Skalární procesory, řetězené zpracování

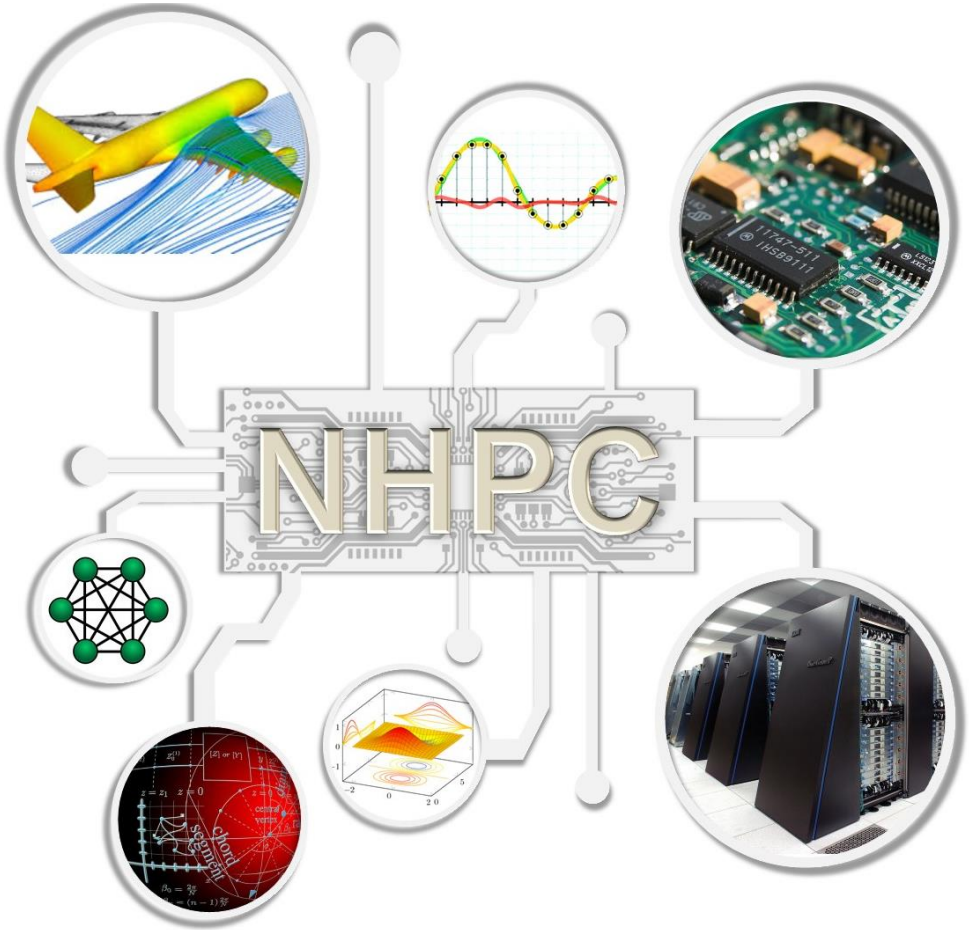
AVS – Architektury výpočetních systémů

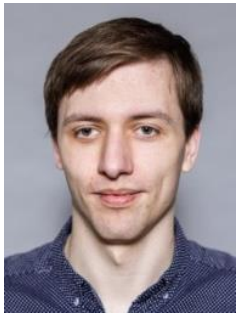
Týden 1, 2024/2025

Jirka Jaroš

Vysoké učení technické v Brně, Fakulta informačních technologií
Božetěchova 1/2, 612 66 Brno - Královo Pole
jarosjir@fit.vutbr.cz

Výzkumná skupina Superpočítačových technologií





Mimo FIT ☺

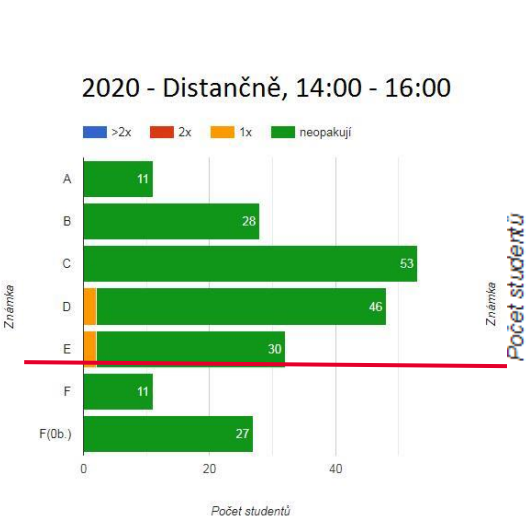
https://www.facebook.com/skupina.stone

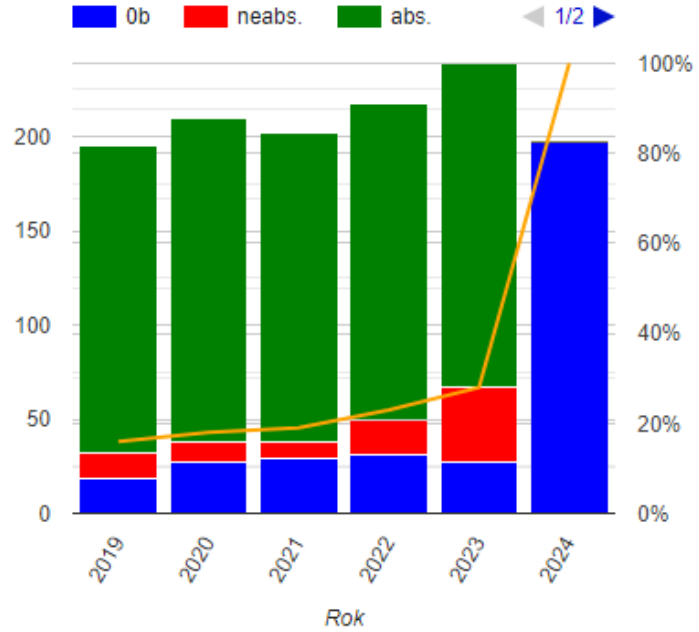
Organizace předmětu

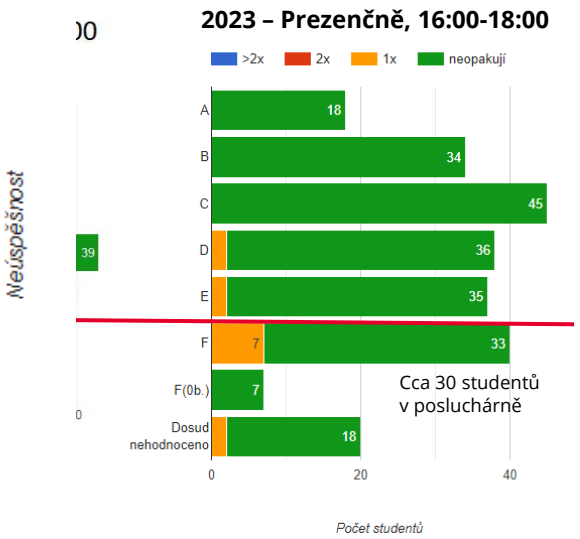
- **Přednášky**
  - Pátek 8:00 – 9:50, místnost E112, E104, E105.
- **Počítačové laboratoře**
  - Celkem 6 laboratoří v 4., 5., 6., 9., 10. a 11. týdnu, učebna N204, N105.
  - Laboratoře probíhají na superpočítači Barbora.
- **Projekty (10 + 20 bodů)**
  - Měření výkonnosti a vektorizace kódu **(08.11.)**
  - Paralelizace kódu na systémech se sdílenou pamětí **(06.12.)**
- **Půlsemestrálka (10 bodů)**
  - 1. listopadu v 9:00 (7. týden, druhá polovina přednášky)
- **Bonusové body (5 bodů)**
  - Nadprůměrná aktivita na přednáškách, cvičeních, či projektech
- **Zápočet (min 20 bodů ze semestru + 1 bod z každého projektu)**
- **Zkouška 60 bodů (minimum 20 pro složení)**
- **Studijní materiály v Moodle + streaming a záznamy + Intel webináře + knihy + …**

Proč chodit na přednášky a cvičení

- **Nejrychlejší a nejefektivnější způsob, jak se něco naučit.**
- **Nebojte se zeptat (nás výuka baví).**
- **Pokud máte pocit, že je předmět k ničemu nebo vás nebaví, dejte nám zpětnou vazbu.**

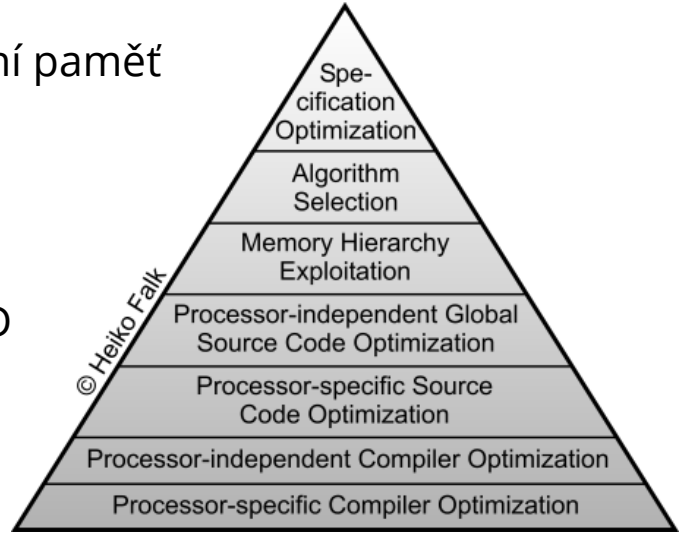






Obsah kurzu – Co se naučím

- **Jádro procesoru**
  - Skalární, superskalární, SIMD
  - Predikce skoků, přednačítání dat a instrukcí, virtuální paměť
- **Paměti cache**
  - Hierarchie, adresování, koherence
- **Vícejádrové procesory (NUMA i UMA)**
  - Hyperthreading, propojovací sítě, ukázky Intel i AMD
- **Vícesoketové systémy (NUMA i UMA)**
  - Rozsáhlé serverové systémy
- **Nízkopříkonové systémy (VLIW, RISC-V, ARM)**
  - Techniky snižování příkonu, architektury s nízkou spotřebou
- **Grafické karty**
  - Architektura a základní principy programování



Potřeba zvyšovat výpočetní výkon je trvalá…

- **Jedna možnost**: zvyšovat počet tranzistorů.
  - Moorův zákon platí již 50 let (1971). Říká, že počet tranzistorů na čipu se zdvojnásobuje každé 2 roky při zachování stejné ceny.
    Otázka je, jak tyto tranzistory využít…
  - Se snižováním rozměrů (dnes běžně 10 – 7 nm a připravuje se 5 nm) se rychlost tranzistorů zvyšuje, příkon snižuje.
- **Větší počet tranzistorů dovoluje paralelismus**:
  - paralelní provádění instrukcí (ILP, Instruction Level Parallelism),
  - střídání vláken na CPU (TLP, Thread Level Parallelism),
  - zpracování dat paralelně (DLP, Data Level Parallelism),
  - rozdělení úloh na vlákna/procesy na více jader.

What's the Opportunity?

Matrix Multiply: relative speedup to a Python version (18 core Intel)

Version	Speed-up	Optimization
Python	1	
C	47	Translate to static, compiled language
C with parallel loops	366	Extract parallelism
C with loops & memory optimization	6,727	Organize parallelism and memory access
Intel AVX instructions	62,806	Use domain-specific HW

from: "There's Plenty of Room at the Top," Leiserson, et. al., *to appear.*

32

Amdahl's law – Paralelizace není zadarmo!!!

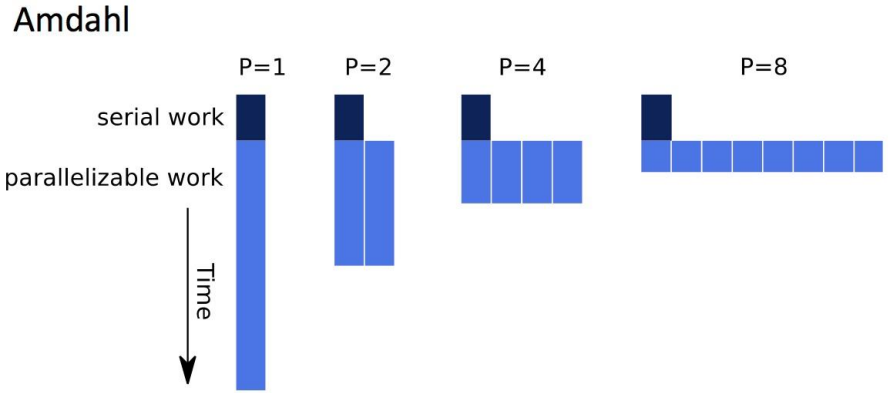
Rozdělení zátěže:

- 1 CPU αW ……. z podstaty sekvenční část,
- P CPU $(1-\alpha)W$ …… paralelizovatelná **část práce**

$$S(P) = \frac{T_S}{T(P)} = \frac{W/R}{W\left(\frac{\alpha}{R} + \frac{1-\alpha}{P \cdot R}\right)} = \frac{P}{1+\alpha(P-1)}$$

Zrychlení $\lim_{P\to\infty} S(P) = 1/\alpha$

Efektivita $\lim_{P\to\infty} E = 0$



Gene M. Amdahl: Validity of the single processor approach to achieving large scale computing capabilities, 1967.
http://dl.acm.org/citation.cfm?doid=1465482.1465560

Gustafson's law: S počtem CPU roste zátěž

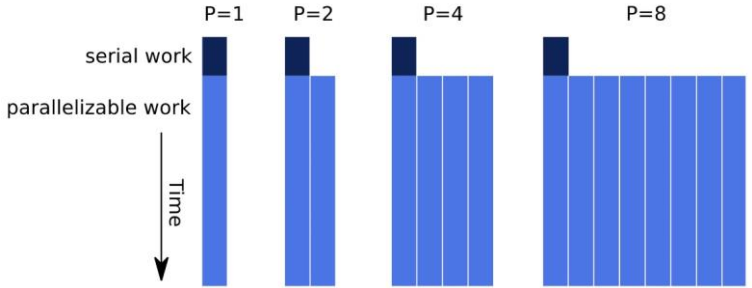
- S rostoucí velikostí úlohy (a tedy i práce W) často α relativně klesá a dostáváme dobré zrychlení.
- Při zachování stejného T_P pak roste T_S.
- Jestliže paralelní počítač pracoval na úloze **dobu T_P**, z toho dobu $\alpha_G T_P$ jen jedno jádro a po dobu $(1-\alpha_G)\,T_P$ všech P jader, je dosažené zrychlení S a účinnost E:

$$S = \frac{T_S}{T_P} = \frac{\alpha_G T_P + P(1-\alpha_G)T_P}{T_P} = P - \alpha_G\,(P-1)$$

$$E = 1 - \alpha_G + \frac{\alpha_G}{P} \qquad \lim_{P\to\infty} E = 1 - \alpha_G$$

!

