az Intel Xeon és az AMD Opteron.

8.7.5. Gyakorlati példák

Quick Path Interconnect

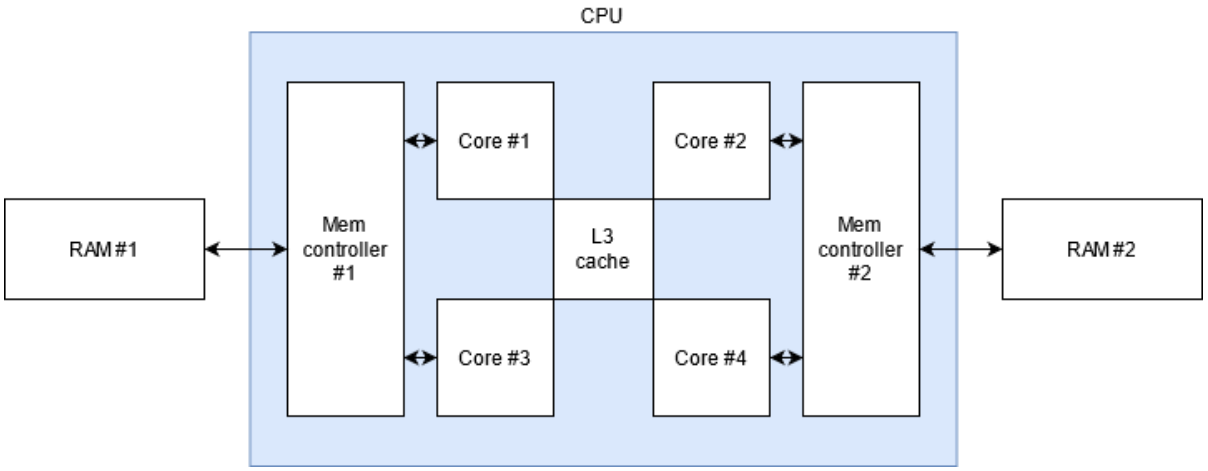
A modern többmagos processzorokban a klasszikus FSB felépítés helyett a QPI (Quick Path Interconnect) technológiát használják, ahol a CPU-k egy nagy sebességű buszon (QPI) kommunikálnak egymással (8.4. ábra). Ennek előnye, hogy a memória kontrollerek a processzorok lapkáira vannak integrálva, így a memória elérése gyorsul. Mivel így már a CPU-knak külön memóriaterületet kell kezelniük, ez egy NUMA architektúrának számít.



8.4. ábra. A Quick Path Interconnect felépítése

Intel Core i

A mai Core i architektúráknál már a processzorokon belüli magok is eltérő memória területeket kezelnek. Egy 4 magos CPU-nál két-két mag kap egy közös memóriavezérlőt, a magok pedig egy, a QPI-hez hasonló buszrendszeren kommunikálnak. A gyors adatelérés érdekében a magok közös L3 cache-t használnak (8.5. ábra).



8.5. ábra. A modern Intel Core i processzorok felépítése

8.8. Összegzés

A modern több magos és több processzoros rendszerekben inkább NUMA felépítést alkalmaznak. Az előbb bemutatott Core i processzor is egy példa arra, hogy a nagy, több gépes rendszereken kívül a mikroarchitektúrák szintjén is megtalálhatók a NUMA jellemzők. A szerverekben cache koherens NUMA rendszerek dominálnak, a nem cache koherens technológiákkal pedig több gépből álló rendszerek építhetők.

9. fejezet

A DDR4 memória

9.1. Bevezetés

A DDR3 memóriák viszonylag alacsony sávszélessége és magas fogyasztása miatt szükségessé vált a RAM-ok továbbfejlesztése, így született meg a DDR4.

9.2. Összehasonlítás

A DDR4-es memóriák alacsonyabb feszültségen üzemelnek, így kevesebbet fogyasztanak. Ezen kívül több pines csatlakozót használ, így nem kompatibilis a DDR3-al. Mindkettő 8n prefetch eljárást alkalmaz. Újdonság a DDR4-nél a bank groupok létrehozása. Míg a DDR3-nál 1 chip 8 különálló bankot tartalmaz, DDR4-nél 1 chip 4 db 4 bankból álló csoportot.

