

# Korszerű számítógép architektúrák I.

Simon Péter <sup>1</sup>

2021. március 25.

<sup>1</sup>Durczy Levente előadásai alapján

1. fejezet

Első előadás

## 2. fejezet

# Függőségek

### 2.1. Bevezetés

A függőségek gátolják a párhuzamos végrehajtást.

### 2.2. Típusai

- adat
- vezérlés
- erőforrás

### 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:** feltételezzük, 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, r1 ; r3 = r1 * r2
; I2
SHL r3 ; r3 * 2
```

Az utasítások végrehajtásának időbeli sorrendje:

	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>
I <sub>1</sub>	F <sub>MUL</sub>	D <sub>r1, r2</sub>	E <sub>MUL</sub>	W/B <sub>r3</sub>	
I <sub>2</sub>		F <sub>SHL</sub>	D <sub>r3</sub>	E <sub>SHL</sub>	W/B <sub>r3</sub>

A probléma, hogy I<sub>2</sub> végrehajtása során, a dekódolási fázisban (t<sub>3</sub> időpillanat) szükség lenne az r3 regiszter értékére, viszont az csak t<sub>4</sub> időpillanatban áll elő (I<sub>1</sub> 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 <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>
I <sub>1</sub>	F <sub>MUL</sub>	D <sub>r1, r2</sub>	E <sub>MUL</sub>	W/B <sub>r3</sub>			
I <sub>2</sub>		F <sub>SHL</sub>	NOP	NOP	D <sub>r3</sub>	E <sub>SHL</sub>	W/B <sub>r3</sub>

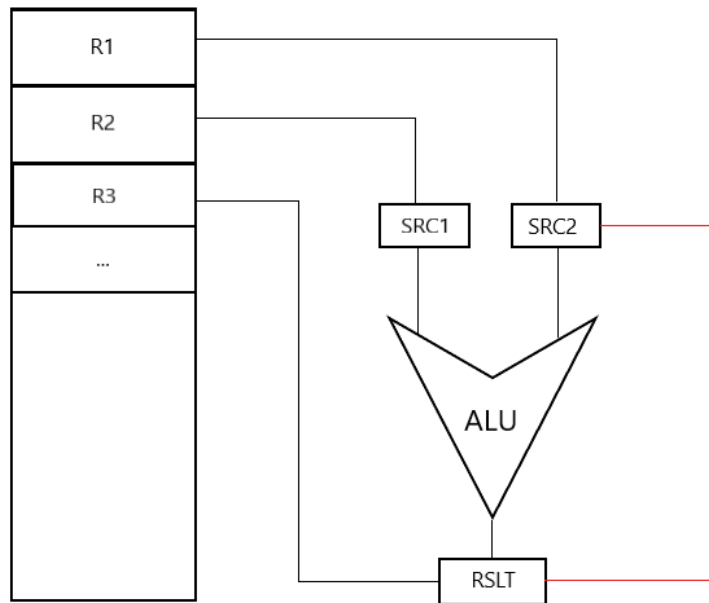
**Következmény:** a műveleteknek várakozniuk kell egymásra, két óraciklus késés keletkezik a futószalagon. Az ezeket követő utasításokhoz is be kell szűrni két NOP-ot, mivel a dekóder foglalt. Ezt a jelenséget teljesen nem lehet megszüntetni, viszont a fékező hatást csökkenthetjük.

**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 mutatja. Alapesetben a MUL utasítás végrehajtása során az adat az r<sub>1</sub> és r<sub>2</sub> regiszterekből az src<sub>1</sub>, illetve src<sub>2</sub> regiszterekbe kerül, majd a művelet elvégzése után a rslt rejtett regiszteren keresztül visszaírásra kerül r<sub>3</sub>-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 <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>
I <sub>1</sub>	F <sub>MUL</sub>	D <sub>r1, r2</sub>	E <sub>MUL</sub>	W/B <sub>r3</sub>			
I <sub>2</sub>		F <sub>SHL</sub>	D	E <sub>SHL</sub>	W/B <sub>r3</sub>		

Mivel már t<sub>3</sub> 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.

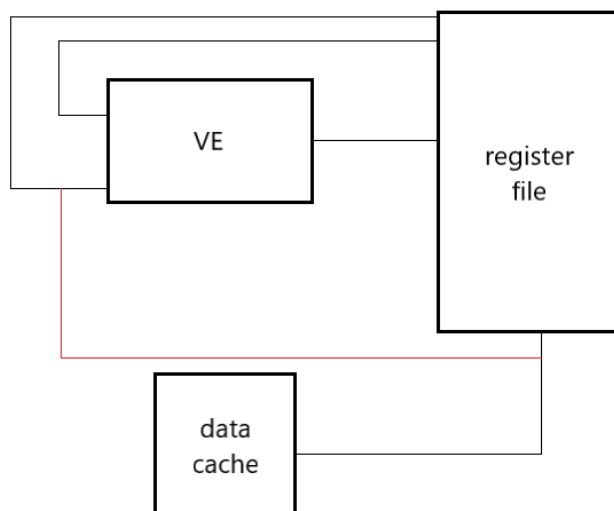


2.1. ábra. Az eredmény visszavezetése a forrás regiszterbe

### 2.3.3. Lehívási adatfüggőség

**Probléma:** a regiszterekben 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 előfordulhat, 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 assembly 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 előfordulhat, hogy a párhuzamos végrehajtás során  $I_1$  később fut le, mint  $I_2$ . 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 adat-függőségeknél. 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 <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>
MUL	F <sub>MUL</sub>	D	E	W/B		
JMP		F <sub>JMP</sub>	D	E	W/B	
ADD			F <sub>ADD</sub>	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 architektúrális 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 <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>
JMP	F <sub>JMP</sub>	D	E	W/B			
MUL		F <sub>MUL</sub>	D	E	W/B		
DIV			F <sub>DIV</sub>	D	E	W/B	
SHL				F <sub>SHL</sub>	D	E	W/B

A sorrend megváltoztatásával elértük, hogy amíg az ugrás végre nem hajtódik (t<sub>3</sub>), 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<sub>1</sub> később végez, mint I<sub>2</sub>, 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 ; túlcsordul -> 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.