Gustafson

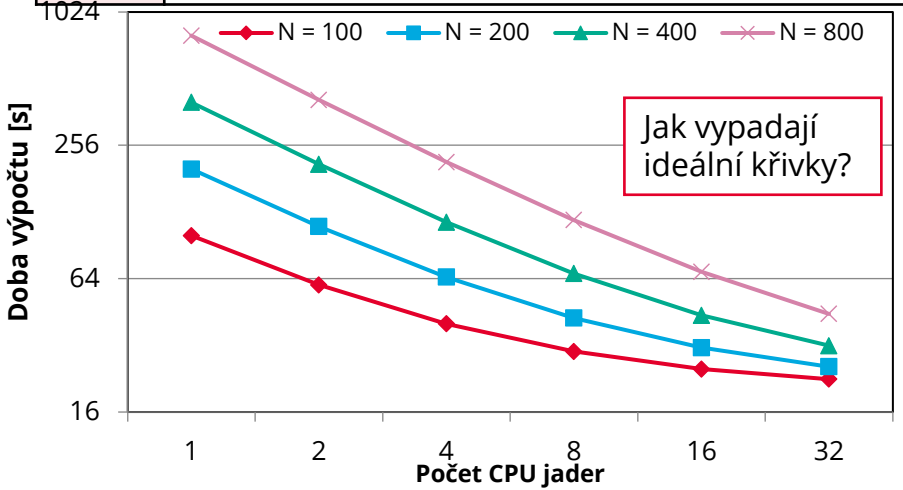


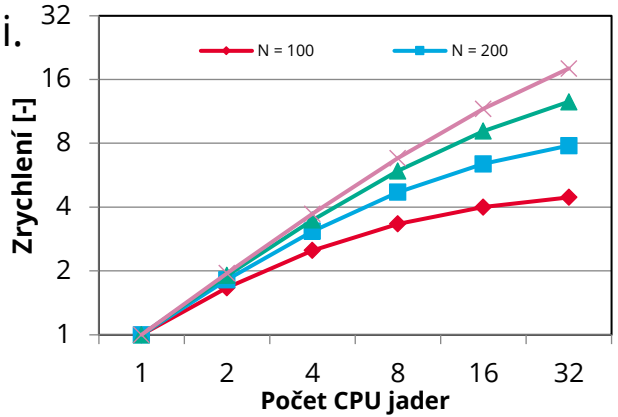
John L. Gustafson: Reevaluating Amdah's law (1998) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.6348

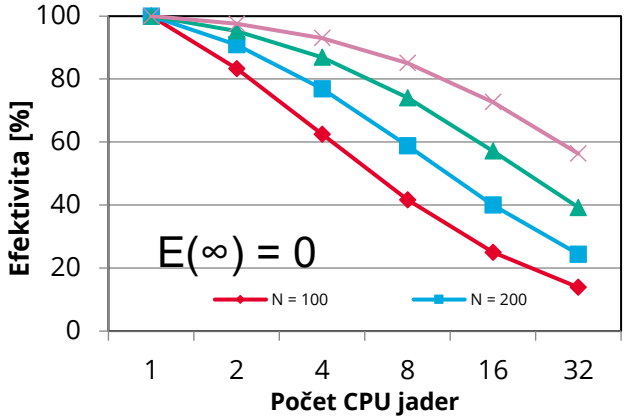
Silné škálování – Strong scaling

- **Silné škálování** – Konstantní celková práce (Amdahl)
  - Snažíme se vykonat úlohu dané velikost *N* co nejrychleji.

Doba výpočtu úlohy o velikosti *N* na *P* jádrech						
Úloha / P	1	2	4	8	16	32
N = 100	100	60	40	30	25	22.5
N = 200	200	110	65	42.5	31.25	25.625
N = 400	400	210	115	67.5	43.75	31.875
N = 800	800	410	215	117.5	68.75	44.375



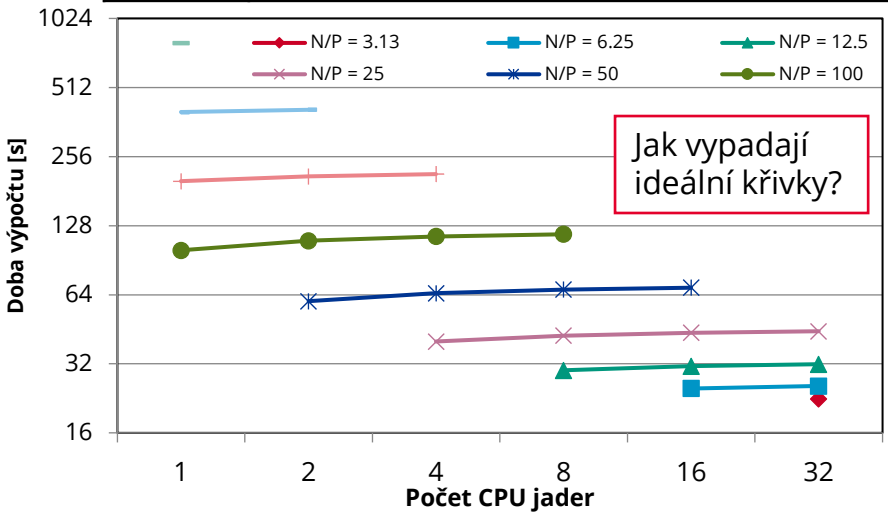




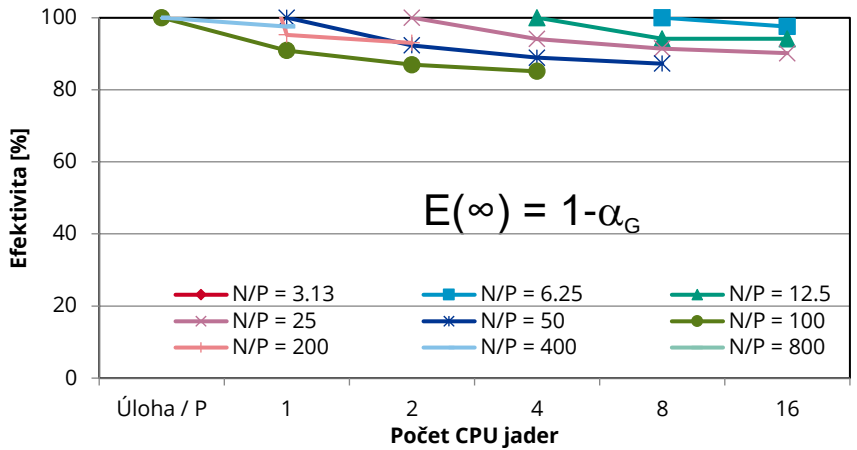
Slabé škálování – Weak scaling

- **Slabé škálování** – Konstantní čas výpočtu na jádr
  - Chceme řešit vetší problémy na větším stroji za stejnou dobu (Gustafson).

Doba výpočtu úlohy o velikosti *N* na *P* jádrech						
Úloha / P	1	2	4	8	16	32
N = 100	100	60	40	30	25	22.5
N = 200	200	110	65	42.5	31.25	25.62
N = 400	400	210	115	67.5	43.75	31.87
N = 800	800	410	215	117.5	68.75	44.37



Doba výpočtu úlohy velikosti *N* na *P* jádrech						
Úloha / P	1	2	4	8	16	32
N/P = 3.13						22.5
N/P = 6.25					25	25.62
N/P = 12.5				30	31.25	31.87
N/P = 25			40	42.5	43.75	44.37
N/P = 50		60	65	67.5	68.75	
N/P = 100	100	110	115	117.5		
N/P = 200	200	210	215			
N/P = 400	400	410				
N/P = 800	800					



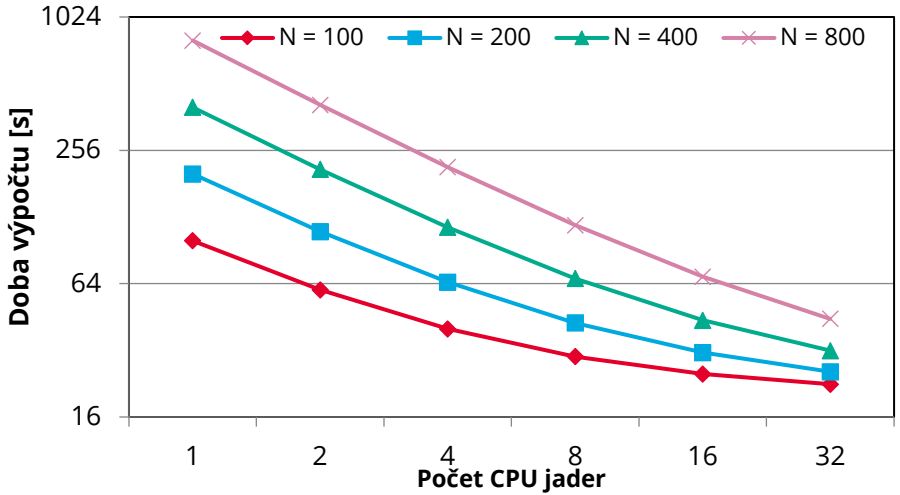
Cena výpočtu

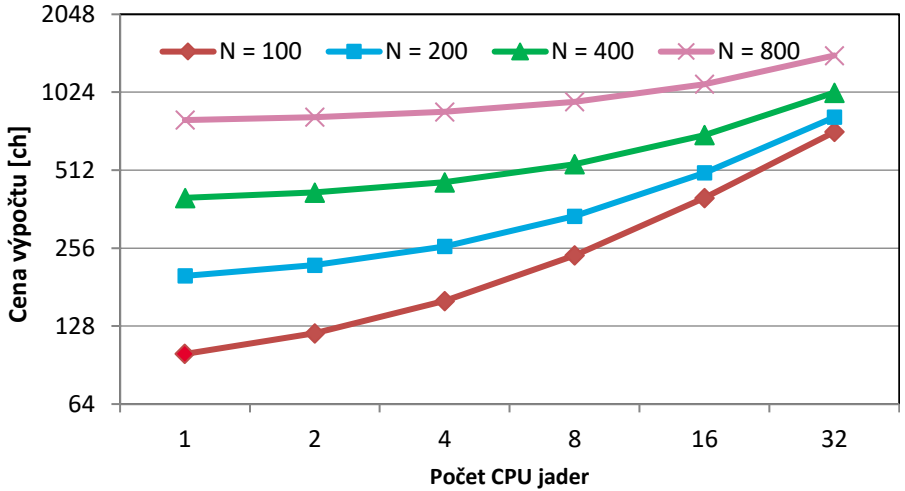
- **Běžně účtovaná v jádrohodinách, nově ale i kWh**

$$C = P * T_P$$

Doba výpočtu úlohy o velikosti *N* na *P* jádrech						
Úloha / P	1	2	4	8	16	32
N = 100	100	60	40	30	25	22.5
N = 200	200	110	65	42.5	31.25	25.62
N = 400	400	210	115	67.5	43.75	31.87
N = 800	800	410	215	117.5	68.75	44.37

Cena výpočtu úlohy o velikosti *N* na *P* jádrech						
Úloha / P	1	2	4	8	16	32
N = 100	100	120	160	240	400	720
N = 200	200	220	260	340	500	820
N = 400	400	420	460	540	700	1020
N = 800	800	820	860	940	1100	1419





Vývoj mikroprocesorů

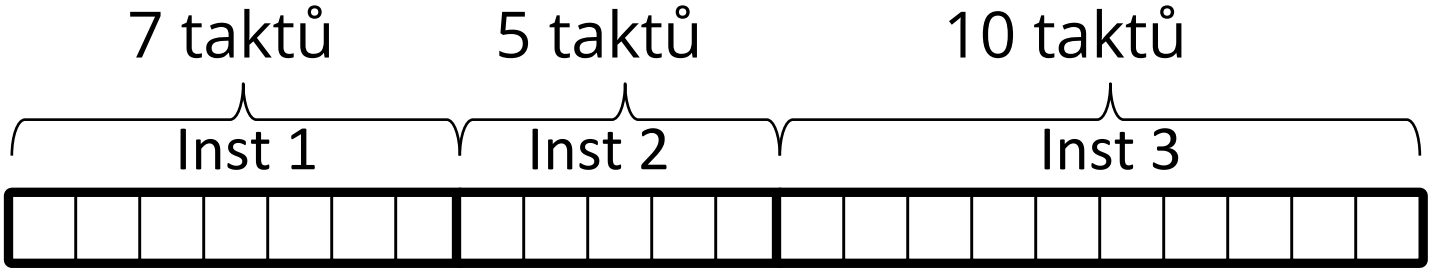
	Architektura	Vydávání instrukcí	Provádění instrukcí
ILP	Subskalární von Neumann	Sekvenční 0 až 1	Sekvenční Několik taktů
ILP	Skalární (řetězené)	Sekvenční 0 až 1	Paralelní CPI > 1
ILP	Superskalární a VLIW	Paralelní 0 až *m*	Paralelní IPC < *m*
TLP	Vícejádrové s časovým MT	1 jádro	Z více vláken
TLP	Vícejádrové s časovým a prostorovým MT	Více jader	Z více vláken na každém jádru

CPI – Clocks per instruction

IPC – Instructions per clock

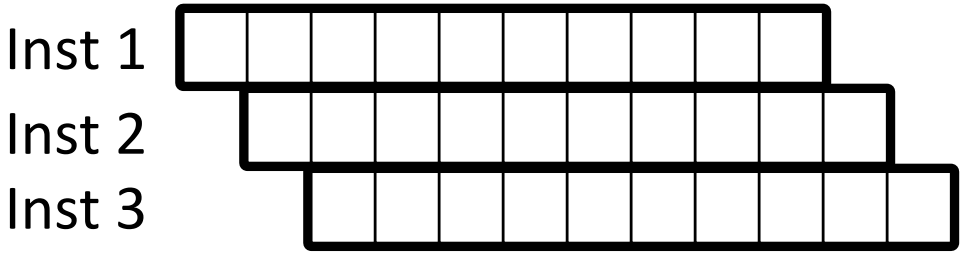
Sub-skalární oproti skalárním CPU

- **Neřetězená (sub-skalární, μ-programovaná) CPU**



3 instrukce, 22 taktů, **CPI = 7,33**

- **Řetězená (skalární) CPU**

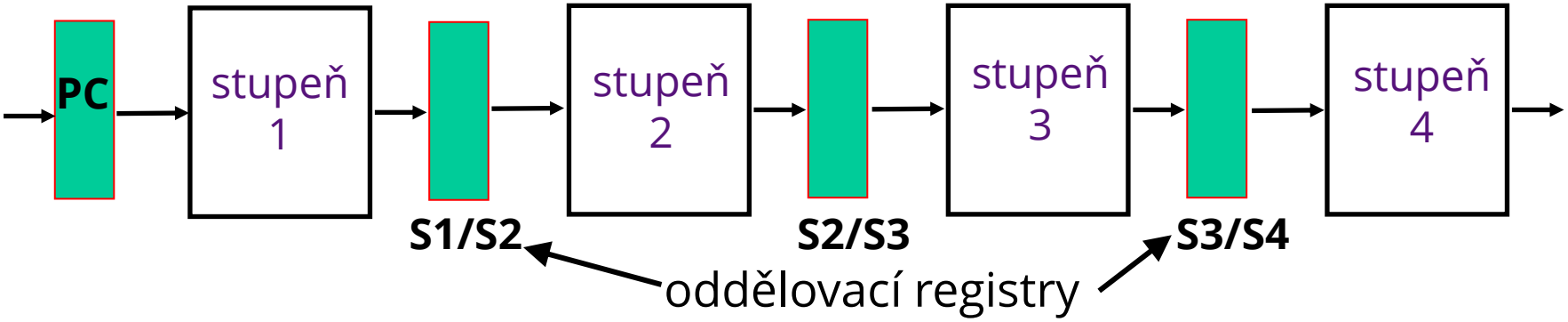


CPI = (10 + 2) / 3 = 4

ale v limitě $N \rightarrow \infty$

je ideálně CPI $\rightarrow 1$

Ideální řetězená linka

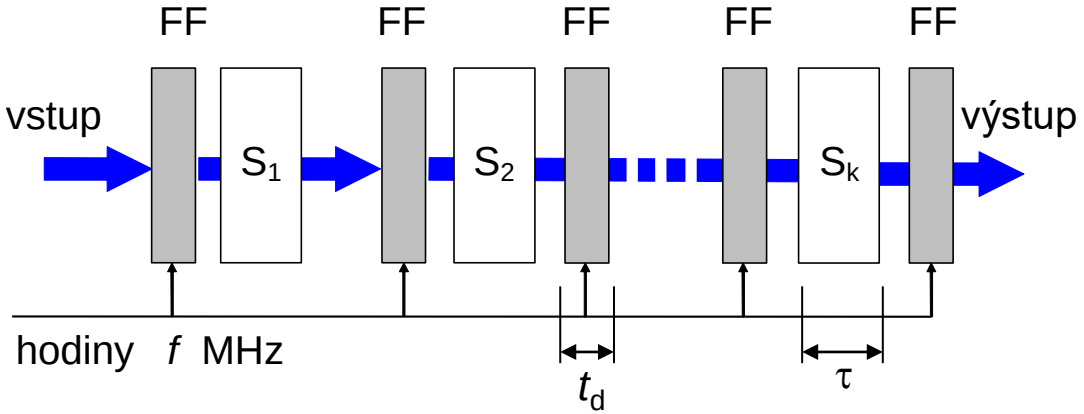


- Všechny objekty procházejí stejnými stupni (uniformní délka instrukcí).
- Žádná dvojice stupňů nesdílí HW prostředky, doba průchodu stupni je stejná.
- Načasování objektu vstupujícího do linky není ovlivněno objekty v jiných stupních.
- ***Tyto podmínky obecně platí pro průmyslové linky nebo VŠ výuku, ale instrukce mohou záviset jedna na druhé!***
- Závislosti řeší **SW** (kompilátor vkládáním NOP) nebo **HW** (zpoždění závislé instrukce – vkládání NOP a pozastavování stupňů linky).