	DDR3	DDR4
Feszültség	1,5-1,35V	1,2V
Csatlakozó	240 pin	284 pin
Prefetch	8n	8n
Bankok	8 x 1	4 x 4

9.3. A bank groupok

A bankok csoportosításának előnye, hogy a groupokból a vezérlés egyszerre kettőt vagy négyet is kiválaszthat, ami a sávszélesség növekedéséhez vezet (mivel a memória műveletek párhuzamosan futnak le).

9.4. Prefetch

A prefetch technológia két okból maradt ugyanúgy 8n:

- a 16n-hez még több pinre lett volna szükség és
- a 16n nem illeszkedik a 64 byte-os cache vonal mérethez (ez csökkentette volna a teljesítményt).

9.5. További újítások

- CRC (Cyclic Redundancy Check) ellenőrzés → véletlenszerű anomáliákat is érzékeli (és akár javítja is), biztosabb az adat olvasása.
- Chipenkénti extra paritás.
- ODT (On-Die Termination): chipenkénti lezárás, a feszültség szabályozásban segít, így alacsonyabb feszültségen is stabilan működik a memória.

9.6. Értékelés

A kisebb feszültség eléréséhez szükséges volt a memóriacellák órajelének csökkentése a DDR3-hoz képest, ez látható a következő táblázatban:

Effektív órajel	f_{core}	Max. sávszélesség	Késleltetés (időzítés)	Elérési idő
DDR3-2133 MHz	266 MHz	17066 MB/s	14-14-14-34	~13,1 ns
DDR4-2133 MHz	133 MHz	17066 MB/s	15-15-15-35	~14,1 ns
DDR4-2400 MHz	150 MHZ	19200 MB/s	16-16-16-39	~13,3 ns

Látható, hogy a DDR4-es memóriák alacsonyabb frekvencia mellett nagyobb sávszélességet tudnak biztosítani, mint a DDR3, ugyanakkor növekedett a késleltetés és az elérési idő. A cache-ek használata (L3) miatt viszont a sávszélesség fontosabb, mint a késleltetés mértéke, ezért a gyakorlati alkalmazásokban (pl. multimédia) a DDR4 előrelépést jelentett. További előnye a DDR4-nek, hogy az effektív órajel tovább növelhető az alacsonyabb magfrekvencia miatt. Míg a DDR3 felső korlátja 2133 MHz, a DDR4 3000 MHz fölé is mehet.

10. fejezet

Cache tárak

10.1. Bevezetés

A cache tárak (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 címezhező, így a kezelését a programozó helyett a hardver végzi. Elhelyezkedése legtöbbször a processzor lapkáján található.

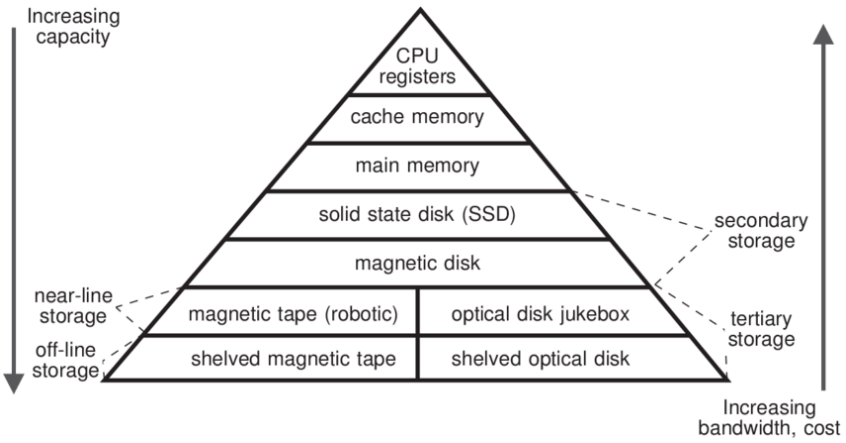
Definíció: a cache az adatok és utasítások átmeneti tárolására szolgáló, gyors működésű tároló.

10.2. Történeti áttekintés

A cache elterjedése és fejlődése kb. 1980-tól kezdődött, előtte még terveztek processzorokat cache nélkül. 1980-tól 2000-ig kb. milliószorosára nőtt a processzorok teljesítménye, míg a memória elérés csak kb. százszorosára. Ez tette szükségessé a gyorsítótárak fejlesztését.

10.3. Elhelyezés a tárolók piramisában

A piramis második szintjén helyezkedik el a cache (10.1. ábra). Az ábrán felsorolt tárolók közül a felső hármat kezeli közvetlenül a processzor.



