Korszerű számítógép architectúrák I.

Simon Péter [1]

2021. március 28.

[1] Durczy Levente előadásai alapján

Tartalomjegyzék

1. fejezet

Első előadás

2. fejezet

Függőségek

2.1. Bevezetés

A függőségek gátolják a párhuzamos végrehajtást.

2.2. Típusai

- adat
- vezérlés
- erősforrás

2.3. Adat függőségek

Probléma: az utasítás végrehajtáshoz 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}

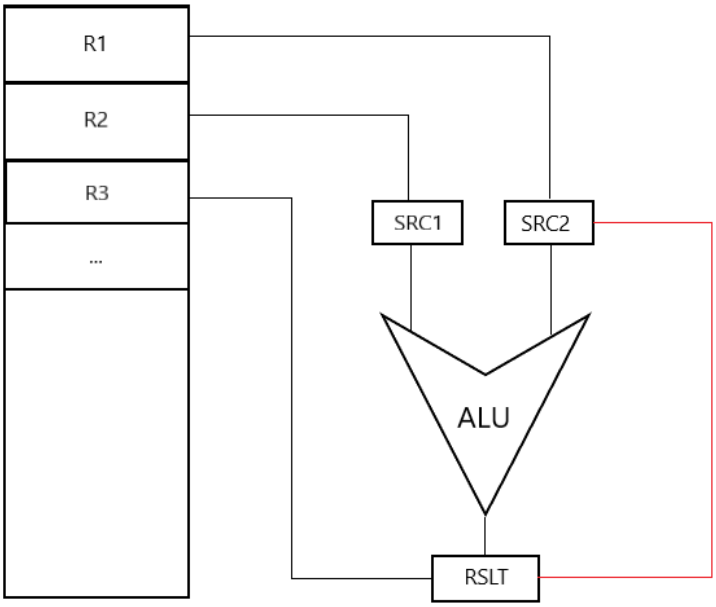
Következmény: a műveleteknek várakozniuk kell egymásra, két óraciklus késés keletkezik a futószalagon. Az ezeket követő utasításokhoz is be kell szúrni két NOP-ot, mivel a dekóder foglalt. Ezt a jelenséget teljesen nem lehet megszüntetni, viszont a fékező hatást csökkenthetjük.

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.

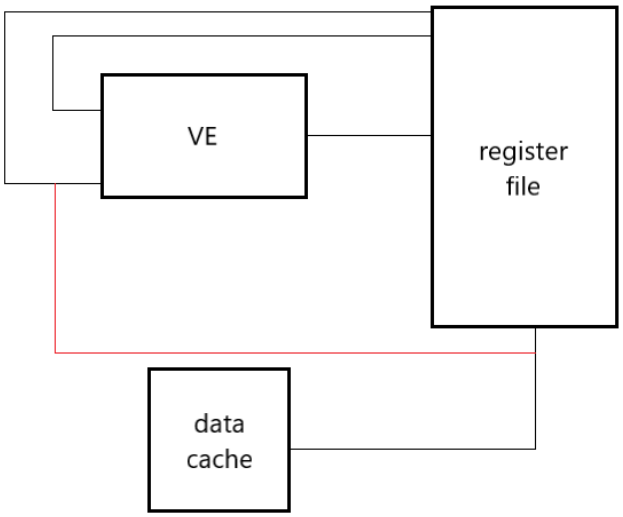


2.1. ábra. Az eredmény visszavezetése a forrás regiszterbe

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

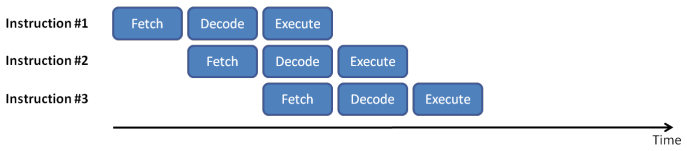
- átrendező puffer segítségével (pl. Intelnél ROB - ReOrder Buffer) vagy
- címkézéssel.