Předpoklady a režie zřetězeného zpracování

- **Předpoklady:**
  - **Nepřetržitý přísun dat** nebo kódů, které je třeba zpracovávat podobným způsobem.
  - Zpracování musí být možno rozdělit na sekvenci **nezávislých** kroků, realizovaných jednotlivými stupni řetězu.
  - Trvání jednotlivých kroků by mělo být přibližně **stejné**.

- **Co má vliv na celkové dosažitelné zrychlení:**
  - Nezbytné přestávky při zpracování vlivem **závislostí**.
  - **Náběh a doběh** řetězeného zpracování při konečném počtu *N* zpracovaných položek.
  - Zpoždění **oddělovacích registrů**.



Průměrná doba trvání instrukce u neřetězené linky: t_1

Doba taktu řetězené linky: $\tau + t_d$, kde $\tau = t_1 / k$

$$S_N = \frac{N\,t_1}{T_k} = \frac{Nk\tau}{(k+N-1)(\tau+t_d)} \qquad S_\infty = \frac{k\tau}{(\tau+t_d)}$$

1. **Neřetězená CPU:** průměrné trvání jedné instrukce

t_1 = 20 ns, výkonnost $R = 1 / t_1$ = 50 MIPS

2. **Řetězená CPU**: 200 MHz, $\tau + t_d = 4 + 1 = 5$ ns,
ideálně žádné prostoje (CPI = 1), k = **5** stupňů

Výkonnost = počet instrukcí / čas = 100 / [(5 + 99) * 5 ns] = 192,3 MIPS

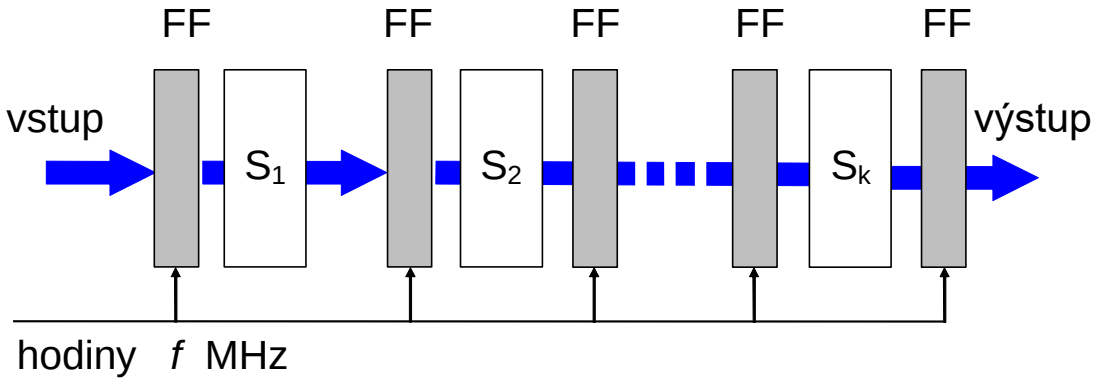
$$S_N = \frac{N\,t_1}{T_k} = \frac{Nk\tau}{(k+N-1)(\tau+t_d)} = \frac{100*20ns}{(5+99)(4+1)} = 3{,}84 = \frac{192{,}3\,MIPS}{50\,MIPS}$$

Pro počet instrukcí $N \to \infty$: $S_\infty = \frac{k\tau}{\tau+t_d} = \frac{20}{4+1} = 4$

Zrychlení při pozastavování linky

- Různé kolize způsobují pozastavování linky
  - Dobu zastavování linky můžeme zprůměrovat na pokutu ***q*** taktů vztaženou na jednu (každou) instrukci.
  - Počet taktů na 1 instrukci je pak CPI = 1 + *q*.

$$\lim_{N\to\infty} S = \frac{k\tau}{(\tau + t_d)(1+q)} \xrightarrow[\tau >> t_d]{} \frac{k}{1+q} = \frac{k}{CPI}$$



Příklad 2: Zrychlení s pokutami

- Četnost instrukcí load je **25** %
  - předpokládejme 100 % zásahů v D-cache,
  - instrukce po load vždy čekají (pokuta 1 takt).
- Četnost skoků je 20 %
  - 2/3 z nich se provede (pokuta 3 takty),
  - zbytek instrukcí je bez pokut.
- Počet stupňů linky je $k = 5$.
- Najděte výsledné CPI a zrychlení proti subskalární CPU pro $N \to \infty$ a $t_d << \tau$.

Řešení: CPI = 1 + pokuty vztažené na 1 instrukci

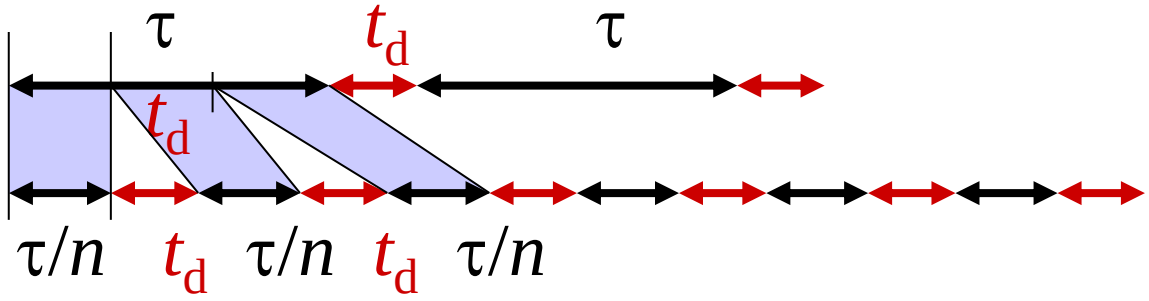
$$CPI = 1 + q = 1 + (0{,}25 * 1 + 0{,}2 * 2/3 * 3) = 1{,}65$$

$$S = k / (1 + q) = k / \mathrm{CPI} = 5 / 1{,}65 = \mathbf{3{,}03}$$

Superřetězená linka s $k \times n$ stupni

- **Stupně linky rozsekneme na menší části**
- **Tyto části vykonávají dílčí úkoly v daném stupni**

$$\underset{N \to \infty}{S} = \frac{\textit{doba taktu řetězení}}{\textit{doba taktu super-řetězení}} = \frac{\tau + t_d}{\tau / n + t_d} > 1$$



Např.

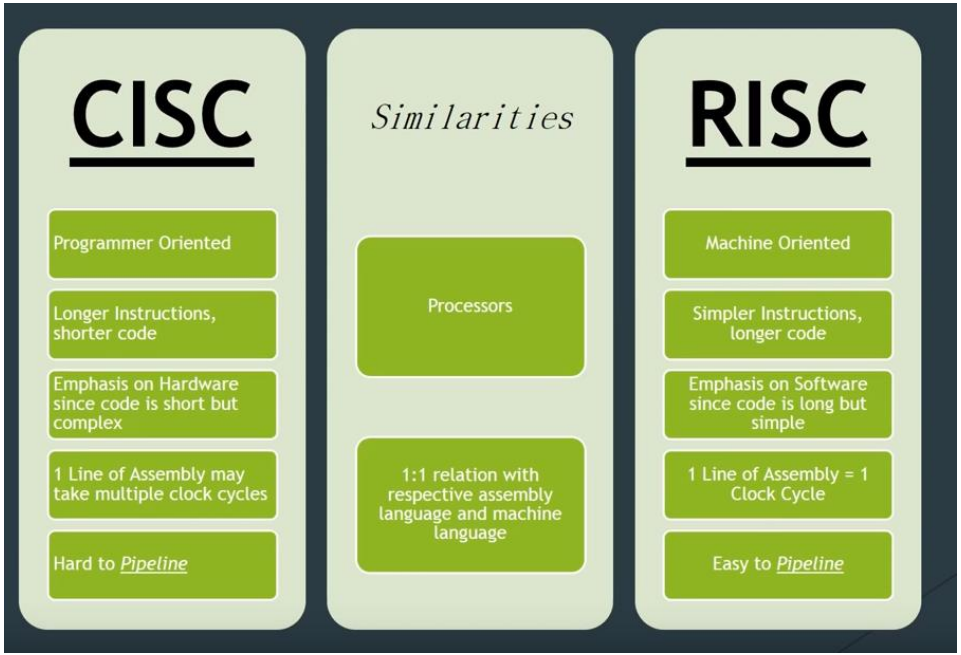
τ = 2 t_d , n = 3: S = 3 / (2 / 3 + 1) = 1,8

ŘETĚZENÝ PROCESOR RISC

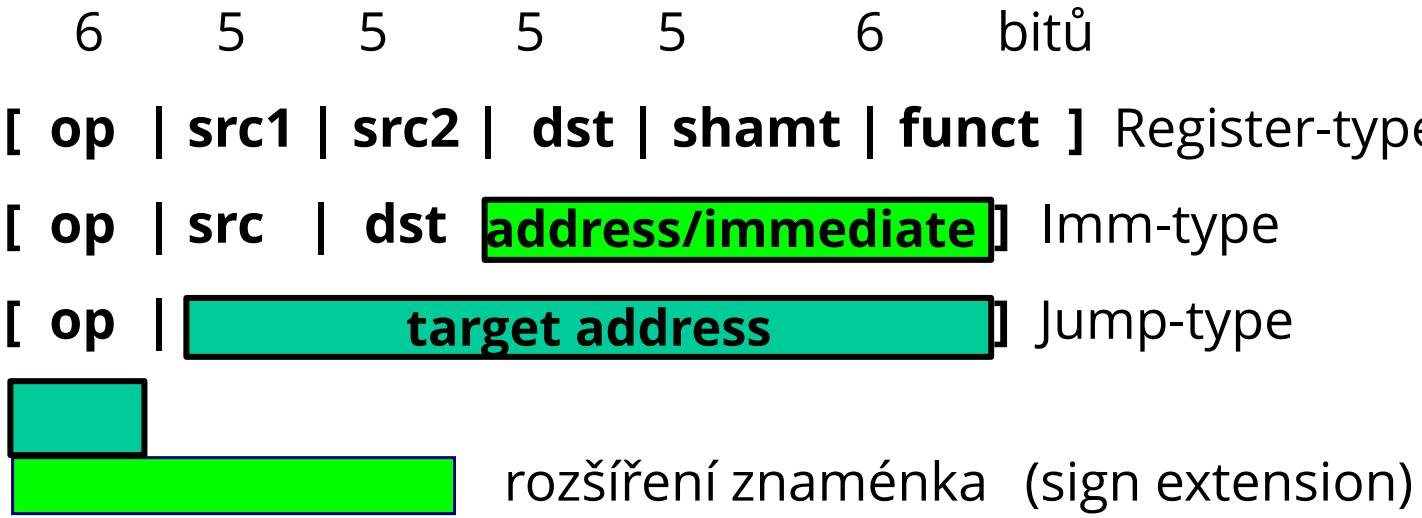
ARCHITEKTURY DLX

Charakteristika RISC instrukcí

- **Všechny instrukce mají 32 (64) bitů**
  - Instrukce x86 mají od 1 do 17 bajtů
- **Jen málo formátů instrukcí a formáty pravidelné**
  - Snadněji se načítají a dekódují v 1 taktu
- **Adresování paměti jen pomocí Load / Store**
  - adresu (offset + obsah registru) lze spočítat ve 3. stupni (EX),
  - přístup do paměti ve 4. stupni (MA)
- **Zarovnání paměťových operandů v bloku cache**
  - Fáze MA trvá jen jeden takt

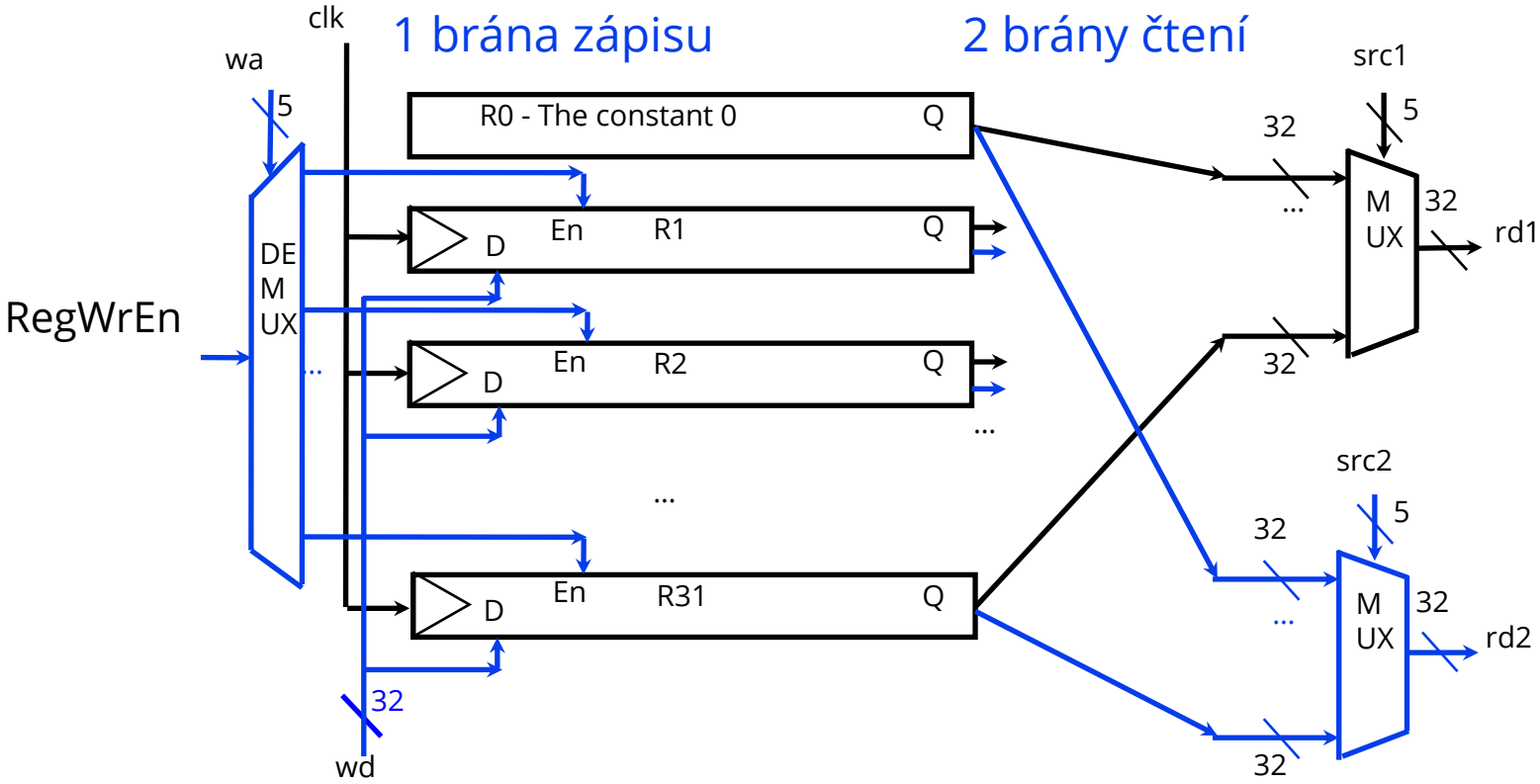


Formáty instrukcí RISC (MIPS/DLX 32 bit)



- funct = funkce ALU
- shamt = shift amount
- immediate = přímý operand | offset | rel. adresa skoku

Soubor registrů RF procesoru DLX



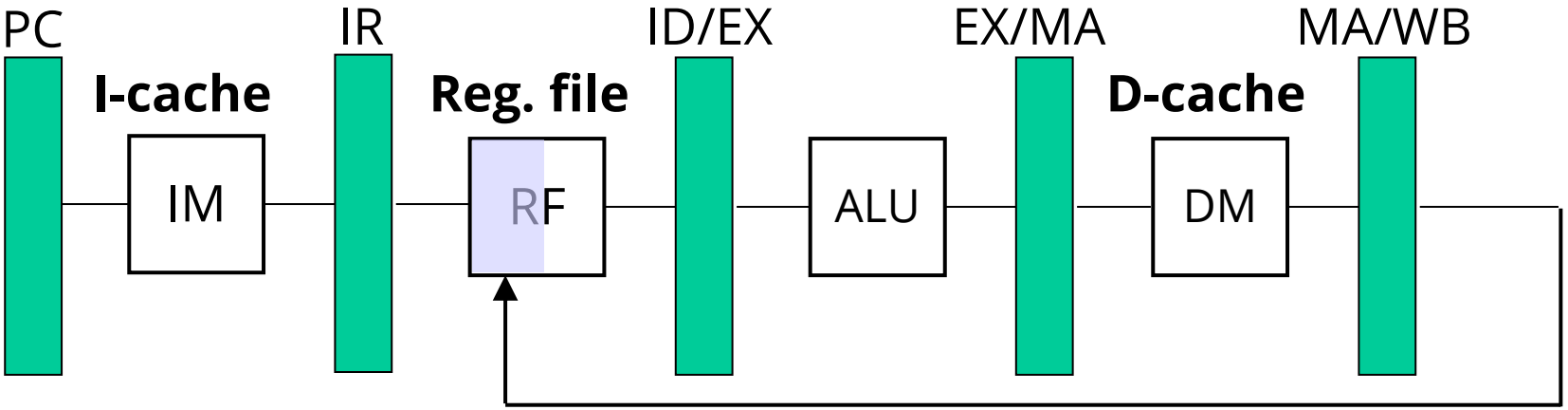
Příklad (extrém): IRF 128 reg., 12 bran čtení a 10 zápisových!