10.1. ábra. A tárolók piramis modelle

10.4. Adatátvitel

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 adatátvitel ezzel szemben lehet byte szintű is. Fontos, hogy a cache kisebb mérete miatt mindig dönteni kell, hogy a memóriából mely adatokat töltse be a rendszer.

10.5. Felosztás

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

- az adatok és utasítások általában elég függetlenül kezelhetők
- előfordulhat, hogy egy programrészlet sok adattal dolgozik (ekkor közös cache esetén előfordulhatna, hogy a sok adat miatt kikerülnek a programhoz tartozó utasítások, ami lassuláshoz vezetne)
- utasítás cache-nél csak az olvasást kell gyorsítani, míg adatoknál az írást és az olvasást is → eltérő technológiák

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

10.6. Tervezési szempontok

A cache tervezésének alapvető dilemmája, hogy el kell dönteni melyik a fontosabb: a sebesség, vagy a találti arány. Mivel a gyorsítótárakban az adatokat keresni kell, minél nagyobb a kapacitása, annál hosszabb időbe telik a keresett adatot megtalálni. Ezért egy olyan kompromisszumot kell kötni, amikor még nem túl sok idő megtalálni egy bizonyos adatot, de már nem kell túl gyakran az operatív tárhoz fordulni. A processzorok gyorsulásával egyre több gyorsítótárra volt szükség, de a kapacitást már az előbb említettek miatt már nem volt hatékony növelni. Erre a megoldás a többszintű gyorsítótár bevezetése. Először az L1 mellett az L2, majd később az L3 cache is megjelent. A legkisebb és leggyorsabb az L1 cache, a legnagyobb és leglassabb pedig az L3.

10.7. A cache szintek összehasonlítása

Az L1 cache a regiszter utáni leggyorsabb tároló, általában a CPU órajelén üzemel. Mérete kicsi, nagyságrendileg 2x32 vagy 2x64 KByte. Késleltetése alacsony, kb. 1,3-1,5 ns (3-4 óraciklus). A késleltetések a többi cache szint között a következő táblázatban láthatóak.

	Elérési idő (sec)	Elérés idő (ciklus)
L1	1,3-1,5 ns	3-4
L2	4,5-8	10
L3	12-20	20-40
RAM	60-80	50-200

10.8. Gyakorlati példák

- Intel 80486: 128 kB L1
- Intel P1/P2: 128 kB L1, 512 kB L2
- Intel P4: 256 kB L1, 1 MB L2

- Intel Core 2: 512 kB L1, 2-8 MB L2
- Intel Core i: 2x32 kB L1, 256 kB, 3-8 MB L3

10.9. Memória és cache közötti kapcsolat

10.9.1. Példa a címzésre

Egy 2 GB RAM felosztható 512 darab 4 MB-os blokkra. A címeket kétutas asszociatív cache esetén kétfelé lehet osztani és ezeket eltárolni a blokkokban.

10.9.2. Adat szinkronizáció

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.

10.9.3. Címek tárolása

A cache-ben a memória egyes, egymást követő rekeszeinek tartalmát tároljuk, de ezek mellé el kell tárolni az adatok memóriabeli 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 meghatározható. A címnek azt a részét, amelyet a cache-ben elhelyez a CPU, és ami alapján a kiválasztás történik, tagnek 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 fizikai, a másodikban virtuális címekből történik a tag származtatása. A virtuális címek használatának hátránya, hogy a tárolandó tag nagyobb, mivel a virtuális címtér is nagyobb és hosszabbak a címek. További hátrány, hogy 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 tageket használnak.

10.9.4. Visszakeresés

A visszakeresés módja az ún. tartalom szerinti visszakeresés (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 hitről beszélünk, ellenkező esetben cache miss lép fel.

10.10. 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%.

10.11. Cache miss

Ha a keresett adat nem található meg a gyorsítótárban, a CPU a RAM-ból olvas, viszont a regiszeren kívül a cache-be is betölti.

10.12. Replacement policy

A cache tároló tartalmának cseréjekor a találati arány fenntartása érdekében szükséges a megfelelő helyettesítési 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.

10.13. Állapot bitek

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:

- V (valid) bit
- D (dirty) bit

10.13.1. Valid bit

A valid bit a cache tartalmának érvényességét jelzi a cache sorra vonatkozóan. Ha be van állítva, az adat a megadott című tárolóhelyhez tartozik, és aktuálisan érvényes. Például törlés (flushing cache line) esetén a V bit 0-ra lesz beállítva, ezzel jelzi a processzor számára, hogy az adott területre szabadon lehet írni. Minden blokkhoz legalább egy V bit tartozik.

