Korszerű számítógép architectúrák I.

Simon Péter [1]

2021. március 26.

[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}

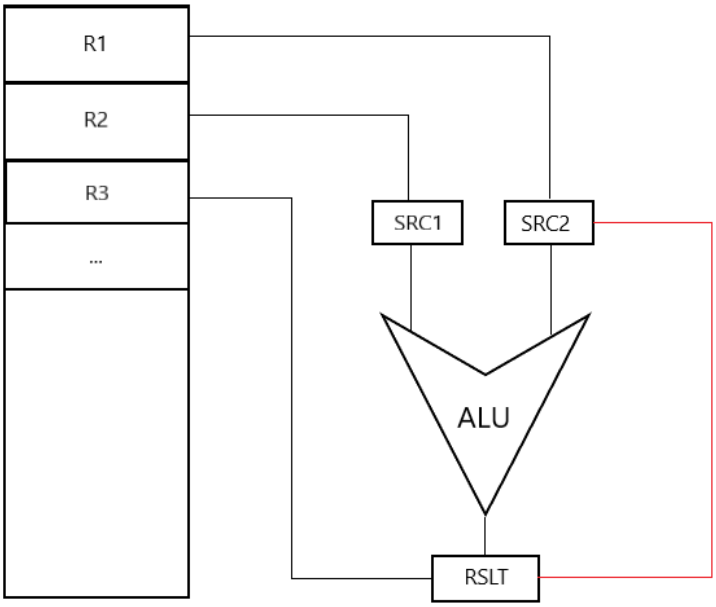
Következmény: a műveleteknek várakozniuk kell egymásra, két óraciklus késés keletkezik a futószalagon. Az ezeket követő utasításokhoz is be kell szúrni két NOP-ot, mivel a dekóder foglalt. Ezt a jelenséget teljesen nem lehet megszüntetni, viszont a fékező hatást csökkenthetjük.

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.

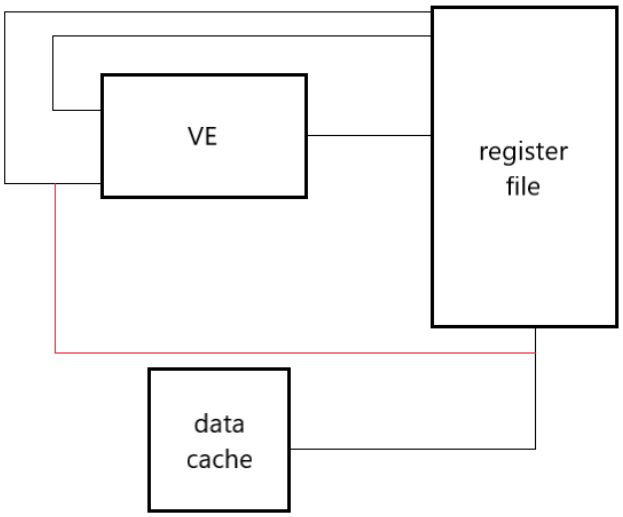


2.1. ábra. Az eredmény visszavezetése a forrás regiszterbe

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

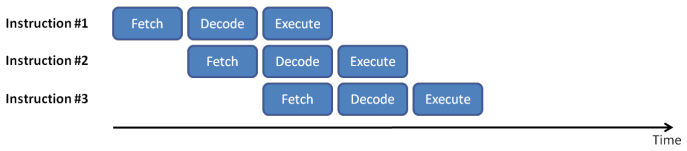
- átrendező puffer segítségével (pl. Intelnél ROB - ReOrder Buffer) vagy
- címkézéssel.

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:

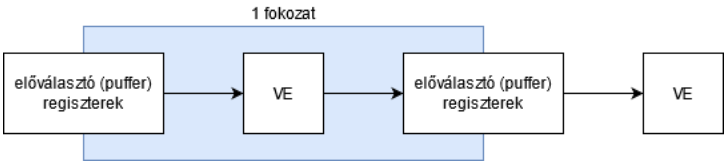
1. Fetch
2. MAR ← PC
   MDR ← [MAR]
   IR ← MDR
   PC ← PC+1

3.8. Fizikai megvalósítás

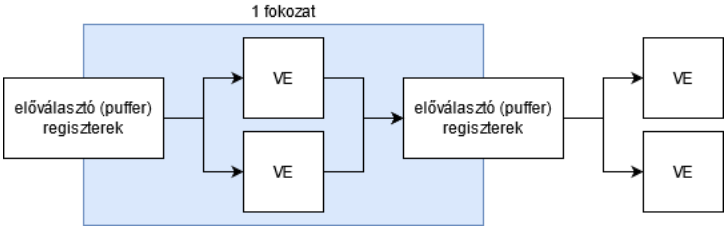
Alkalmazásuk alapján megkülönböztetünk univerzális és dedikált futószalagokat. Az univerzális minden művelet elvégzésére alkalmas, míg a dedikált speciális műveletekre képes. Hardveres szempontból az univerzális futószalag előnytelen, mivel sok tranzisztorra van szükség, a kialakítása bonyolult és drága,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

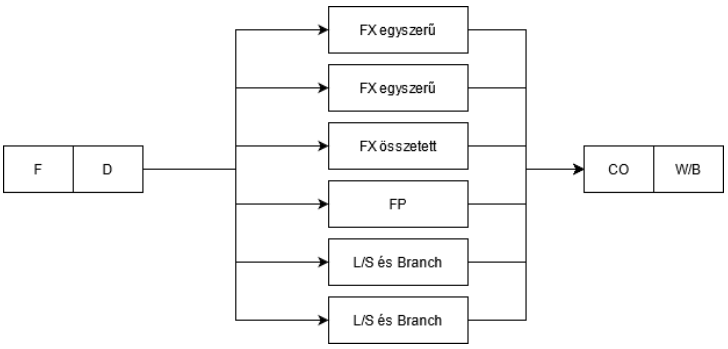


3.3. ábra. A végrehajtó egységek párhuzamosítása

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).