Stupně zřetězeného zpracování instrukcí RISC

IF = Instruction Fetch	načtení instrukce
ID = Instruction Decode	dekódování instrukce načtení registrových operandů
EX = Execute	provedení instrukce (výpočet adresy)
MA, CA = Memory Access	přístup do paměti cache
WB = Write Back	zápis výsledku do cílového registru

AG = Address Generation	generování adresy
E/C = Execute / Cache access	

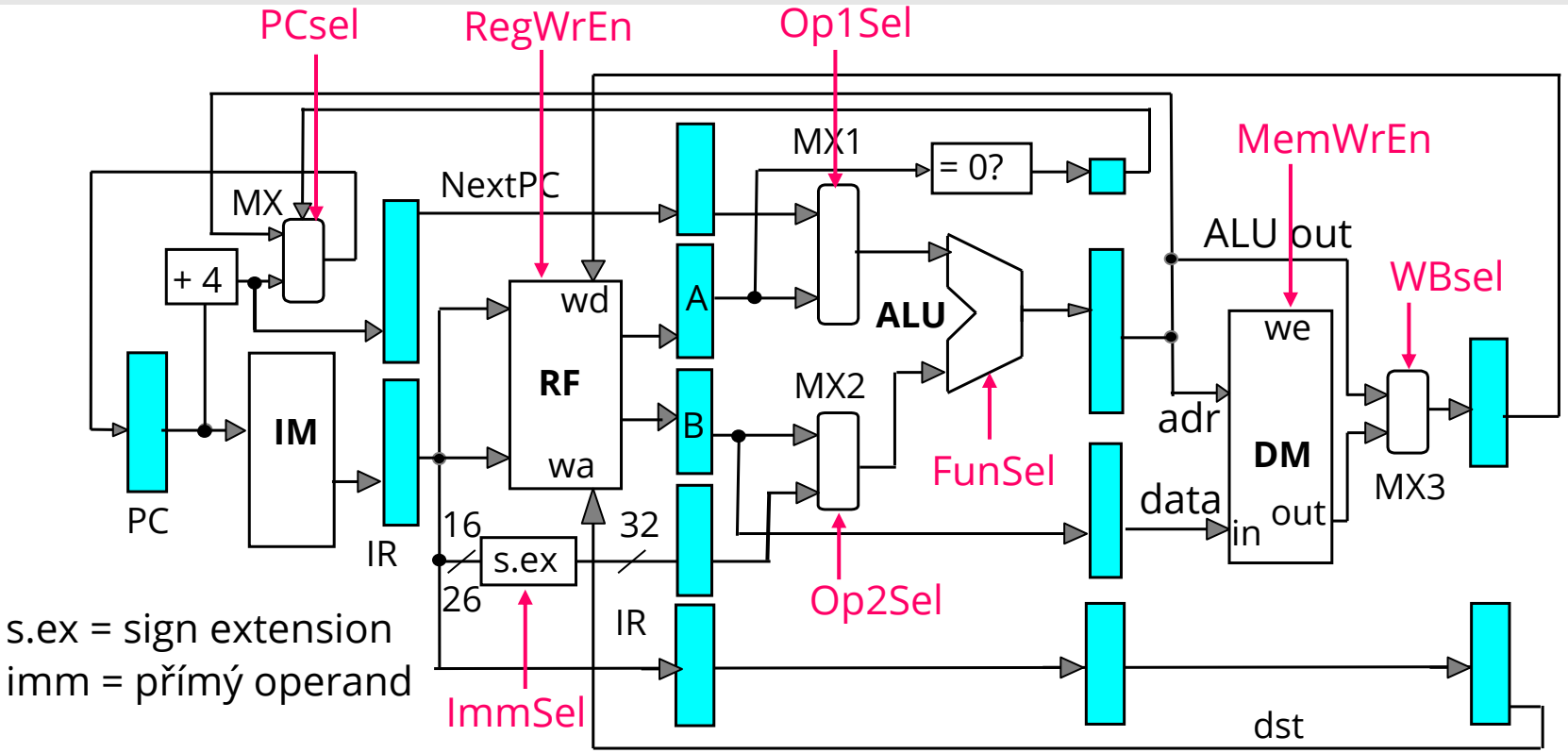
5 stupňová RISC linka procesoru DLX



IFetch - IDecode - EXecute - Memory - Write
načtení instrukce - načtení operandů - výpočet (adresy) - (Cache) Access - Back do reg.

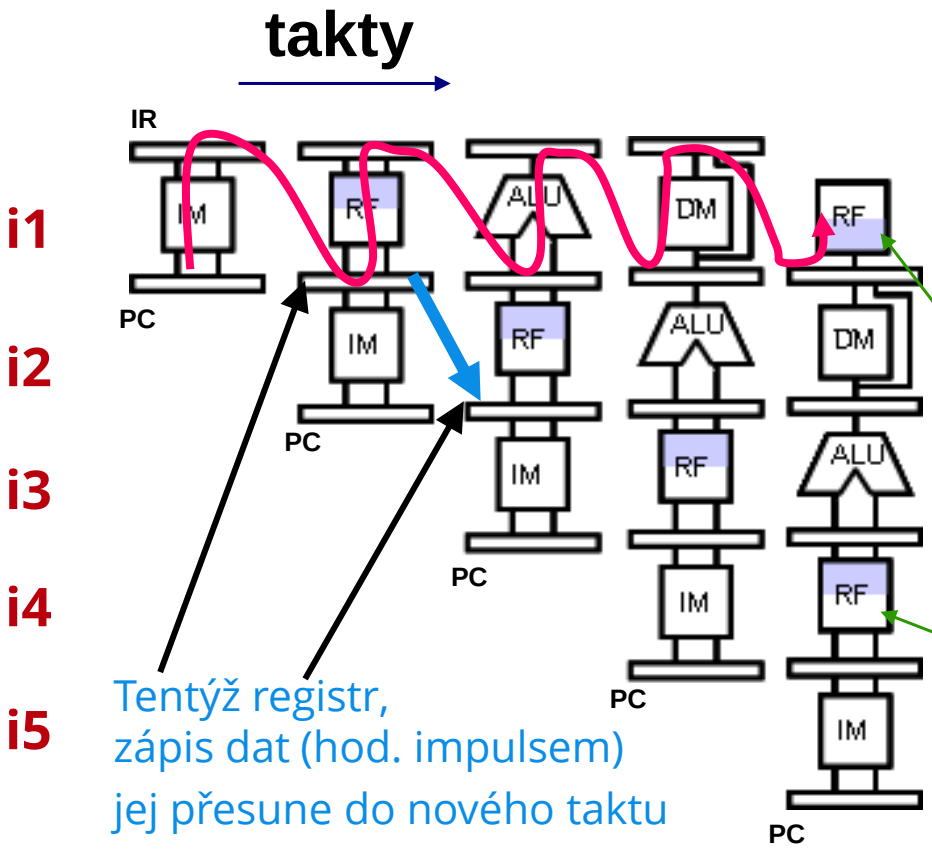
Ve druhé části taktu lze z RF číst data zapsaná v první části taktu (nebo dříve).

Klasická RISC Pipeline (MIPS/DLX)



- Linkou putuje **op-kód**, **src1**, **src2** a **dst** dokud je třeba.
- Op-kód je dekódován ve stupni ID, **řídicí signály** putují linkou přes oddělovací registry do míst určení.
- https://comparch.edu.cvut.cz/

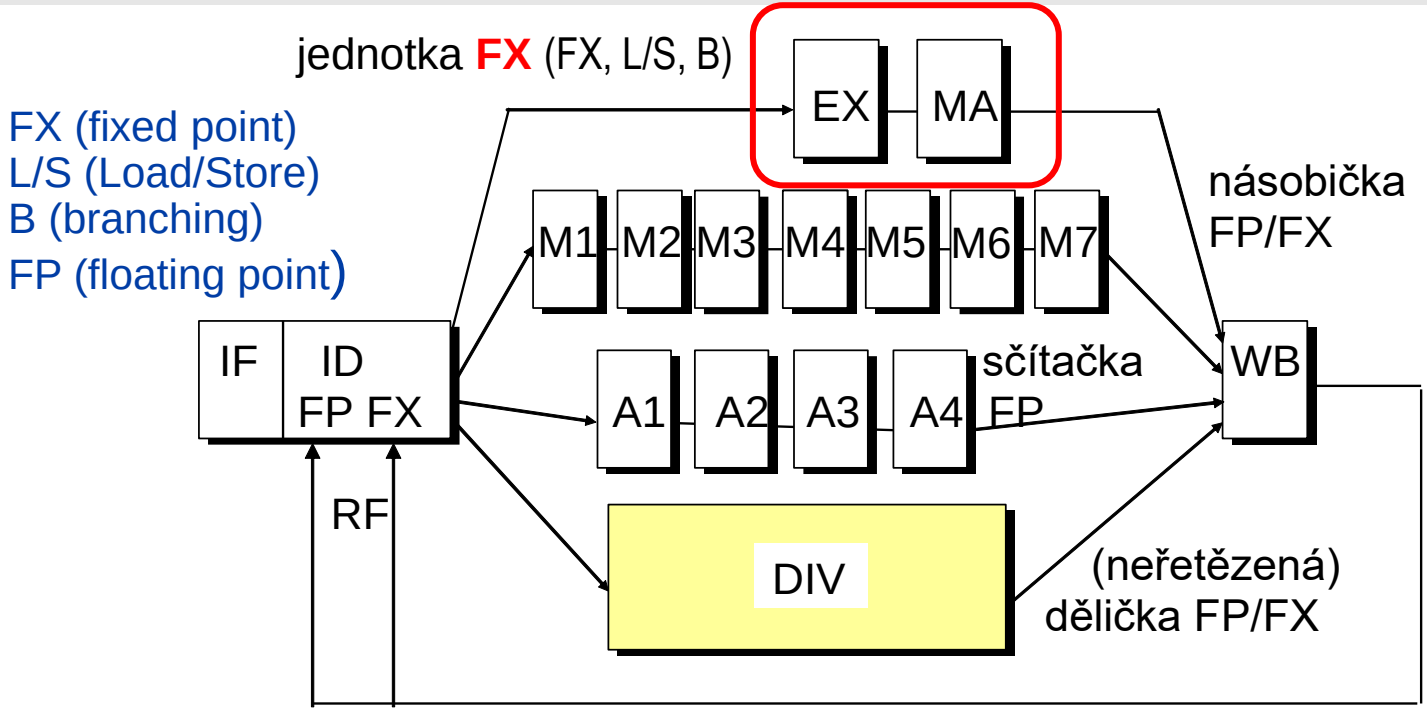
Model řetězeného zpracování



Pozor, HW linka směřuje nahoru a posunuje se v diskrétním prostoru času – instrukcí diagonálně doprava dolů!

RF: Zápis (WB) na začátku a čtení (ID) operandů na konci doby taktu je O.K.!

Zřetězené zpracování instrukcí FX, B, L/S, FP



- Do výkonných stupňů může vstoupit nejvýš 1 instrukce/takt
- Ve stupních EX (FX, FP) může být více instrukcí současně, do WB jen 1
- Instrukce mohou zakončovat mimo pořadí pokud nejsou na sobě závislé

KONFLIKTY PŘI ŘETĚZENÉM ZPRACOVÁNÍ INSTRUKCÍ

Klasifikace konfliktů

- Instrukce může záviset na něčem co produkuje dřívější instrukce
  - závislost se může týkat hodnoty dat
    → **datová závislost (konflikt)**
  - závislost se může týkat adresy příští instrukce
    → **řídicí závislost (větvení, výjimky)**

- Instrukce v řetězené lince může potřebovat prostředek, který právě používá jiná instrukce
  → **strukturní závislost (konflikt)**

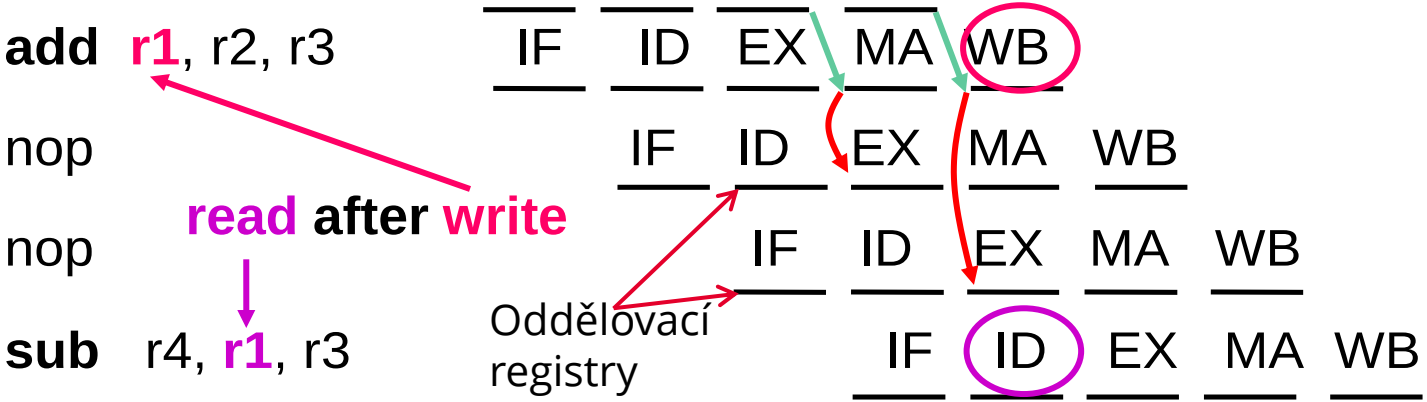
DATOVÉ KONFLIKTY

Datové závislosti mezi instrukcemi

- **Pravé** (true dependencies), též postupové (flow):
  - RAW (Read After Write), instrukce načítá nebo počítá s výsledkem generovaným dřívější instrukcí.
  - **Řešení**: čekání na výsledek, předávání dat, přeuspořádání operací kompilátorem tak, aby byly operace dokončeny v předstihu.
  - **Dva typy**:
    - načtení – použití
    - výpočet – použití
- **Nepravé** (false/name dependencies) – vznikají změnou pořadí vykonání instr.
  - WAR – protiproudé (anti-)
  - WAW – výstupní (output)
  - Instrukce zapisuje tam, odkud předchozí instrukce četla nebo kam zapsala. Nejde o tok dat, ale o konflikt jmen.
  - **Řešení**: přejmenováním.