10.13.2. Dirty bit

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

10.14. Jellemzők

A cache-eket jellemző paraméterek:

- méret (32kB - 20MB)
- elhelyezkedés (on chip - processzorlapkán, vagy off-chip - különálló)
- blokk méret: a fő tár és a cache között egy egységben mozgatott adatmennyiség (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 algoritmus (replacement policy): meghatározza azt a módot, ahogy a felesleges blokkot a cache-ben kiválasztjuk egy szükséges új blokk betöltésekor. Ilyen algoritmusok pl.:
  - FIFO - legrégebben betöltött blokkot írja felül
  - LIFO - legutoljára betöltött blokkot írja felül
  - LFU (Least Frequently Used) - legritkábban használt blokkot írja felül
  - LRO (Least Recently Used) - legrégebben használt blokkot írja felül
- adat aktualizálási módszer
  - write through - az adat változása esetén azonnal visszaírásra kerül az operatív tárba (biztosabb)
  - write back - csak az adott cache line felülírása előtt aktualizál (gyorsabb, gyakrabban használt, de áramszünet esetén adatvesztés léphet fel, ezért pl. a RAID vezérlők esetében akkumulátorral támogatják a cache-t)
- koherencia mechanizmus: az a módszer, amely biztosítja a fő tár és a cache tárak tartalmának egyezőségét

10.15. Típusok

10.15.1. Teljesen asszociatív cache

Egy beolvasott blokk a cache-ben bárhová (bármelyik sorba) kerülhet. Előnye a nagy találati arány és a rugalmasság, mivel mindig a legoptimálisabb adatokat tölthetjük be. Keresésnél a CPU a tageket vizsgálja minden sorban, egyszerre, mivel az adat bárhol lehet. Ennek megvalósításához szükséges n darab párhuzamos összehasonlító áramkör, ami viszont drága, bonyolult és nagyobb a fogyasztása.

10.15.2. Direct mapping

Ennél a megoldásnál minden memória blokk csak egy bizonyos cache line-ba tölthető be. A blokk helyét a blokk sorszám határozza meg. Mivel egy cache cellához 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. Előnye viszont a gyors visszakeresés és az olcsó megvalósítás (mivel csak egy összehasonlító áramkör kell).

10.15.3. n-way (set) associative cache

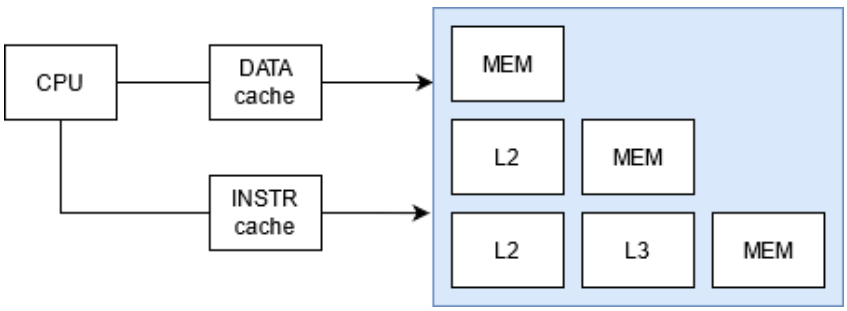
Az előző két módszer közötti kompromisszumot jelenti ez a módszer. n jellemzően 2, 4, 8 vagy 16. Az n változó meghatározza, hogy egy adott blokk hány cache line-ba kerülhet. Például egy 4 utas asszociatív cache-nél egy blokk 4 cache line-ba kerülhet. Eredmény, hogy a CPU a csoport index alapján 4 darab blokkra tudja szűkíteni a keresést, tehát csak 4 összehasonlító áramkör kell. Előny, hogy rugalmasabb és nagyobb találati arányú mint a direct mapping, de olcsóbb, mint a teljesen asszociatív. Az Intel általában 8 utas asszociatív cache-t használ.

10.15.4. Sector mapping cache

A gyakorlatban ritkán használt, itt a blokk helye kötött a csoportokon belül, viszont a csoport bárhová kerülhet.

10.16. Cache hierarchia

A gyorsítótárak hierarchiája látható a 10.2. ábrán. Régebbi rendszerekben csak L1 adat és utasítás cache volt, ezek fölött csak a memória állt, később megjelent az L2, majd az L3 cache is.