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:

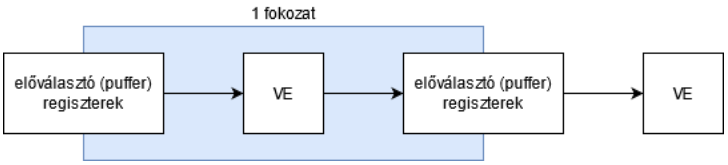
1. Fetch
2. MAR ← PC
   MDR ← [MAR]
   IR ← MDR
   PC ← PC+1

3.8. Fizikai megvalósítás

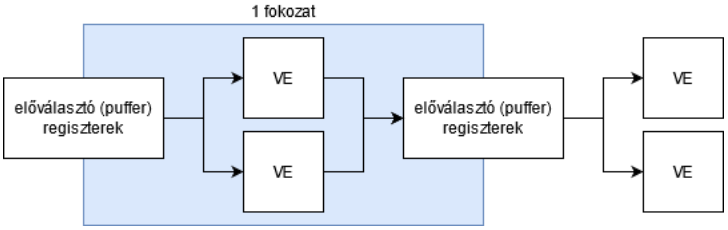
Alkalmazásuk alapján megkülönböztetünk univerzális és dedikált futószalagokat. Az univerzális minden művelet elvégzésére alkalmas, míg a dedikált speciális műveletekre képes. Hardveres szempontból az univerzális futószalag előnytelen, mivel sok tranzisztorra van szükség, a kialakítása bonyolult és drága, 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

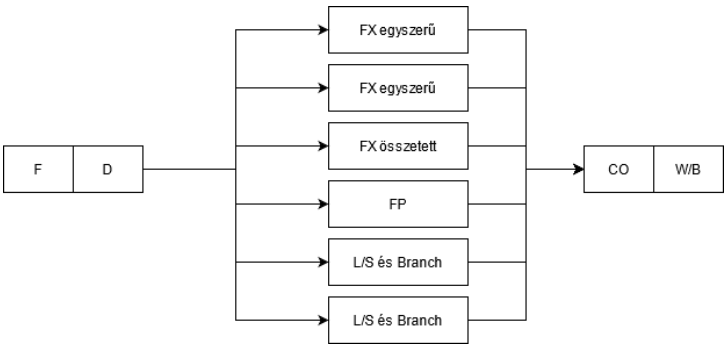


3.3. ábra. A végrehajtó egységek párhuzamosítása

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.

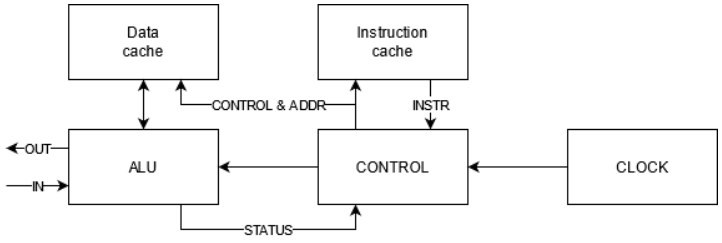
4.3. Harvard architektúra

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).

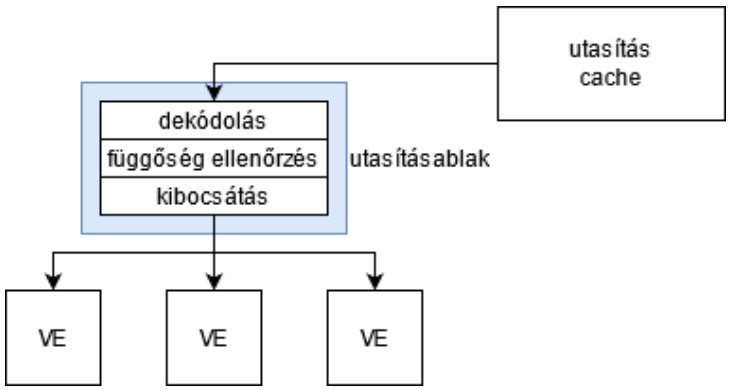
4.4. Első generációs szuperskalárok (keskeny szuperskalárok)