Konflikt vzniká, když pořadí RAW, WAR nebo WAW není dodrženo.

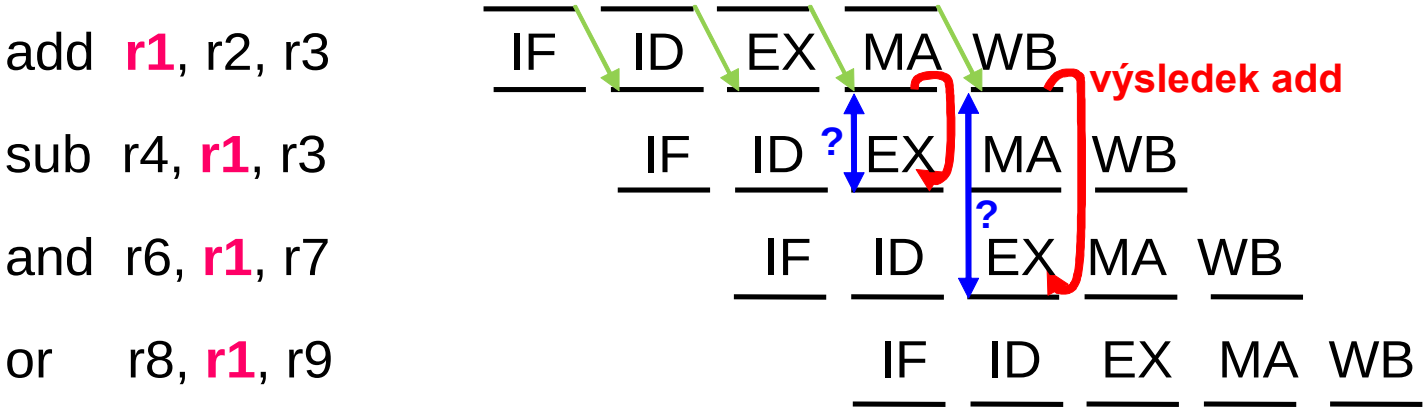
Závislost RAW (read after write)



1. Instrukce **sub** čte výsledek zapsaný instrukcí **add**.
2. Ten je k dispozici až ve fázi **WB**, takže sub musí **2 takty čekat**.

- To by bylo příliš pomalé! Nemusíme čekat až na WB, data jsou k dispozici již dříve (hned po EX fázi add v oddělovacím registru)!
- Nová datová cesta, tzv. ***bypass*** (zkratka), může dostat data z výstupu ALU (resp. z výstupu MA) přímo na vstup ALU bez zdržení (pokuta = 0 taktů) → nopy lze nahradit užitečnými instrukcemi.

Předávání dat mezi instrukcemi podrobněji



? dst (add) je porovnána se **src1**(sub) a **src2**(sub)

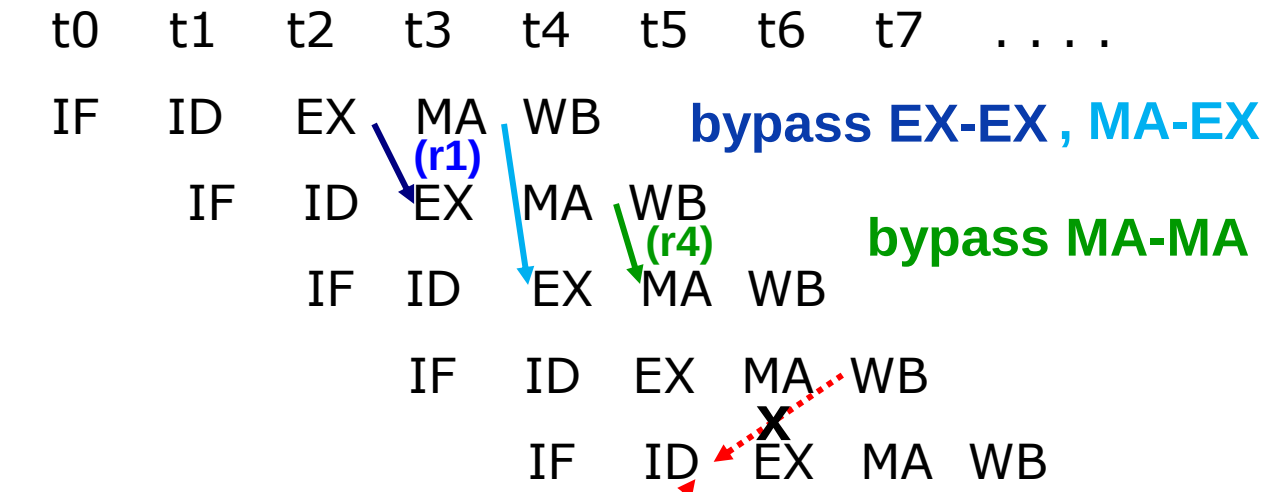
? dst (add) je porovnána se **src1**(and) a **src2**(and)

- shoda + další podmínka pro přípustné op-kódy instrukcí aktivuje **bypass** EX → EX resp. MA → EX
- (Bypass je naznačen jen při aktivaci, ale v HW linky je trvale.)

Předávání dat při RAW (načtení – použití)

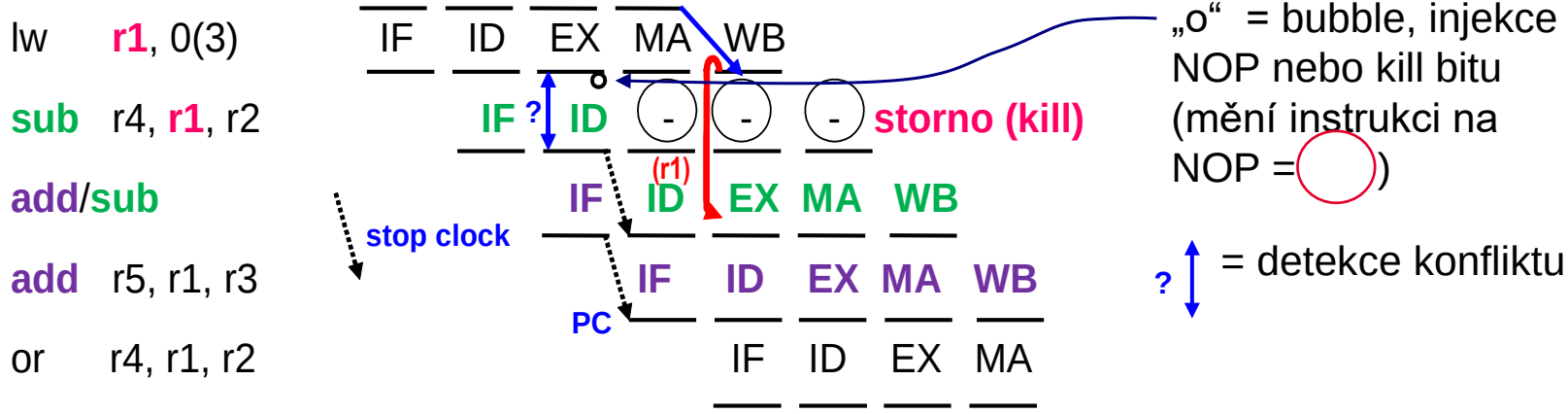
memcpy

```
add  r1,r2,r3
lw   r4,0(r1)
sw   r4,12(r1)
lw   r2,0(r1)
sub  r4,r1,r2
```



- **(r1) = výsledek add, který teprve bude zapsán do reg. r1**
- Kopírování dat lw – sw bez pokuty.
- **Předávat data do minulého taktu nelze!!**
  sub musí 1 takt počkat! (viz příští slajd).

HW řešení: zastavení (stall) dvou stupňů na 1 takt + bypass



Původní verze sub je stornována, zpožděná sub pokračuje;
add (resp. cokoliv po sub) musí být též zpožděno.

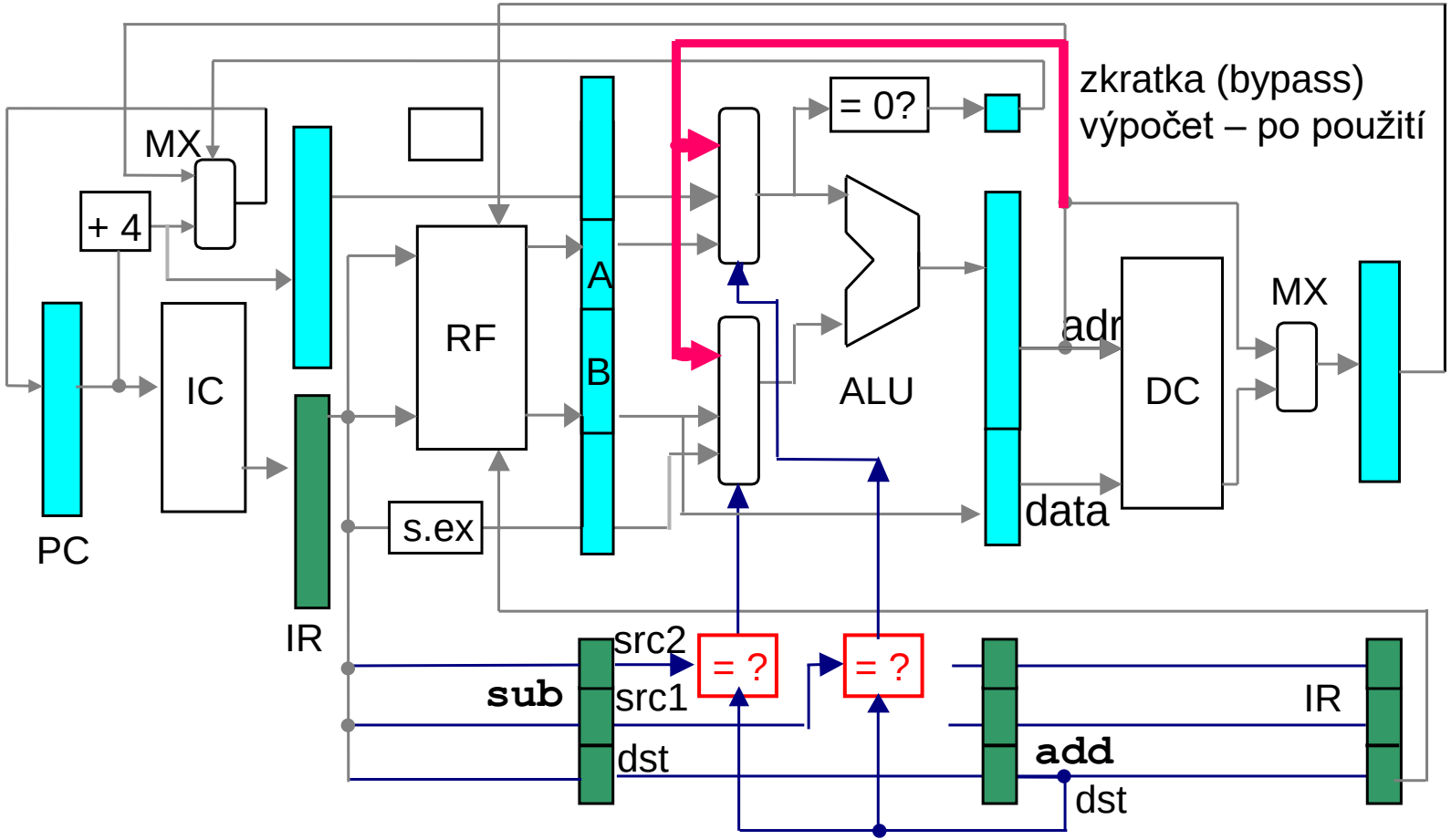
Stručně:

lw	IF	ID	EX	MA	WB			
sub		IF	ID	ID	EX	MA	WB	
add			IF	IF	ID	EX	MA	WB
or					IF	ID	EX	MA

(r1)

stop clock = stall

HW podpora předávání dat EX → EX (RAW)



Nepravé datové konflikty WAR a WAW

WAR

```
i0: fdiv   f0, f2, f4
i1: fadd   f6, f0, f8
i2: fsub   f8, f10, f14
```

Může vzniknout jen při změně pořadí provedení instrukcí, i2 před i1: **i2 změní chybně hodnotu f8 pro i1, čekající na f0**

WAW

pořadí zápisů změněno

```
i1: fmul   f1, f2, f3   ;  IF  ID  EX  EX  EX  EX  CA  WB
i2: fadd   f1, f4, f5   ;      IF  ID  EX  EX  CA  WB
```

```
i1:   fmul f1, f2, f3
i2:   fadd f6, f4, f5
```

odstranění konfliktu přejmenováním cílového registru

V klasické 5 stupňové lince FX konflikty WAR a WAW vzniknout nemohou.

ŘÍDÍCÍ KONFLIKTY

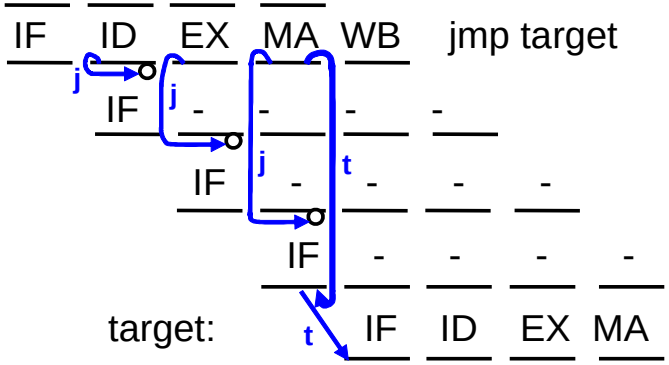
Řídící konflikty

- V průměru je každá 6. – 9. instrukce skoková
  - nepodmíněný skok – jump j, jump register jr
    – do podprogramu: jump and link, jal
  - podmíněný skok
    - bnez/beqz r1, target (test 1 registru ve stupni ID)
    - bne/beq r1, r2, loop (test 2 registrů ve stupni EX)

- Jaká data skok potřebuje:
  - nepodmíněný: op-kód = **j**, PC, rel. adresu (pole Imm 26 bitů) nebo obsah registru
  - podmíněný: op-kód = **b**, PC, rel. adresu (pole Imm 16 bitů), vyhodnocenou podmínku.

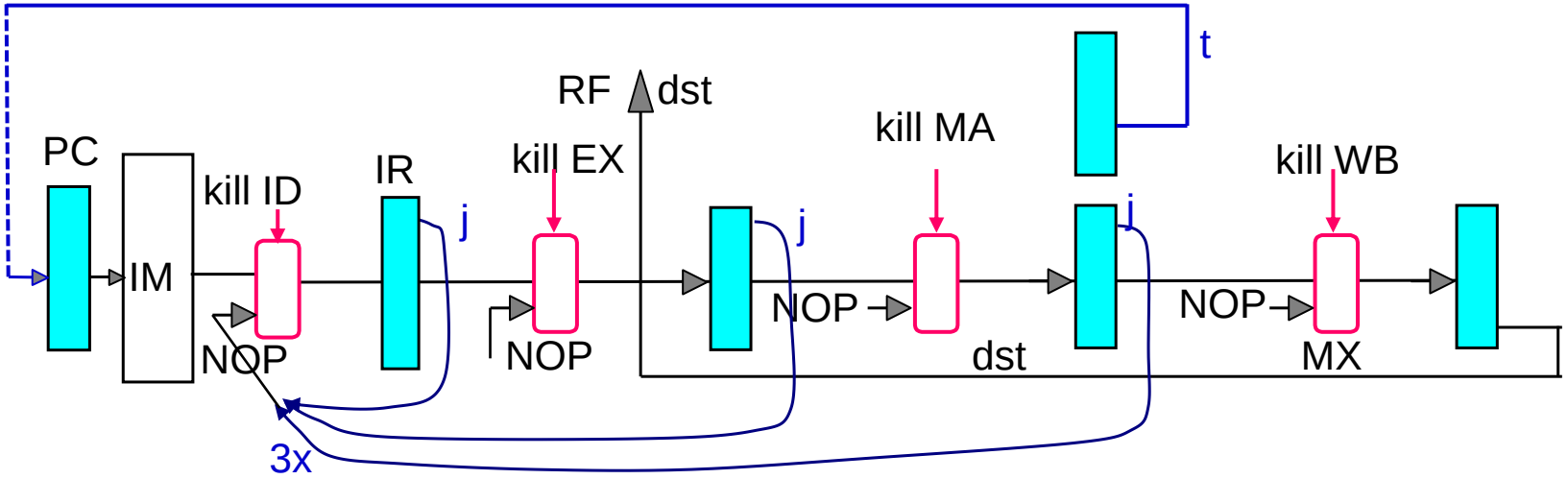
Výpočet cílové adresy (PC + rel. adresa) a podmínky je třeba co nejvíce urychlit!

