ÓBUDAI EGYETEM

SZÁMÍTÓGÉP ARCHITEKTÚRÁK ALAPJAI

11-12. előadás

Durczy Levente

2025/2026/1

ÓBUDAI EGYETEM

PÁRHUZAMOS ARCHITEKTÚRÁK

1. Számítógép architektúrák osztályozása

Miben mérik ma egy CPU teljesítményét?

Pl.: FLOPS (Intel 486 CPU, 1989-től integrált FP végrehajtó egység)

Az AI is főleg lebegőpontos számításokkal operál!

Desktop CPU: ~ 10-20 tera FLOPS (10^{13}) kb. 16-32 FP64 utasítás / óraciklus

illetve kb. 32-64 FP32 utasítás / óraciklus

Szuper számítógépek (~ 2017-től): 10^{17} FLOPS (100 billiárd, vagy 100 peta FLOPS)

Ennél talán csak az 1946 augusztus 1-én bevezetett új Forint értéke volt nagyobb:

1 Ft - 400 kvadrillió pengőt (400.10^{24}) ért. ☺

Hogyan érhető ez el? ⟶ Párhuzamosítással!

Mai CPU tervezési irány:

- többmagos rendszerek (SoC), párhuzamos végrehajtás maximalizálása
- nagyteljesítményű magok
- energiahatékony magok
- dedikált neurális hardver gyorsító (AI neural engine)

Pl.: Apple M3 max (2023 okt., 3nm-es technológia):

- 92 milliárd tranzisztor (GAAFET)
- 16 mag
- 30-40 magos GPU
- $18x10^{18}$ művelet másodpercenként (neural engine)

Összegezve:

Ebben a félévben magasabb szinteken ismerjük meg a számítógép architektúrák felépítését.

A párhuzamosság típusai:

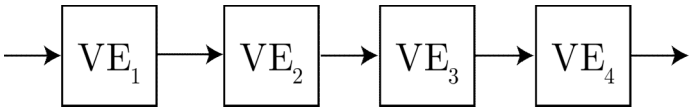
Funkció szerint:

- rendelkezésre álló párhuzamosság: a feladatokban/programokban rejlő párhuzamosság

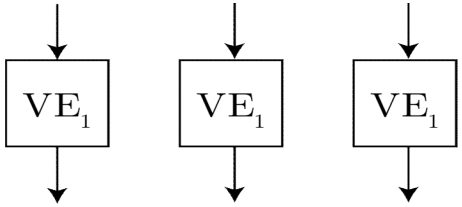
- kihasználható párhuzamosság: a végrehajtás során valóban kihasználható párhuzamosság
  (pl.: régi architektúrák ⟶ nem voltak képesek sok párhuzamosságra)

Elhelyezkedés szerint:

- <u>időbeli párhuzamosság</u>: több végrehajtó egység működés csak időben elcsúsztatva egymás mellett és általában nem ugyanazt a feladatot csinálják. Például: futószalag



- <u>térbeli párhuzamosság</u>: több, azonos típusú végrehajtó egység egyidejű működése.



Típus szerint:

Adatpárhuzamosság:

két módon hasznosítható:

- adatpárhuzamos architektúrák segítségével: adat elemeken párhuzamos vagy futószalag elvű műveletek végrehajtását teszik lehetővé

- **átalakítható** funkcionális párhuzamossággá: az adat elemeken végrehajtandó műveletek ciklusok formájában történő megfogalmazása

funkcionális párhuzamosság:

- a feladat logikájából következő párhuzamosság. Ez a feladatmegoldás minden formális leírásában megjelenik kisebb-nagyobb mértékben.
  (pl.: folyamatábrák, adatfolyamgráfok, …)

Parancsnyelveken írt programok esetén a rendelkezésre álló funkcionális párhuzamosságot különböző szinteken tudjuk megkülönböztetni.
A parancsok valójában kérelmek, melyeket az OS magjának (Kernel) szolgáltató rutinjai, vagy önálló processzek szolgálnak ki (scriptek, shell, bash)

A funkcionális párhuzamosság szintjei:

- Utasítás szintű párhuzamosság (ILP – Instruction Level Parallelism): program utasítások párhuzamosan történő végrehajtása
- Ciklus szintű párhuzamosság: egymást követő iterációk párhuzamos végrehajtása. Ezt a (egymást követő iterációk közötti) függőségek akadályozhatják.
- Eljárás szintű párhuzamosság: párhuzamosan végrehajtható eljárások formájában jelenik meg. Az ezen a szinten rendelkezésre álló párhuzamosság mértéke leginkábban a feladat jellegétől függ.
- Program szintű párhuzamosság: egymástól független programok párhuzamos futtatása
- Felhasználó szintű párhuzamosság: több, egymástól független felhasználó egyidejű kiszolgálása. Például szerverek, időosztásos rendszerek.