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:

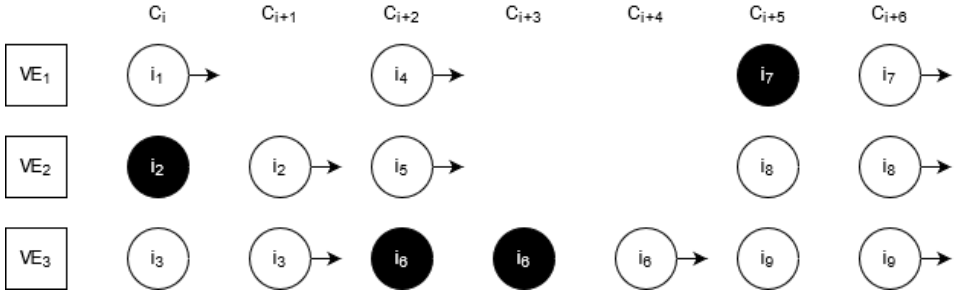
- A kibocsátott utasításokat egyenként pótoljuk.
- A kibocsátott utasításokat egyszerre pótoljuk (megvárjuk, hogy az összes utasítás kiürüljön).

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

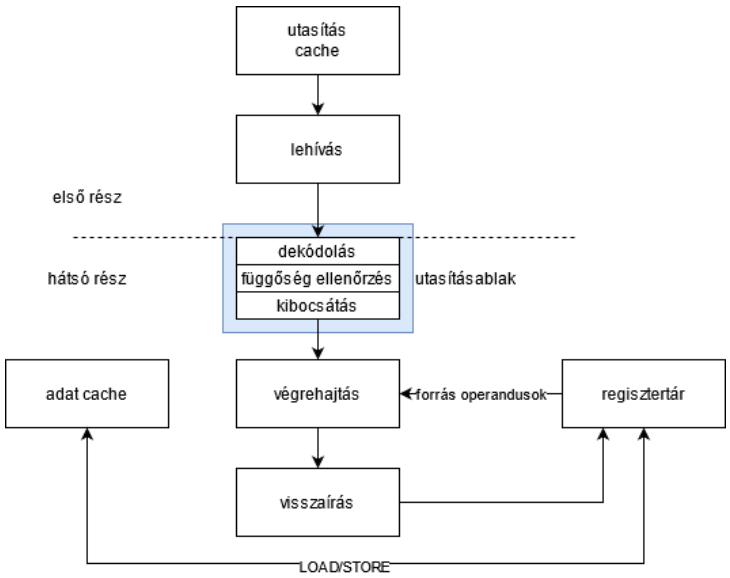
Egyszerre pótló és sorrendben kibocsátó utasításablak értékelése: a tapasztalat azt mutatja, hogy az ilyen elven működő processzor kibocsátási rátája közelít az 1 utasítás/ciklushoz. Tehát a jóval több tranzisztor és bonyolultabb felépítés ellenére nem sokkal gyorsabb, mint a futószalag processzorok.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