Nepodmíněný skok **j** relativně k PC

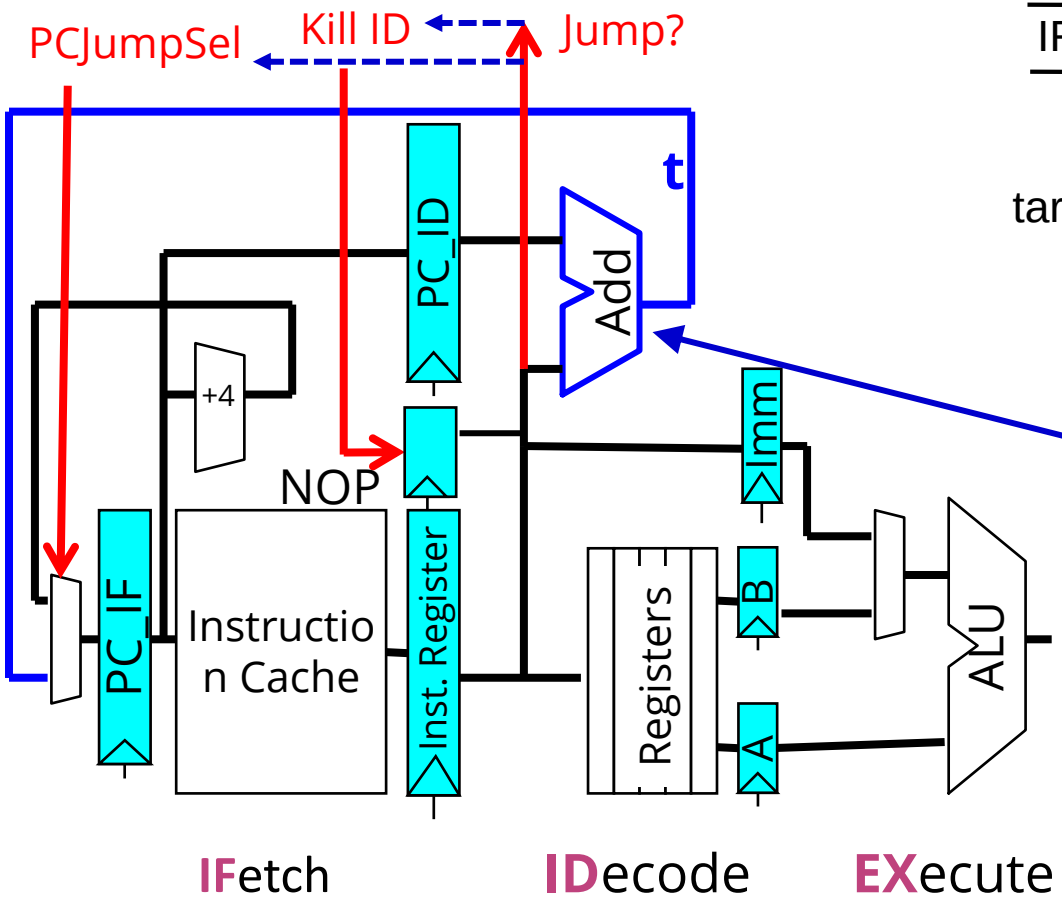


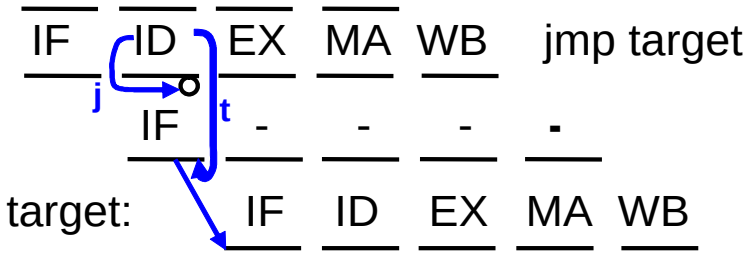
Adresa se spočte v EX, během MA se přepíše do PC → pokuta 3 takty.

Storno již načtené nebo rozpracované instrukce (kill): injekce NOP do IR.



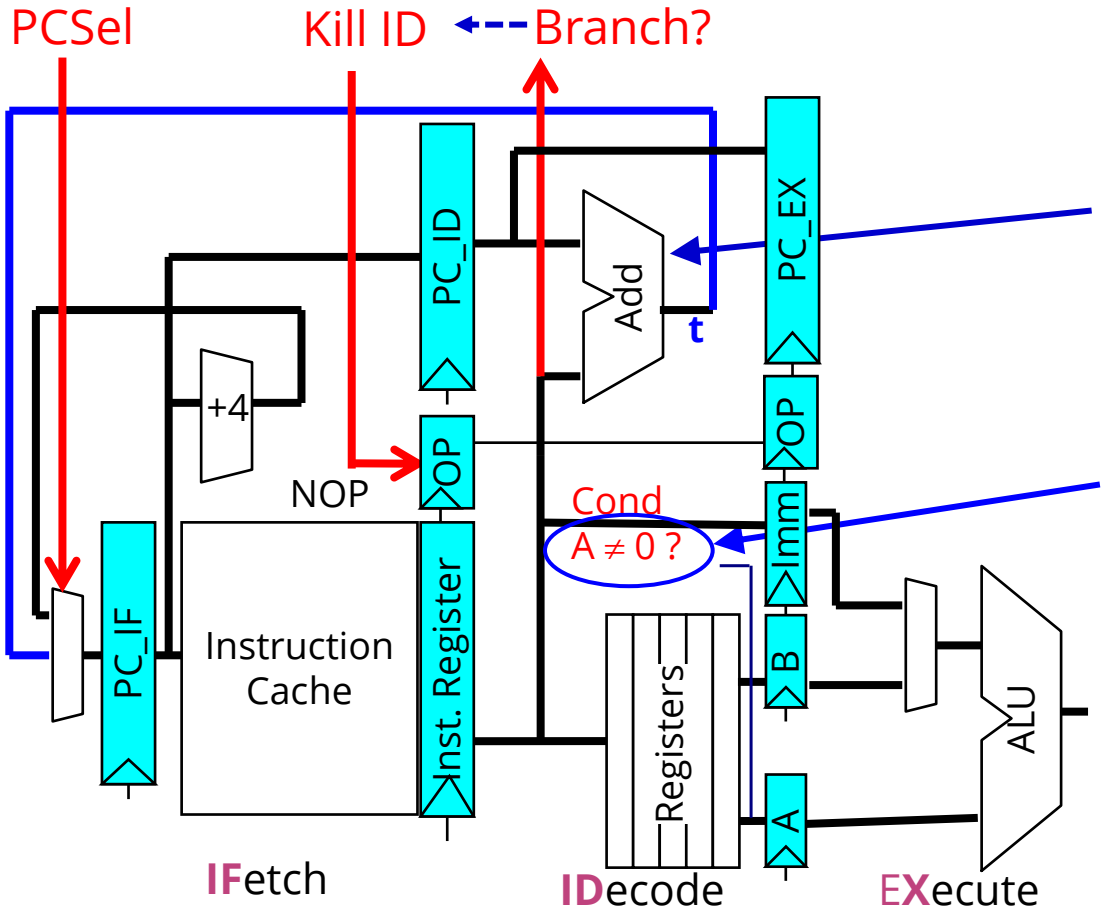
Redukce pokuty u nepodmíněného skoku





Nová sčítačka pro výpočet adresy **t** ve stupni ID: pokuta jen 1 takt

Úprava HW pro podmíněné skoky bnez, beqz



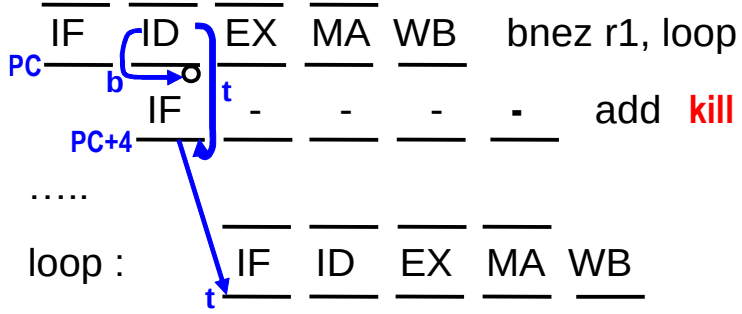
Stejná sčítačka pro výpočet adresy bnez a beqz jako pro jmp

Detekce nuly ve stupni ID

Pokuty u instrukcí bnez/beqz (test na nulu)

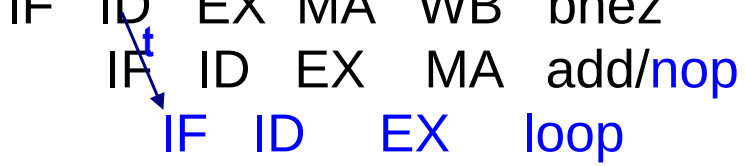
Pokud je testovaný registr zapsán s předstihem, je **pokuta 0 nebo 1 takt**:

```
loop:   …
        bnez  r1, loop
        add   …
```



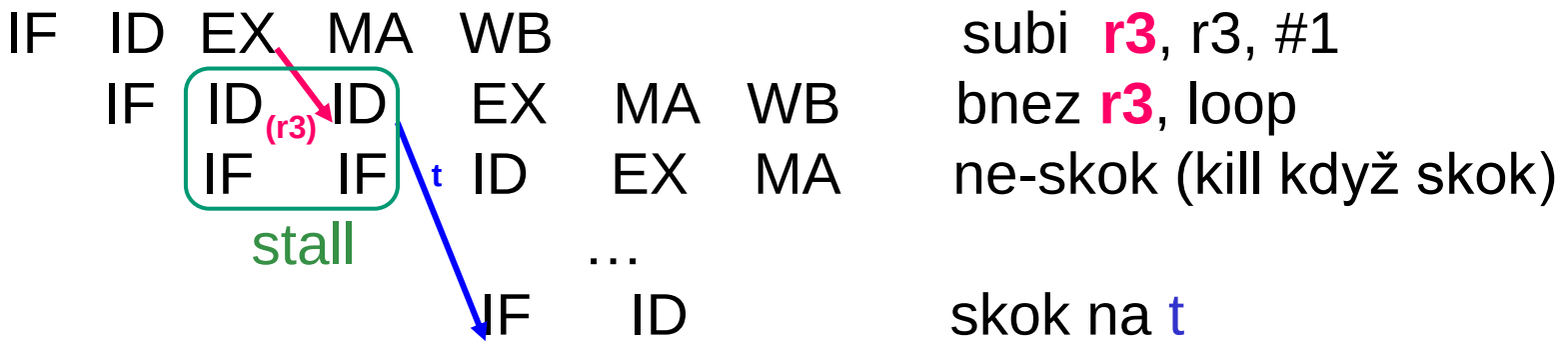
Fixní negativní predikce: jedeme dál s add. Pouze je-li test true, stornujeme add a aktivujeme bypass pro cílovou adresu *t* (skok na loop). Pokuta pak bude:

- 1 takt když se skočí na loop
- 0 když pokračuje add.



Pokuty u instrukcí bnez/beqz

- Pokud je obsah test. registru generován těsně před instrukcí **bnez**, vzniká datová závislost (**r3**) a **pokuta** se zvýší o 1 takt:



Pokuta je tedy

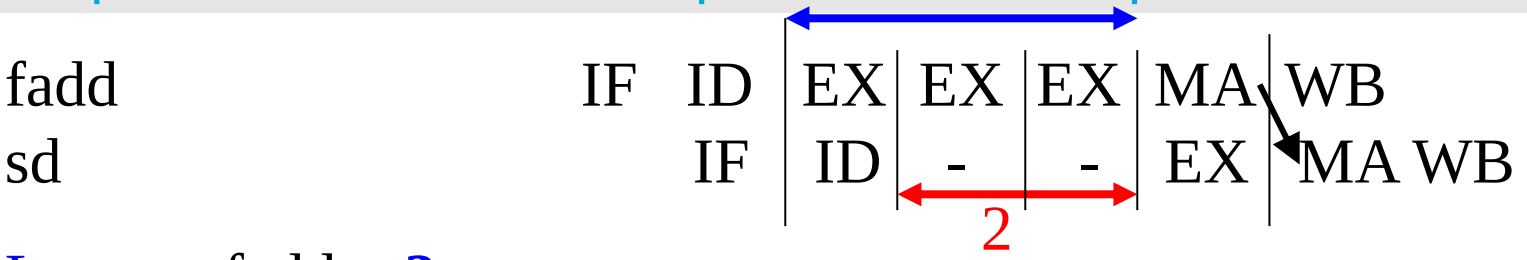
- zpoždění instrukce **bnez** čekající na **subi**: 1 takt
- při skoku na **t** storno instrukce ne-skok: 1 takt

Celkem **1 takt** (ne-skok) **nebo 2 takty** (skok).

Asistence kompilátoru při řetězovém zpracování

- Přejmenování registrů (odstraní nepravé závislosti)
- Vyplnění prázdných taktů užitečnými instrukcemi
- Přehození pořadí instrukcí bez změny sémantiky programu s hlídáním zpoždění **mezi operacemi**, případně i spekulativní přehození pořadí
- Rozbalení smyček
- SW řetězení smyček v programu, popřípadě v kombinaci s jejich rozbalením

Kompilátor zná latence operací a mezioperační latence

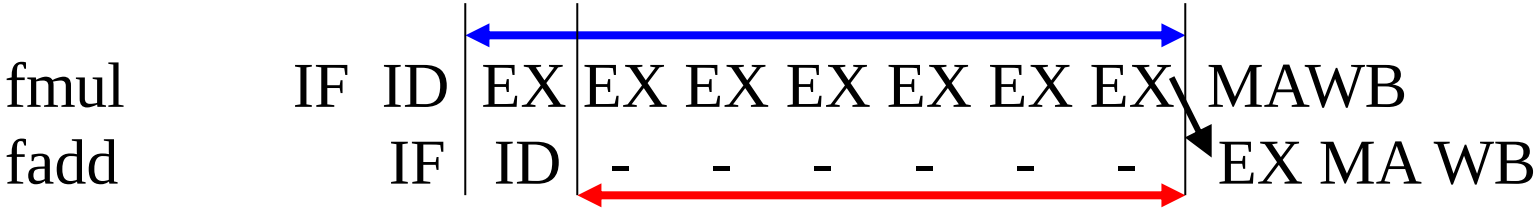


Latence fadd = **3**

Mezioperační latence (fadd, sd) = **2**

(*producent*, *konzument*)