A számítások felgyorsítása érdekében az architektúrák, az OS-ek és a fordító programok (Compiler) egyaránt törekednek a rendelkezésre álló párhuzamosságok hasznosítására, melyek funkcionális párhuzamosságok esetében a következő szinteken hasznosíthatók:

Rendelkezésre álló párhuzamosságok hasznosítása:

Utasítás szinten rendelkezésre álló párhuzamosság:

- utasítás szinten párhuzamos architektúrával (ILP), vagy
- erre a célra szolgáló fordítóprogram segítségével

Ciklus és eljárás szintű párhuzamosság:

Általában szálak vagy folyamatok formájában használhatóak ki.

A szál vagy folyamat a tárgykód legkisebb önállóan végrehajtható része ➡ **Párhuzamosítható**!

(olyan részfeladatokat fogalmaznak meg, melyek bizonyos feltételek mellett párhuzamosan hajthatóak végre!)

Szálakat vagy folyamatokat létrehozhat:

- Programozó: párhuzamos nyelveket használva, amiben lehet elágazásoka és párhuzamosan végrehajtható részeket létrehozni (FORK, JOIN)
- OS, ami támogatja a többszálas vagy többfeladatos végrehajtást (MS Windows 98-tól)
- Párhuzamos fordító: magas szintű programnyelvek esetében

Program és felhasználói szintű párhuzamosság:
Általában programok vagy párhuzamos rendszerek segítségével használható ki. Szükséges hozzá megfelelő hardver és szoftver, ami ezt támogatja.

Szemcsézettség:

A funkcionális párhuzamosság kihasználásánál különböző szemcsézettségi szinteket tudunk megkülönböztetni:

Finom szemcsézettség:	Utasítás szint	(alacsony szintű párhuzamosság)
⬇	Szál szint	
	Folyamat szint	
Durva szemcsézettség:	Felhasználó szint	(magas szintű párhuzamosság)

Általában kijelenthető, hogy alacsony szinteken rendelkezésre álló párhuzamosság nagy valószínűséggel párhuzamos architektúrákkal vagy párhuzamos compilerekkel közvetlenül használhatóak ki, míg a magasabb szintű párhuzamosság inkább többszálas vagy többfeladatos OS-ek alatti konkurens végrehajtással hasznosítható.

ÓBUDAI EGYETEM

Compiler:

A fordítóprogram, ami lefordítja a magas szintű programnyelven írt programot a processzor számára érthető formátumra. Két feladata van:

1. Analizálni a forrásnyelvű program karaktersorozatát

   Részei:

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

2. Szintetizálni (létrehozni) a tárgykódot

   Részei:

- Kódgenerálás: ez lehet assembly kód vagy gépi kód

  (Amennyiben assemblyből generálódik a gépi kód, azt az assembler nevű compiler végzi)

- Kódoptimalizálás: párhuzamos végrehajtásnál független részeket keres, amiket végre lehet hajtani mindenféle előfeltétel nélkül!

Párhuzamos architektúrák osztályozása

Klasszikus modell - Flynn-féle osztályozás (Michael J. Flynn prof. Stanford university)**:**

Elhelyezés a tervezési térben:
Flynn-féle osztályozás a rendelkezésre álló vezérlő és a feldolgozási egységek számán alapul.
Négy fogalmat vezetett be:

SI (single instruction stream): egyszeres utasításfolyam. Az architektúra egyetlen vezérlő egységgel rendelkezik.
MI (multiple instruction stream): többszörös utasításfolyam. Az architektúra több egymástól elkülönülő utasításfolyamot tud egyidőben végrehajtani. Olyan vezérlőegységgel rendelkezik, mely egy időben több utasításfolyamot tud generálni.
SD (single data stream): egyszeres adatfolyam. Egyetlen CPU egyetlen adatfolyamot dolgoz fel.
MD(multiple data stream): többszörös adatfolyam. Több végrehajtó egység dolgoz fel több, egymástól független adatfolyamot.