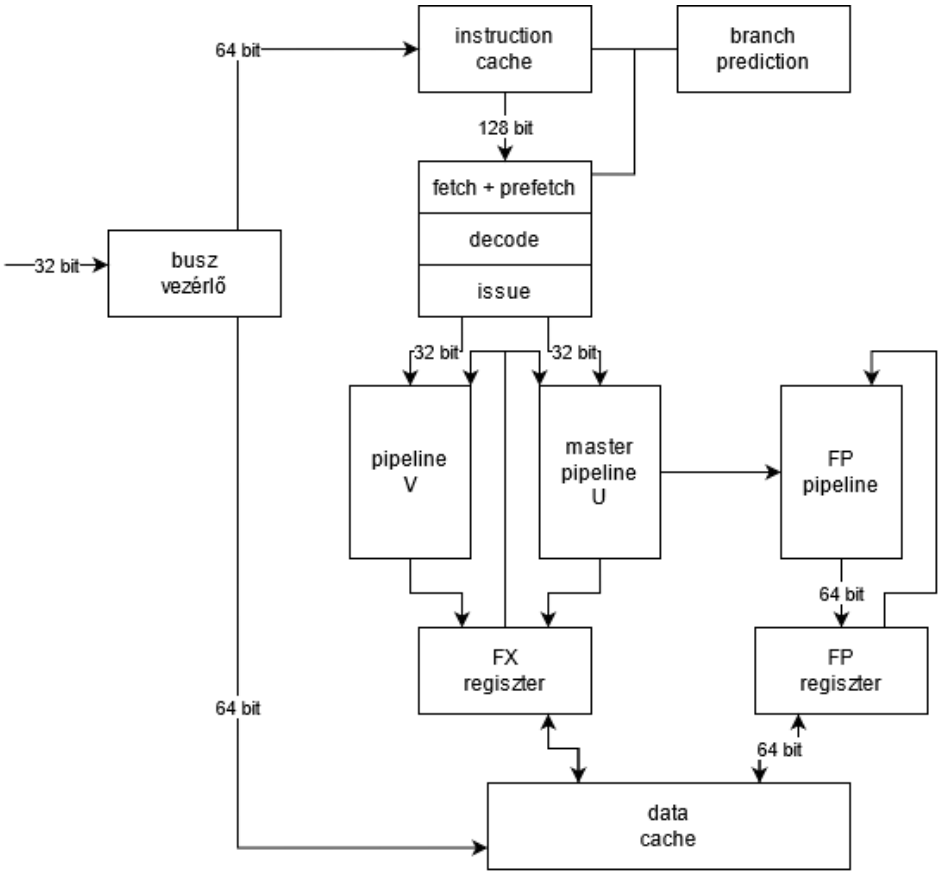
Az első generációs szuperskalároknál a következő szűk keresztmetszetek lépnek fel:

- 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

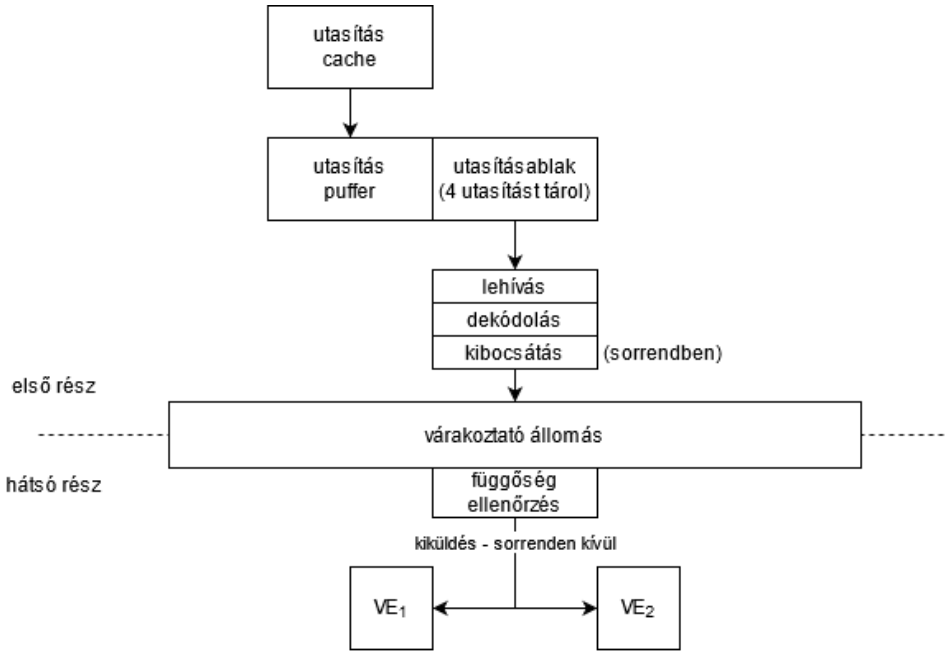
Mivel legtöbbször az elágazásnál ugrás történik, ezért a történet bitek értéke kezdetben 11. Amennyiben a következő elágazásnál mégis hibás volt az ugrás, a történeti bitek értéke 10-ra, gyenge elágazásra változik, tehát az ezt követő alkalommal megint meg fogja próbálni az ugrást. Ha harmadszorra is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