Latence fmul = **7**



Mezioperační latence (fmul, fadd) = **6**

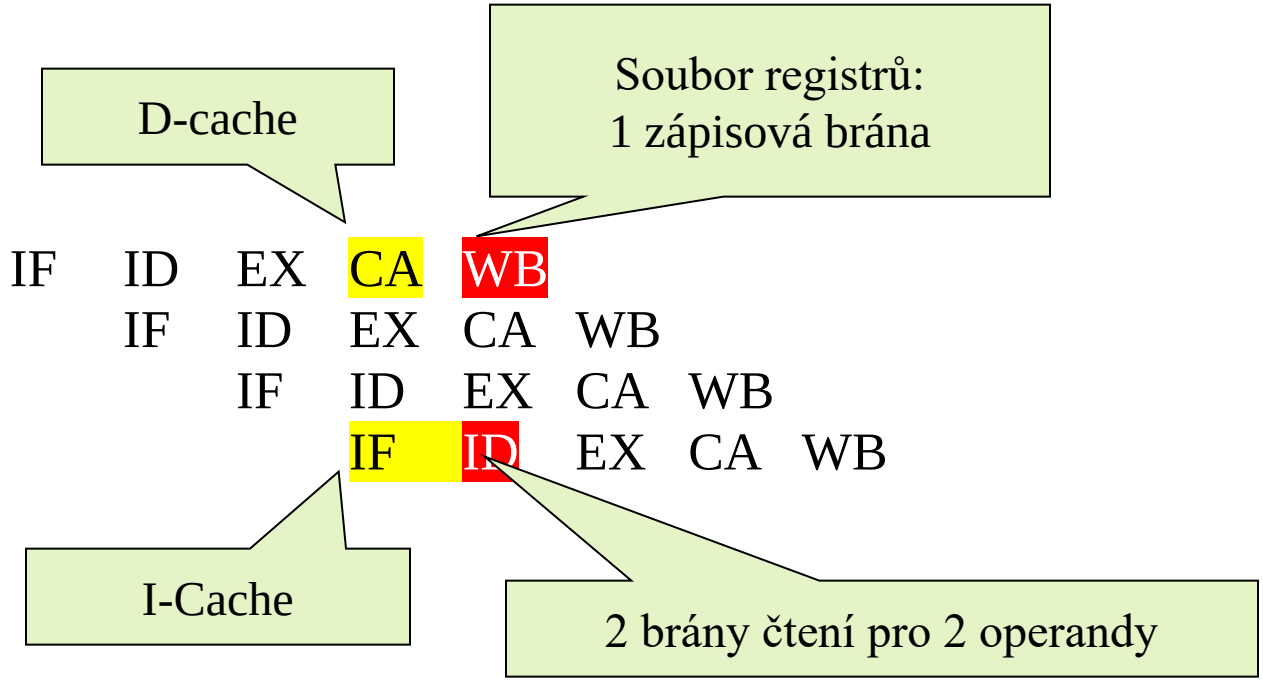
STRUKTURNÍ KONFLIKTY

Strukturní konflikty

… když 2 instrukce potřebují stejný prostředek.

```
div   r1, r2, r3
div   r4, r2, r5
```

serializováno



PŘÍKLADY

Příklad 3: Inkrementace prvků vektoru (FX)

```
for (i = 0; i < 1000; i++) x[i]++;
```

```
loop  lw    r1,0(r2)
      addi  r1,r1,#1
      sw    r1,0(r2)
      addi  r2,r2,#4
      sub   r4,r3,r2
      bnez  r4,loop
```

- 5 stupňová řetězená linka
- v cache 100% úspěšnost

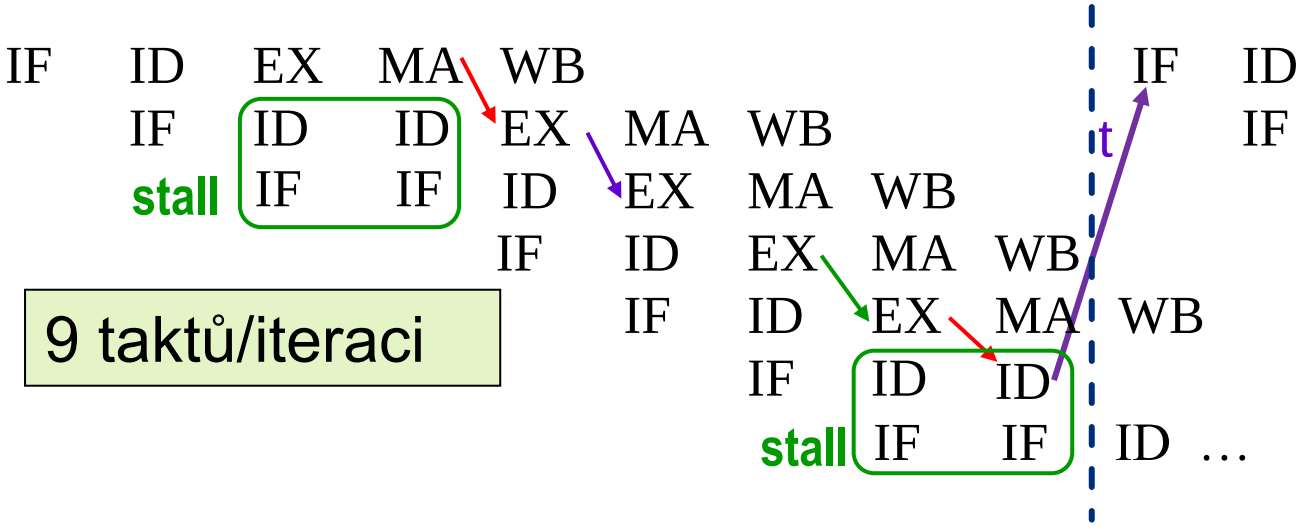
Kolik taktů trvá smyčka

1. bez předávání dat?
2. s předáváním dat?
3. s předáváním dat a s přeskládáním instrukcí (provádí kompilátor).

Nakreslete diagramy řetězového zpracování.

Příklad 3: Smyčka s předáváním dat

```
loop:
lw   r1, 0(r2)
addi r1, r1, #1
sw   r1, 0(r2)
addi r2, r2, #4
sub  r4, r3, r2
bnez r4, loop
…
```

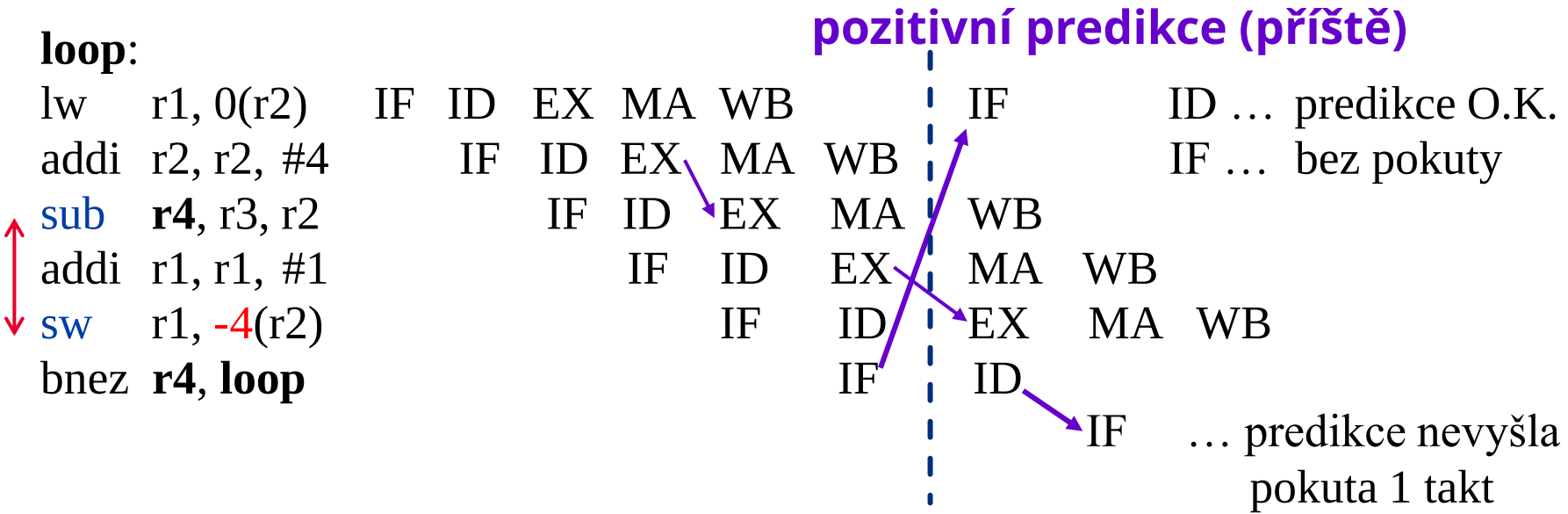


9 taktů/iteraci

bnez zapíše cílovou adresu *t* do PC na konci fáze ID (bnez)

Bez předávání dat by musela každá závislá instrukce (konzument) počkat s načtením operandu až by se fáze WB (producent) kryla s fází ID(konzument) → **17 taktů/iteraci!**

Příklad 3: Smyčka naplánovaná kompilátorem



Po úspěšné **pozitivní** predikci ve stupni IF(bnez) hned na cílovou adresu: 6 taktů/iteraci.

Příklad 4: Floating point smyčka

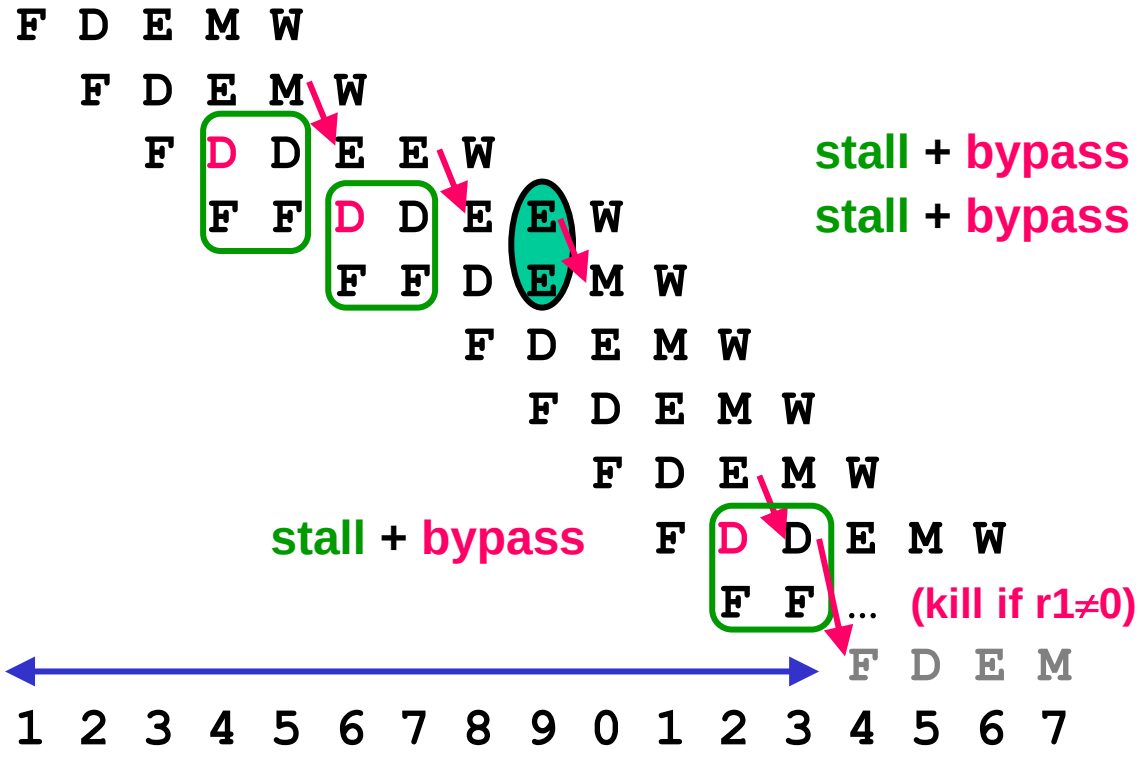
- Pro následující úsek kódu s operacemi FP (2 takty EX)

```
loop:ld     f0,0(r1)
     ld     f4,0(r2)
     fmul   f0,f0,f4
     fadd   f2,f2,f0
     sd     f2,0(r3)
     addi   r3,r3,#-8
     addi   r2,r2,#-8
     addi   r1,r1,#-8
     bnez   r1,loop
```

- Naznačte časování a najděte počet taktů těla smyčky
  - při 100 % zásahů v cache
  - s předáváním dat
  - a když jsou podmíněné skoky rozhodnuty ve stupni ID.

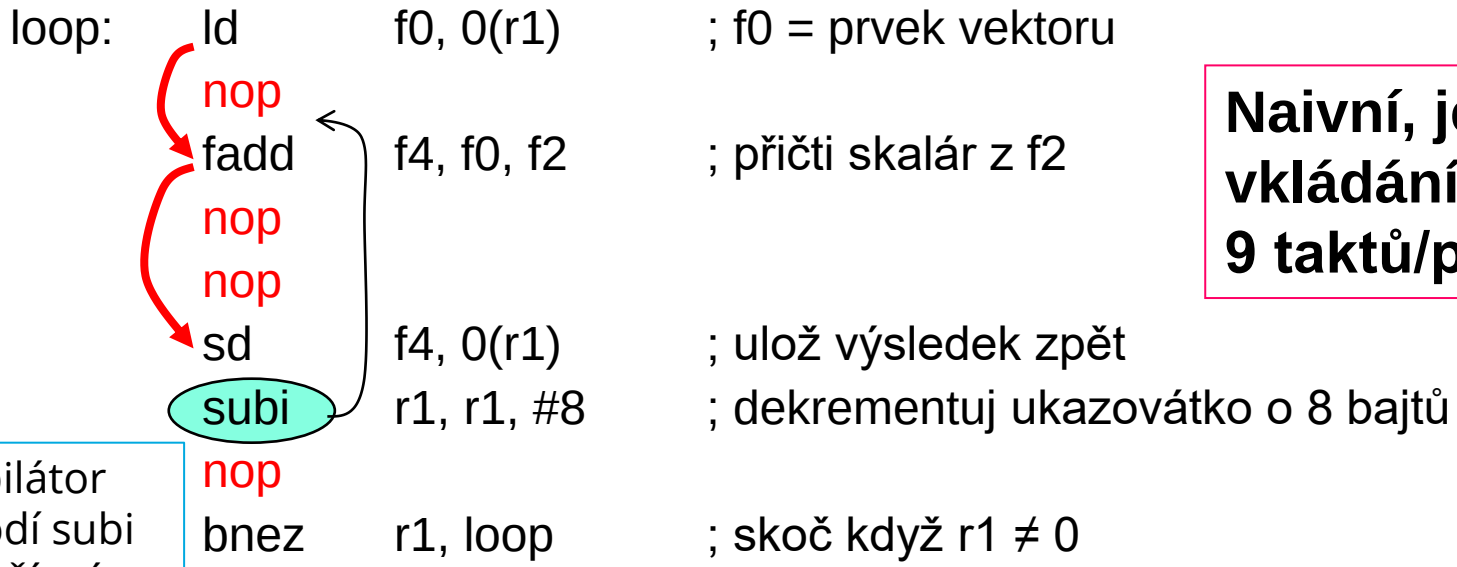
Příklad 4: Řešení

```
L: ld     f0,0(r1)
   ld     f4,0(r2)
   fmul   f0,f0,f4
   fadd   f2,f2,f0
   sd     f2,0(r3)
   addi   r3,r3,#-8
   addi   r2,r2,#-8
   addi   r1,r1,#-8
   bnez   r1,L
   ...
   ld     f0,0(r1)
```



- Ve stupních E (FX, FP) může být více instrukcí současně
- Instrukce mohou zakončovat mimo pořadí pokud nevznikne konflikt WAR nebo WAW