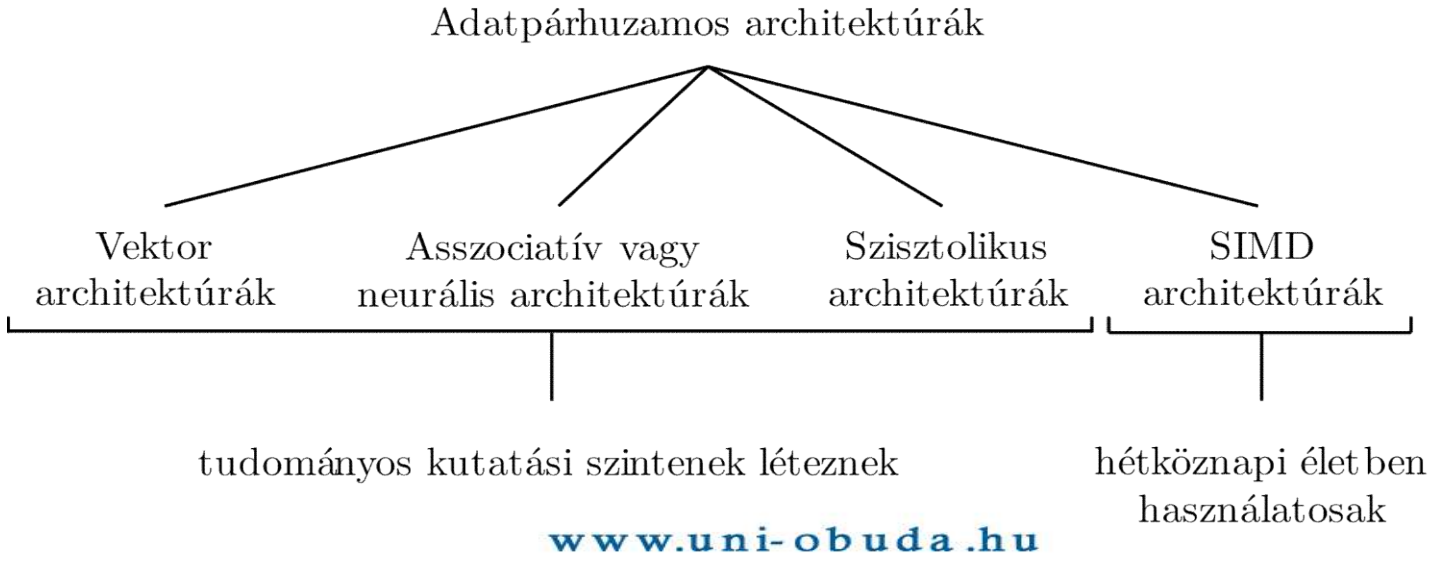
Ezekre építve a szempontok kombinálásával 4 osztályba lettek sorolva az architektúrák:

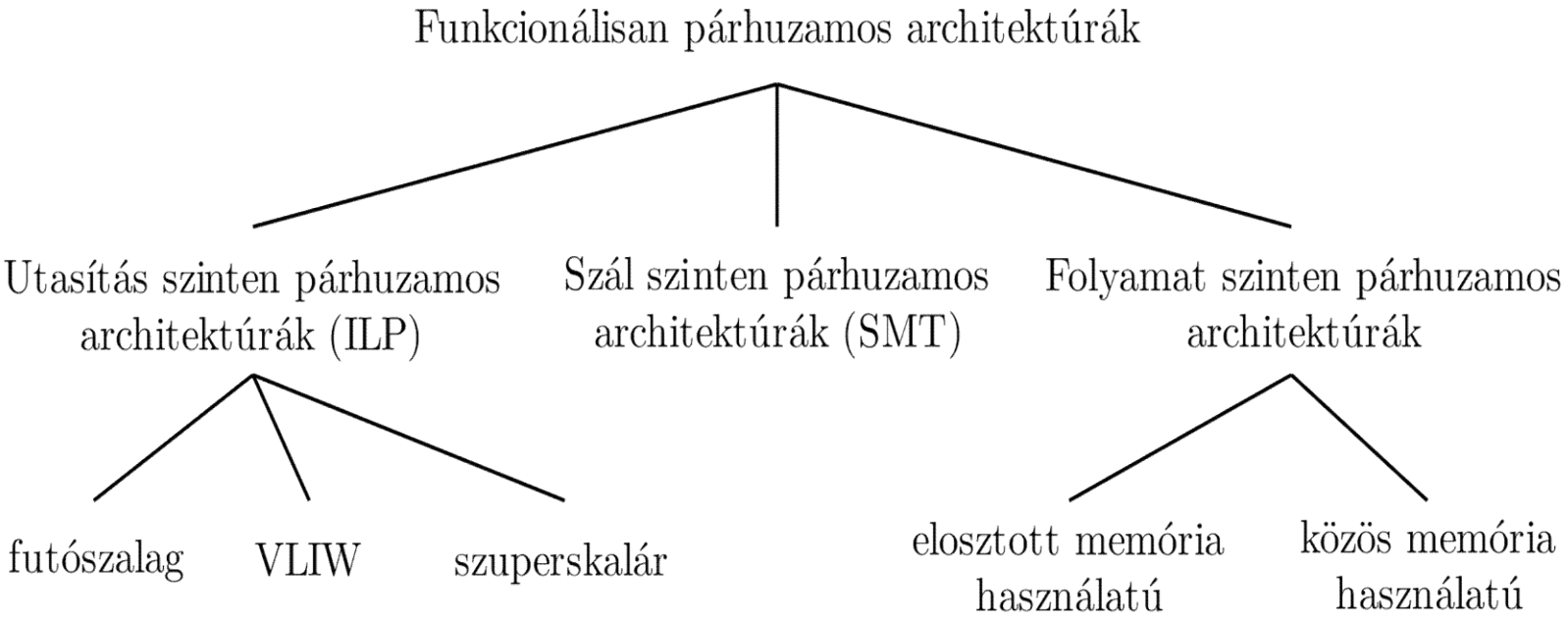
ÓBUDAI EGYETEM

SISD	SIMD	MISD	MIMD
Neumann modell (szekvenciális végrehajtás)	Multimédia feldolgozás (ugyanazon műveletek végrehajtása sok adaton)	Elméleti kategória	Teljes párhuzamos feldolgozás

Világos és egyszerű modell, de nem utal a párhuzamosság fajtájára, szintjére, módjára.