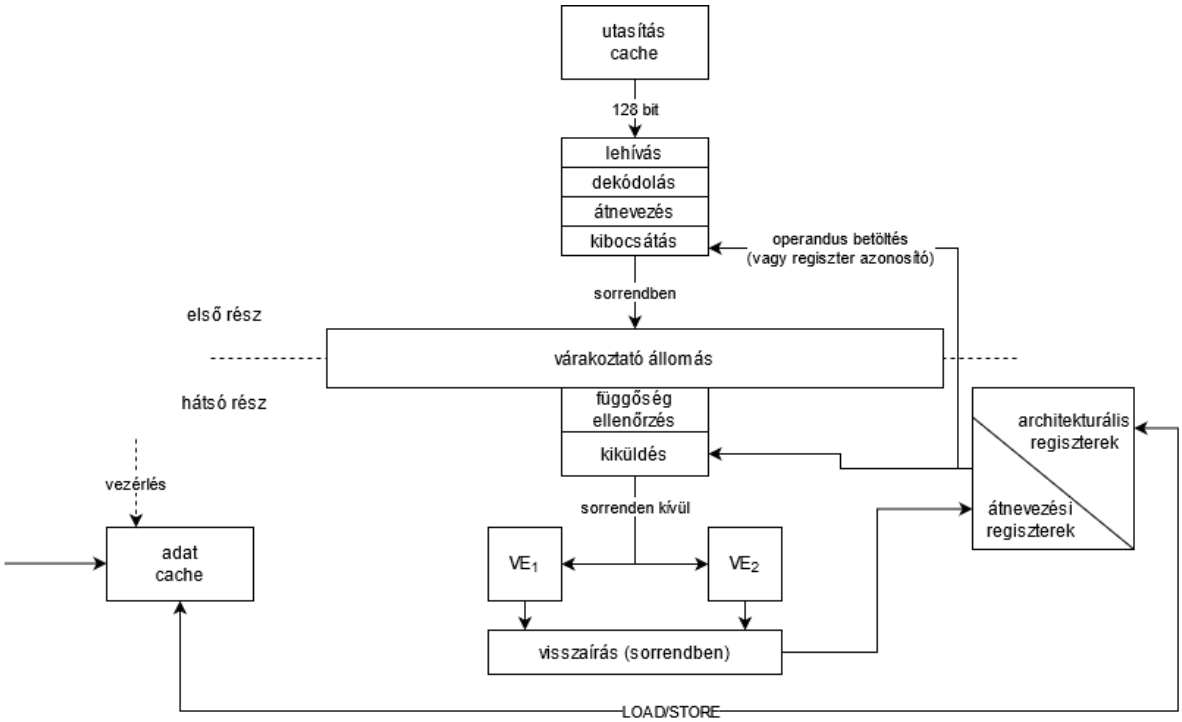
itt, a várakoztató állomásban történik meg, ahonnan a végrehajtó egységek felé sorrendben kerülnek kiküldésre az utasítások. Az operatív tár elérése a RISC architektúra miatt egy adat cache-en keresztül valósul meg, ami az architekturális regiszterekkel kétirányú kommunikációt folytat. Az utasítások 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