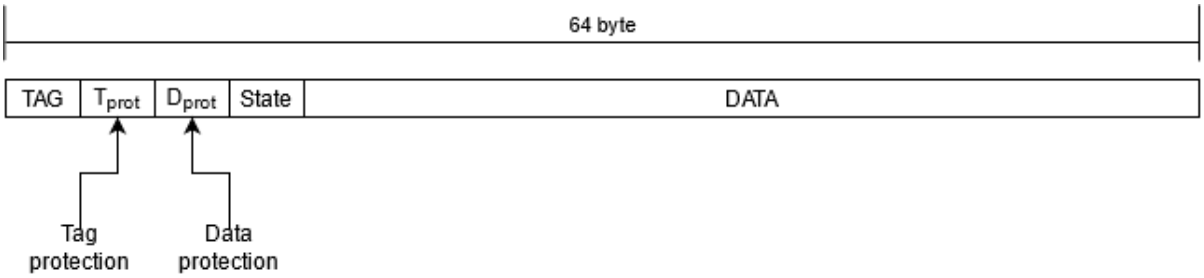
10.2. ábra. A gyorsítótárak hierachiája

10.17. Cache line felépítése

Példa: 64 byte-os cache line esetén a cache line részei:

- Directory Entry: tag és egyéb állapotjelzők.
- Data: maga az adat.

Ennek a felépítése látható a 10.3. ábrán.



10.3. ábra. A cache line felépítése

10.18. Adat szervezési módok

Többszintű cache-ek esetén megkülönböztetünk inclusive és exclusive gyorsítótárakat.

10.18.1. Exclusive cache

Exclusive cache esetében a tárolók egymástól függetlenek, nem tartalmazzák egymás adatait, így ugyanaz az adat nem lehet jelen egyszerre több tárban. Az adatok betöltése kétféleképpen történhet. Az egyik lehetőség, hogy először az L1-be töltjük be az adatokat, majd ami nem fér bele, azt az L2-be, és ami oda se, azt az L3-ba. Ilyenkor nevezzük az L3-at victim cache-nek is.

10.18.2. Inclusive cache

Inclusive cache-eknél az L2 tartalmazza az L1 adatainak másolatát is. Hátrány, hogy csökken L2 szabad mérete és kétszer kell írni, viszont előny, hogy L1-ben minden sor szabadon cserélhető, mert L2-ben megvan (ahonnan majd kiírásra kerülhet). A másik előny, hogy több mag esetén, amikor egy másik mag cache-ében kell keresni az adatot, a duplikálás miatt elég L2-ben keresni. Mivel manapság elég nagy méretű cache-eket használnak, az előnyök miatt inkább inclusive cache-t alkalmaznak (főleg több magos processzoroknál).

10.19. Data prefetch logic

További gyorsítási lehetőség a data prefetch logic, ami általában 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.

10.20. Cache koherencia probléma

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

- snoopy
- snorf (ritka)
- könyvtár alapú
- MESI

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 cellát (valid bit segítségével)
- felülírja az új állapottal

Az első módszernél 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. A felülírásos rendszernél ilyen probléma nem áll fent, a CPU vagy mag egyszerűen direktben küldi el a változott adatot a többi CPU-nak vagy magnak. A gyakorlatban mégis inkább az invalidációt alkalmazzák, mivel egy invalidálási üzenet sokkal kisebb, mint egy teljes cache cella tartalma.

10.20.1. Snoopy protokoll

A magok vezérlői folyamatosan figyelik, "szaglásszák" a közvetítő adat buszt. Olyan tranzakciót keresnek, amik hatással vannak önmagukra. Például ha egy cache írási művelettel találkozik, aminek a tagje olyan, amit ő is tárol, azonnal invalidálja azt a cache line-t és elkéri az aktuális adatot a másik magtól. Előnye az egyszerű kiépíthetőség, de hátrány, hogy terheli a buszrendszert.

10.20.2. 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, a könyvtár vagy felülírja, vagy invalidálja a többi cache tartalmát. Ezzel a megoldással nincs szükség külön buszra. Elsősorban NUMA rendszerekben használatos.

10.20.3. MESI protokoll

A MESI a Modified Exclusive Shared Invalid rövidítése. A protokoll állapotjelzői:

- Modified: egy tár valid, a többi invalid - egyik tárban módosult de még nincs visszaírva
- Exclusive: érvényes adat, egyezik a fő tárral, és csak egy helyen van meg
- Shared: az adat a fő tárral egyezik, de több tárolóban is megtalálható
- Invalid: az adat érvénytelen