Az modern osztályozás mindezeket figyelembe veszi.
Megkülönböztet adatpárhuzamos és funkcionálisan párhuzamos architektúrákat.





A tárgy keretében jellemzően a funkcionálisan párhuzamos architektúrákat tárgyaljuk + SIMD.

Fejlődés iránya

1950-es években mikor megjelentek az első processzorok, akkor azok Neumann elvű processzorok voltak, szekvenciális utasítást végrehajtást biztosítottak.

1980-as években a személyi számítógépek esetében a párhuzamosság első megjelenési formája az időbeli párhuzamosság volt, mely elsősorban futószalag feldolgozás formájában jelentkezett.

Egyre nagyobb igény jelentkezett a sebesség növekedésre ➡ párhuzamos utasítás végrehajtás bevezetése!

Ennek két módja lehet:
- Többszörözés (több végrehajtó egység, térbeli vagy kibocsátási párhuzamosság)
- Futószalag elv (időbeli párhuzamosság)

Ezek a futószalag elvű processzorok az utasításokat sorrendben bocsátották ki, ezért ezeket skalár ILP CPU-knak nevezték (párhuzamos végrehajtás, de soros kibocsátás!)

1990-es évek elején jelentek meg a szuperskalár processzorok, melyek párhuzamos végrehajtással és párhuzamos kibocsátással is rendelkeztek).
Def.: óraciklusonként 1-nél több utasítást képes kibocsátani a dekódoló egységből!
Később kiegészültek utasításon belüli párhuzamossággal is (multimédia).

Az utasítás végrehajtás és az utasítás kibocsátás szorosan összefüggnek!

Minél több utasítást tud egy CPU párhuzamosan végrehajtani, annál több utasítás kibocsátására van szükség egy óraciklusban a hardver erőforrások hatékony kihasználásához!

2000-es évektől a fejlődésnek két iránya jelent meg: evolúciós és revolúciós.

Evolúciós: a feldolgozási szélesség 32 bitről 64 bitre nőtt. A logikai architektúra változatlan.
Például: x86 utasítás készlet (ISA) → x86_64

Revolúciós: teljesen új logikai architektúra (VLIW) + új utasításkészlet (IA64) (felhagytak vele)

A fejlődés eredményeként megjelentek a többmagos rendszerek, melyek a mai napig irányadóak.

Az utasítás végrehajtás elemi műveletei:

Egy utasítás végrehajtás négy elemi műveletre osztható:
F (fetch) - lehívás
D (decode) – dekódolás
S/O (source operand) – forrás operandus lehívás
E (execute) – végrehajtás
W/B (writeback) – visszaírás

Kibocsátási párhuzamosság:

A kibocsátás a CPU dekódoló egységéből történik, ezeket a dekódolt utasításokat hajtja végre a végrehajtó egység. Ha a CPU képes párhuzamosan több utasítás végrehajtására, a kibocsátási kapacitást is növelni kell.
Kibocsátási párhuzamosságnak nevezzük, ha dekódoló egység óraciklusonként egynél több utasítás kibocsátására képes (szuperskalár).

A párhuzamos feldolgozásnak minden ILP CPU-ra vonatkozó követelménye:

- Az utasítások párhuzamos végrehajtása során minden ILP CPU-nak figyelembe kell vennie az utasítások közötti függőségeket.
- Meg kell őrizni a soros végrehajtás konzisztenciáját (párhuzamos végrehajtás esetén is ugyanúgy kell viselkednie a programozó által írt soros végrehajtású programnak).

Függőségek:

Egy programban az egymást követő utasítások függhetnek egymástól!
A függőségek akadályozzák párhuzamos végrehajtás teljesítmény növelésének maximalizálását, valamint gátolhatják a párhuzamos adat vagy utasítás végrehajtást.

Például egy futószalag fokozatai számának növelésével a teljesítmény nem sokszorozható a végtelenségig két ok miatt:

- az egyes futószalag fokozatok a gyakorlatban különböző időt igényelnek, így komoly időveszteségek léphetnek fel
- a függőségek miatt

A gyakorlatban kimutatták, hogy a teljesítmény csak egy bizonyos határig növekszik, a fokozatok számának további növelése később teljesítmény csökkenést eredményez általános célú alkalmazások esetén.

Típusai:
- **Adatfüggőség**
- **Vezérlésfüggőség** (feltétel nélküli és feltételes vezérlő utasítások esetén ➡ elágazás!)
- **Erőforrásfüggőség**

Adatfüggőség:

Adatfüggőségről beszélünk, ha az egymást követő utasítások ugyanazt az adatot használják.