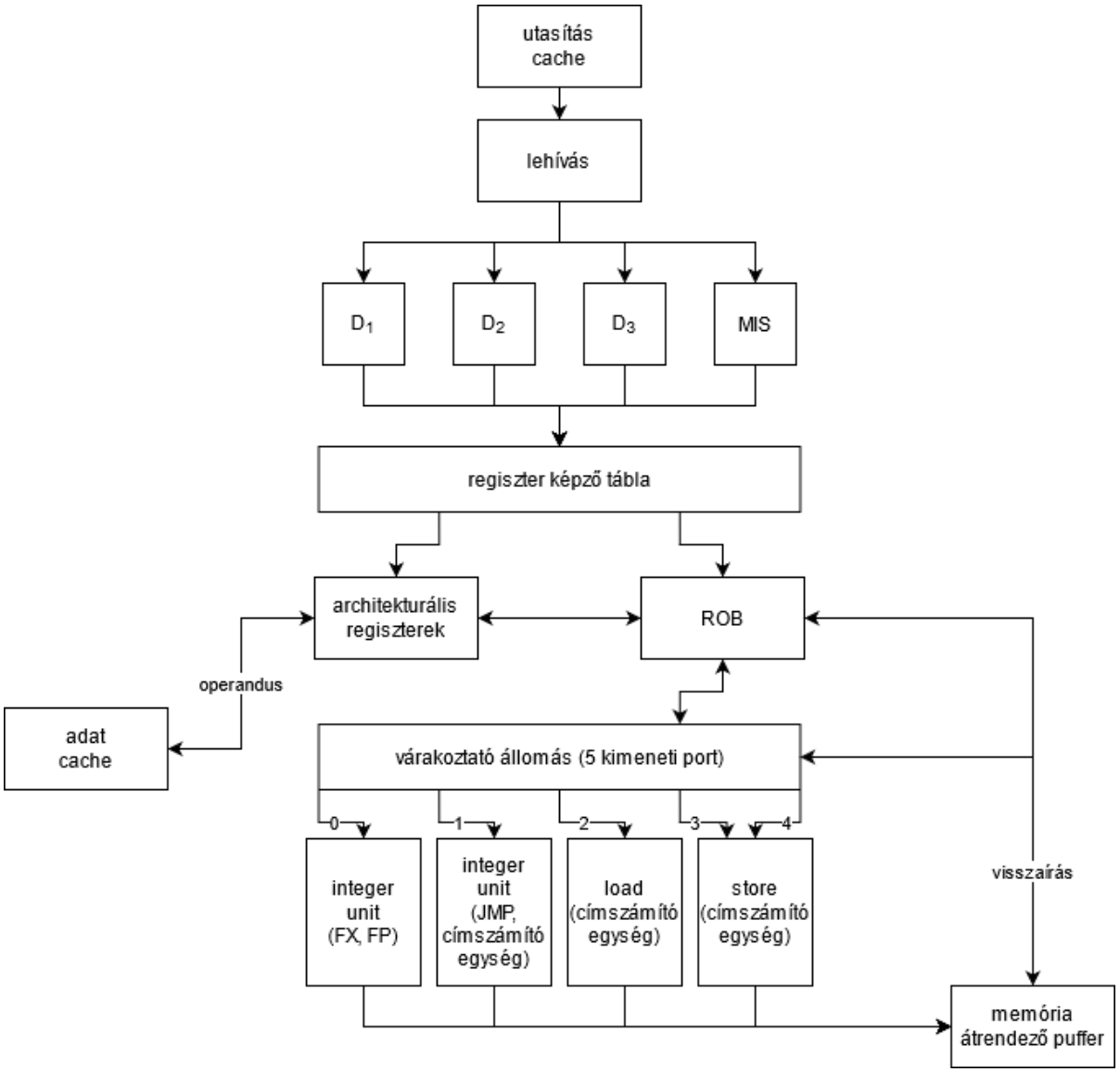
Tulajdonságai

- 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.