Příklad 5: Inkrementace prvků vektoru ve FP

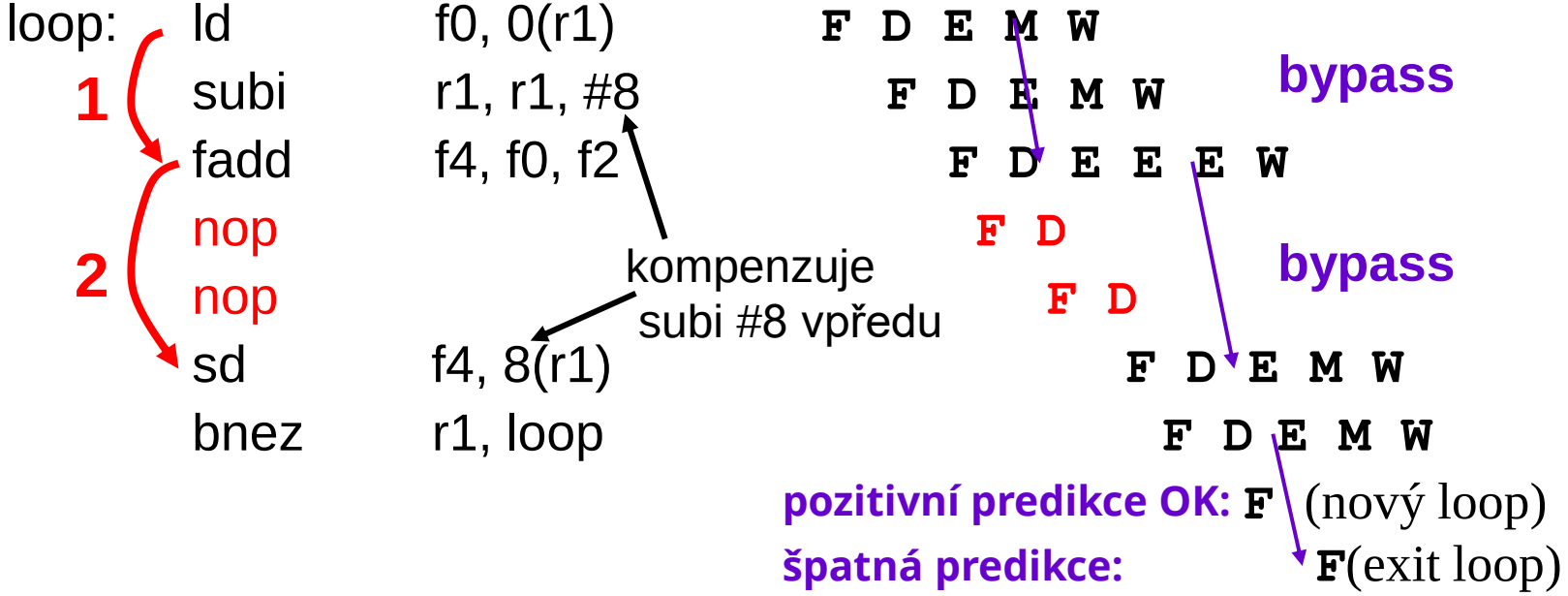


Naivní, jen vkládání NOP, 9 taktů/prvek

Kompilátor přehodí subi na lepší místo

Producent	Konzument	Meziop. latence
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Příklad 5: Základní naplánování kompilátorem



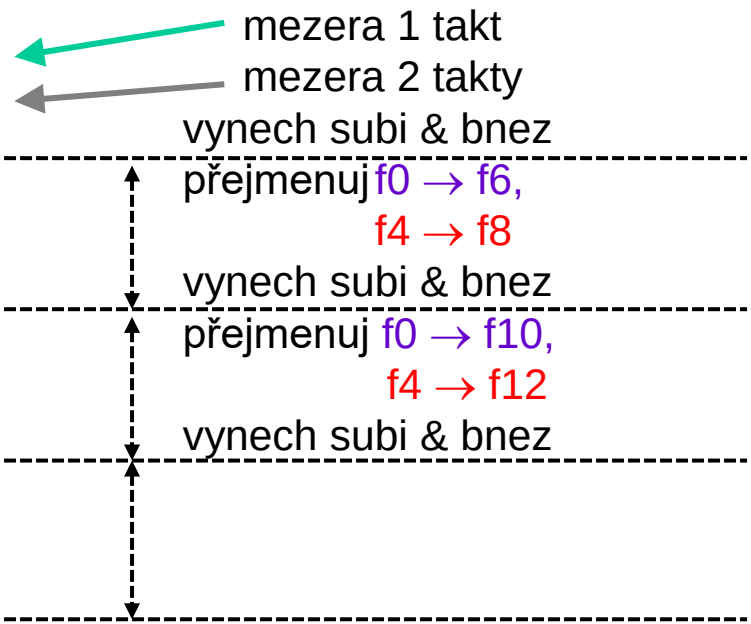
1 a 2 takty mezioperační latence splněny

7 taktů na element vektoru, jen 3 užitečné (výpočet).
Další zmenšení režie je možné pomocí rozbalení smyčky.

Příklad: Naivní rozbalení smyčky 4x

přejmenování registrů a změna odsazení (offsetu) paměť. operandů

```
loop:   ld      f0, 0(r1)
        fadd    f4, f0, f2
        sd      0(r1), f4
        ld      f0, -8(r1)
        fadd    f4, f0, f2
        sd      -8(r1), f4
        ld      f0, -16(r1)
        fadd    f4, f0, f2
        sd      -16(r1), f4
        ld      f0, -24(r1)
        fadd    f4, f0, f2
        sd      -24(r1), f4
        subi    r1, r1, #32
        bnez    r1, loop
```



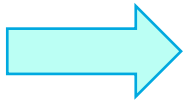
- Toto rozbalení smyčky RAW konflikty neodstraní
- Stále je nutné vkládat NOPs
- Pouze redukována režie smyčky

27 taktů včetně prázdných, 6,8 taktů/iteraci

Příklad: Rozbalení smyčky minimalizující prostoje

```
loop:   ld      f0, 0(r1)
        fadd    f4, f0, f2
        sd      0(r1), f4
        ld      f0, -8(r1)
        fadd    f4, f0, f2
        sd      -8(r1), f4
        ld      f0, -16(r1)
        fadd    f4, f0, f2
        sd      -16(r1), f4
        ld      f0, -24(r1)
        fadd    f4, f0, f2
        sd      -24(r1), f4
        subi    r1, r1, #32
        bnez    r1,loop
```

27 taktů včetně prázdných
6,8 taktu/element



```
loop:   ld      f0, 0(r1)
        ld      f6, -8(r1)
        ld      f10, -16(r1)
        ld      f14, -24(r1)
        fadd    f4, f0, f2
        fadd    f8, f6, f2
        fadd    f12, f10, f2
        fadd    f16, f14, f2
        sd      0(r1), f4
        sd      -8(r1), f8
        sd      -16(r1), f12
        subi    r1, r1, #32
        sd      8(r1), f16
        bnez    r1, loop
```

14 taktů, žádný prázdný
3,5 taktu/element

Změna pořadí zpracování instrukcí

4 iterace prolnuty

Odstraněny všechny RAW konflikty

Přerušení a výjimky při řetězeném zpracování

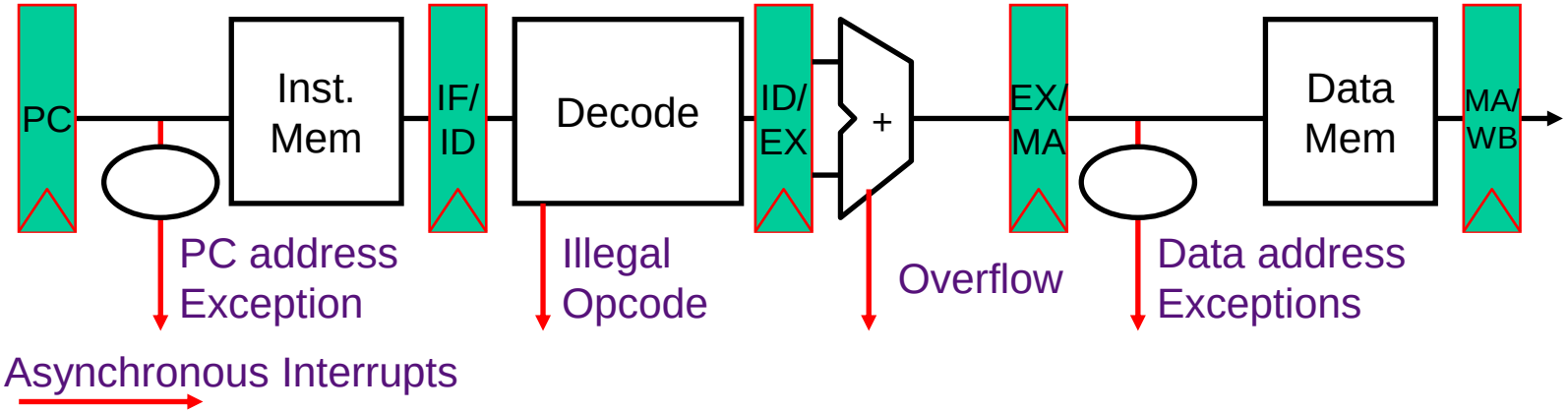
Jde o události, které vyžadují zpracování systémovým programem.

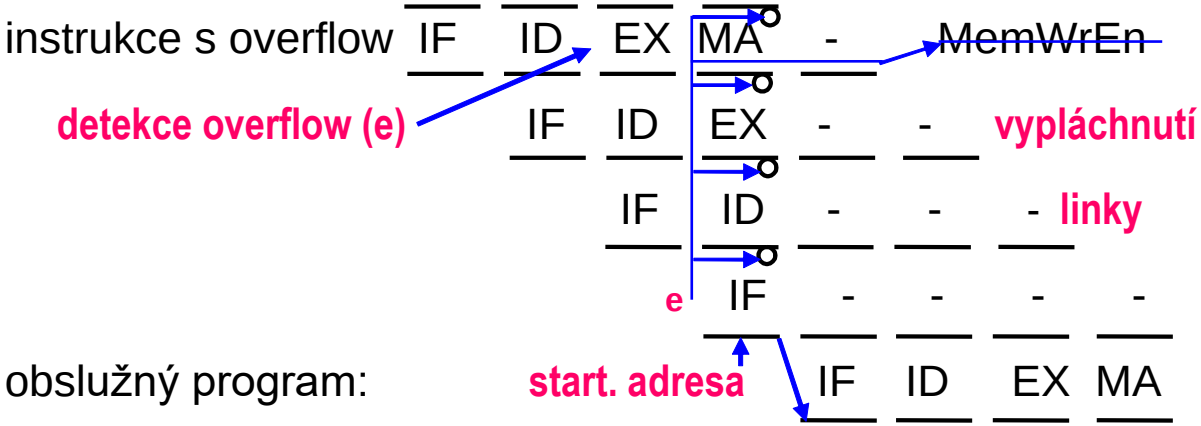
- **Výjimka (exception)**: neobvyklá interní událost při zpracování konkrétní instrukce (např. dělení nulou, nedefinovaný op-kód, přetečení, výpadek stránky) Obecně instrukce nemůže být dokončena a musí být *restartována* po zpracování výjimky
  - to vyžaduje anulaci účinků jedné nebo více částečně provedených instrukcí (zotavení).
  - Trap: speciální instrukce volání systému –přechod do privilegovaného módu kernelu (SW přerušení).
- **Přerušení:** HW signál přepínající procesor na nový proud instrukcí při výskytu nějaké *externí události (*požadavek na obsluhu zařízení I/O, signál časovače, porucha napájení, HW porucha)

Obsluha přerušení

- Když se procesor rozhodne zpracovat přerušení
  - zastaví běžící program u instrukce I_i, a dokončí všechny instrukce až do I_{i-1}
  - uloží PC instrukce I_i do speciálního registru (EPC, Extra PC),
  - zablokuje další přerušení a předá řízení určenému programu obsluhy přerušení běžícímu v kernelu OS.
- Obslužný program: přečte registr *Cause* (indikuje příčinu)
  - Buď ukončí program nebo restartuje u instrukce I_i
  - Používá zvláštní instrukci nepřímého skoku RFE *(return-from-exception)*, která
    - obnoví uživatelský režim CPU, EPC → PC
    - obnoví stav hardwaru a stav řízení
    - povolí přerušení.

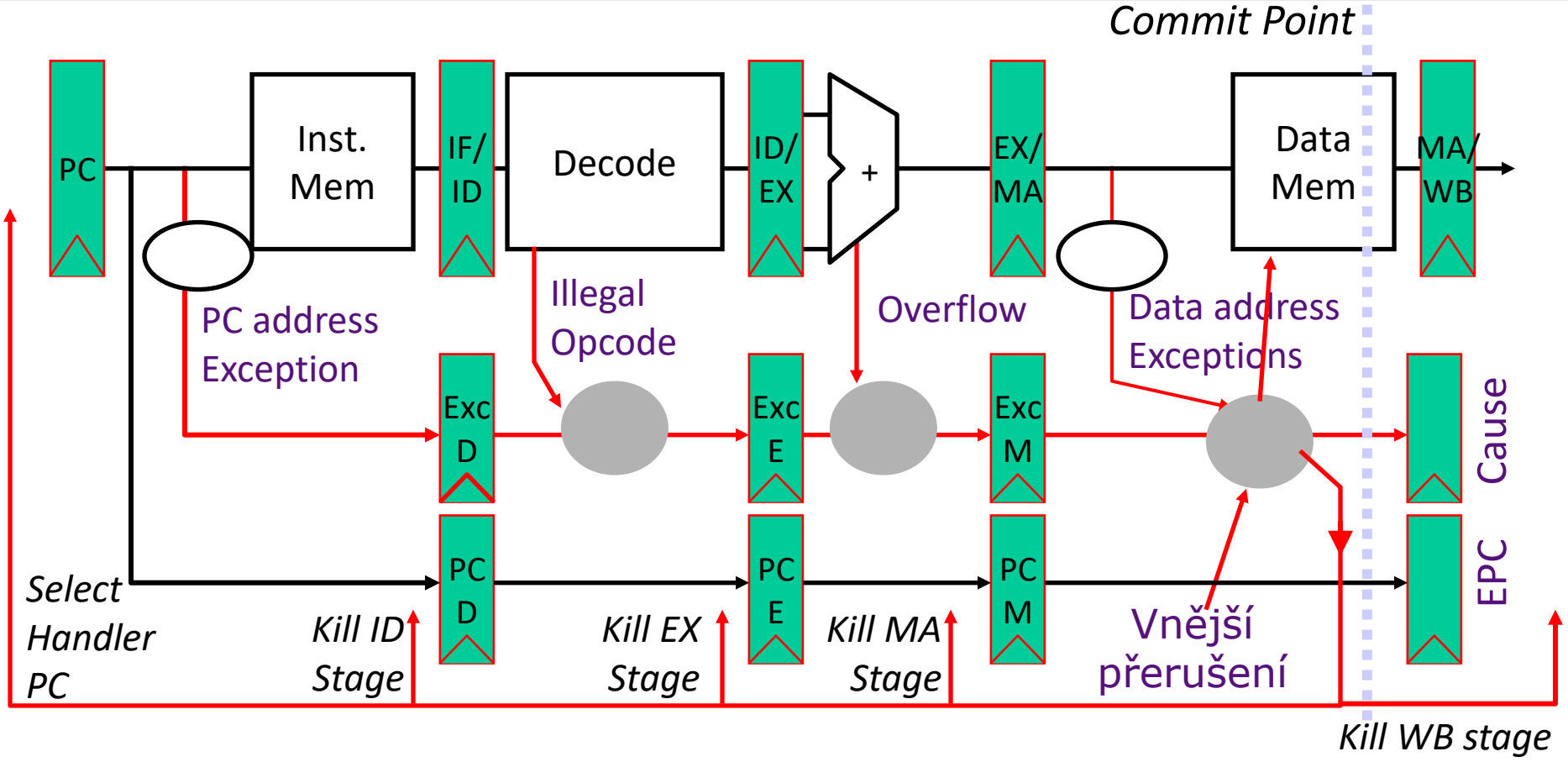
Zpracování výjimek (5 stupňová linka)





NOP do všech
4 registrů podél linky
= kill 4 rozpracované
instrukce

Zpracování výjimek (5 stupňová linka)



Zpracování výjimek (5 stupňová linka)

- Příznak výjimky Exc je udržován v lince až do „bodu zlomu“ (commit point, stupeň MA). Zápis do paměti nebo do registru je nevratný, u instrukce s výjimkou nesmí proběhnout (**MemWrEn, RegWrEn**)

- Vnější přerušení jsou injektovány do bodu zlomu a uplatní se přednostně před výjimkami.
- Když výjimka doputuje do bodu zlomu: aktualizuj stavový registr Cause a EPC register, zahoď (kill) částečně provedené instrukce (NOP do oddělovacích registrů, do sekce op-kódu).
- Injektuj adresu obslužného programu do PC.

Pokračování příště