Csoportosítása:

Jellege szerint:

utasítás szekvenciában jelenlevő (lineáris utasításfeldolgozás):

- valós adatfüggőség (RAW – Read After Write)
  - műveleti adatfüggőség
  - behívási adatfüggőség
- ál adatfüggőség
  - WAR (Write After Read)
  - WAW (Write After Write)
- ciklusban jelenlevő

Operandus típus szerint:

- regiszter
- memória

Valós műveleti adatfüggőség (RAW):

Példán keresztül bemutatva (probléma felvetés):
- Feltételezzük, hogy a CPU 3 operandusos utasításokat használ, valamint egy 4 fokozatú futószalagot (F, D, E, W/B).
- Két számot össze akarunk szorozni, majd az eredményeket össze akarjuk adni.

I_1	MUL r_3, r_2, r_1	(r_1, r_2 tartalmát összeszorozzuk r_3-ba)
I_2	SHL r_3	(r_3 értéket eltolással megduplázzuk)

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

Probléma: t_3 időpontban szükségünk van r_3 értékére a dekódoláshoz, viszont az csak t_4 végén fog előállni.
Valós műveleti adatfüggőség, mert műveletet akarunk végezni egy regiszter tartalmával, ami nem áll rendelkezésre. Ilyen futószalagos párhuzamos végrehajtásnál fordulhat elő, ahol az utasítás lehívás (Fetch) és végrehajtás (Execute) átfedheti egymást. Ilyenkor a művelet nem folytatódhat, míg a függősége fel nem oldódik → elakadás, várakozás következik be (**csökken a hatékonyság**!).

Kezelés: a vezérlés NOP utasításokat (No Operand) használ

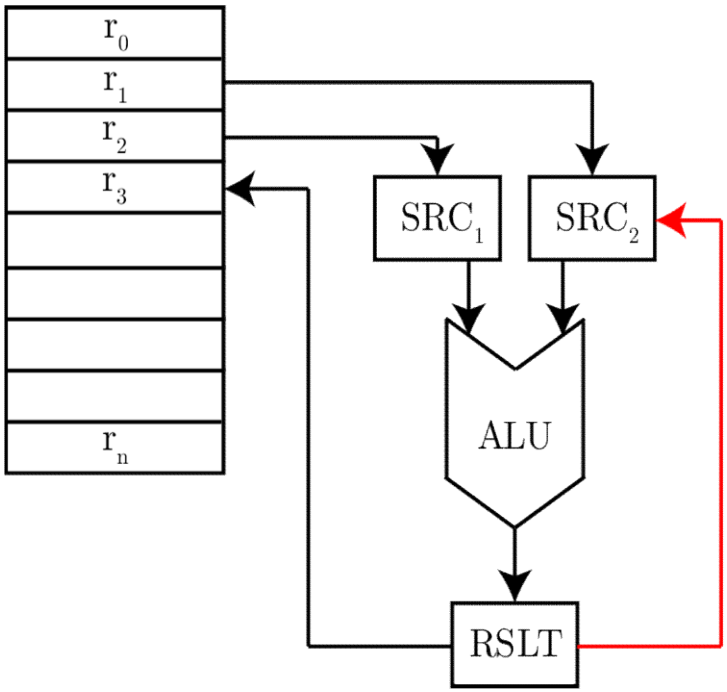
	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F_{MUL}	$D/SO_{r1, r2}$	E_{MUL}	W/B_{r3}			
I_2		F_{SHL}	NOP	NOP	D/SO_{r3}	E_{SHL}	W/B_{r3}

Következmény: két óraciklussal hosszabb ideig tart az utasítás végrehajtás.
Ezáltal minden utána következő utasítás is csúszni fog.

Lassító hatás csökkentése: operandus előrehozással, ami extra hardvert igényel (dinamikusan)

Műveleti valós adatfüggőség (folyt.):

Egy extra hardver segítségével az eredményt visszavezetjük valamelyik forrásregiszterbe. Így már t_3 végére elérhető az eredmény az ALU bemenetén és az esetek nagy részében kiküszöbölhető egy-két várakozó ciklus, mivel például a szorzást követő átvezetés nagyon gyorsan megtörténik és rendelkezésre fog állni az adat az eltolás végrehajtásához.



Ezt minden mai CPU használja.

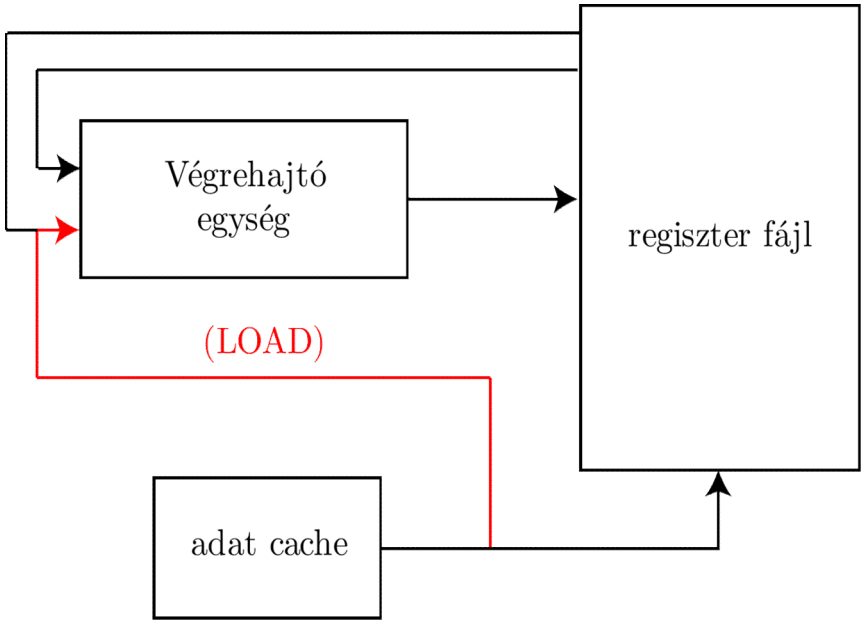
Természetesen az eredményt a regiszterbe is vissza kell írni!

Lehívási valós adatfüggőség (RAW):

Probléma: A regiszterekbe az operatív tárból vagy adatcache-ből töltjük be a szükséges operandusokat, majd az ALU a regiszterekből hívja le őket.

Amennyiben a kért adat éppen nem elérhető a regiszterben, akkor be kell tölteni, de a cache és főleg az operatív tár elérése sok időt vesz igénybe.

Kezelés: betöltési operandus előrehozása. Extra hardver segítségével nem csak a regiszterekbe, hanem az ALU bemenetére is betöltjük az operandust. Legalább 1 óraciklussal gyorsítjuk a végrehajtást.



Ál adatfüggőségek:

WAR (write after read):

Az ál adatfüggőség teljesen megszüntethető. Egyik típusa az olvasás utáni írás. Viszonylag ritka függőség, de valóban okozott problémát régebben.

Probléma:

I1	MUL r3 r2 r1	r3-ba szorozzunk össze két számot,
I2	ADD r2 r4 r5	a szorzó forrás regiszterébe töltjük egy gyorsabb művelet eredményét

Előfordulhat bizonyos architektúráknál, hogy az ADD utasítás eredménye hamarabb előáll, mint a megelőző utasítás operandusainak beolvasása. Mivel I2 módosította az I1 bemeneti operandusát, a MUL utasítás hibás eredményt fog adni ⟶ sérül a szekvenciális konzisztencia.

Megoldás: r2 tartalmát egy ideiglenes regiszterbe irányítjuk (r23)

I1	MUL r3 r2 r1
I2	ADD r23 r4 r5

Az r23 ⟶ r2 hozzárendelést nyilván kell tartani. Majd, amikor a MUL utasítás végzett, vissza kell írni a r23 tartalmát r2-be. Az ilyen ideiglenes regisztereket átnevezési regisztereknek hívjuk.

Átnevezési (vagy piszkozat) regiszterek tulajdonságai:

- új, önálló regiszterek
- saját címtartománnyal rendelkezik
- programozó számára transzparens
- extra hardvernek számít

Ezáltal két féle regiszterkészletet különböztetünk meg:

- architektúrális regiszterkészlet: ezeket látja (tudja használni) a programozó
  (pl.: AX, BX, CX, DX, SI, DI, …regiszterek)
- átnevezési regiszterkészlet: a vezérlés használja arra, hogy az ál adatfüggőségeket kiküszöbölje

<u>WAW (write after write):</u>

Ez a gyakoribb ál adatfüggőség.
Probléma:

I_1 MUL r_3 r_2 r_1 r_3-ba szorozzunk össze két számot
I_2 ADD r_3 r_4 r_5 majd ugyanabba a regiszterbe töltjük egy gyorsabb művelet eredményét

WAW (folyt.):

Az ADD utasítás sokkal gyorsabb, mint a MUL, ezért előfordulhat, hogy a párhuzamos végrehajtás során I_1 később fut le, int I_2. Mivel az eredményt ugyanabba a regiszterbe írják, ezért az I_1 felülírja I_2 eredményét → sérül a szekvenciális konzisztencia.

Megoldás: r_3 átirányítása egy átnevezési (piszkozat) regiszterbe.

A regiszter átnevezés egyébként történhet statikusan compiler segítségével, vagy dinamikusan átnevezési egységgel.

Ciklusbeli függőség:

Probléma: egy ciklus mindig az eggyel korábbi iteráció eredményét használja, azaz az aktuális eredményhez szükség van az előző iteráció eredményére.

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

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

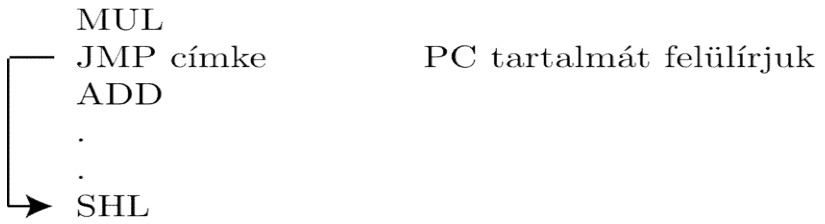
Vezérlés függőség:

Vezérlés függőségnek akkor léphet fel, ha feltétlen vagy feltételes elágazáshoz érünk.

Feltétel nélküli elágazás:

(ugrás)

Probléma:



	t_1	t_2	t_3	t_4	t_5	t_6
I_1	F_{MUL}	D	E	W/B		
I_2		F_{JMP}	D	E	W/B	
I_3			F_{ADD}	D	E	W/B

A JMP utasítás az Execute fázisban (t4) állítja át a PC-t, ezzel végzi el az ugrást. A futószalag végrehajtás miatt azonban ekkora már a következő utasítás, az ADD már lehívásra került, architektúrától függően akár már az azt követő utasítás is.

- Feleslegesen hívjuk le az ADD utasítást!
- előfordulhat, hogy a lehívott utasítások be is fejeződhetnek, mire a JMP végrehajtásra kerül, ami veszélyezteti az architektúrális regiszterek tartalmát (hibás működés!).

Kezelés (nem megoldás!):

- lehet **statikus, dinamikus,** vagy **spekulatív:** jelentése eltér az eddig bevezetett jelentéstől

a) Ugrási buborék (statikus kezelés):

A JMP utasítás mögé egy vagy több NOP utasítás kerül be, ezzel lassítva a futószalagot, míg elő nem áll az ugrási cím. Az ugrási buborék nagysága **n-1**, ahol n a futószalag fokozatok száma.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
I_1	F_{MUL}	D	E	W/B				
I_2		F_{JMP}	D	E	W/B			
I_3			NOP	NOP	NOP	F_{SHL}	D	E

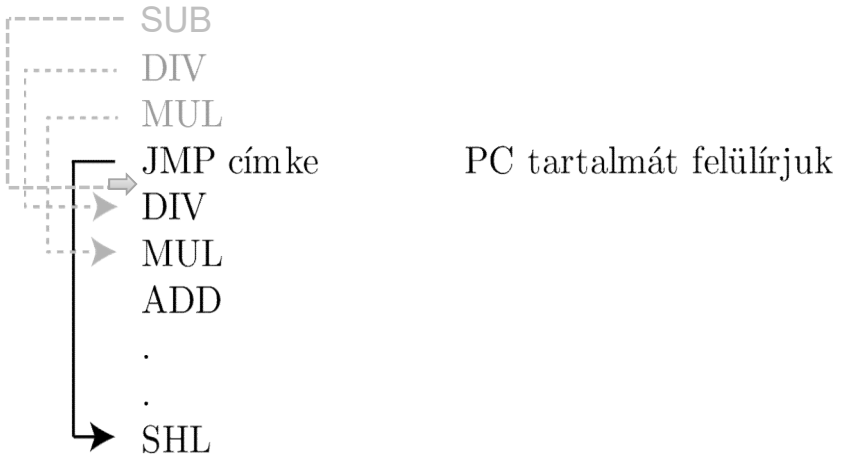
Ez egyszerű kezelési mód, compilerrel megoldható.
Értékelés: bár felesleges műveleteket végzünk, de nem veszélyeztetjük az architektúrális regiszter tartalmakat.

b) Utasítások átrendezése optimalizáló compiler segítségével (<u>dinamikus kezelés</u>):

ha a compiler JMP utasítást talál akkor megpróbálja átrendezni az utasítások sorrendjét úgy, a JMP előttről rak át adatmanipuláló utasításokat a JMP mögé (tulajdonképpen előbbre hozza a JMP-ot).

A sorrend megváltoztatásával elértük, hogy amíg az ugrás nem hajtódik végre (t4), addig csak olyan utasításokat hívnunk le, amiknek még az ugrás előtt kell végrehajtódniuk. Mire az ADD utasításhoz leérnénk, felülíródik a PC és a megfelelő utasítás hívódik le (SHL).

<u>Hátránya</u>: a hatása a futószalag fokozatok számának növekedésével rohamosan csökken



Például:
2 fokozat esetén a feltöltés valószínűsége ~85%
3 fokozat esetén már csak ~50%

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
I_1	F_{JMP}	D	E	W/B				
I_2		F_{SUB}	D	E	W/B			
I_3			F_{DIV}	D	E	W/B		
I_4				F_{MUL}	D	E	W/B	
I_5					F_{SHL}	D	E	W/B

Az utasítások átrendezését a korai RISC processzorokban alkalmazták.

Manapság 15-25 fokozatúak a futószalagok, itt már a compiler nem képes elég adatmanipuláló utasítást átcsoportosítani ➡ NOP utasítások is kellenek, ami késleltet ➡ lassul a végrehajtás.

A mai rendszerekben előlehívás történik, ennek során egy vezérlés elágazó utasításokat keres, hogy ezek címét előre kiszámíthassa (feltétel nélküli esetben).

Feltételes elágazás vezérlés függősége:

Kezelése ma már csak dinamikusan, a végrehajtás során történhet, hiszen a feltétel teljesülésétől vagy NEM teljesülésétől függ, hogy ugrás vagy soros folytatás következik-e.

Ennek a fékező hatását mérséklő elágazás becslést és a később megjelenő spekulatív elágazás-kezelést (branch prediction) később tárgyaljuk.

Erőforrás függőségek:

Akkor lép fel, ha több utasítás akarja ugyanazt az erőforrást használni. Ilyenkor az egyiket várakoztatni kell. Erőforrások lehetnek például:

- Architektúrális regiszterek
- Puffer regiszterek
- Dedikált végrehajtó egységek, …

ÓBUDAI EGYETEM

Tervezési szempont:

Ne okozzon az erőforrás függőség szűk keresztmetszetet!

Példa:

A logikai futószalagok különböző célokra dedikált végrehajtó egységekben vannak megvalósítva. Ilyen például 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.

Kezelés:

Úgy kell tervezni a processzort, hogy az erőforrás függőség nem okozzon szűk keresztmetszetet. Ez például az erőforrások többszörözésével érhető el.

Fontos szempont a hatékonyság, 70-80%-os kihasználtság az általános.

A Szekvenciális (soros) konzisztencia megőrzése:

A programozó a programot szekvenciális logika szerint készíti el és a hardver vagy a compiler feladata, hogy az utasítások egymásutániságában a lehető legtöbb rendelkezésre álló párhuzamosságot ki tudja használni, miközben a logikai integritás megmarad.

Típusai:

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

Tehát két vonatkozásban is felvetődik a soros konzisztencia kezelésének problémája:

- Normál utasítás feldolgozás során
- Kivétel kezelésnél

Az utasítás feldolgozás soros konzisztenciája (processzor konzisztencia):

Probléma: párhuzamos végrehajtás (több végrehajtó egység) 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 ⟶	gyorsabb
I3	JZ címke	ha az eredmény 0, akkor ugrás

Mivel a JZ utasítás mindig a legutoljára végzett utasítás eredményét használja fel a feltételes ugrás eldöntéséhez, ha I1 később végez, mint I2, előfordulhat, hogy a JZ a **DIV** eredménye alapján fog ugrani, ami hibás működéshez vezethet.

Tehát a párhuzamos végrehajtás bevezetése megsértheti a soros végrehajtás logikáját!

Megoldás: a hardvert úgy kell tervezni, hogy ilyen hiba ne fordulhasson elő. A feltételes utasítás csak az előtte lévő utasítás eredményét vegye figyelembe, ne azt, ami utoljára befejeződik!
Például plusz flag-ek bevezetése.

A kivétel-kezelés soros konzisztenciája:

Párhuzamos végrehajtás esetén az utasítások végrehajtás során keletkező kivételek is sorrenden kívüli megszakításkérést okozhatnak!

Amennyiben ezeket a kéréseket a CPU azonnal fogadja, akkor a megszakításokat a soros CPU-tól eltérő sorrendben fogja kezelni.

Ebben az esetben a CPU pontatlan megszakításkezelést végez!

Probléma:

tegyük fel, hogy az alábbi kódrészletben az ADD túlcsordul, de a MUL még nem fejeződött be.

I1	MUL r3 r2 r1	
I2	ADD r5 r6 r7	túlcsordul és kér egy megszakítást
I3	JZ címke	ha az eredmény 0, akkor ugrás

Az ADD utasítás túlcsordulása miatti megszakításkérést kezelni kell. Ilyenkor a processzor a regiszterek állapotát (kontextust) elmenti egy verem regiszterbe. Miután a kivétel lekezelődött, 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 r3 regiszter definiálatlan állapotba kerül ⟶ hibákhoz vezethet!

Pontatlan kivétel-kezelés volt a korai szuperskalároknál.

Pontos kivétel-kezelés:

Másnéven precíz megszakításkezelés. Minden mai CPU a megszakítás kéréseket kizárólag az utasítások eredeti sorrendjében fogadja el!

Az előző példában a CPU csak akkor fogadja el az ADD megszakítás kérését, ha az előtte lévő utasítások befejeződtek és nem küldtek megszakításkérést.

Megvalósítás:

- átrendező puffer segítségével: a CPU a megszakítást csak akkor fogadja, ha az adott utasítást kiírjuk ebből az átrendező pufferből. (például: Intelnél ROB (Reorder Buffer))
- címkézés: az utasításokat sorszámokkal látjuk el és akkor fogadjuk el a megszakítást, ha egyetlen megelőző sorszámú sem kért.